Génération de séquences de test pour l'accélération d'assertions

L. Damri

UNIVERSITÉ DE GRENOBLE

THÈSE

Pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ DE GRENOBLE

Spécialité : **Nanoélectronique et nanotechnologies**

Arrêté ministériel : 7 août 2006

Présentée par

Laila DAMRI

Thèse dirigée par Mme. **Laurence PIERRE**

Préparée au sein du Laboratoire ‘‘Technique de l'Informatique et de la Microélectronique pour l'Architecture de systèmes intégrés (TIMA)’’

dans l'École Doctorale Electronique, Electrotechnique, Automatique & Traitement du Signal

Génération de séquences de test pour l'accélération d'assertions

Thèse soutenue publiquement le **17/12/2012,**
devant le jury composé de :

M. Lakhdar SAIS
Professeur, Université d'Artois, Rapporteur
Mme. Hélène COLLAVIZZA
Maitre de conférences HDR, Université de Nice-Sophia Antipolis, Rapporteur
M. Régis LEVEUGLE
Professeur, Institut Polytechnique de Grenoble, Président
Mme. Laurence PIERRE
Professeur, Université Joseph Fourier, Directrice de thèse

ISBN : 978-2-11-129171-3

Génération de séquences de test pour l'accélération d'assertions

Sous la direction de Pr. Laurence PIERRE

Laila DAMRI

Le 17 Décembre 2012

Grenoble

France

ISBN : 978-2-11-129171-3

Remerciements

Cette thèse doit beaucoup aux nombreuses personnes qui m'ont encouragé, soutenu et conforté au long de toutes ces années. Qu'elles trouvent dans ce travail l'expression de mes plus sincères remerciements.

Je souhaite remercier Mme. Laurence Pierre, Professeur et Responsable de l'équipe VDS d'avoir encadré cette thèse. Grâce à elle j'ai pu apprendre beaucoup de choses importantes et dont certaines fortes utiles pour la suite de ma carrière professionnelle. Je n'oublie pas aussi son aide précieuse dans la relecture et la correction de ma thèse. La justesse de ses critiques a été très constructive et utile ; je lui exprime ma très profonde gratitude.

Je tiens à remercier Mme. Dominique BORRIONE, Professeur et Directrice du Laboratoire TIMA pour m'avoir accueilli au sein du Laboratoire TIMA.

Je remercie les membres du jury d'avoir accepté de participer à mon jury de thèse. M. Lakhdar SAIS, Professeur à l'université d'Artois, et Mme. Hélène COLLAVIZZA, Maître de conférences HDR à l'‘université de Nice-Sophia Antipolis pour avoir accepter de rapporter ma thèse. Je remercie également M. Régis LEVEUGLE Professeur à l'Institut Polytechnique de Grenoble d'avoir accepter de présider le jury. Je leur remercie aussi pour leurs remarques pertinentes sur les travaux que j'ai réalisé dans ma thèse, ce qui m'a permis de m'orienter à avancer vers de nouvelles perspectives.

J'adresse mes remerciements à tous les membre (anciens et nouveaux) de l'équipe VDS : Renaud, Yann, Katel, Negin, Giorgios, Luca, Alex, pour tous les échanges techniques et scientifiques, et pour leur sympathie et leur accueil pendant ces trois ans de thèse. J'aimerais spécialement remercier Zeineb pour son amitié et son grand soutien durant ma thèse, ainsi lui souhaiter bon courage et bon continuation. Je remercie également Oussama pour l'aide et le soutien qu'il m'a apporté.

J'aimerais également remercier mon oncle Abdelhak, pour tout l'aide qu'il m'a apporté durant mon séjour en France, et je tiens à lui exprimer ma sincère gratitude.

Je tiens aussi à remercier mes amis de Montpellier, grâce à eux, mes weekends sont devenus des moments très agréables qui m'ont aidé à surmonter plusieurs périodes difficiles durant ma thèse : Hajer, Azhar, Mohamed et Nawfal. Ma très chère amie Kaouthar (ex-Montpelliéraine et marsellienne actuellement), je ne peux que lui dire un grand merci pour son amitié durant toutes ses années, son soutien et ses conseils personnelles et professionnelles, et lui souhaiter bon continuation. Je remercie Amine pour son grand soutien et ses encouragements pour me pousser à avancer quand ça n'allait pas ; ses remarques et ses conseils m'ont étaient d'une grande utilité et grâce à lui aussi j'ai pu arriver à finir la rédaction et soutenir ma thèse.

Mes dernières pensées iront vers ma famille à qui j'adresse tous mes remerciements, ma gratitude et mon énorme amour : mon père Mohamed, ma mère Fatima, mon frère Reda, mes sœurs Nora, Houda et Imane, qui m'ont permis de poursuivre mes études jusqu'à aujourd'hui, grâce à leur encouragement et leur énorme soutien, et je tiens leur dire que sans eux je n'en serais pas là.

Enfin, je tiens à remercier toutes mes connaissances de m'avoir accompagné et aidé pendant mes années d'études.

Table des Matières

Listes des Figures

Chapitre 1.

Introduction générale

I. Introduction

L'amélioration constante de la technologie des circuits intégrés conduit les industriels à développer des circuits plus complexes, plus performants et qui impliquent de plus en plus l'utilisation de grands composants préexistants (blocs IP), mais nécessitent plus d'efforts pour leur vérification. C'est un problème qui ne cesse de croître désormais de jour en jour. Les industriels ont besoin de solutions qui garantissent le bon comportement des circuits, en passant par la vérification et aussi la correction de la fabrication à l'aide de différentes étapes de test.

I.1. Vérification des circuits intégrés

La vérification constitue une étape principale dans le flot de conception (voir Figure I.1). Découvrir les failles dans le fonctionnement d'un circuit avant sa fabrication peut épargner de grandes pertes. Cependant, avec la croissance de la complexité des systèmes, le processus de vérification pendant la conception est devenu une tâche de plus en plus difficile. Les méthodes formelles et semi-formelles peuvent apporter des solutions, notamment la vérification à base d'assertions (Assertion-Based Verification, ABV). Énoncer des assertions logiques et temporelles en cours de conception du circuit peut apporter des avantages et permettre de découvrir des erreurs de conception le plus tôt possible. Les exigences de la vérification dépendent directement à la complexité des circuits. Plus le circuit gagne en complexité, plus la vérification devient une tâche rude. Le développement de modèles fiables est lié à la manière dont ils sont conçus, mais plus encore à la façon dont ils seront vérifiés.

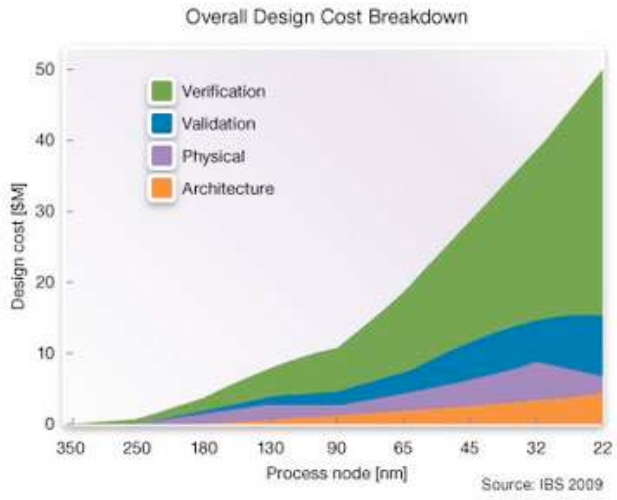


Figure I.1: Le coût de la vérification dans le processus de conception d'un circuit

Dans le flot de conception des systèmes numériques à très grande densité d'intégration, les phases de vérifications fonctionnelle et temporelle deviennent des étapes de validation incontournables. Elles visent à s'assurer que le circuit, une fois fabriqué, réalisera la fonction spécifiée.

Des assertions peuvent permettre la spécification, et peuvent être utilisées de plusieurs façons selon les besoins des concepteurs. Il existe deux grands axes de vérification :

- La vérification statique (voir sous-section I.1.1). Ce type de vérification regroupe les outils de vérification formelle comme le *model-checking* et *l'equivalence-checking*.
- La vérification dynamique (voir sous-section I.1.2). C'est une vérification effectuée en cours de simulation ou d'émulation.

I.1.1. La vérification statique

La vérification statique est assurée par des outils de vérification formelle. Les méthodes formelles sont des approches qui permettent de raisonner exhaustivement, dans une logique mathématique, afin de démontrer la validité d'un système par rapport à une spécification donnée. On distingue deux grandes familles d'outils :

1. Les *Equivalence Checkers* : ces outils se basent sur la présence de deux modélisations du système : la première modélisation représente le modèle de référence, et la deuxième le modèle à comparer et vérifier avec la référence.
2. Les *Model Checkers* : ces outils permettent de vérifier si un modèle du système satisfait une spécification, généralement associée à une (ou des) assertion(s) logico-temporelle(s).
   Le Model Checking [1] fait partie de la famille des techniques de vérification automatisées. Cette technique consiste à créer d'abord un modèle du circuit à vérifier sous forme d'automate d'états fini. Chaque nœud représente un état du système et chaque transition représente une évolution possible du système d'un état donné vers un autre état. Des techniques algorithmiques permettent de parcourir symboliquement cette structure d'automate afin d'y vérifier la validité des propriétés. Si le système contient une erreur, le Model checking va générer un contre-exemple qui peut être utilisé pour identifier la source de l'erreur. Par contre, cette technique souffre du problème d'explosion combinatoire en fonction du nombre d'états. Ce problème se produit surtout dans les systèmes qui ont plusieurs composants qui peuvent interagir les uns avec les autres, ou pour les structures de données qui peuvent prendre un grand nombre de valeurs, ce qui peut mener à un nombre d'états globaux très important.
   Le Model Checking est une méthode automatique qui vérifie qu'un système d'état fini satisfait une propriété temporelle. La structure de BDD (Binary Decision Diagram) [2] peut être utilisée pour représenter symboliquement l'espace d'état fini. Comme alternative, des méthodes améliorées comme le BMC (Bounded Model Checking) [3] sont proposées. La validation d'un système en utilisant cette méthode peut garantir la couverture nécessaire. Dans cette approche une propriété p est considérée sur un nombre fini d'applications de la fonction de transition δ. Le BMC est l'une des meilleures techniques utilisées pour la vérification formelle.

I.1.2. La vérification dynamique

La vérification dynamique est une vérification des assertions effectuée en cours de simulation ou d'émulation. La simulation est l'approche la plus répandue pour la vérification des systèmes numériques. Ce type de vérification est basé sur un modèle HDL (Hardware Description Language) du système étudié. Ce modèle décrit le comportement global ou partiel du système. Il est stimulé avec une série de vecteurs de test et les résultats obtenus sont comparés avec des valeurs prévues dans les mêmes conditions. L'avantage majeur de la vérification par simulation apparaît dans le fait qu'elle ne nécessite pas un prototype matériel. Cependant, cette méthodologie est gourmande en temps CPU, en outre elle requiert la génération de stimuli.

Une bonne couverture de fonctionnalité d'un circuit exige une vérification et génération exhaustive des vecteurs de test, ce qui devient très lourd en terme de temps de simulation. Bien que la génération aléatoire des vecteurs de test puisse être intéressante, elle manque de précision et peut conduire à une couverture faible des comportements à vérifier dans un circuit. Une solution alternative consiste à assurer une génération dirigée des vecteurs de test précis qui ciblent l'observation d'une fonctionnalité ou d'une erreur donnée.

I.2. Contexte de cette thèse

La vérification dynamique permet de détecter des défauts de fonctionnement près de leurs sources, ce qui permet un gain de temps considérable de débogage. Un circuit est initialement décrit à l'aide d'un langage de description matérielle tel que VHDL (*VHSIC Hardware Description Language*, et VHSIC signifie *Very High Speed Integrated Circuit*) [4] ou Verilog [5]. Ces langages permettent aux concepteurs de représenter les comportements attendus du circuit, ainsi que son architecture. Les instructions traduisent une configuration logique de portes et de bascules (niveau "portes logiques"), ou une description du fonctionnement du circuit sous la forme des machines d'états finis (FSM) (niveau RTL).

Bien que la simulation permette de tester et vérifier le fonctionnement du circuit, il reste difficile de détecter quand il y a un problème de fonctionnement durant la simulation. La méthode de vérification à base d'assertions apporte une solution à ce problème. Dans la vérification à base d'assertions (ABV) [6], les assertions capturent les comportements attendus des systèmes, elles permettent de reporter des informations sur les erreurs et la couverture. Les assertions :

- Sont intégrées facilement dans la description pour capturer les fonctionnalités du modèle
- Permettent de détecter les erreurs près de leur source, ce qui donne la possibilité d'éliminer plus rapidement les défauts
- Donnent à la fois la possibilité d'une analyse formelle et d'une analyse dynamique.

Ces assertions peuvent être décrites avec des langages spéciaux, capables d'être intégrés dans des descriptions HDL. Les langages qui permettent d'exprimer des assertions les plus connus

sont le langage PSL [7] et SVA [8]. Dans le cas de PSL, les assertions peuvent être placées dans des commentaires précédés par le mot clé "PSL" (voir exemple Figure I.2). Ce mot clé permet aux outils de simulation comme ModelSim de différencier les simples commentaires de ceux qui contiennent des assertions à vérifier. Les langages de spécification de propriétés seront abordés plus en détail dans le chapitre suivant.

La Figure I.2 montre le squelette d'une description matérielle représentant un circuit reconnaisseur de code BCD (que nous verrons plus en détail plus loin) et comment on peut y intégrer des assertions écrites en langage PSL pour les simuler dans l'outil ModelSim (Mentor Graphics). Pendant la simulation, elles sont prises en compte et le simulateur montre en parallèle avec la simulation des signaux du circuit, l'état de ces assertions (vérifiées ou violées). L'assertion output_O traduit une partie de la spécification du système : la sortie ne peut être à '0' que si le nombre sur 4 bits reçu en entrée est fini d'être analysé (S3 et S4 sont à '1') et ça n'est pas un code BCD valide (bit de poids fort à '1' et l'un des bits numéro 2 ou 1 à '1' aussi).

Cet exemple simple va nous permettre d'introduire la problématique posée dans cette thèse. Cette assertion est un cas simple d'implication dans une spécification. Dans le cas où la partie gauche (prémisse) de l'implication ne se produit jamais, la spécification est considérée comme vérifiée mais n'est en fait jamais activée. Ce phénomène est appelé la vacuité dans la vérification.

```
library ieee;
use ieee.STD_LOGIC_1164.all;

entity BCD_checker is
port( I  : in  STD_LOGIC ; clk : in  STD_LOGIC ;
      O  : out STD_LOGIC);
end BCD_checker;

architecture STRUCT of BCD_checker is
signal S1, S2, S3, S4 STD_LOGIC ;

-- psl default clock is (clk' event and clk='1');
-- psl property output_O is
-- always (O = '0' → (I = '1' and S3 = '1' and S4 = '1' and (S1 = '1' or S2 = '1'));
-- psl assert output_O;
begin

(...VHDL...)

end STRUCT;
```

Figure I.2 : Exemple d'intégration des assertions PSL dans une description VHDL

Dans le contexte des travaux de l'équipe, des moniteurs de surveillance synthétisables sont construits à partir de ces assertions, et peuvent être connectés au circuit à vérifier, ce qui permet la vérification en simulation, émulation sur FPGA, ou même d'embarquer certains de ces moniteurs pour la surveillance en ligne. Si l'assertion n'est pas activée, cela se traduit par la non activation des moniteurs.

Dans cette thèse, on va se concentrer sur les assertions ainsi vérifiées en cours de simulation. Un problème majeur de l'introduction des assertions est celui de la génération des séquences de test à utiliser pour les simulations. Elles doivent garantir une bonne couverture des conditions d'activation des moniteurs de surveillance. On souhaite vérifier que le système satisfait un certain nombre de propriétés. L'objectif est de développer des solutions qui permettent de s'assurer que ces propriétés sont réellement vérifiées et ne passent pas d'une façon *vide* (moniteur jamais activé et répondant *vrai* par défaut).

I.3. Plan de la thèse

Hormis cette introduction, ce manuscrit est divisé en trois chapitres. Il est organisé de la manière suivante :

- Le chapitre 2 présente une introduction générale au sujet de cette thèse tout en discutant le contexte, les motivations et la problématique. Il rapporte aussi un état de l'art sur des travaux connexes. Ce chapitre permet au lecteur de situer le thème de la vérification à base d'assertions et de comprendre les motivations des réalisations de cette thèse qui sont expliquées dans le chapitre suivant.
- Le chapitre 3 décrit l'approche complète proposée dans le cadre de cette thèse pour la génération des vecteurs de test capables d'activer suffisamment souvent les assertions destinées à observer les comportements d'un circuit.
- Le chapitre 4 expose des cas d'études qui montrent l'amélioration du taux de couverture d'assertions en appliquant l'algorithme proposé dans cette thèse.
- Enfin une conclusion résume le travail réalisé et suggère des axes de recherche futurs.

Chapitre 2.

Contexte et état de l'art

II. Contexte et état de l'art

II.1. Technologie de vérification "HORUS"

II.1.1. Langage PSL

PSL (Property Specification Language) [7] est un langage formel qui permet d'écrire une spécification à l'aide de propriétés logico-temporelles. C'est un langage rapide à assimiler, basé sur une syntaxe relativement simple, il peut être associé à divers langages de description, comme VHDL [4] et Verilog [5]. Il est destiné à être utilisé pour la spécification fonctionnelle comme entrée pour les outils de vérification et de simulation, et en vérification formelle. PSL peut être utilisé pour surveiller le comportement global d'un système, ainsi que les « hypothèses » (*assumptions* en anglais) sur l'environnement dans lequel le système est censé fonctionner.

Une spécification PSL se compose des assertions qui modélisent des propriétés portées sur le système à vérifier. Une propriété est construite à l'aide des expressions booléennes, qui décrivent le comportement au cours d'un cycle, des expressions séquentielles, qui décrivent le comportement multi-cycles et des opérateurs temporels, qui décrivent des relations sur plusieurs cycles.

La vérification en simulation s'appuie sur un sous-ensemble nommé *sous-ensemble simple* de PSL (voir plus loin), voici un exemple de propriété dans ce sous-ensemble :

always *(req* -> **next !***req)*

Chaque fois que *req* est à '1', au cycle d'horloge suivant il doit être à '0' (autrement dit, le signal *req* n'est jamais maintenu sur plus d'un cycle).

Le langage PSL est structuré en quatre couches (voir Figure II.1) :

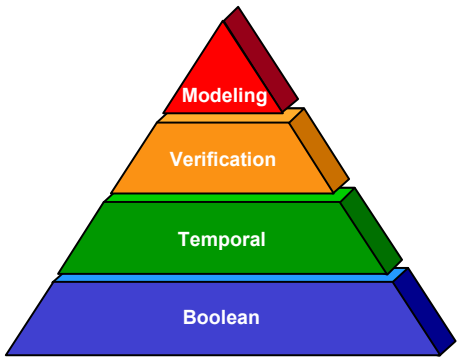


Figure II.1 : structure générale du langage PSL

- ***Couche booléenne :*** constituée des expressions booléennes. Cette couche est construite sur les operateurs booléens classiques and, or...
- ***Couche temporelle :*** c'est la couche la plus importante dans PSL, elle permet de spécifier les relations entre les expressions booléennes dans le temps.

Par exemple : always (*req* → next *ack*) est une propriété temporelle qui exprime que l'activation de *req* est toujours (exprimé par always) suivie un cycle après (exprimé par → next) par l'activation de *ack*.

Cette couche est formée de trois sous-ensembles : **FL** (Foundation Language), les **SEREs** (Sequential Extended Regular Expressions), et **OBE** (Optional Branching Extension). La sémantique des propriétés **FL** et des **SEREs** est fondée sur la logique LTL (Linear Temporal Logic), contrairement à celle de OBE qui est basée sur la logique CTL (Computation Tree Logic).

- ***Couche vérification :*** cette couche fournit des directives qui précisent comment utiliser les propriétés à l'aide de mots clés.

  Par exemple : **assert always** (*req* -> **next** *ack*) ;

  On trouve ici la directive assert qui indique que la propriété doit être vérifiée. Il existe aussi la directive assume qui définit le comportement des entrées.
- ***Couche modélisation :*** cette couche fournit le moyen de définir le modèle d'environnement dans lequel est effectuée la vérification, en modélisant le comportement des entrées, et en déclarant et donnant un comportement aux signaux et variables auxiliaires. L'environnement, ainsi que les propriétés sont regroupés dans une structure appelée vunit (Figure II.2).

```
Vunit (Arbiter){
default clock is rising_edge(clk);
assume always (req1 → next(ack1 before req1));
Assume always (req2 → next(ack2 before req2));
assert   always (req1 → eventually! ack1 );
Assert   always (req2 → eventually! ack2 );
Assert   never(req1 and req2);
}
```

Figure II.2 : Exemple d'unité de vérification vunit

Comme dit précédemment, la couche temporelle présente le cœur du langage PSL. Cette couche est formée de trois sous-ensembles :

- ***Le sous-ensemble FL (Foundation Language) :*** regroupe les opérateurs logiques et les opérateurs temporels comme always, before, until, next, …

  Exemple de FL :

  property FL1 **is assert always** (*req* → **eventually !** *ack*) ;

  Cette propriété signifie que toute production de *req* doit inévitablement être suivie de *ack* (sans spécifier quand exactement).

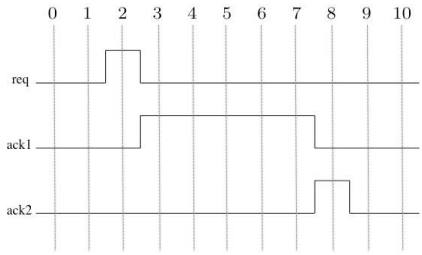
  Il existe deux types d'opérateurs temporels : *faibles* et *forts* (un opérateur temporel *fort* est reconnu avec un point d'exclamation à la fin). La différence entre l'opérateur faible et fort devient importante dans le cas d'une trace courte, quand on veut déterminer si oui ou

non, tout le comportement décrit par la propriété est satisfait sur la trace de simulation. Un opérateur faible est tolérant, dans le cas où une trace s'achève rapidement avant de vérifier la propriété jusqu'à la fin. Dans ce cas, et tant qu'il n'a pas d'erreur de fonctionnement, aucune violation n'est signalée même si la trace ne couvre pas la totalité de la fonctionnalité décrite par la propriété. Cependant, un opérateur fort est strict dans le cas des traces courtes. Cet opérateur exige qu'aucune erreur de fonctionnalité ne soit signalée et demande que la condition de fin d'une expression se produise avant la fin d'une trace.

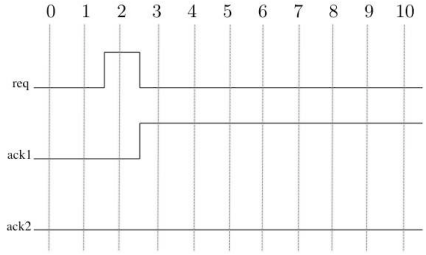
Exemple :

property FL2.1 **is assert always** (*req* → **next** (*ack1* **until** *ack2*)) ;
property FL2.2 **is assert always** (*req* → **next** (*ack1* **until !** *ack2*)) ;

La propriété FL2.1 signifie que la production de *req* doit être suivie par *ack1* au cycle prochain, *ack1* reste actif jusqu'à la production de *ack2*.



Trace 1 : FL2.1 et FL2.2vérifiées



Trace 2 : FL2.1 vérifiée et FL2.2 non vérifiée

Figure II.3 : Opérateur until fort vs opérateur until faible

La Figure II.3 montre des traces vérifiant la propriété FL2.1 (opérateur **until** faible), et la propriété FL2.2 (opérateur **until** fort). Dans la première trace, FL2.1 et FL2.2 sont vérifiées. Cependant, dans la trace 2, seule FL2.1 (opérateur until faible) est vérifiée, car la condition de fin *ack2* ne se produit pas dans la trace.

- ***LE sous-ensemble OBE (Optional Branching Extension) :*** Utilisé uniquement pour la vérification statique. Cet ensemble supporte les opérateurs A (tous les chemins d'exécution doivent vérifier la propriété) et E (il existe au moins un chemin d'exécution respectant la propriété) de la logique **CTL**.

Exemple de propriété OBE :

property OBE1 **is assert A** (**never** (*ack* _1 **and** *ack* _2)) ;

Cette propriété interdit la production de *ack_1* et *ack_2* simultanément sur tous les chemins d'exécution possibles.

- ***Le sous-ensemble des SEREs (Sequential Extended Regular Expressions) :*** Ce sous-ensemble est fortement influencé par la syntaxe des expressions régulières permettant d'exprimer des propriétés sur des séquences. Il contient par exemple les expressions qui expriment la séquentialité { ; , :}, la répétition consécutive (ex : f[*], [*i]g) et non consécutive (f[→i], [=i]g).

  Exemple de SERE :

  property SERE1 **is assert** {*ack_1* ; *ack_2*}[*]

  Cette propriété signifie que la trace doit contenir une succession de motifs {*ack_1*; *ack_2*}, comme montré sur la Figure II.4.

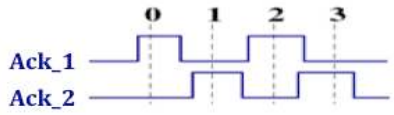


Figure II.4 : Trace qui satisfait la SERE1

Le *"sous-ensemble simple"* de PSL est destiné à la vérification dynamique. Le but de ce sous-ensemble est d'imposer des restrictions sur le langage pour permettre la vérification en cours de simulation, c'est à dire que la composition des propriétés temporelles est faite de sorte à assurer que l'évaluation de la propriété va de gauche à droite dans le diagramme de temps. Les expressions utilisant certains des opérateurs always, never, next, next_event, before, until, etc doivent respecter des restrictions pour appartenir à ce sous-ensemble. Comme par exemple, pour l'opérateur eventually! et never l'opérande doit être un booléen ou une séquence, les opérandes de l'opérateur before doivent être des booléens, etc.

II.1.2. Comparaison avec le langage SVA

Tout comme PSL, le langage SVA (SystemVerilog Assertions) offre les moyens de décrire des comportements complexes. C'est un langage de spécification de propriétés logico-temporelles standardisé en 2005 [9] au cœur de la norme SystemVerilog. Avec SVA unifié syntaxiquement avec le reste de *SystemVerilog*, le concepteur est en mesure d'intégrer les assertions directement en ligne avec le système et d'autres codes de vérification permettant aux outils de déduire une grande quantité d'informations.

La sémantique de SVA est définie de telle sorte que l'évaluation de ces assertions soit garantie équivalente entre la simulation (par évènement), et la vérification formelle (par cycle).

Le langage SVA contient deux types d'assertions :

- *Immédiates :* Ces assertions vérifient si une expression logique est vraie à tout instant donné. Elles peuvent être placées dans des blocs procéduraux (des blocs *initial* ou *always*) et seront exécutées comme des instructions classiques.

- *Concurrentes :* ces assertions offrent la possibilité de spécifier un comportement au fil du temps d'une manière concise, évalué sur des points discrets dans le temps (horloge). Une assertion concurrente peut se trouver dans des blocs procéduraux et aussi dans des modules.

  Exemple d'assertion concurrente :

  Assertion_concurrente:

  assert property @(**posedge** *clk*) (req **##**1 *grant* **##**10 *!req* **##**1 *!grant*);

  else

  $error("échec de vérification…");

  La séquence dans l'exemple ci-dessus signifie : *req* doit être vrai immédiatement, suivi par *grant* à vrai au cycle d'après. Puis, après 10 cycles d'horloge, *req* doit être à faux, suivi au cycle d'après par *grant* à faux.

Le langage SVA est structuré en quatre couches (Figure II.5) :

- ***Couche booléenne :*** comme celle de PSL, elle constituée des expressions booléennes classiques comme : {&&, ||, !}.
- ***Couche de séquence :*** constituée des expressions booléennes au fil du temps : séquence, répétition, etc. Équivalente au sous-ensemble SERE du PSL.
- ***Couche de propriété :*** cette couche offre la possibilité d'exprimer le comportement du système en se basant sur les expressions de la couche booléenne et de séquence.

  Une propriété a la fome suivante:

  property *nom_de_propriété* (*liste_des_arguments*);

  déclaration_variables_pour_assertion ;

  spécification_de_propriété;

  endproperty
- ***Couche assertion :*** contient les directives {assert, assume, cover} pour décrire comment une propriété est supposée être utilisée pendant la simulation.

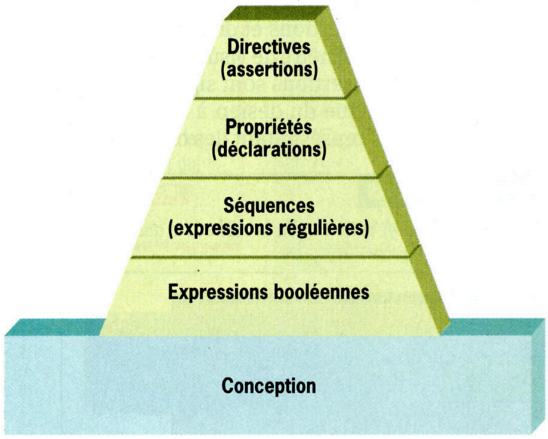


Figure II.5 : Structure générale du langage SVA

Bien que les langage PSL et SVA aient quelques similitudes, il existe des différences fondamentales entre les deux langages [10] :

- Contrairement à **PSL**, **SVA** ne donne pas la possibilité de différencier entre des formules fortes et formules faibles.
- PSL offre un support supplémentaire à la vérification formelle à l'aide de **OBE** (à base de **CTL**), ce qui n'est pas le cas pour **SVA**. Cela rend PSL efficace dans la vérification statique et dans la vérification dynamique, contrairement à **SVA** qui s'oriente plus vers la vérification dynamique.
- PSL est riche en opérateurs, il est basé sur les expressions régulières (**SEREs**) et les opérateurs **FL** alors que **SVA** n'est basé que sur les expressions régulières (**Sequences**)

II.1.3. Assertion-Based Verification (ABV) - Technologie HORUS

Ces dernières années, une méthode de vérification à la popularité grandissante a fait son apparition dans les domaines académiques et industriels : la *vérification à base d'assertions* (ABV) [6]. Cette approche utilise des assertions qui représentent les comportements que le circuit doit satisfaire. Le nombre des assertions nécessaires pour vérifier un circuit augmente avec la complexité de sa description RTL (Register Transfer Level). Cette approche a largement prouvé son efficacité, et son utilisation peut augmenter l'efficacité de la phase de test jusqu'à 50%. L'utilisation des assertions tout au long du flot de conception permet de détecter des erreurs au plus près de leur source, ce qui offre un gain de temps considérable au niveau du débogage.

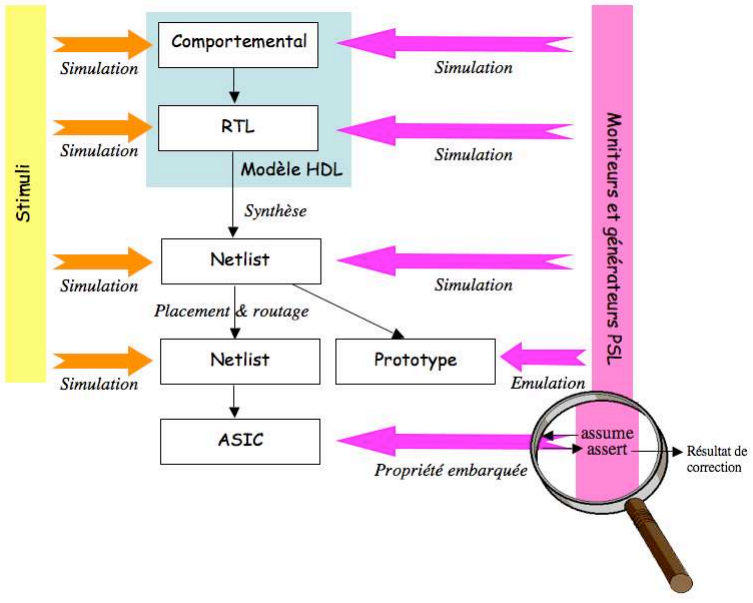


Figure II.6 : Flot de conception usuel et flot de vérification par assertion [11]

La Figure II.6 compare le flot de conception usuel avec le flot de vérification par assertions proposé par l'équipe. La vérification à base d'assertion utilise deux types de propriétés :

- Hypothèse (assumption) : décrit l'environnement du circuit à tester, incluant les contraintes sur les entrées.
- Assertion : décrit les comportements que le circuit doit assurer. Ces propriétés peuvent porter sur des entrées, des sorties et même des signaux internes du circuit à vérifier.

Tout cela permet de construire un environnement complet de vérification, où les comportements du circuit sont analysés tout en respectant des protocoles de communication avec l'environnement.

Les propriétés peuvent être synthétisées en composants matériels utilisés par les outils de vérification :

- Les hypothèses sont synthétisées en générateurs. Ces composants produisent des séquences de signaux correspondant aux hypothèses temporelles spécifiées.
- Les assertions sont synthétisées en moniteurs. Ces composants analysent le comportement du circuit et reportent une erreur si celui-ci ne vérifie pas les propriétés.

L'outil HORUS [12] est une plateforme de vérification développée au sein de l'équipe VDS. Il produit des moniteurs de surveillance et des générateurs [13]. Dans l'ensemble, cet outil aide l'utilisateur à instrumenter un circuit dans le but de faciliter d'éventuelles étapes de debug. Pour cela, l'instrumentation d'un certain circuit se fait en quatre étapes :

1. Sélectionner le circuit avec toute sa hiérarchie.
2. Définir les propriétés pour la vérification du circuit, sélectionner le langage HDL cible et faire la construction des moniteurs et générateurs. La Figure II.7 montre l'interface graphique d'Horus pour cette étape.

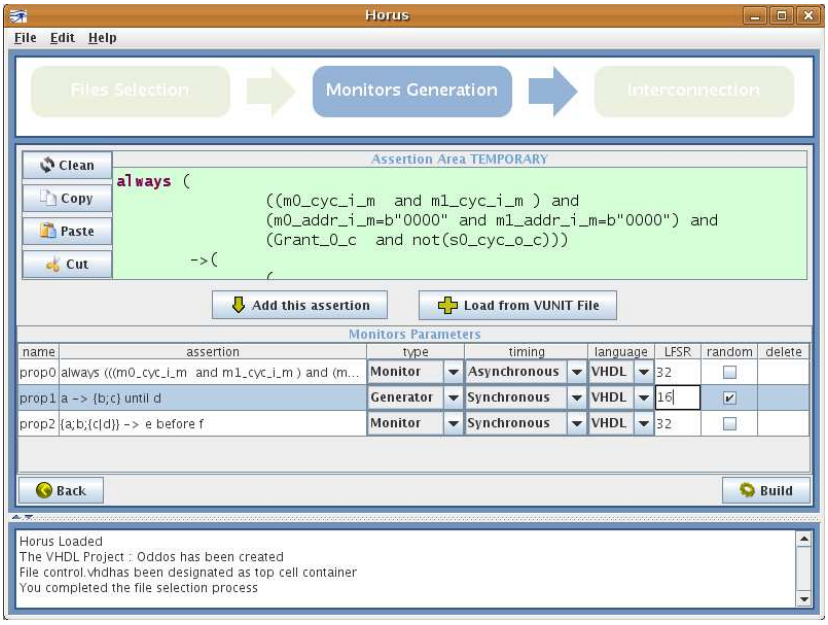


Figure II.7 : Interface graphique d'Horus [13]

3. Connecter les moniteurs et les générateurs au DUV (Design Under Verification). L'utilisateur peut sélectionner les signaux à connecter à chaque composant de vérification (donne la possibilité de connecter des signaux internes).
4. Générer le circuit instrumenté en prenant en compte, si des signaux internes sont accédés, une légère modification qui est faite pour remonter ces signaux sur les sorties primaires du DUV.

Pour une propriété donnée, un moniteur est construit sous forme d'un module (Figure II.8) qui prend en entrée :

- L'entrée *Clk :* pour synchroniser le module avec le circuit vérifié.
- L'entrée *Reset :* pour la réinitialisation.
- L'entrée *Start :* pour déclencher l'évaluation.
- Les entrées *Opérandes* : correspondent aux signaux du DUV qui sont des opérandes dans la propriété.

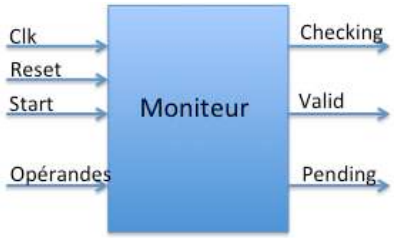


Figure II.8 : Interface de moniteur d'Horus

On trouve les trois sorties suivantes :

- La sortie *Checking* : indique que la vérification de la propriété a démarré (sortie passe à 1), et que la sortie *Valid* sera valide dans le prochain cycle.
- La sortie *Valid* : donne une information sur le résultat de l'évaluation (sortie à 1 en cas de vérification réussite, et passage à 0 quand il y a violation de propriété).
- La sortie *Pending* : cette sortie quand elle est à 1, indique que le moniteur a été déclenché et que le résultat de la satisfaction est encore en attente. Cette sortie est significative en cas des opérateurs forts.

La construction du moniteur à partir d'une propriété se fait d'une façon modulaire. Des moniteurs primitifs, définis pour chaque opérateur élémentaire de PSL sont interconnectés entre eux suivant l'arbre syntaxique de la propriété.

Exemple illustratif : Prenons l'exemple simple d'une partie de contrôleur de barrière de parking (ce système sera repris plus loin). On considère la propriété suivante :

property P **is Always** (*ticket_insere* → **next!** (**not** *ticket_ok* → **not** *Barriere_out_ouverte*))

Cette propriété signifie qu'une fois que le ticket est inséré, puis que le mécanisme de vérification de sa validité informe qu'il n'est pas valide, alors la barrière de sortie doit rester fermée. L'architecture du moniteur de la propriété P est représentée par la Figure II.9.

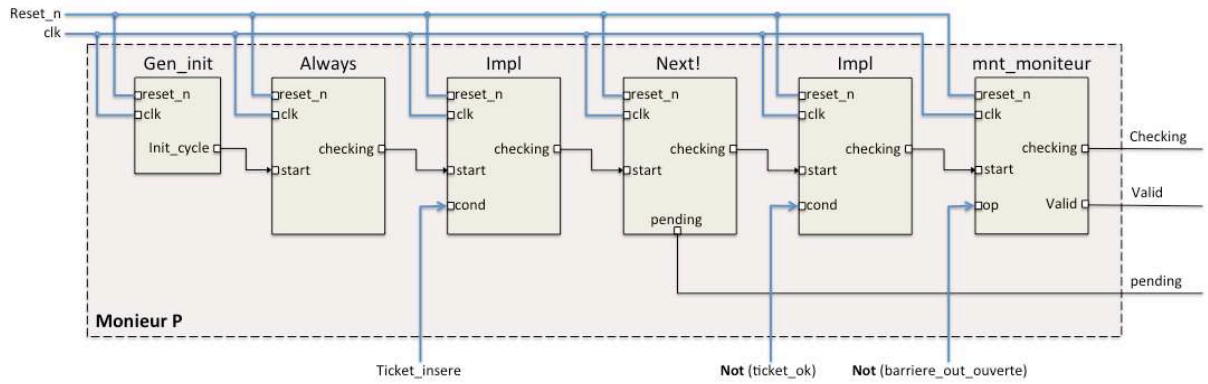


Figure II.9 : Architecture du moniteur de la propriété P

Après construction du moniteur P, il est connecté au DUV pour surveiller les signaux inclus dans la propriété durant la simulation (Figure II.10).

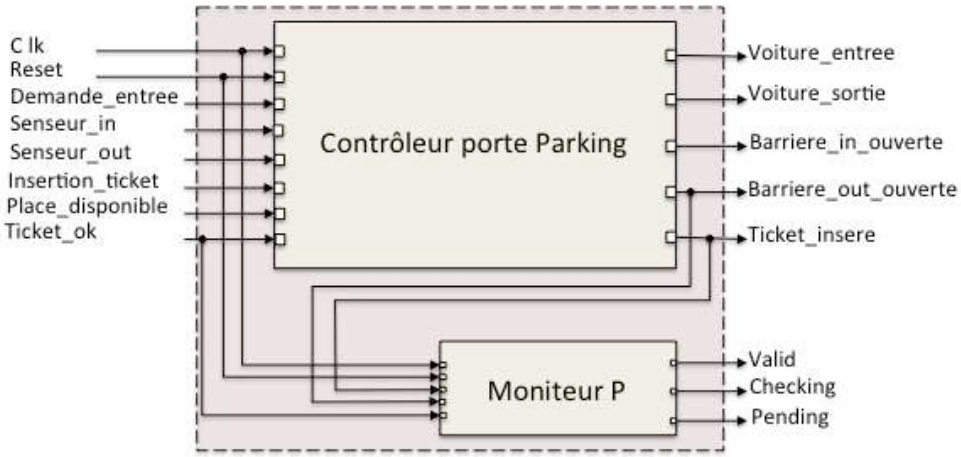


Figure II.10 : Interconnexion entre DUV et le moniteur P

La simulation classique montre l'évolution des signaux dans le temps (voir Figure II.11), un comportement défectueux est peu évident à détecter pendant la simulation; c'est là où intervient le moniteur connecté au circuit qui, à l'aide de sa sortie Valid, nous permet de vérifier aisément le comportement du circuit durant la simulation. La sortie Valid est maintenue à 1 quand la propriété correspondante est vérifiée; un passage à 0 indique que la propriété P a été violée.

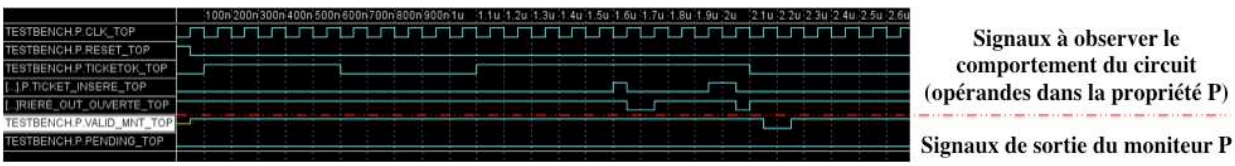


Figure II.11 : Simulation des signaux du circuit et des sorties du moniteur P

L'idée de créer des moniteurs à partir des assertions a été introduite et commercialisée grâce à l'outil FoCs (FOrmal CheckerS) d'IBM [14], [15]. Contrairement à Horus, cette approche

de synthèse est basée sur les automates, elle transforme des propriétés écrites en PSL en des moniteurs VHDL, Verilog ou SystemC. Les circuits générés sont un peu trop volumineux pour être utilisés sur FPGA. De nombreuses autres approches ont vu le jour, en tentant d'obtenir des composants le plus simples possible, pour minimiser l'impact des moniteurs à la fois sur la taille du circuit à vérifier en cas d'implémentation de ce dernier, et sur le temps de simulation. Les auteurs de [16] proposent une approche pour la synthèse des assertions logico-temporelles écrites en langage **SVA** en modules **BSV** (BlueSpec System Verilog). L'idée principale de cette approche et de transformer les assertions synchrones en des actions atomiques, ce qui nécessite la multiplication des FSMs autant de fois qu'il y a réentrance, ce qui peut causer l'explosion de la taille des moniteurs. Le traitement des opérateurs temporels non bornés est interdit, ce qui limite l'utilisation de cette méthode sur un sous-ensemble de System Verilog. Une méthode à base de résidus a été proposée dans [17], elle s'est intéressée à la génération des moniteurs à partir des automates d'états finis pour les SEREs PSL. En pratique, les caractéristiques en synthèse et le temps de construction des moniteurs n'ont pas été étudiés. La méthode proposée dans [18] est basée sur le HLDD (High Level Decision Diagram). Des moniteurs logiciels sont générés à l'aide de l'outil FoCs et transformé en HLDD. Bien que l'introduction du HLDD ait amélioré le temps de simulation et réduit son impact d'un facteur de 10, les inconvénients des moniteurs générés par FoCs restent présents.

L'approche [19] traite les assertions écrites en PSL ou SVA, et les transforment en des modules hardware aptes à être connectés avec le DUV. Le principe de cette approche est de traiter suivant des modes prédéfinis les parties gauche et droite d'une implication. Ces modes prédéfinis (must-mode, if-mode, et mode scope) correspondent à la manière symbolique d'évaluation de la trajectoire dans une spécification. Par exemple on prend le cas simple: (a $\rightarrow$ b), le traitement de cette implication sera comme suit : la partie gauche de l'implication est transformée en if-mode. Le résultat de cette partie gauche seras pris en compte et considéré comme pré-condition pour traiter la partie droite de l'implication, qui sera transformée à son tour en must-mode. Le résultat retourné par la partie droite est bien le résultat de l'implication. Cette approche prédéfinit des automates pour un sous-ensemble d'opérateurs primitifs, et traite les autres opérateurs en utilisant des règles de réécriture. Pour construire des moniteurs pour des propriétés plus complexes il suffit d'interconnecter les moniteurs primitifs qui correspondent aux opérateurs utilisés dans cette propriété. [20] est une continuation de l'approche décrite dans [19] et présente la nouvelle version de MBAC, destiné à la génération des moniteurs en transformant les assertions correspondantes en une représentation sous forme d'automate, avant de les convertir en description RTL.

II.2. Qualité de la vérification - Activation des assertions

La technologie Horus a été intégrée à certains des outils d'aide à la conception de la société *Dolphin Integration,* dans l'environnement SLASH (rassemble les outils SLED et SMASH), [21]. Dans le cadre de cette coopération, les travaux de cette thèse se baseront sur l'utilisation de ces outils :

- SLED : outil de saisie de schémas. Il permet notamment de construire et rassembler graphiquement des unités de vérification (moniteurs et générateurs) avec le circuit à vérifier.

Cet outil permet, après construction du prototype du circuit instrumenté avec des unités de vérification, de créer les fichiers qui permettent de simuler ce circuit. Ces fichiers sont utilisables dans SMASH.
- SMASH : outil de simulation mixte analogique/numérique. Il permet notamment la simulation des assertions écrites en PSL en parallèle avec le circuit. À l'aide de cet outil, nous pouvons savoir le nombre de fois qu'une assertion est vérifiée et/ou violée. Cela nous permet de pouvoir calculer le taux de couverture d'activation d'assertions et faire des comparaisons par la suite.

Un problème majeur se pose quand on veut faire de l'ABV par simulation ou émulation. Une assertion surveille en partie la fonctionnalité du DUV, cependant il nous faut savoir si le comportement visé par cette propriété a été réellement vérifié ou non, autrement dit si cette assertion a bien été activée (problème de vacuité). La génération des séquences de test qui permettent de couvrir les parties de fonctionnalité décrites par les assertions, et ainsi les activer durant la simulation est primordiale. Des séquences de test générées d'une façon aléatoire risquent de conduire à un faible taux de couverture d'activation de ces assertions, voire à ne jamais les déclencher.

Prenons comme exemple le cas simple d'un circuit qui contrôle l'entrée et la sortie des voitures d'un parking. La Figure II.12 présente l'interface de ce circuit.

Figure II.12 : Interface du contrôleur de barrière de stationnement

Ce circuit prend comme entrées essentielles:

- Demande_entree : demande pour entrée.
- Place_disponible : pour indiquer s'il y a des places disponibles.
- Senseur_in : signal provenant du capteur d'entrée pour signaler qu'une voiture est entrée.
- Insertion_ticket : insertion d'un ticket de sortie.
- Ticket_ok : pour indiquer que le ticket de sortie est valide, donc la voiture peut sortir.
- Senseur_out : signal provenant du capteur de sortie pour signaler qu'une voiture est sortie.

Il a les sorties suivantes:

- Barriere_in_ouverte : pour commander l'ouverture de la barrière d'entrée.
- Barriere_out_ouverte : pour commander l'ouverture de la barrière de sortie.

- Voiture_entree : commande envoyée au mécanisme pour incrémenter le nombre de voitures.
- Ticket_insere : commande pour signaler au mécanisme vérificateur qu'un ticket a été inséré.
- Voiture_sortie : commande envoyée au mécanisme pour décrémenter le nombre de voitures.

Quand une voiture est présente devant la barrière d'entrée, le conducteur appuie sur le bouton d'entrée, s'il y a des places disponibles dans le parking, la barrière d'entrée doit s'ouvrir et rester ouverte jusqu'à ce que le capteur détecte que la voiture est bien entrée. À ce moment, un signal est envoyé au mécanisme de comptage pour incrémenter le nombre des voitures dans le parking. Dans le cas d'une voiture sortante, le conducteur doit passer payer et recevoir un ticket de sortie valide d'une machine. Quand il se présente devant la barrière de sortie, il insère son ticket de sortie. La barrière s'ouvre si le ticket est valide. À ce moment, un signal est envoyé au mécanisme de comptage pour décrémenter le nombre de voitures dans le parking. Considérons alors les propriétés suivantes :

(P_1) **not** *barriere_out_ouverte* **until** *ticket_insere*
(P_2) **always** *(ticket_insere →* **next !** *(***not** *ticketOK →* **not** *barriere_out_ouverte))*
(P_3) **always** *({barriere_out_ouverte ;* **not** *barriere_out_ouverte} |→ (***not** *barriere_out_ouverte)* **until** *ticketOK)*

La propriété P_1 signifie que la barrière de sortie ne pourra s'ouvrir qu'après la première insertion de ticket de sortie.
La propriété P_2 vérifie que si le ticket inséré n'est pas valide, la barrière de sortie ne doit pas se lever.
La propriété P_3, la barrière de sortie reste fermée jusqu'à la prochaine insertion d'un ticket de sortie.

Mesurons le nombre de vérifications effectives des propriétés P_2 et P_3, dans le cas où les vecteurs de test sur les entrées du circuit sont générés de façon totalement pseudo-aléatoire (la propriété P_1 n'est par définition activée qu'une seule fois, puisqu'elle n'a pas d'opérateur *always*).

	Rapport de propriétés PSL		
propriétés	Nombre de vérification	Nombre de violations	Temps de simulation
P2	2622	-	20000 cycles d'horloge
P3	1069	-	

Nous reprendrons ces mêmes exemples par la suite, pour comparer avec les résultats obtenus par application de nos algorithmes.

II.3. État de l'art

Divers types de travaux ont visé la générations des séquences de test efficaces pour une bonne vérification :

- Certains se sont focalisés sur le problème de la vacuité dans les assertions.
- D'autres approches de type ATPG (Automatic Test Pattern Generation) sont plus orientées vers l'étude au niveau structurel. En considérant le circuit, ces approches génèrent des vecteurs de test suivant des critères précis (endroit et nature de faute localisée) pour couvrir des fautes.
- Enfin, des approches ont visé la garantie d'une meilleure couverture au niveau RTL, dont les modèles sont généralement des automates.

II.3.1. Étude de la vacuité

C'est la satisfaction d'une propriété d'une façon vide, par exemple dans le cas d'une implication dont l'antécédent (partie gauche de l'implication) est toujours faux. Le problème de la vacuité a été adressé pour la première fois dans les années 1990. Les chercheurs ont remarqué ce problème et ont montré qu'il est possible de l'éviter en vérifiant le système à l'aide des formules écrites manuellement qui assurent la satisfaction des pré-conditions dans une spécification. Cependant, cette méthode peut être sujette à erreurs. C'est pour cela que l'introduction des techniques automatiques pour détecter la vérification "vide" est devenue indispensable.

Dans [22], les auteurs considèrent les formules de w-ACTL (ACTL correspond à CTL sans les opérateurs ‘‘E’’, et les formules de w-ACTL sont les formules de ACTL pour lesquelles tous les opérateurs binaires ont au moins un opérande qui est une formule propositionnelle). Ils proposent un algorithme qui construit une ‘‘formule témoin’’ w(φ) pour toute formule φ de w-ACTL : cette formule détecte la vacuité et, s'il n'y a pas vacuité fournit un exemple positif de φ. Cette solution, comme les suivantes, permet de savoir si une propriété présente un défaut de vacuité, mais ne vise pas à produire des séquences de test pour éviter ce phénomène. Dans [23], le même problème est résolu, mais sans limitation à w-ACTL. Une méthode plus générale est développée pour la génération de ‘‘formule témoin’’ (chemins de la spécification considérée qui satisfont φ de façon non vide) pour des spécifications en CTL*.

Le travail présenté dans [24] cherche également à identifier la vacuité des propriétés, plus précisément d'assertions exprimées dans le sous-ensemble simple de PSL. La vacuité est prise au sens de [25], qui traite le problème par model-checking : une formule φ est satisfaite de manière vide dans un modèle M si elle est satisfaite dans M et il existe une sous-formule ψ de φ qui peut être changée arbitrairement sans affecter le résultat du model-checking. Les principes utilisés dans [24] sont identiques à ceux utilisés dans [25], l'idée étant essentiellement d'analyser l'effet de la substitution d'une sous-formule par faux. Alors que [25] met en œuvre une solution à base de méthodes formelles (model-checking), [24] décline

la solution dans un contexte dynamique en réalisant des mutations de checkers pour propriétés PSL (produits par l'outil FoCs d'IBM [15]).

Bien que ces méthodes de détection de vacuité puissent s'avérer efficaces pour améliorer la qualité de la vérification (soit dynamique ou statique), elles se limitent sur le traitement du problème d'une façon très logique en se basant sur les formules sans prendre en compte le circuit concerné par les assertions. Elles permettent de savoir si une propriété présente un défaut de vacuité, mais ne visent pas à produire des séquences de test pour éviter ce problème. Ce qui n'est pas le cas que nous visons, nous cherchons à activer une propriété de façon non vide, en tenant compte du circuit étudié.

II.3.2. Techniques d'ATPG (Automatic Test Pattern Generation)

L'ATPG est un processus qui permet de générer des séquences de test d'une façon automatique (Figure II.13) pour simuler et analyser les réponses d'un circuit afin de détecter des fautes. C'est une technologie qui est relativement mature et qui a montré un grand succès d'analyse dans le domaine de test, surtout pour les structures complexes des circuits. Elle se base sur les informations du niveau structurel pour guider les processus de recherche des vecteurs de test.

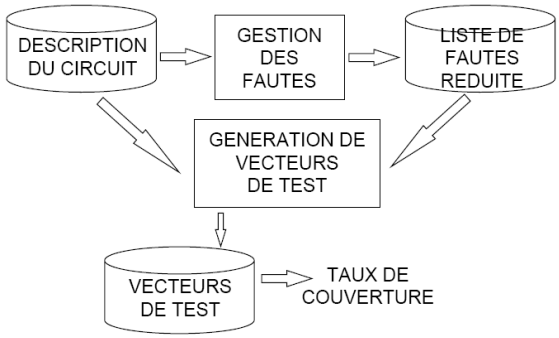


Figure II.13 : Architecture d'un système d'ATPG

Il y a deux phases principales. La première (création de test), est la génération des vecteurs de test pour les fautes ciblées. Cette phase implique en général une boucle qui évalue et affine la séquence de test progressivement jusqu'à ce que la couverture de fautes soit satisfaite. La deuxième phase (application de test), consiste à injecter la faute dans un circuit, ensuite, essayer d'activer la faute de façon à ce que la cause de son effet se propage et se manifeste à la sortie du circuit. Le résultat trouvé sur la sortie est différent de la valeur attendue sans fautes, par conséquent la faute est détectée.

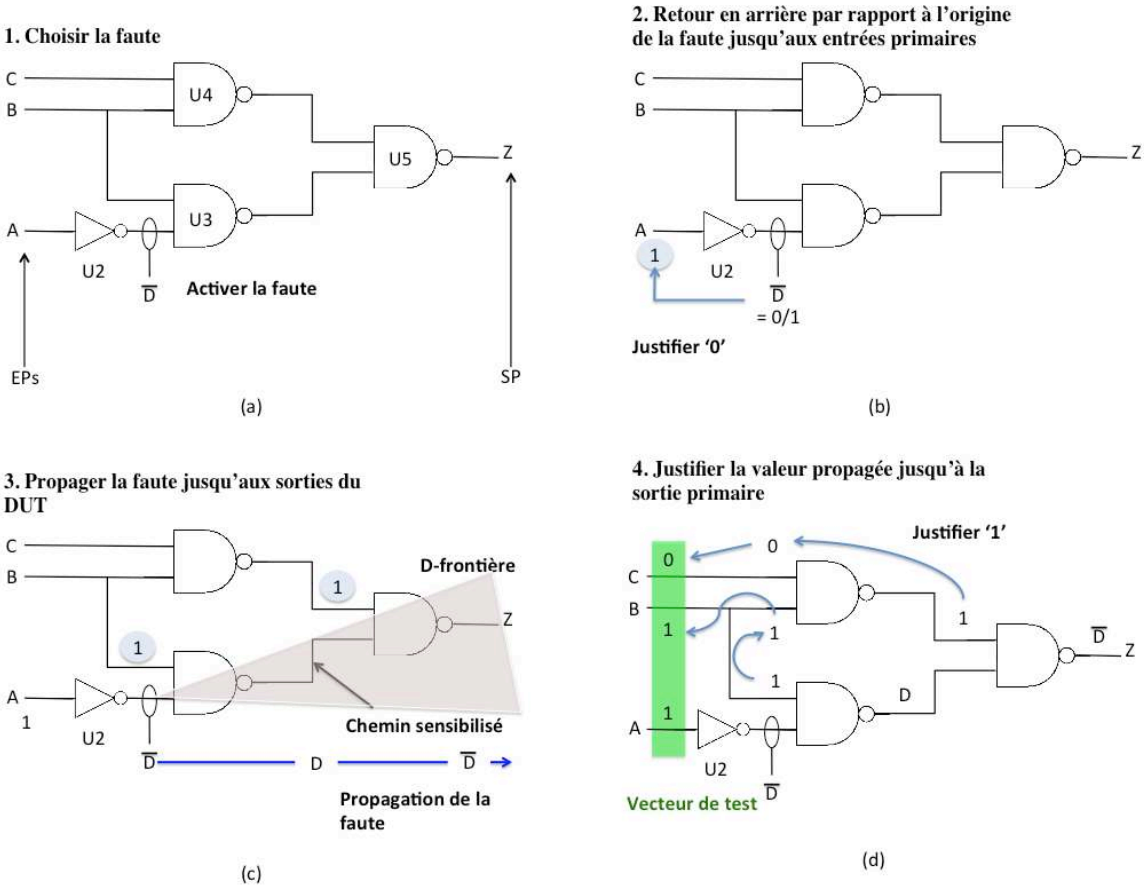


Figure II.14 : Exemple pour l'algorithme de base d'ATPG

La Figure II.14 présente un algorithme basique d'ATPG pour l'exemple $\overline{A}B + BC$ et détaille les actions nécessaires pour détecter une faute et générer les séquences de test qui mettent cette faute en évidence [26]. La faute est détectée d'abord par l'activation (ou l'excitation) de la faute. Pour ce faire, on conduit le nœud en faute à la valeur inverse de la faute. L'exemple de la figure montre une faute de collage à '1' sur la sortie d'***U2***, tout en détaillant les différentes étapes suivies pour trouver la séquence de test qui mette en évidence la faute :

(a) sensibiliser la faute qu'on veut mettre en évidence,
(b) retourner en arrière depuis la faute jusqu'au entrées primaires pour justifier la valeur qui la sensibilise.
(c) propager la faute en sensibilisant le chemin qui va être traversé pour atteindre la SP (Sortie Primaire).
(d) retourner en arrière et justifier toutes les valeurs trouvées pour faire remonter la faute jusqu'aux entrées primaires, pour générer la séquence de test correspondante.

Les systèmes d'ATPG proposent des algorithmes bien connus pour la génération des vecteurs de test comme :

- D-algorithme : Le principe de base du D-algorithme [27] est le suivant. Premièrement, on définit un test pour une faute f donnée en terme d'entrée et de sortie de la porte fautive.

Ensuite, on détermine tous les chemins sensibilisables du site de la panne à toutes les sorties du circuit. Finalement, on essaye de construire un vecteur de test sur les entrées primaires qui réalisent toutes les assignations effectuées dans la phase précédente.

- PODEM : c'est un algorithme d'énumération [27], [28] dans lequel toutes les combinaisons d'entrée sont implicitement (mais exhaustivement) examinées comme vecteurs de test pour une faute donnée. La structure du circuit est utilisée de manière analogue à celle du D-algorithme afin de guider les essais successifs des combinaisons d'entrée.
- FAN : L'algorithme FAN [27], [29] est directement issu des travaux de PODEM. Il propose de meilleures heuristiques permettant en particulier de réduire le nombre de tentatives en particulier dans le cas de divergences (''FAN OUT'') d'où son appellation.

Plusieurs recherches ont ciblé l'amélioration la qualité des vecteurs de test générés pour détecter les fautes. Cela a entrainé l'utilisation de la méthode de SAT en se basant sur des informations du niveau structurel pour générer des vecteurs de test satisfaisant une assertion donnée et les comparer avec ceux produits par l'ATPG [30]. Certains travaux ont étudié la possibilité de combiner les deux méthodes (ATPG et SAT), ciblant toujours l'amélioration du taux de détection des fautes dans un circuit donné.

SAT est le problème de déterminer si les variables dans une formule booléenne donnée peuvent être affectées de manière à avoir la formule à VRAI (formule satisfiable), ou déterminer qu'il est impossible de satisfaire une telle condition (formule insatisfiable).
Prenons l'exemple suivant :

La formule X **AND** Y est satisfiable car $X = vrai$ et Y = vrai, rend $X\ AND\ Y = vrai$.

SAT est connu comme un problème NP-complet. C'est à dire, il n'existe pas d'algorithme connu pour résoudre efficacement toutes les instances de SAT. Cependant il existe une classe d'algorithmes nommée les SAT solvers qui peut résoudre efficacement un sous-ensemble assez large d'instances de SAT, ce qui peut être très utile dans plusieurs domaines pratiques tels que la conception des circuits, et les preuves de théorèmes automatiques. Les formulations basées sur la satisfiabilité booléenne (SAT) représentent des techniques efficaces pour résoudre les problèmes de l'ATPG. Dans ces méthodes, les conditions de testabilité sont transformées en des formules CNF (Conjunctive Normal Form, c'est à dire conjonctions de disjonctions) équivalentes. Si la formule est satisfiable, le SAT solver retourne une affectation satisfaisante à partir de laquelle l'extraction de vecteur de test est possible.

Dans ce contexte, il arrive d'utiliser un circuit dit "Miter" : il compare les sorties de deux circuits combinatoires N et N* à l'aide de la porte logique XOR (Figure II.15). La sortie est considéré comme satisfiable si seulement si les deux circuits à comparer ne sont pas fonctionnellement équivalents (si pour les mêmes valeurs d'entrées, les circuits N et N* produisent des valeurs de sorties différentes).

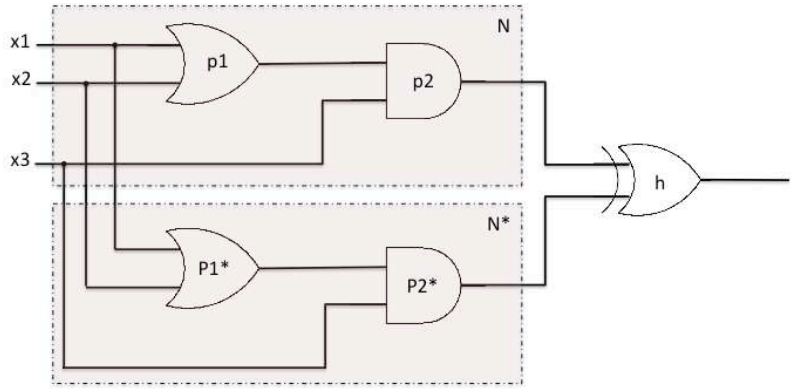
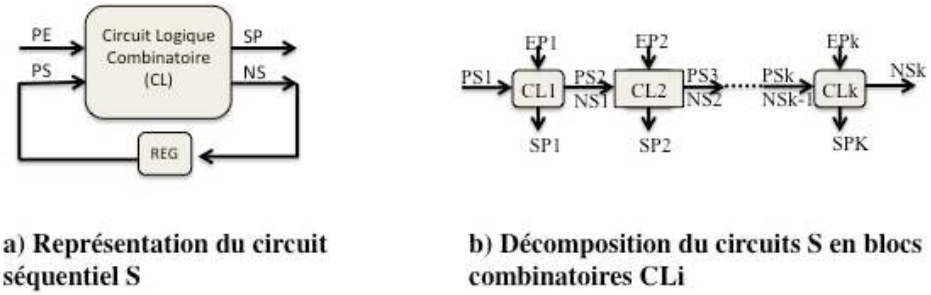


Figure II.15 : Exemple de circuit Miter

La Figure II.15 montre un circuit Miter pour deux copies N et N* du même circuit (N représente le circuit non défectueux et N* représente le circuit défectueux). N et N* ont le même ensemble de variables d'entrée, mais différentes variables intermédiaires.

Dans le cas d'un circuit séquentiel S (Figure II.16 (a)), il sera déroulé en plusieurs blocs combinatoires liés par des registres qui introduit le concept de séquentialité à l'aide de l'entrée état précèdent (PS) et de la sortie état suivant (NS) du bloc combinatoire *CLi* (Figure II.16 (b)).



a) Représentation du circuit séquentiel S

b) Décomposition du circuits S en blocs combinatoires CLi

Figure II.16 : Décomposition du circuit séquentiel en blocs combinatoires

Pour réaliser un circuit Miter pour deux copies C_k et C_k^f d'un circuit séquentiel (Figure II.17), on compare chaque sortie PO_i du bloc combinatoire de la copie C_k avec la sortie PO_i^f correspondante du bloc combinatoire de la copie C_k^f à l'aide des portes XOR. Les résultats des comparaisons sont assemblés à l'aide de la porte OR, qui indique si au moins une sortie du XOR est passée à 1 (ce qui veut dire qu'il y a une non équivalence fonctionnelle dans l'un des blocs combinatoires).

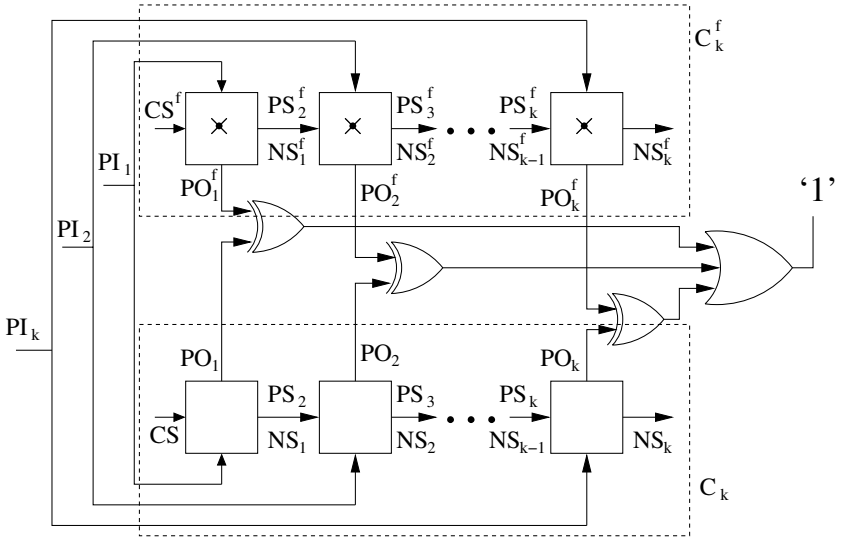


Figure II.17 : Circuit « miter » pour comparer des circuits séquentiels

Des fonctions booléennes sont associées aux circuits C_k et C_k^f du miter. La différence des valeurs de sorties peut être présentée sous la forme d'un problème de satisfiabilité, et l'équation résultante peut être codée directement en CNF et résolue à l'aide d'un SAT Solver.

Dans [31], les auteurs s'intéressent à la génération de tests pour circuits séquentiels, pour assurer la propagation de fautes de collage. La méthode est basée sur la satisfaction booléenne ; le but est de produire un nombre de vecteurs de test « presque minimal ». Elle reprend une solution pour circuits combinatoires et étend aux circuits séquentiels de façon classique en déroulant sur un nombre donné de cycles la partie combinatoire, comme expliqué ci-dessus. En addition à l'ensemble de formules qui décrivent le problème de propagation, d'autres clauses sont passées au solver de satisfiabilité, notamment des « clauses actives » qui permettent de spécifier que les valeurs des "lignes actives" (lignes qui propagent une divergence) dans le circuit fauté doivent être différentes des valeurs dans le circuit non fauté.

Le travail de [32] propose deux algorithmes d'ATPG pour circuits combinatoires et s'intéresse aussi à la propagation de fautes de collage. Ces algorithmes sont aussi basés sur SAT et améliorent la solution de [31]. Dans le premier cas, le temps d'exécution de SAT est amélioré en ajoutant à l'ensemble de clauses passées au solver de satisfiabilité des clauses qui sont associées aux CODCs (Compatible Observability Don't Cares), sous-ensemble des ODCs (Observability Don't Cares) qui permet d'éviter de recalculer les ODCs pendant l'optimisation des formules. Dans le deuxième cas, la justification des fautes est accélérée en ajoutant des « j-active clauses » pour certains nœuds dans le TFI (Transitive Fan-In) de l'emplacement fauté.

Les auteurs dans [33] proposent une méthode qui combine l'outil séquentiel ATPG « STRATEGATE » [34] avec des techniques (déterministes) de SAT « Chaff » [35] (le

problème est converti en une satisfaction des sorties d'un circuit « miter » qui compare les sorties du circuit) pour améliorer la génération des vecteurs de test et avoir une meilleure couverture des fautes. La méthode proposée consiste à utiliser l'outil STRATEGATE pour générer d'une part les vecteurs de test pour certaines fautes et d'autre part une première liste de fautes non testées. Cette liste est passée dans le SAT Solver Chaff qui produit à son tour une deuxième liste non résolues par le SAT pour la faire passer finalement dans HITEC [36].

L'article [37] propose une solution pour améliorer les séquences de test produites par l'ATPG à base de SAT en utilisant les propriétés structurelles des circuits à vérifier et en appliquant les techniques 'don't care' local, pour augmenter le nombre des variables non significatives dans les vecteurs de test, et ainsi pouvoir produire des séquences de test plus petites.

Ces techniques ne sont pas adaptées à l'objectif recherché dans cette thèse. En effet, elles ne visent pas générer des vecteurs de test permettant d'activer une propriété logico-temporelle exprimée sur les états et les sorties du DUV.

II.3.3. L'analyse de couverture

Dans les domaines du matériel et logiciel, des méthodes liées à la production des suites de test ont été développées afin de pouvoir garantir une meilleure couverture. Cela a donné lieu à une vaste littérature soit dans le contexte du logiciel ou du matériel. Les techniques utilisées dans les deux cas ne sont pas très éloignées lorsqu'on se place au niveau RTL et que les modèles considérés sont des automates d'états finis. La qualité de la vérification peut être mesurée par l'analyse de la couverture, qui peut être un moyen pour guider la détection des erreurs.

Le choix des métriques de couverture dépend de la représentation spécifique du système ou du but de vérification spécifique. Toutes les métriques se réfèrent à un ensemble de séquences d'entrée. Les auteurs de [38] proposent une description des principales métriques de couverture. Nous allons les rappeler ici :

Couverture de code. Les métriques de couverture de code s'appliquent sur les descriptions HDL qui décrivent le circuit, ou sur des CFG (Control Flow Graphs). Cette méthode est assez coûteuse, mais cependant très répandue. Les métriques de couverture de code les plus utilisées sont les couvertures d'expressions, de lignes ou de branches.

Couverture du circuit. Les métriques de couverture qui se basent sur la structure du circuit identifient les parties physiques du circuit qui sont couvertes. Cependant, cette information sur la couverture n'est pas facile à utiliser (contrairement à la couverture de code) pour générer des nouvelles séquences de test qui stimulent directement les régions inexplorées. Les métriques de couverture de circuit les plus utilisées sont les couvertures de registres, chaque registre doit être initialisé, chargé et lu. On trouve également les couvertures de compteurs.

Couverture de FSM. Les métriques de couverture de FSM nécessitent généralement une abstraction de la FSM, en projetant sa description symbolique sur un sous-ensemble de variables d'état [38], ce qui rend l'interprétation de l'information de couverture difficile (difficultés de relier les parties non couvertes dans la FSM au parties non couvertes dans le programme HDL). Un état ou une transition de la FSM abstraite est dit couvert s'il est visité durant l'exécution d'une séquence d'entrée. Il existe plusieurs métriques de couverture de FSM, parmi les plus utilisées on trouve :

- La couverture d'états : exprime le nombre d'états visités dans l'automate (Figure II.18 (a)).
- La couverture de transitions : l'intention est de traverser toutes les transitions dans un automate (Figure II.18 (b)).

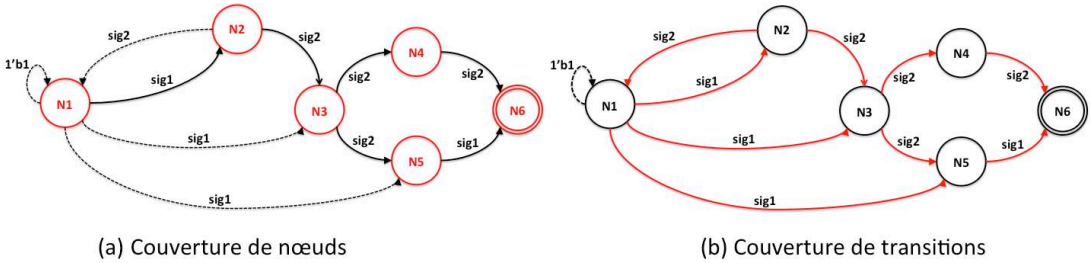


Figure II.18 : Métriques de couverture d'états (nœuds) et de transitions

- La couverture du chemin complet : cette métrique consiste à couvrir tous les chemins possibles. Un chemin part du nœud initial, et traverse l'automate à travers des transitions jusqu'au nœud final. Cette couverture devient infaisable quand l'automate contient des cycles, car cela peut conduire à traverser des chemins de longueurs infinies.
- La couverture du chemin limité (m-chemin) : cette métrique est apparue pour résoudre le problème des automates avec des cycles dans le cas des chemins complets. C'est le même principe que dans la couverture de chemins complets, mais en fixant une longueur m du chemin qui inclut toutes les transitions et les cycles dans un automate.

Couverture fonctionnelle. Ce cas est le plus complexe, il concerne par exemple des séquences d'instruction d'un processeur ou des séquences de transactions sur un bus. Il faut alors mettre en œuvre des scénarios d'exécutions. Ceci peut se faire par des moniteurs associés par exemple à des assertions. Il peut être nécessaire d'examiner le comportement sur un seul cycle d'horloge, ou alors sur plusieurs cycles, ce qui fait appel à des opérateurs temporels. Ceci nécessite un important effort manuel, et de l'expérience.

Par ailleurs [39] s'intéresse à des métriques liées à la **couverture d'assertions**, dans le sens où il s'agit de mesurer si les assertions ont été complètement exercées lors d'une simulation. Il mentionne essentiellement les métriques suivantes : assertion step coverage (compte si toutes les étapes possibles ont été exercées : par exemple pour la SERE x [*0:7], tous les nombres

d'occurrences possibles de x entre 0 et 7 ont-ils été vus ?), assertion variable coverage (toutes les valeurs possibles des variables apparaissant dans l'assertion ont-elles été testées ?), assertion fault coverage (l'assertion permet-elle de détecter des comportements défectueux ?). Enfin il décrit l'outil de TransEDA [40], qui permet d'obtenir de telles mesures de couverture pour une simulation.

Nos travaux se rapprochent de cette problématique, mais se concentrent sur la non vacuité des tests par assertions. Ils ne visent pas à donner des mesures en fin de simulation pour estimer sa qualité a posteriori, mais à améliorer la qualité de la simulation en produisant des séquences de test garantissant une meilleure activation des assertions.

Dans [41], les auteurs se concentrent sur la couverture de la transition dans un graphe de transitions d'états. Les variables de la FSM sont classées en trois sous-ensembles, coverage model set (permettent de différencier des transitions), ignore set (ne sont pas suffisamment discriminatrices), et care set (doivent être considérées car peuvent être discriminatrices). En se basant sur ces informations, l'algorithme de génération de suites de tests procède essentiellement en considérant toutes les transitions et en utilisant des techniques du model-checking pour produire des contre-exemples à la négation de chaque transition.

[42] se base également sur les méthodes de model-checking, et propose un outil CAT (Coverability Analysis Tool). Deux problèmes sont notamment considérés : l'atteignabilité des valeurs d'une variable (en prenant en compte un ensemble fini de valeurs possibles), et la couverture d'instructions. En substance, l'approche utilisée pour étudier l'atteignabilité de la valeur Val_i pour une variable V revient, comme dans l'article présenté ci-dessus, à trouver les contre-exemples de la négation de la formule EF ($V = Val_i$) grâce à un model checker. L'idée de base est similaire pour la couverture d'instructions, puisque chaque instruction Si est remplacée par $V_i=1$; Si; et on concentre sur les contre-exemples de la formule ! (EF $V_i=1$).

Il s'agit de problèmes classiques, traités dans ces deux articles avec un raisonnement assez classique aussi, qui présente l'avantage de pouvoir être mis en œuvre facilement et de façon complétement automatisée, mais qui souffre de tous les inconvénients des techniques de model-checking (explosion combinatoire).

On retrouve une approche peu éloignée du contexte logiciel dans [43], qui s'intéresse à la génération de cas de test pour des circuits digitaux dont des parties sont décrites en Esterel. L'étude se base sur la couverture d'états, définie comme le ratio entre le nombre d'états atteints par simulation et le nombre d'états théoriquement atteignables (déterminé grâce à l'outil Xeve). L'ensemble des états atteignables non atteints par simulation est appelé missing state set. L'algorithme proposé vise à identifier les séquences de test qui conduiraient à cet ensemble d'états ; il exploite également model checking et négations de propriétés. La possibilité de prendre en compte des contraintes induites par l'environnement est aussi discutée.

Ces méthodes sont toutes basées sur des modèles à base d'automates, qui peuvent engendrer un problème d'explosion combinatoire. De plus elles génèrent une seule solution à la fois pour chaque cas traité, dont on n'a pas la certitude que ça soit la meilleure. Notre objectif est de trouver une méthode capable de générer un ensemble de séquences de test mais surtout d'une façon optimisée, ce qui va nous permettre de sélectionner les vecteurs de test les plus efficaces pour améliorer la qualité de la vérification, en se basant sur les informations fournies par la structure du circuit à vérifier.

L'approche de [44] est complémentaire à la nôtre. Le but ici n'est pas de contraindre les séquences de test de façon à ce qu'une certaine couverture soit garantie, mais d'évaluer la qualité du testbench en ce qui concerne le nombre des assertions activées. Les assertions sont écrites en langage ITL (langage propriétaire de OneSpin Solutions), sous la forme d'implications logiques sur des valeurs des signaux sur un intervalle de temps fini $t_{min}..t_{max}$. Elles sont transformées en "micropropriétés", dont la partie hypothèse booléenne est utilisée pour calculer le degré de la couverture d'assertion.

La motivation du travail de [45] est principalement la production de tests dirigés permettant de couvrir des cas limites et autres caractéristiques importantes du système vérifié. La solution préconisée aboutit à la question suivante "comment peut-on contrôler des signaux internes à partir des entrées primaires ?", autrement dit quelles sont les contraintes sur les entrées primaires permettant de produire la valeur 1 ou 0 sur un signal interne donné. Cette question est similaire au problème auquel nous allons aboutir dans le chapitre 3. La solution proposée ici est très différente de la nôtre. Elle ne prend pas en compte la structure du circuit concerné, mais se base sur des tests existants (à améliorer) et les traces de simulation résultantes. A partir des traces de simulation qui peuvent amener à la valeur souhaitée sur un signal donné, la méthode vise à extraire des contraintes telles qu'on puisse avoir une probabilité importante d'avoir la valeur voulue à l'endroit voulu. La méthode n'est donc pas exacte.

L'approche dans [46] s'intéresse à la génération des séquences de test pour la vérification dynamique, qui peuvent produire la réponse attendue du circuit ou révéler une fonctionnalité défectueuse. Les auteurs s'intéressent plus spécialement aux assertions sous la forme des automates générés à l'aide de l'outil MBAC, et proposent aussi un ensemble de métriques de couverture d'automates qui représentent les assertions, et les relient aux objectifs spécifiques de la couverture d'assertions, pour permettre de rapporter le problème de couverture d'assertion à celui de couverture d'automate et l'appliquer sur l'automate généré. Les algorithmes proposés visent donc principalement la couverture de nœuds ou de transitions. Les résultats sont des séquences de variables (qui étiquettent les transitions) qui conduisent au succès ou à l'échec de l'assertion. Cette approche considère exclusivement l'assertion et ne prend pas en compte le circuit. Elle permet donc d'identifier des séquences de variables (qu'elles soient entrées, sorties, ou signaux internes) qui exercent la propriété mais ne permet pas de produire des séquences d'entrées garantissant l'absence de vacuité.

II.3.4. Autres approches

L'article [47] se concentre particulièrement sur la génération de test dirigée, pour microprocesseurs pipelinés. Comme déjà rencontré ci-dessus, la solution est basée sur l'utilisation de techniques de model-checking pour produire des contre-exemples à la négation de formules (par exemple relatives à l'activation des interactions dans le pipeline). L'approche consiste à décomposer le système à vérifier et les propriétés correspondantes en des sous-propriétés. Les négations de ces sous-propriétés sont traitées par du Bounded Model Checking à base de SAT, et des contre-exemples 'locaux' sont générés. À la fin, les contre-exemples locaux sont fusionnés pour avoir un seul contre-exemple global. Cette approche est efficace du point de vue réduction d'espace de recherche dans le BMC (le choix du bound). Elle présente toutefois la caractéristique de ne générer qu'une seule solution. A l'inverse, nous recherchons la production d'une variété de solutions afin de pouvoir opérer la sélection de solutions les plus efficaces possibles pour l'activation des assertions.

Bien que ne visant pas la génération de séquences de test, [48] s'intéresse à un problème proche de celui qui nous concerne, en le considérant sur des modèles M de type automates, ou plus précisément structures de Kripke (ensembles d'états munis d'une relation de transition et d'une fonction d'étiquetage associant les états à des propositions atomiques). Le problème est appelé *trigger querying* et revient à chercher un ensemble de scenarios qui déclenchent une formule φ dans un modèle M (autrement dit, si une exécution de M a un préfixe qui reflète le scenario, alors son suffixe satisfait φ). Le problème est restreint à la recherche de solutions sous forme d'expressions régulières, c'est à dire qu'une solution *r* est l'expression régulière maximale qui satisfait $M \models r \rightarrow \varphi$, où → correspond à l'opérateur *suffix implication* de PSL. Les auteurs démontrent que la recherche de solutions à ce problème a une complexité polynomiale dans la taille du système. Pour des raisons pratiques, ils suggèrent donc de se limiter à la recherche de solutions partielles. Ils se ramènent donc au problème suivant : existe-t-il un mot *w* de longueur maximum n tel que $M \models w \rightarrow \varphi$? ils proposent deux méthodes pour résoudre ce problème, la première basée sur une technique BDD (Binary Decision Diagram), la deuxième sur SAT ; l'avantage de la première est qu'elle calcule tous les mots solutions, alors que la deuxième ne renseigne que sur l'existence de solution. L'algorithme à base de BDD procède en gros en calculant (symboliquement) tous les mots possibles de longueur *n* puis en ne conservant que ceux qui *déclenchent* φ (i.e. φ est satisfaite dans l'état terminal du mot). Enfin, des variantes du problème sont considérées, comme le *relevant trigger querying* (élimination des solutions *vides*) ou le *constrained trigger querying* (recherche de scenarios satisfaisant des contraintes données).

Chapitre 3.

Méthode développée

III. Méthode développée

III.1. Objectif

Le problème qui se pose dans cette thèse est celui de la vacuité dans la vérification des assertions. Les travaux menés dans cette thèse sont destinés à résoudre cette problématique pour améliorer la qualité de vérification à l'aide des assertions pendant la vérification dynamique (simulation), et assurer une meilleure couverture d'assertion. Dans l'exemple très simple $P: always\ (a \rightarrow b)$ il faudrait ainsi garantir que a est suffisamment souvent vraie. Comme nous cherchons à couvrir toute propriété du sous-ensemble simple, la solution n'est généralement pas aussi triviale.

Dans cette thèse, on va s'affranchir des problèmes liés à la manipulation d'automates. Nous allons considérer des modèles structurels des circuits (descriptions obtenues après synthèse logique). Le but est de développer une méthode qui donnera en résultat la spécification de générateurs de séquences de test activant suffisamment les propriétés. En général, ils positionneront les entrées primaires du circuit suivant des conditions sur les états des registres internes du circuit. La Figure III.1, montre les interconnexions du générateur de vecteurs de test sur les entrées, conditionnés par des états de registres internes à satisfaire. Les séquences de test ont pour rôle d'activer suffisamment souvent les signaux opérandes de la propriété à vérifier.

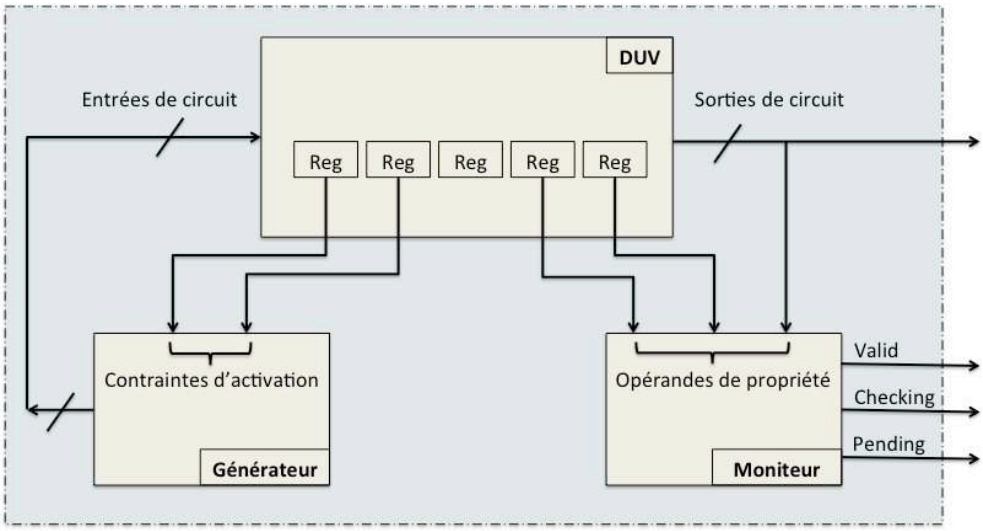


Figure III.1 : interconnexions entre DUV, moniteur et générateur de vecteurs de test

Nous présentons ici la solution que nous avons décrite dans [49].

III.2. Solution préconisée

III.2.1. Démarche

Il est nécessaire de commencer par caractériser les conditions d'activation pour les

moniteurs. Si la propriété est déjà sous la forme d'une implication, les conditions d'activation sont automatiquement prises de la partie gauche de l'implication. Par exemple :

Property **P** *:* ***Always*** *(****not*** *req1* ***and not*** *req2)* → *ack1*

Dans cette propriété la condition d'activation est une combinaison de signaux *(****not*** *req1* ***and not*** *req2)*. Dans ce cas, l'activation sera déduite en combinant les contraintes de chaque signal de la combinaison.

Si la propriété n'est pas une implication, il faut ramener cette propriété à la forme d'une implication. Cette solution consiste à transformer une propriété en une SERE sous la forme d'implication suffixe L |→ R. Des ensembles de règles de réécriture des propriétés pour le langage PSL ont été proposés dans [50], [51], et [52].

Règles de réécriture des propriétés. La transformation des propriétés sous forme des SEREs est vérifiée formellement dans [50]. Le but des règles est de ramener la majorité des opérateurs à un petit ensemble de cas de base, et le meilleur moyen est de les réécrire sous forme de SERE. L'article cite trois ensembles de règles de réécriture. On a par exemple celui de Singh & Garg [52] qui est utilisé pour transformer des expressions du sous-ensemble simple de PSL dans des formules d'implication en SERE.

Prenons l'exemple de deux opérateurs FL du langage PSL *eventually* et *never*. Ces opérateurs peuvent être transformés sous forme d'implication :

DR1 : ***eventually!*** $r \vec{=} \{true\} |\rightarrow \{[*]; r\}$!
DR2 : ***never*** $r \vec{=} r |\rightarrow \{false\}$

Exemple de transformation de propriété :

Si on reprend les propriétés qu'on a définies dans la section II.2, on trouve que les propriétés *(P2) et (P3)* sont déjà sous forme d'implication, dans ce cas il suffit de prendre les parties gauches comme conditions d'activation. Cependant, l'exemple de la propriété *(P1)* a la forme d'un opérateur FL. On peut transformer cette propriété sous forme d'une implication :

(P1) ***not*** *barriere_out_ouverte* ***until*** *ticket_insere*

Cette propriété peut être transformée en utilisant la règle suivante de [51] :

$$p \textbf{ until } b \bar{=} \{(\textbf{not } b)[+]\} |\rightarrow p$$

On obtient l'assertion équivalente suivante :

(P'1) $\{(\textbf{not } ticket\_insere)[+]\} |\rightarrow$ ***not*** *barriere_out_ouverte*

Cette transformation sous la forme d'implication va nous permettre d'identifier et isoler toutes les conditions d'activation de la propriété. Ces conditions seront prises en compte par

la suite pour générer des solutions capables de les satisfaire.

Une fois les conditions identifiées et isolées, l'étape suivante consiste à faire remonter ces conditions aux entrées primaires ou s'arrêter sur des registres internes. Ces conditions représentent les contraintes à prendre en considération pour l'activation des propriétés. Pour chaque contrainte sur un signal S au temps t : $S\ (t) = v$, le but est de déterminer toutes les solutions qui satisfassent cette contrainte.

III.2.2. Propagation des contraintes aux entrées et registres

Nous allons propager cette contrainte en revenant en arrière dans le modèle structurel du circuit, et en passant dans toutes les composants structurels (portes logiques et registres) qui se trouve sur la cône d'influence du signal S (Figure III.2). Les modèles structurels des circuits traités par l'algorithme ne contiennent que des composants simples. Le but est de remonter ces contraintes jusqu'aux entrées primaires du circuit. Toutefois le circuit peut contenir des boucles structurelles qui empêchent la propagation jusqu'aux entrées, et qui pourrait causer la création de boucles infinies. On cherche surtout à éviter ce phénomène. L'algorithme s'arrête donc lorsqu'il est remonté aux entrées primaires, ou aux entrées primaires et à un ensemble de registres déjà rencontrés.

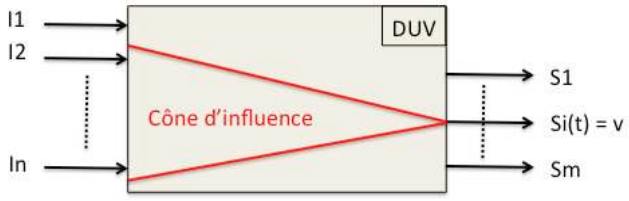


Figure III.2 : Cône d'influence d'un signal si dans DUV

L'objectif est de déterminer les conditions suffisantes pour avoir la valeur v sur le signal S. Deux cas de figures sont possibles :

1. L'algorithme arrive à remonter jusqu'aux entrées primaires : il suffit alors d'appliquer le vecteur de test correspondant pour générer la valeur v désirée sur le signal S.

2. L'algorithme s'arrête sur des registres et des entrées primaires : le générateur va devoir produire des vecteurs sur les entrées du DUV lorsque les conditions correspondantes sur les registres sont satisfaites.

L'algorithme est divisé en deux grandes parties :

- La première partie prend en charge la construction d'un arbre qui contient toutes les solutions possibles qui peuvent activer une condition donnée.

- Une fois que l'arbre des solutions est construit, on procède à l'extraction des solutions. Deux méthodes sont proposées :

- Méthode de parcours direct de l'arbre des solutions à l'aide de l'identification des portes conjonctives et disjonctives.
- Méthode utilisant un codage sous forme de BDD de la fonction booléenne représentée par l'arbre des solutions.

Pour **construire l'arbre des solutions**, on suppose que toutes les boucles structurelles dans le circuit contiennent au moins un registre (élément mémorisant). L'algorithme est capable de détecter la présence d'un registre déjà visité et s'arrête pour empêcher une boucle infinie. Il parcourt le modèle structurel du circuit en traversant les composants qui se trouvent sur le cône d'influence du signal en appliquant les règles de la Figure III.3.

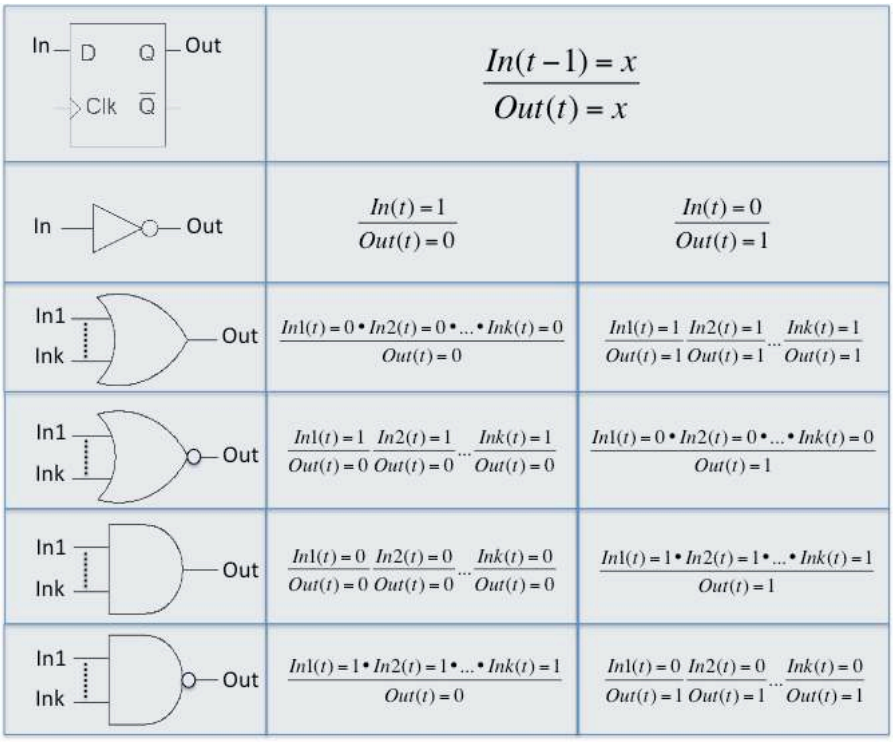


Figure III.3 : Ensemble de règles pour la remontée des contraintes

Ces règles prennent en compte le type du composant et la valeur du signal à la sortie :

- Registre : une contrainte sur sa sortie au temps t est rapportée sur son entrée au temps $(t - 1)$.
- Porte NOT : c'est le complément de la contrainte qui est rapporté vers l'entrée du composant.
- Porte OR : si la contrainte sur la sortie est 1, alors il suffit d'avoir la valeur 1 sur l'une de ses entrées. Dans le cas où la contrainte est 0, il faut avoir la valeur 0 sur toutes les entrées. On applique le même raisonnement sur les autres opérateurs binaires.

À l'aide de ces règles, l'algorithme construit un arbre de solutions pour déterminer les conditions suffisantes pour satisfaire la contrainte $S(t) = v$. Le parcours du modèle structurel commence à partir du nœud représentant le signal S pour une contrainte v sur sa sortie. Un

ensemble de dépendances *ens_dep* est construit ; au début il contient tous les nœuds qui précèdent le nœud *S* d'un pas sur son cône d'influence. Cet ensemble reflète les signaux qui doivent être traités après l'étape courante. Ces signaux seront traités par la suite un par un et de la même manière que le premier signal *S*, pour faire remonter les contraintes en construisant à chaque fois des ensembles qui correspondent aux signaux qui les précèdent.

```
// Initialisations
nœud *nœud1= créer_nœud (sig, x);
arbre_solution arbre = nœud1;
ensemble ens_sig = vide();
Insérer(ens_sig, sig);
int fini = 0;
// remonter jusqu'aux entrées primaires (tant que ens_dep ne contient pas que des entrées primaires)
while (!fini)
 {
     fini = 1;
     for (s = ens_sig->début; s != NULL; s = s->sig_suivant)
      {
          // mettre à jour l'arbre des solutions et retourner les nouveaux signaux à traiter
          ens_dep = traverse_porte(s, &arbre);
          // vérifier si ens_dep ne contient que des entrées
          b = entrées_seulement(ens_dep);
          fini = fini && b;
      }
     // mettre à jour l'ensemble ens_sig en utilsant ens_dep
     construire_ens_sig(ens_sig, ens_dep);
 }
```

Figure III.4 : Algorithme de remontée de contrainte

L'algorithme de la Figure III.4 montre les étapes générales suivies pour traverser les composants et construire en parallèle les ensembles de dépendances. D'abord un ensemble *ens_sig* est créé et contient le signal avec la contrainte à faire remonter. Ce premier signal passe dans la fonction *traverse_porte* (construite à base des règles ci-dessus). Elle génère deux résultats: le premier est l'ensemble de dépendance *ens_dep,* le deuxième est le nouvel arbre de solutions (*arbre*) issu de l'ensemble des signaux traités. On vérifie alors si *ens_dep* ne contient que des entrées primaires. S'il contient des signaux internes, on vérifie si ces signaux n'existent pas déjà dans une liste où l'algorithme sauvegarde tous les ensembles *ens_dep* des signaux traversés, si oui on supprime ce signal. Enfin, on construit un nouvel ensemble *ens_sig* et on répète l'exécution pour chaque signal dans l'ensemble *ens_sig* pour pouvoir remonter les contraintes dans le cône d'influence du signal. À la fin, on obtient un ensemble *ens_sig* vide qui signale la fin de l'exécution, et cela veut dire que l'algorithme a pu remonter les contraintes aux entrées primaires du circuit et/ou s'arrêter sur des signaux internes pour éviter de créer des boucles infinies. Le résultat est sauvegardé sous forme d'un arbre. Cet arbre représente l'ensemble des solutions possibles qui puissent satisfaire à une contrainte donnée.

Exemple illustratif : circuit BCD

Pour mieux expliquer le déroulement de cet algorithme, on va prendre le cas du circuit de la Figure III.5 qui représente une simple implémentation d'un reconnaisseur de code BCD [53]. Ce circuit a une seule entrée primaire *In*, une seule sortie *Out*, et contient 4 registres internes S_1, S_2, S_3 et S_4. On cherche à produire toutes les solutions pour la contrainte sur le signal de sortie, $Out\ (t) = faux$.

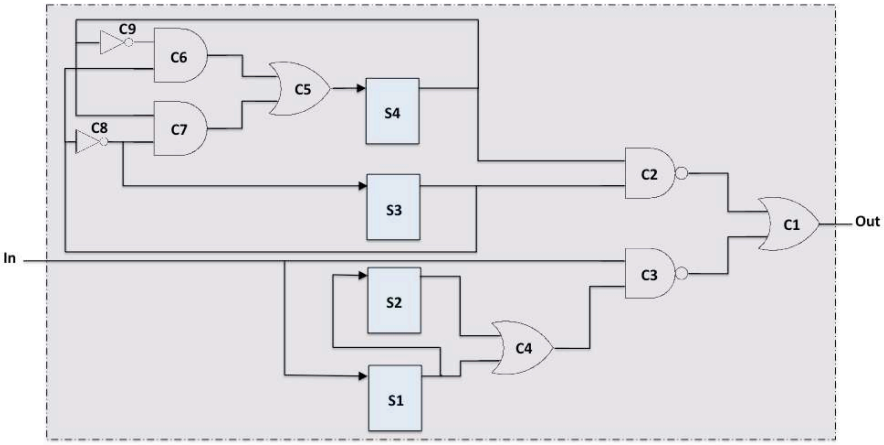


Figure III.5 : Circuit reconnaisseur de code BCD

Les détails de l'application de l'algorithme sont donnés en Annexe 1. L'algorithme génère finalement un arbre (Figure III.6) qui rassemble toutes les solutions possibles qui satisfassent la contrainte $Out\ (t) = faux$.

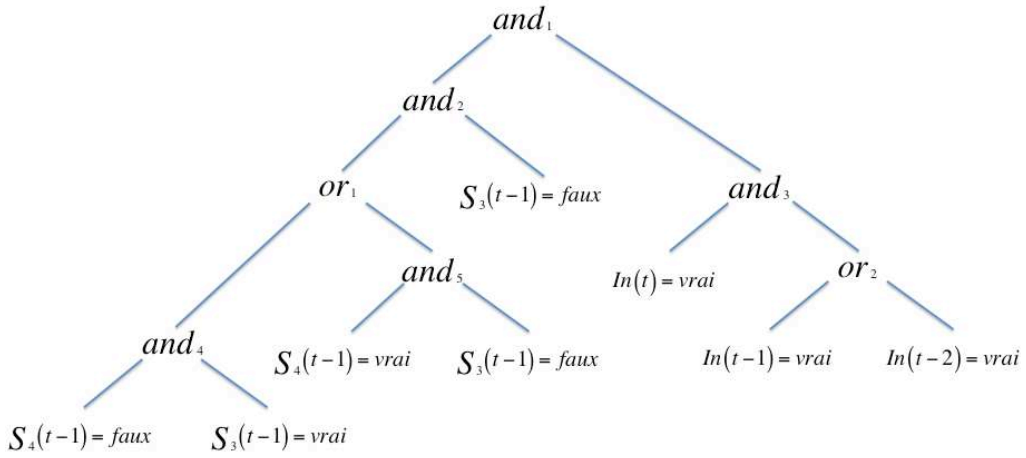


Figure III.6 : Arbre produit par l'algorithme pour satisfaire la contrainte Out (t) = faux

Après la génération de l'arbre des solutions, il est nécessaire d'**extraire les solutions de cet arbre**. La raison de produire plusieurs solutions capables de satisfaire des contraintes

données réside dans l'observation que dans ces solutions, certaines sont plus adéquates que les autres pour produire des générateurs de test.

Méthode 1 : avec parcours direct de l'arbre des solutions

Dans cette première méthode l'extraction se fait directement à partir de l'arbre de solutions, en tenant compte du type de dépendance (conjonctif ou disjonctif) entre les signaux dans cet arbre. La fonction construit les solutions suivant les opérateurs qu'elle traverse :

- Operateur conjonctif (and) : si la fonction le traverse, elle continue la construction de la solution en court en ajoutant la conjonction des conditions sur tous les fils de cet opérateur (Figure III.7 (a)).
- Operateur disjonctif (or) : si la fonction traverse cet opérateur, le nombre de solutions possibles augmente linéairement avec le nombre de fils (Figure III.7 (b)).

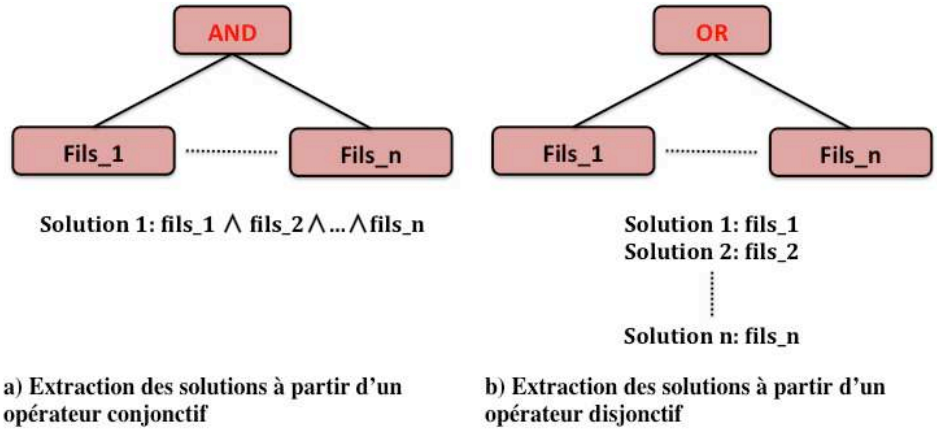


Figure III.7 : Opérateurs parcourus dans l'arbre des solutions

Exemple illustratif : circuit BCD

Reprenons l'exemple d'arbre de solutions de la Figure III.6. La fonction parcourt l'arbre à partir de la racine. Le premier opérateur traversé est un composant conjonctif and_1. Une première solution est en cours de construction, cette solution est la conjonction des deux fils de cet opérateur :

$$Solution\ 1 : and_2 \wedge and_3$$

Si les deux fils du premier opérateur ne sont pas des feuilles, la fonction passe aux fils de l'opérateur traversé. Le premier fils (and_2) est un opérateur conjonctif. On remplace and_2 par la conjonction de ses deux fils, tout en restant sur la même première solution :

$$Solution\ 1 : or_1 \wedge (S_3\,(t-1) = faux) \wedge and_3$$

On continue sur l'opérateur disjonctif (or_1) à deux fils. Deux possibilités sont créées : soit prendre en compte le premier fils, ou le deuxième. Cela exige la construction d'une deuxième solution :

$$\textbf{Solution 1} : and_4 \wedge (S_3\,(t-1) = faux) \wedge and_3$$
$$\textbf{Solution 2} : and_5 \wedge (S_3\,(t-1) = faux) \wedge and_3$$

On continue sur l'opérateur and_4 ; c'est un opérateur conjonctif à deux fils (feuilles) :

$$\textbf{Solution 1} : (S_3\,(t-1) = vrai) \wedge (S_4\,(t-1) = faux) \wedge (S_3\,(t-1) = faux) \wedge and_3$$

Même chose pour l'opérateur and_5, qui un opérateur conjonctif à deux fils :

$$\textbf{Solution 2} : (S_3\,(t-1) = faux) \wedge (S_4\,(t-1) = vrai) \wedge (S_3\,(t-1) = faux) \wedge and_3$$

Et ainsi de suite jusqu'à obtenir :

$$\textbf{S1'} : S_3\,(t-1) = vrai \wedge S_4\,(t-1) = faux \wedge S_3\,(t-1) = faux \wedge In\,(t) = vrai \wedge In\,(t-1) = vrai$$
$$\textbf{S2'} : S_3\,(t-1) = vrai \wedge S_4\,(t-1) = faux \wedge S_3\,(t-1) = faux \wedge In\,(t) = vrai \wedge In\,(t-2) = vrai$$
$$\textbf{S3'} : S_3\,(t-1) = faux \wedge S_4\,(t-1) = vrai \wedge S_3\,(t-1) = faux \wedge In\,(t) = vrai \wedge In\,(t-1) = vrai$$
$$\textbf{S4'} : S_3\,(t-1) = faux \wedge S_4\,(t-1) = vrai \wedge S_3\,(t-1) = faux \wedge In\,(t) = vrai \wedge In\,(t-2) = vrai$$

La fonction doit également prendre en compte les contradictions qui peuvent se produire dans certaines solutions, comme dans le cas des solutions **S1'** et **S2'**, où on trouve l'existence du signal $S_3\,(t-1)$ à valeur vraie et fausse dans la même solution. Elle supprime donc toutes les solutions contenant des contradictions. La Figure III.8 représente la liste finale des solutions générée ici. Après la suppression des contradictions on obtient deux solutions possibles.

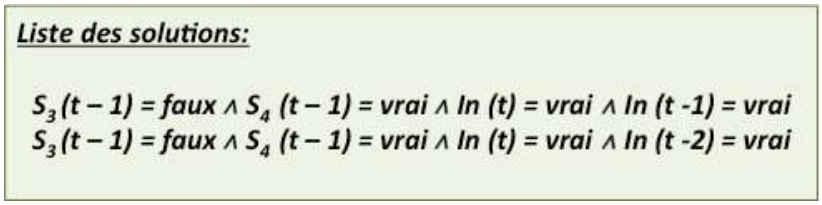


Figure III.8 : Solutions générées pour satisfaire la contrainte Out (t) = faux

Cette méthode présente l'avantage de fournir une précision optimale dans les solutions produites. Elle est cependant limitée à des circuits de taille moyenne car la construction des solutions consomme beaucoup de mémoire. La tâche de détection des solutions non contradictoires peut ajouter un léger surcoût en temps CPU.

Méthode 2 : avec CUDD et construction du BDD

Cette méthode se base sur la construction d'un BDD associé à la fonction booléenne exprimée par l'arbre des solutions afin d'en extraire les combinaisons de variables rendant cette fonction vraie.

BDD (Binary Decision Diagrams). Les BDDs sont des modèles de représentation particulièrement efficaces pour la manipulation des fonctions logiques, introduits en 1978 par Akers [2]. En 1986, Bryant propose les OBDDs (Ordered Binary Decision Diagrams), une représentation canonique des BDD, ainsi que des algorithmes pour calculer efficacement les opérations booléennes sur ces structures de données.

Un arbre de décision binaire est construit en appliquant récursivement la première forme du théorème de Shannon sur toutes les variables d'une fonction logique.

$$f(x_1,x_2,...,x_i,...,x_n) = \overline{x_i}.f(x_1,x_2,...,0,...,x_n) + x_i.f(x_1,x_2,...,1,...,x_n)$$

Exemple : On applique le théorème de Shannon sur la fonction $f(a,b,c) = \overline{a}.b.c + a.c$ (Figure III.9).

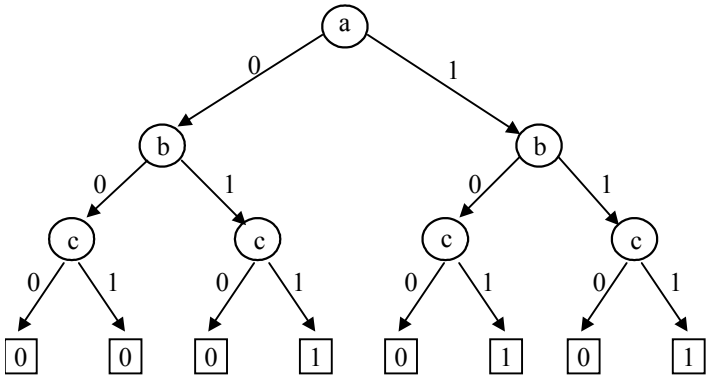
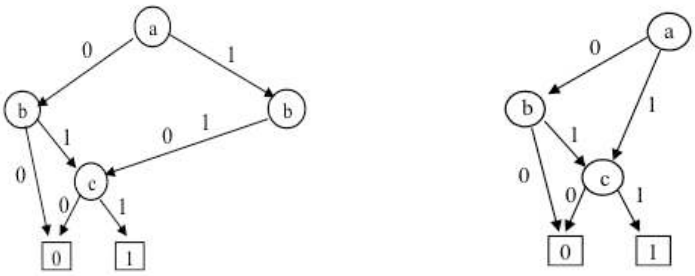


Figure III.9 : Arbre de décision binaire associé à la fonction $f = \overline{a}.b.c + a.c$

Pour réduire la complexité d'un arbre de décision binaire, il est possible de réduire sa taille par transformation en un graphe acyclique orienté appelé BDD. On cherche à avoir une présentation plus simplifiée en ne conservant qu'une seule représentation des feuilles et des sous-graphes identiques. On peut réduire encore plus le graphe de décision binaire en éliminant les sommets redondants (sommet avec fils gauche et fils droit identiques).

La Figure III.10 présente les étapes de réduction pour obtenir le BDD pour l'exemple précèdent de la fonction *f*.



a) Réduction par élimination des feuilles et sous-graphes identiques

b) Réduction par élimination des sommets redondants

Figure III.10 : Le Diagramme de Décision Binaire (BDD) pour $f = \overline{a}.b.c + a.c$

Le BDD est une représentation canonique dont les variables ne peuvent apparaître qu'une seule fois sur chaque chemin entre la racine et une feuille, et dont la complexité de la structure dépend fortement de l'ordre dans le quel sont considérées les variables. Ainsi, pour deux ordres différents, les BDDs peuvent être de tailles très différentes.

CUDD Package (CU Decision Diagram). Le package CUDD (Colorado University Decision Diagram) [54] est un ensemble de fonctions qui permettent la création et la manipulation de diagrammes de Décision binaires (BDD) ainsi que de ZDD (Zero-suppressed Decision Diagrams) et de ADD (Algebraic Decision Diagrams). La méthode d'extraction des solutions proposée ici se base sur la construction du BDD et utilise le package CUDD.

L'arbre de solutions peut être transformé en une forme exploitable par CUDD pour construire un BDD. La transformation vers les fonctions de création de BDD est décrite en Annexe 2. Le BDD généré par la fonction de la Figure A2.3 est présenté dans la Figure III.11 :

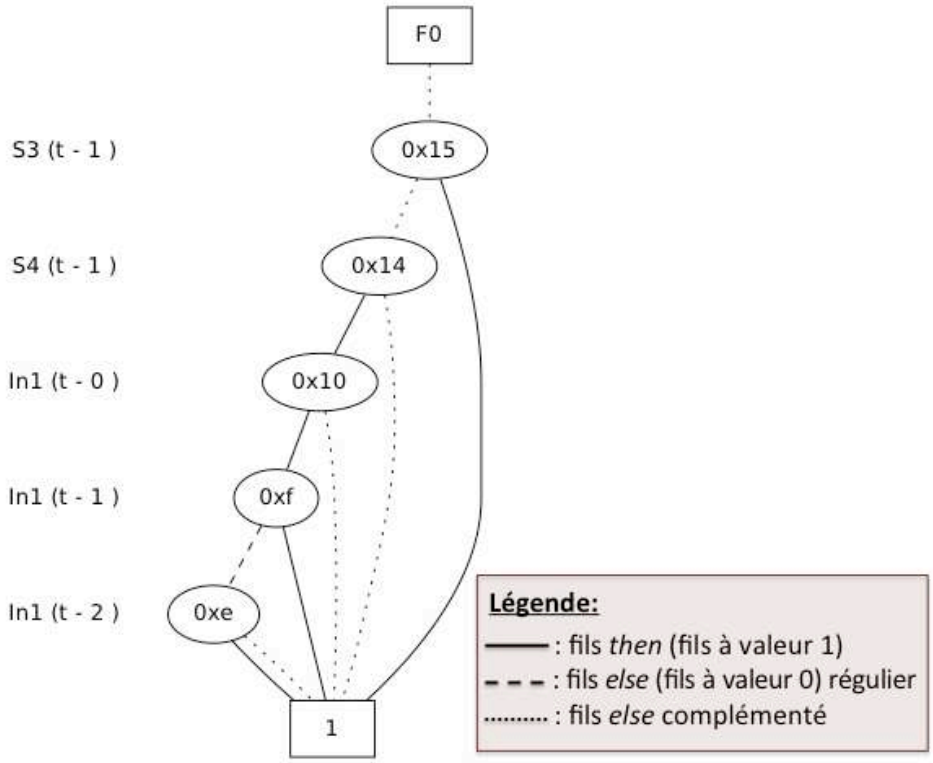


Figure III.11 : BDD pour la contrainte Out (t) = faux sur la sortie du circuit BCD

Une fois que le BDD est construit, il faut le parcourir pour extraire tous les chemins de la racine vers la feuille 1. Pour ce faire, on peut créer une fonction de parcours du BDD. Reprenons l'exemple du circuit BCD, et la contrainte $Out\ (t) = faux$. Un BDD correspondant à l'arbre de solution est montré dans la Figure III.11. La Figure III.12 représente les chemins possibles qui existent dans le BDD.

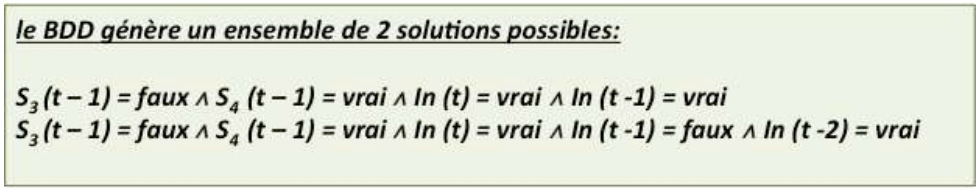


Figure III.12 : Solutions extraites à l'aide de la fonction de parcours de BDD

L'inconvénient de cette fonction est une redondance dans les solutions. Comme le montre la Figure III.12 le passage par le nœud *In (t - 1)* dans la deuxième solution est obligatoire pour continuer vers son sous-arbre. Cela crée une redondance de signaux et affectations des signaux inutile dans la solution proposée (l'affectation du signal *In (t - 1)* n'a pas d'influence sur la deuxième solution).

On peut donc plutôt utiliser la fonction CUDD d'extraction des implicants premiers. Dans cet implicant, seuls les signaux qui sont impliqués dans le trajet minimal sont affectés, les autres sont considérés comme des *''don't care''*. L'ensemble des solutions générées à l'aide de cette fonction représente des vecteurs de test très compacts. Si on reprend l'exemple de BCD pour la contrainte $Out(t) = faux$ sur sa sortie, la fonction génère les solutions de la Figure III.8.

Pour résumer, la méthode d'extraction de solutions à base de la fonction de parcours direct de l'arbre des solutions produit un ensemble de solutions qui portent sur toutes les variables de l'arbre. Ces solutions contiennent plus d'informations sur l'affectation des signaux. Cependant, elles peuvent être très longues, et les signaux de l'arbre sont indépendants, ce qui peut causer des contradictions, ce qui entraine une tâche de vérification supplémentaire pour chaque solution produite. Cette méthode est efficace dans le cas des circuits de taille moyenne. Dans la méthode d'extraction à base de BDD, les solutions sont générées sans contradictions, grâce à la forme canonique du BDD créé. Cependant, les solutions générées contiennent moins d'informations que dans celles générées par la première méthode. Les signaux ne sont affectés que s'ils ont une réelle influence pour satisfaire une contrainte donnée, sinon, ils sont considérés comme des signaux à valeur *''don't care''*. Les solutions dans ce cas sont plus compactes et plus faciles à utiliser.

III.2.3. Choix des solutions

Il s'agit maintenant de choisir les solutions qui sont les plus adéquates pour créer des générateurs de test. Une solution contient des conditions sur des entrées du circuit et sur des états internes. Des critères s'imposent dans le choix des solutions pour pouvoir aisément les transformer en générateurs de vecteurs de stimulis d'entrée conditionnés par l'état du circuit.

Un premier critère de choix concerne les *paramètres temporels des entrées primaires et composants de mémorisation* dans les solutions. En effet, les conditions portant sur l'état interne doivent être antérieures aux contraintes indiquant comment forcer les valeurs des entrées primaires en conséquence. Le générateur doit réagir en fonction des conditions sur les états internes du circuit et produire des valeurs présentes ou futures sur les entrées.

Reprenons l'exemple de la liste des solutions (Figure III.8) générée pour satisfaire la contrainte $Out\ (t) = faux$. On remarque que la première solution est la seule solution viable : le générateur produira la séquence $\{1\ ;\ 1\}$ sur l'entrée In quand il détectera la condition sur les états internes du circuit $S_3\ (t-1) = faux \wedge S_4\ (t-1) = vrai$. La deuxième solution est irréaliste, car les conditions sur les états internes du circuits sont vérifiées à $(t-1)$, tandis que la solution indique la génération d'une valeur vraie sur l'entrée In à temps $(t-2)$, cela veut dire une génération dans le passé, ce qui est impossible. Un générateur ne peut pas prévoir quelle valeur doit être générée sur l'entrée avant la production des conditions sur les états internes du circuit.

D'autre part, il existe des solutions qui ne portent que sur des états internes. Ces solutions sont évidemment inutiles car elles ne fournissent pas de vecteurs de test pour les entrées. Reprenons par exemple le cas du circuit reconnaisseur de code BCD, mais avec la contrainte $Out\ (t) = vrai$. On obtient l'ensemble des solutions de la Figure III.13.

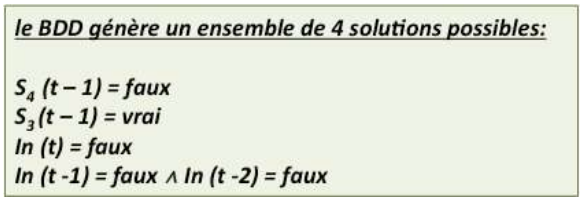


Figure III.13 : Solutions de la contrainte Out (t) = vrai

Dans ce cas, les choix possibles seront la troisième et la quatrième solution, car la première solution porte uniquement sur l'état interne $S_4\ (t-1)$ et la deuxième porte uniquement sur le registre $S_3\ (t-1)$.

Les signaux rencontrés par l'algorithme n'ont pas de signification particulière pour lui, sauf pour différencier les entrées primaires des signaux internes. On peut donc trouver des solutions qui contiennent des valeurs sur des signaux qui correspondent à des valeurs inhabituelles pour le protocole suivi dans le fonctionnement. Prenons le cas de l'entrée primaire *reset* (on considère par exemple que cette entrée est active à vrai). Des solutions qui exigeraient de maintenir la valeur vrai sur l'entrée *reset* maintiendraient le circuit toujours en état d'initialisation, ce qui ne correspondrait pas à une simulation réaliste. De même dans le cas d'un circuit ayant un signal *enable* (par exemple le circuit HDLC traité plus loin). Suivant le fonctionnement du circuit, ce signal est majoritairement maintenu à la valeur vraie. Les solutions qui exigeraient la valeur faux trop souvent sur ce signal ne donneraient pas non plus un comportement réaliste.

Pour faciliter la sélection des solutions suivant ces critères, un outil a été développé pour automatiser cette tâche [55]. Il prend d'une part la liste des solutions comme fichier d'entrée, et d'autre part les critères à respecter (Figure III.14). Cet outil permet de sélectionner une liste qui ne retient que les solutions les plus souhaitables et qui satisfassent tous les critères qu'on a cités précédemment.

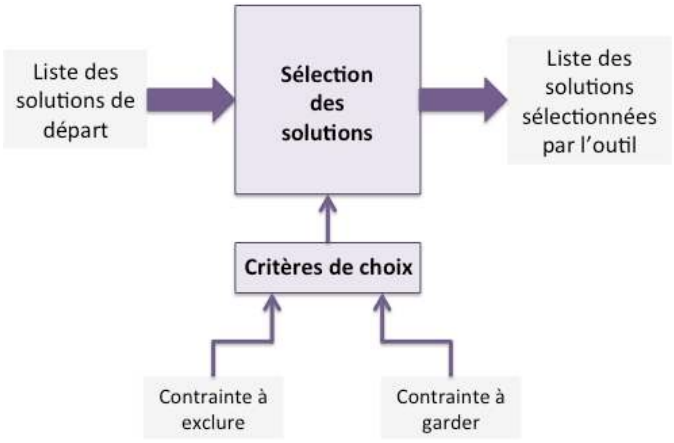


Figure III.14 : Outil de sélection des solutions suivant des contraintes données

Cependant dans la liste des solutions générées, il reste à prendre en compte les conditions sur les états internes et déterminer celles qui sont les plus probables à atteindre durant la simulation. Cela va permettre de choisir des solutions les plus aptes à produire une contrainte donnée sur le signal à traiter. Pendant sa visite dans l'équipe, Kjetil Svarstad (Norwegian University of Science and Technology) a proposé et implémenté une méthode qui s'intéresse à la *probabilité d'atteindre les états impliqués dans les solutions*, afin d'augmenter les chances de contraindre les entrées les plus souvent possibles. Le but de cette méthode est de définir les états qui sont plus fréquemment atteints pendant la simulation. Dans la solution développée, le circuit est transformé en un ensemble de BDDs. On suppose que les valeurs appliquées sur les entrées du circuit ont une probabilité 0,5 d'être à vrai et 0,5 d'être à faux, et une itération de point fixe fournit une estimation de la probabilité d'atteignabilité de chaque état.

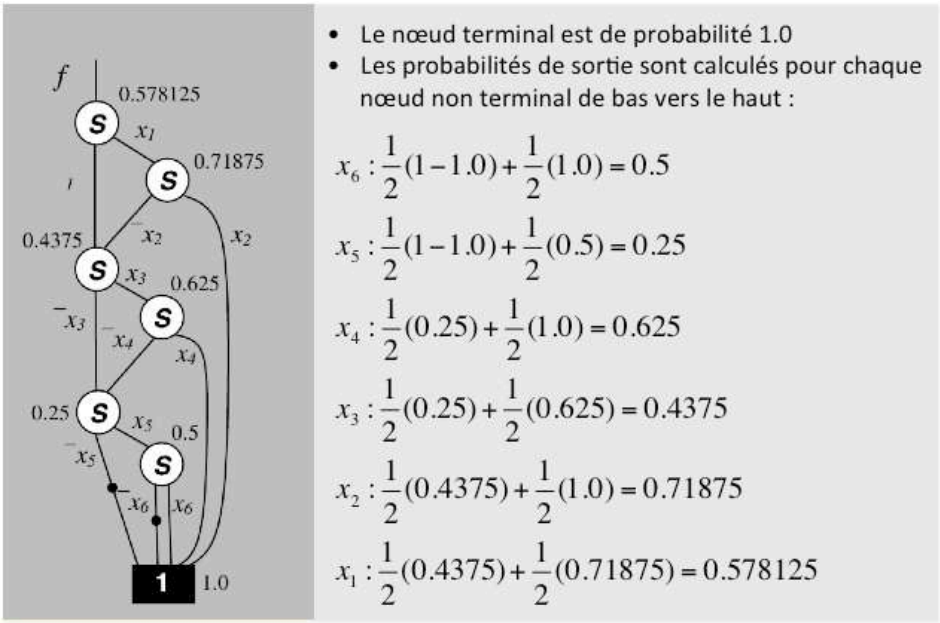


Figure III.15 : Calcul de probabilité des nœuds du BDD de la fonction f

Prenons l'exemple de la fonction $f = x_1x_2 \vee x_3x_4 \vee x_5x_6$. Comme le montre la Figure III.15, le calcul des probabilités pour les nœuds non terminaux se fait dans l'ordre de bas vers le haut. On commence le calcul de la probabilité à partir des nœuds qui sont en bas, dans le cas de cet exemple on commence par x_6. Après application de l'extension de Shannon on obtient la probabilité d'atteindre x_6 est de 0,5. Une fois la probabilité de x_6 calculée on passe au nœud au dessus, jusqu'à arriver au nœud racine x_1. Ce nœud correspond à la fonction f, on trouve que la probabilité sur x_1 est de : 0,578125. On peut facilement vérifier la probabilité de la sortie pour la comparer à ce calcul via le BDD. En utilisant par exemple une table de vérité on trouve que pour six variables il y a 64 possibilités à vérifier pour f, dont 37 parmi elles conduisent à $f = 1$. Donc la probabilité d'avoir f à vrai est : $37/64 = 0{,}578125$.

On va reprendre par exemple le cas du circuit contrôleur de parking décrit dans la section II.2; sa machine à états sera détaillée dans la section IV.1. Pour une contrainte sur la sortie ***barriere_out_ouverte (t) = faux***, l'algorithme produit un premier ensemble de solutions qui rassemble toutes les possibilités qui peuvent produire cette contrainte sur la sortie ***barriere_out_ouverte***. Comme le montre la Figure III.16, le circuit synthétisé a trois registres internes state_0, state_1 et state_2, et les solutions générées contiennent des conditions sur ces registres.

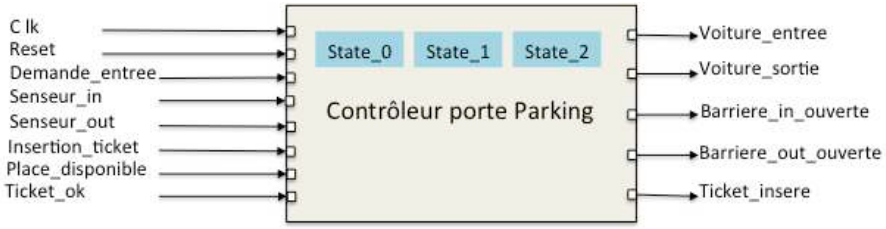


Figure III.16 : États internes du circuit contrôleur de parking

La présence des 3 registres est due à l'utilisation d'un codage binaire pour l'état symbolique lors de la synthèse :

États de machine		Idle	Ouverture_in	Fermeture_in	Verif_ticket	Ouverture_ out	Fermeture_out
Registre	State_0	**0**	**1**	**0**	**1**	**0**	**1**
	State_1	**0**	**0**	**1**	**1**	**0**	**0**
	State_2	**0**	**0**	_	_	**1**	**1**

On applique la méthode concernant le calcul de la probabilité d'atteindre les états impliqués dans les solutions qui produisent la contrainte sur la sortie, en supposant que les valeurs appliquées sur les entrées du circuit ont une probabilité 0,5 d'être à vrai et 0,5 d'être à faux, on obtient les résultats de la Figure III.17. Les résultats des calculs montrent par exemple que la sortie barriere_out_ouverte est majoritairement à faux, et que les états internes state_0, state_1 et state_2 du circuit sont plus fréquemment à faux. En prenant en considération ces

calculs de probabilités, on peut faire un choix dans la liste des solutions et privilégier celles qui contiennent des conditions sur les registres (state_0, state_1 et state_2) à faux.

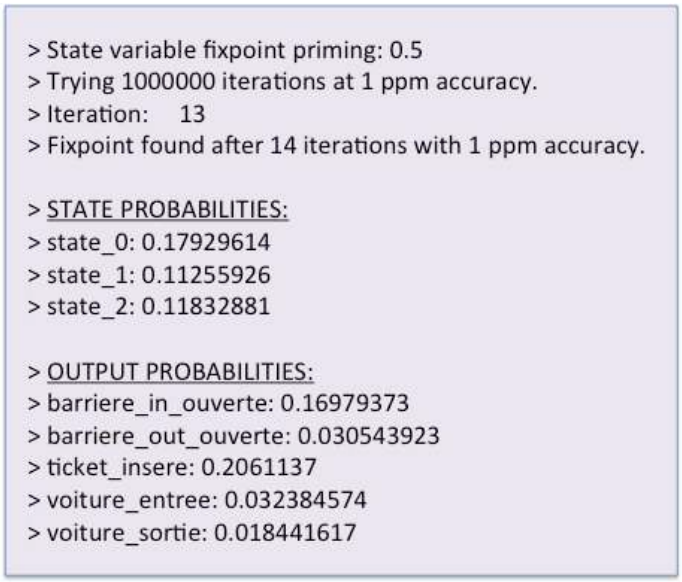


Figure III.17 : Probabilités des états internes et sorties du circuit contrôleur de parking

L'application de cette méthode peut s'avérer très intéressante. Cependant, elle suppose que les probabilités des valeurs affectées aux entrées primaires sont toujours de 0,5. Il restera donc à améliorer la méthode pour le cas où il y a des relations de dépendances entre les entrées, ou plus encore, si le fonctionnement décrit un protocole donné.

III.2.4. Mise en œuvre des générateurs de séquences de test

Les solutions produites dans la section précédente permettent la production d'une contrainte donnée sur le signal traité. Ce signal est un opérande d'un opérateur qui appartient à la partie gauche d'une implication logique ou SERE qu'on veut vérifier au cours de la simulation. La production des contraintes sur ces opérandes garantit l'activation de ces assertions. Suivant les solutions choisies, des conditions qui portent sur des états internes du circuit doivent être surveillées, de façon à appliquer des valeurs ad hoc sur des entrées primaires pour propager une contrainte jusqu'au signal opérande. La dernière étape de l'approche développée consiste à intégrer ces solutions pendant la simulation.

Ces solutions sont transformées en des spécifications logiques, qui permettent de distinguer les conditions et les séquences à appliquer sur les entrées primaires impliquées. Une spécification est constituée de deux parties, une première partie correspond à des conditions à satisfaire sur des états internes du circuit, et la deuxième partie correspond à la séquence de test appliqué sur les entrées primaires un fois que les conditions sont satisfaites au cours de la simulation. Nous avons ainsi la spécification du générateur de séquences de test voulu. La construction de ce générateur de séquences de test peut être faite de deux façons :

- Création d'un "process" VHDL qu'on peut intégrer directement dans le testbench pour simuler le circuit.

- Construction du générateur à l'aide de la méthode développée dans [13], connecté au circuit comme décrit dans la Figure III.1.

III.2.4.1. Écriture d'un process VHDL

Durant la simulation, un testbench est chargé de produire des séquences de test pour dérouler une simulation des signaux du circuit à vérifier. Des séquences de test peuvent être générées d'une façon pseudo-aléatoire ou dirigée, le choix dépend du besoin du concepteur au départ. Cependant ces fichiers testbench restent modifiables et adaptables au changement de besoin. La méthode proposée dans cette section consiste à remplacer le process VHDL de génération pseudo-aléatoire de jeux de test par un process qui prend en compte une solution choisie. Ce processus représente un générateur de vecteurs de test conditionné avec des états internes du circuit. Lorsque les conditions sur ces états internes sont satisfaites, des valeurs données sont appliquées sur les entrées pour satisfaire une contrainte donnée. Des valeurs pseudo-aléatoires sont appliquées sur les entrées primaires lorsque les conditions voulues sur les états internes ne sont pas satisfaites (Figure III.18).

```
process
  variable seed1, seed2: positive := 200;
  variable rand: real;
  variable int_rand: integer;
  variable stim: std_logic_vector(7 downto 0);
  begin
    wait until clk'event and clk='0';
    if (cpt mod 5 = 0) then
      UNIFORM(seed1, seed2, rand);
      int_rand := INTEGER(TRUNC(rand*256.0));
      stim := std_logic_vector(to_unsigned(int_rand, stim'LENGTH));
      entrée_1 <= stim(0);
      entrée_2 <= stim(1);
          ⋮
    end if;
  end process  Processus de génération de séquences de test;
```

Figure III.18 : Génération des séquences pseudo-aléatoires sur les entrées primaires

Pour illustrer, on va reprendre l'exemple du circuit BCD (Figure III.5). Pour une contrainte sur la sortie $Out\ (t) = faux$, et après vérification des critères de choix, on a sélectionné la solution :

$$S_3\ (t-1) = faux \wedge S_4\ (t-1) = vrai \wedge In\ (t) = vrai \wedge In\ (t\text{-}1) = vrai$$

Les conditions portant sur les états internes dans cette solution sont : S_3 à faux et S_4 à vrai. Lorsque la conjonction de ces conditions est satisfaite, une séquence de valeurs { vrai ; vrai } doit être appliquée sur l'entrée *In* du circuit. À partir de ces données on peut construire un process qu'on va intégrer dans le testbench pour simuler le circuit BCD (Figure III.19).

Cela permet d'améliorer la qualité de la vérification en produisant les valeurs attendues sur les signaux opérandes de l'assertion pendant la simulation. Le process VHDL n'étant pas

synthétisable, il faudrait le tranformer en la machine à états finis correspondante pour le réutiliser en émulation matérielle.

```
Process
Begin
      wait until ck'event and ck = '0';
      if (S3 = '0' and S4 = '1') then
            In <= '1';
            wait until ck'event and ck = '0';
             In <= '1';
      else
            -- Génération des valeurs aléatoires pour l'entrée In.
             ⋮
      end if;
End process;
```

Figure III.19 : Processus de génération de séquences de test pour le circuit BCD

III.2.4.2. Générateur de séquences de test

L'outil Horus développé dans l'équipe donne la possibilité de transformer une spécification en un composant HDL, qui peut être soit des moniteurs, des générateurs ou des générateurs étendus. Ces générateurs étendus sont des générateurs qui produisent des séquences de test conditionnées avec des contraintes. Les solutions sélectionnées peuvent être transformées en des spécifications logiques exploitables par Horus. Tout comme ci-dessus, le but est de surveiller le comportement des états internes du circuit jusqu'à satisfaire des conditions précises, qui impliquent la production des séquences sur les entrées. La méthode utilisée dans cette section prend les solutions sélectionnées et les transforme en des générateurs étendus qu'on peut par la suite connecter avec le circuit. Ces modules générés à l'aide de SynthHorus vont générer des séquences de test pseudo-aléatoires grâce au bloc LFSR intégré dans le générateur étendu pour garantir la continuation de la simulation, en parallèle il surveille le comportement des états internes inclus dans la solution. Une fois qu'il reconnait les conditions satisfaisables, il envoie des séquences précises sur les entrées pour produire la contrainte voulue sur le signal opérande traité.

SyntHorus. Il fait partie de l'outil Horus, il se charge de transformer une spécification temporelle écrite dans le sous-ensemble simple du langage PSL en une description HDL correcte par construction. Il est capable de combiner les hypothèses ("assumptions") et les assertions pour créer un composant qui mixe les générateurs et les moniteurs ; ces nouveaux composants sont appelés des générateurs étendus. Tout comme Horus, SyntHorus se base sur une construction modulaire. La construction d'un module complexe se fait en combinant plusieurs modules d'opérateurs de base, combinés avec des composants spécifiques appelés des solveurs [13].

Pour illustrer, on va reprendre l'exemple du circuit du contrôleur du parking présenté dans la section II.2. On considère la propriété P2 :

(P_2) ***always*** *(ticket_insere →* ***next !*** *(****not*** *ticketOK →* ***not*** *barriere_out_ouverte))*
Pour activer suffisamment souvent cette propriété, il faut produire la valeur vrai sur le signal opérande *ticket_insere* de la partie gauche de l'implication. L'algorithme nous génère un ensemble de solutions potentielles. Considérons la solution suivante :
(S_1) : State_0 (t – 1) = vrai, place_disponible (t – 1) = faux, insertion_ticket (t – 1) = faux, place_disponible (t) = faux, insertion_ticket (t) = vrai

La solution présente un ensemble de conditions à satisfaire sur des états internes du circuit, et des séquences à appliquer sur les entrées primaires. La spécification PSL *SP_2* qui correspond à la solution *S_1*, exploitable par SyntHorus est :

(SP_2) : Always (state_0 → (not place_disponible and not insertion_ticket) and next ! (not place_disponible and insertion_ticket)) ;

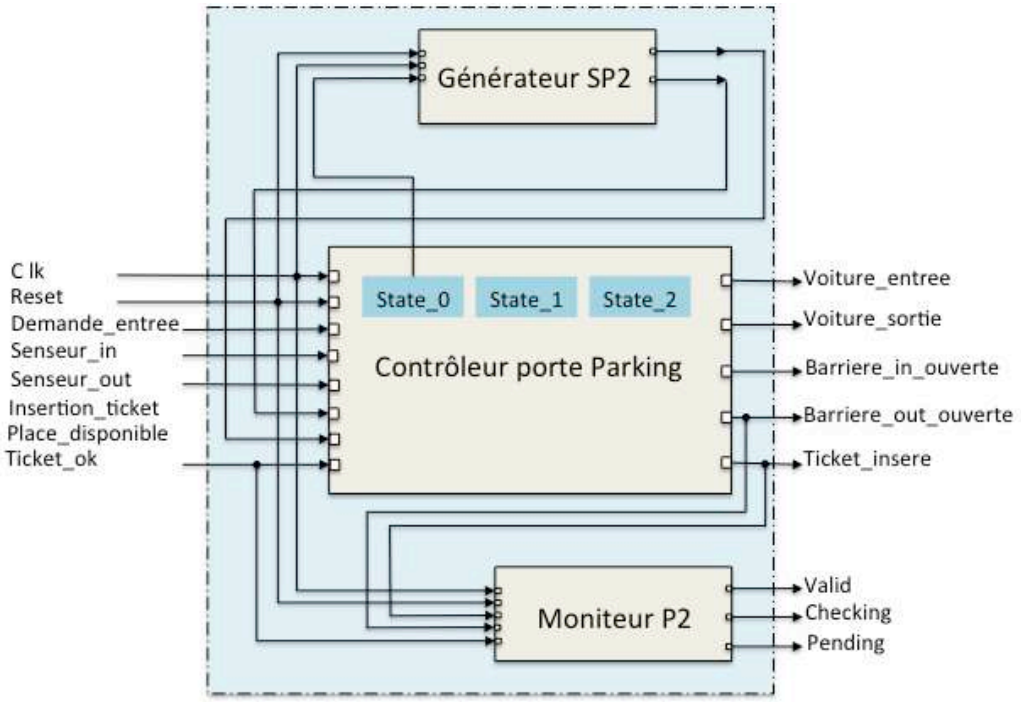


Figure III.20 : connexion du générateur SP2 avec le circuit contrôleur de Parking

Le module générateur SP2 généré à partir de la spécification est connecté avec le circuit à vérifier (Figure III.20). Il assure l'activation du signal *ticket_insere* pour pouvoir vérifier l'assertion P2 transformée en moniteur P2. La condition sur l'état du registre interne *state_0* est surveillée par le générateur SP2. Une fois satisfaite, une séquence de valeurs $\{0\ ;\ 0\}$ est appliquée sur l'entrée *place_disponible*, et une séquence de valeurs $\{0\ ;\ 1\}$ est appliquée sur l'entrée *insertion_ticket*. Nous reprendrons cet exemple de façon plus complète (prise en compte des deux parties gauches d'implications) dans le chapitre IV.

Contrairement à la méthode précédente, la production de générateurs synthétisables permet l'implémentation de cette méthode, ce qui permet la vérification par émulation matérielle.

III.3. Automatisation du procédé

L'approche proposée doit prendre en entrée des descriptions VHDL, et fait appel à des formats de fichiers différents. Une automatisation des transformations a été réalisée (Figure III.21).

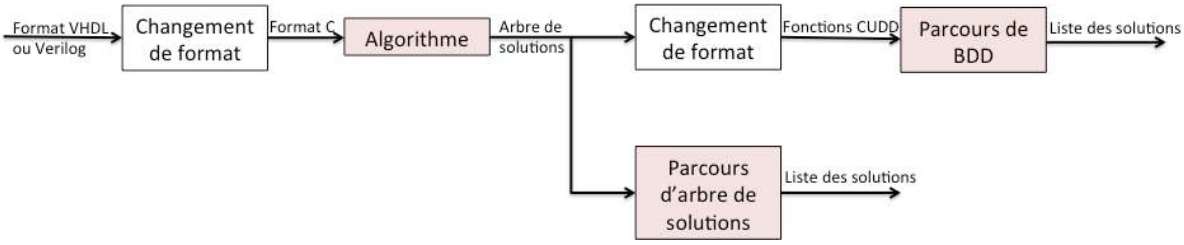


Figure III.21 : Transformation des formats de fichiers

Deux transformations de formats d'entrée sont sollicitées dans cette approche :

- Transformation du format du circuit en VHDL ou Verilog vers un format en C.
- Transformation du format d'arbre de solutions généré par l'algorithme en des fonctions exploitables par CUDD pour la création du BDD correspondant.

La Figure III.22 décrit les étapes suivies pour générer le format en C de la description matérielle du circuit à vérifier, à partir d'une description VHDL ou Verilog au niveau RTL. Ce fichier sera utilisé par la suite comme un fichier d'entrée pour l'algorithme décrit précédemment.

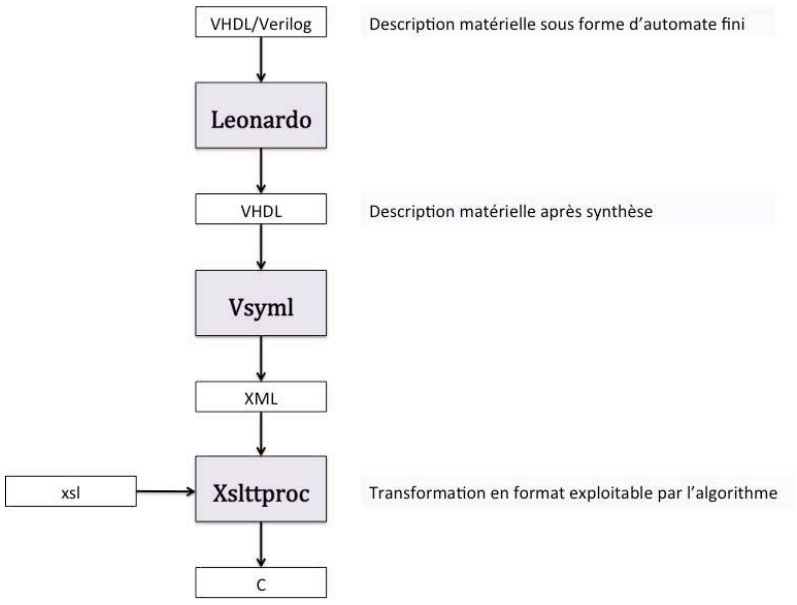


Figure III.22 : Génération du fichier C exploitable par l'algorithme de propagation de contraintes

Afin de se ramener à des modèles structurels de circuits, les descriptions sont synthétisées à l'aide de l'outil Leonardo. Leonardo est un outil de Mentor Graphics, il accepte les systèmes décrits en VHDL et Verilog et effectue une synthèse logique ; il représente une bonne solution pour la synthèse à la fois FPGA et ASIC. Une fois que le résultat de la synthèse est généré, il est analysé et transformé en un fichier XML en utilisant l'outil Vsyml développé dans l'équipe [56]. Cet outil est un analyseur et simulateur symbolique pour VHDL, qui produit une sortie XML [57], il est assez simplement utilisé ici pour récupérer un format XML à partir de code structurel. Le format XML est enfin transformé en format C par un script XSLT.

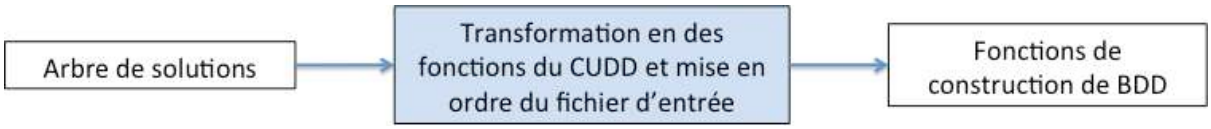


Figure III.23 : Transformation de l'arbre de solutions en des fonctions de construction de BDD

Après la génération de l'arbre des solutions vient l'étape d'extraction des solutions. La première variante réalisée prend l'arbre des solutions produit par l'algorithme et l'exploite directement sans avoir besoin d'intermédiaire. La deuxième variante fait appel au package CUDD pour créer le BDD correspondant à l'arbre. Une transformation de l'arbre des solutions sous la forme de fonctions de construction de BDD à base des fonctions CUDD est nécessaire (Figure III.23). Un nouveau fichier est créé pour représenter l'arbre des solutions sous forme de fonctions CUDD, et construire des fonctions de construction et de parcours de BDD.

Chapitre 4.

Applications

IV. Applications

Dans ce chapitre on va étudier l'application de l'approche développée sur quelques cas d'études. Une étude comparative sur l'amélioration de la qualité de la vérification dynamique est ainsi effectuée. Nous allons commencer par un exemple illustratif simple pour montrer les différentes étapes suivies pour la génération des vecteurs de test. Pour cela, on va reprendre l'exemple du circuit décrit dans la section II.2. Nous allons ensuite décrire les étapes de génération des solutions, les critères de choix et les résultats d'expérimentations sur une étude de cas fournie par Dolphin Integration, et une autre étude de cas (interface pour protocole HDLC) fournie par Thales Communications & Security.

IV.1. Exemple illustratif : Circuit contrôleur de parking

Dans cette section on va reprendre l'exemple décrit dans la section II.2 du circuit contrôleur de parking. En appliquant notre approche on a réussi à augmenter le nombre d'activations des assertions. On va reprendre les étapes suivies dans cette approche pour générer des vecteurs de test satisfaisant des contraintes sur les signaux à traiter.

Comme le montre la Figure IV.1, le fonctionnement de ce petit exemple est réparti sur 6 états symboliques. A partir de l'état idle, la partie droite de la FSM gère une demande d'entrée de voiture, et la partie gauche une demande de sortie.

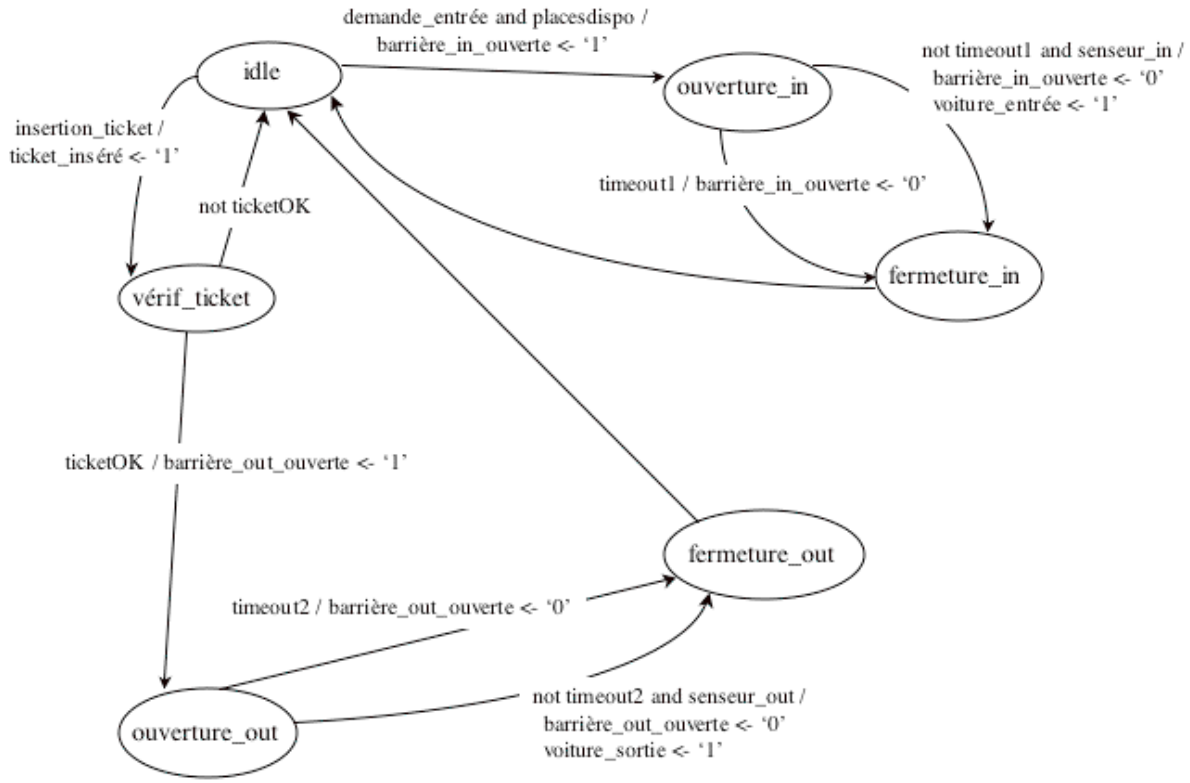


Figure IV.1 : FSM du circuit contrôleur de barrière de parking

A partir de l'état `idle` :

- lorsqu'une voiture demande à entrer, s'il y a des places disponibles dans le parking, le système passe dans un état `ouverture_in` en commandant l'ouverture de la barrière. Si la voiture ne rentre pas dans le laps de temps qui lui est imparti, le système passe dans un état `fermeture_in` en commandant la fermeture de la barrière. Si elle rentre, le système passe également dans l'état `fermeture_in` en commandant la fermeture de la barrière, et en indiquant qu'une voiture est entrée (sert au système de comptage des voitures)
- lorsqu'un automobiliste insère son ticket de sortie, le système passe dans un état `vérif_ticket` après avoir indiqué qu'un ticket a été inséré. Si le ticket est valide, le système passe dans un état `ouverture_out` en commandant l'ouverture de la barrière. Si la voiture ne sort pas dans le laps de temps qui lui est imparti, le système passe dans un état `fermeture_out` en commandant la fermeture de la barrière. Si elle sort, le système passe également dans l'état `fermeture_out` en commandant la fermeture de la barrière, et en indiquant qu'une voiture est sortie.

Nous allons considérer ici une machine après synthèse avec codage *onehot* :

États symboliques	Idle	Ouverture_in	Fermeture_in	Verif_ticket	Ouverture_out	Fermeture_out
États après synthèse	State_0	State_1	State_2	State_3	State_4	State_5

Dans ce cas d'étude, on va reprendre l'assertion P_2 :

*(P_2) **always** (ticket_insere → **next !** (**not** ticketOK → **not** barriere_out_ouverte))*

Cette assertion surveille le fonctionnement du circuit quand le ticket inséré n'est pas valide. Dans ce cas la barrière de sortie ne doit pas s'ouvrir.

IV.1.1. Identification des conditions d'activation

Dans le cas de l'assertion P_2, on trouve la forme d'implication déjà présente, il suffit donc d'isoler les opérandes qui se situent dans la partie gauche. Pour l'assertion P2 les conditions à prendre en considération pour l'activer sont : le signal *ticket_insere* à vrai, et l'entrée *ticketOK* à faux un cycle après. La condition sur l'entrée ne nécessite pas l'utilisation de l'algorithme pour générer la valeur '0', mais l'algorithme va être appliqué pour la contrainte sur la sortie *ticket_insere (t) = vrai*, et l'algorithme va démarrer à partir de ce signal.

IV.1.2. Construction d'arbre de solutions et création de l'ensemble des solutions potentielles

L'algorithme démarre à partir de la sortie *ticket_insere* pour faire remonter la contrainte portée sur ce signal jusqu'aux entrées primaires et états internes du circuit. Un arbre de solutions est généré pour rassembler toutes les possibilités qui puissent satisfaire à la contrainte *ticket_insere (t) = vrai*. Une fois l'arbre généré on extrait les solutions, par

exemple ici avec la variante basée sur le BDD. Les fonctions d'extraction produisent une liste qui contient 24 solutions (Figure IV.2).

insertionticket (t - 0) = true reset (t - 1) = true demandeentree (t - 0) = false

insertionticket (t - 0) = true reset (t - 1) = true placesdispo (t - 0) = false

insertionticket (t - 0) = true reg_state_0 (t - 1) = true demandeentree (t - 0) = false demandeentree (t - 1) = false insertionticket (t - 1) = false

insertionticket (t - 0) = true reg_state_0 (t - 3) = true demandeentree (t - 3) = true placesdispo (t - 3) = true timeout1 (t - 2) = true demandeentree (t - 0) = false reset (t - 3) = false reset (t - 2) = false

insertionticket (t - 0) = true ticketok (t - 3) = true reg_state_3 (t - 3) = true timeout2 (t - 2) = true demandeentree (t - 0) = false reset (t - 3) = false reset (t - 2) = false

⋮

insertionticket (t - 0) = true reg_state_4 (t - 3) = true senseurout (t - 2) = true placesdispo (t - 0) = false reset (t - 3) = false reset (t - 2) = false timeout2 (t - 3) = false senseurout (t - 3) = false

Figure IV.2 : Ensemble des solutions généré pour la contrainte ticket_insere (t) = vrai

IV.1.3. Sélection des solutions

Dans cet exemple, l'entrée reset est un signal actif haut. On préfère donc écarter les solutions qui exigent que l'entrée reset soit à vrai, par exemple les deux premières :

$$\text{Insertion\_ticket (t - 0)} = \text{true reset (t - 1)} = \text{true demande\_entree (t - 0)} = \text{false}$$
$$\text{Insertion\_ticket (t - 0)} = \text{true reset (t - 1)} = \text{true place\_disponible (t - 0)} = \text{false}$$

Une deuxième sélection produit une deuxième liste de solutions qui ne rassemble que les solutions ayant des entrées primaires. Puis, pour chaque solution il faut s'assurer que les conditions portant sur les états internes sont antérieures aux contraintes à appliquer sur les entrées. En effectuant ces sélections, il reste finalement 22 solutions (la réduction est minimale dans cet exemple car la majorité des solutions respectaient les critères voulus). On peut enfin utiliser les résultats obtenus du calcul des probabilités d'atteindre les états du circuit (Figure III.17).

Exemple de solution choisie. On va prendre par exemple la solution S0 de la liste finale des solutions sélectionnées respectant les critères cités ci-dessus.

Solution S0 :
state_0 (t – 1) = vrai ; place_disponible (t – 1) = faux ; insertion_ticket (t – 1) = faux ;
place_disponible (t) = faux ; insertion_ticket (t) = vrai ;

Intuitivement, cette solution correspond à passer, à partir de l'état idle, dans la partie gauche de la FSM (place_disponible à '0' et insertion_ticket à '1'), qui met la sortie ticket_inséré à '1'.

On rappelle qu'il faut ticket_insere (t) = vrai et ticketOK (t +1) = faux. La production de la première contrainte est garantie par l'application de la solution S0. Alors on va globalement utiliser la solution S1 :

Solution S1 :
state_0 (t – 1) = vrai ; place_disponible (t – 1) = faux ; insertion_ticket (t – 1) = faux ; place_disponible (t) = faux ; insertion_ticket (t) = vrai ; tickeOK (t + 1) = faux ;

IV.1.4. Mise en œuvre de la solution sélectionnée et simulation

Opérons une transformation de la solution sélectionnée sous forme de spécification logique.

Spécification logique SPC1 :
Always (state_0 → (not place_disponible and not insertion_ticket) and next[1] ! (not place_disponible and insertion_ticket) and next[2] ! (not ticketOK);

On a d'abord effectué une simulation en utilisant un process qui génère des séquences de test sur les entrées du circuit d'une manière pseudo-aléatoire. Les résultats de cette simulation vont nous servir par la suite comme référence pour pouvoir comparer les résultats obtenus par chaque méthode d'intégration de générateur, et ainsi prouver l'augmentation du taux de couverture d'assertion. On va considérer :

Temps de simulation global = 2 ms
Temps cycle d'horloge = 100 ns

Simulation pseudo-aléatoire. On effectue une première simulation en utilisant un process qui génère des vecteurs d'une façon pseudo-aléatoire, Figure IV.3.

```
Pseudo-aléatoire: process
 variable seed1, seed2: positive := 200;
 variable rand: real;
 variable int_rand: integer;
 variable stim: std_logic_vector(7 downto 0);

 begin
  wait until ck'event and ck='0';
  if (cpt mod 5 = 0) then
    UNIFORM(seed1, seed2, rand);
    int_rand := INTEGER(TRUNC(rand*256.0));
    stim := std_logic_vector(to_unsigned(int_rand, stim'LENGTH));
    demande_entree <= stim(0);
    senseur_in <= stim(1);
    insertion_ticket <= stim(2);
    ticket_ok <= stim(3);
    senseur_out <= stim(4);
    place_disponible <= stim(5);
  end if;
 end process  Processus de génération de séquences de test;
```

Figure IV.3 : Process Pseudo-aléatoire

Ce process crée un entier aléatoire à partir de la fonction UNIFORM. Cet entier est converti en une variable binaire stim dont la taille est fixée avec LENGTH. Chaque bit de la variable stim est utilisé pour produire une valeur sur une entrée du circuit.

En effectuant une simulation purement pseudo-aléatoire, on a obtenu un premier résultat de couverture de l'assertion P2 : **elle était activée 2621 fois**.

Simulation avec générateur dans le process. On reprend la spécification logique **SPC1**, et on crée un process qui permet de surveiller la condition sur le registre state_0 au cours de la simulation ; une fois satisfaite, le processus applique la séquence **{0 ; 0}** sur l'entrée **place_disponible**, et **{0 ; 1}** sur l'entrée **insertion_ticket**, et **{- ; - ; 0}** sur l'entrée **ticketOK**. Sinon, il applique des séquences pseudo-aléatoires sur les entrées du circuit.

```
Processus de génération de séquences de test: process
variable seed1, seed2: positive := 200;
variable rand: real;
variable int_rand: integer;
variable stim: std_logic_vector(7 downto 0);
begin
 wait until ck'event and ck='0';

 if (state_0 = '1') then                                   -- Générateur des vecteurs de test déterminés
   place_disponible <= '0';
   insertion_ticket <= '0';
   wait until ck'event and ck='0';
   place_disponible <= '0';
   insertion_ticket <= '1';
   wait until ck'event and ck='0';
   ticket_ok <= '0';

 elsif (cpt mod 5 = 0) then

   UNIFORM(seed1, seed2, rand);                            -- Génération des séquences pseudo-aléatoires sur les entrées primaires
   int_rand := INTEGER(TRUNC(rand*256.0));
   stim := std_logic_vector(to_unsigned(int_rand, stim'LENGTH));
   demande_entree <= stim(0);
   senseur_in <= stim(1);
   insertion_ticket <= stim(2);
   ticket_ok <= stim(3);
   senseur_out <= stim(4);
   place_disponible <= stim(5);

 end if;
end process  Processus de génération de séquences de test;
```

Figure IV.4 : Processus de génération de séquences de test

Simulation et résultat : On a pu remarquer que la production la valeur '1' sur la sortie ticket_insere a augmenté.

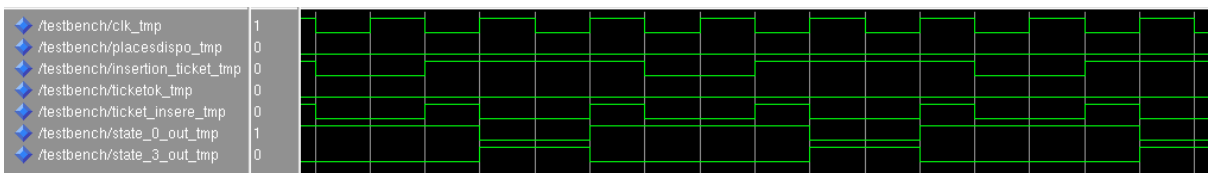


Figure IV.5 : Chronogramme de simulation du circuit avec le process intégrant le générateur

La Figure IV.5 nous donne un aperçu du comportement du circuit pendant la simulation en utilisant le nouveau process. En partant de l'état d'initialisation, et en forçant l'entrée insertion_ticket à '1' et place_disponible à '0' on génère la valeur '1' on se place dans l'état d'ouverture_in (state_3), cependant, la valeur '0' sur ticketOK remet la FSM à son état d'initialisation. On reste principalement entre ces deux états la majorité du temps de simulation, ce qui explique le fait que le circuit revient régulièrement à l'état state_0, et par conséquent, les nombreuses productions de la valeur '1' sur la sortie ticket_insere, ce qui se reflète sur l'augmentation considérable du nombre d'activations de l'assertion P2.

On a pu obtenir le résultat de taux de couverture d'assertion suivant : **l'assertion était activée 4545 fois**.

Méthode à base de générateur de vecteurs de test. On reprend notre spécification `SPC1`, qui est transformée en un générateur étendu à l'aide de *SyntHorus*. Ce générateur est connecté au circuit comme le montre la Figure IV.6.

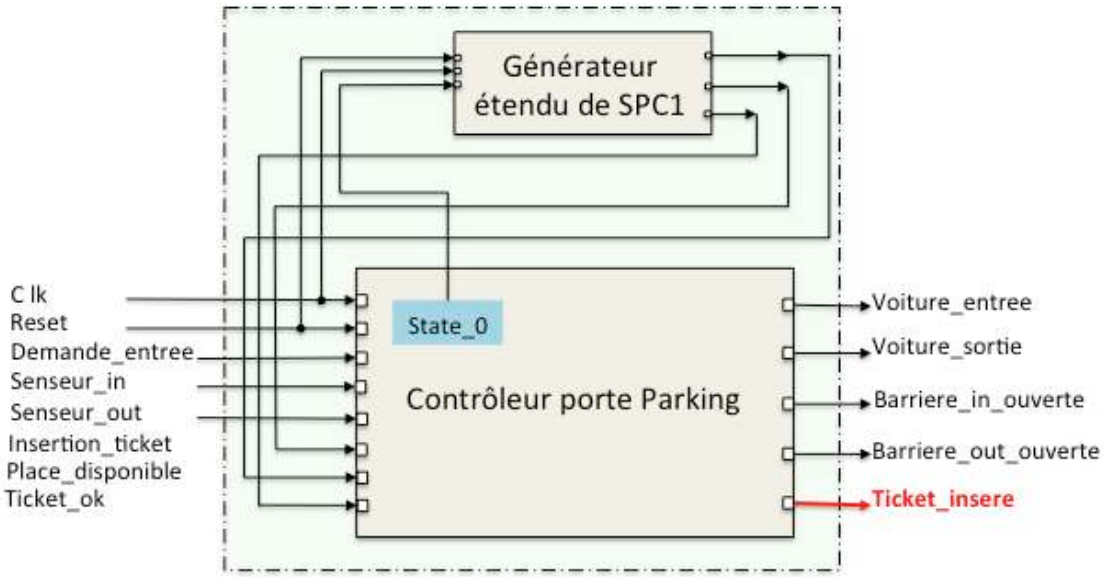


Figure IV.6 : Connexion du générateur étendu de SPC1 avec le circuit contrôleur porte de parking

Ici la solution est transformée en un générateur matériel, connecté au circuit via son état interne state_0 et ses entrées primaires : insertion_ticket, place_diponible et ticketOK. Le générateur surveille le comportement de state_0 pendant la simulation. Il est constitué de deux parties. À chaque fois une seule partie est activée suivant state_0 : si state_0 est à '1', des séquences de test déterminées sont envoyées aux entrées insertion_ticket, place_diponible et ticketOK. Sinon, c'est la partie LSFR qui génère des séquences pseudo-aléatoires qui est activée. Ce module (Figure IV.7) correspond à un générateur qui prend comme entrées les conditions d'activation de la spécification et génère des séquences envoyées vers les entrées du circuit pendant la simulation. Le générateur étendu `G_SPC1` est connecté au circuit comme le montre la Figure IV.6 et un nouveau circuit top est construit.

Figure IV.7 : Interface du générateur étendu G_SPC1 correspondant à la spécification SPC1

Simulation et résultat :

Le taux de la vérification a remarquablement augmenté pour l'assertion P2. **L'assertion était activée 5448 fois**. Cette augmentation est probablement due au bloc LSFR du générateur étendu, qui recalcule les valeurs envoyées aux entrées à chaque cycle, donc la variation sur les valeurs des entrées est plus fréquente.

Les résultats obtenus montrent que l'utilisation de l'approche développée présente des améliorations de la qualité de la vérification des assertions pendant la simulation.

IV.2. Circuit ADC

IV.2.1. Présentation du circuit

Considérons le cas du circuit présenté dans le tutoriel PSL de Dolphin Integration [58]. Ce circuit transmet des données venant de convertisseurs analogiques/ numériques (ADC) doubles. Chaque convertisseur produit deux mots de 12 bits. Ces quatre résultats sont transmis via des ports séries. L'interface de ce circuit est présentée dans la Figure IV.8.

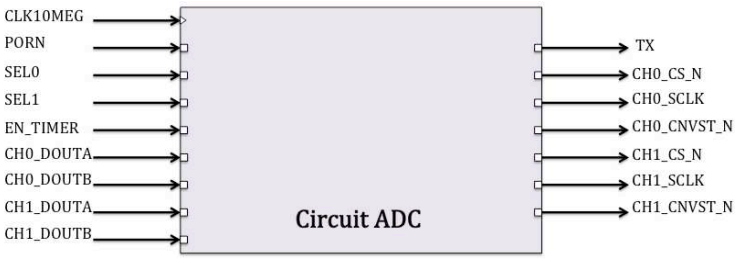


Figure IV.8 : Interface du circuit ADC

Ce circuit prend comme entrées :

- `PORN` : signal d'initialisation du système.
- `CLK10MEG` : signal d'horloge du système.
- `SEL0 et SEL1` : bits 0 et 1 du signal de définition du facteur de division de l'horloge CLK10MEG.
- `EN_TIMER` : signal déclencheur de la transmission série des données converties en des mots de 12 bits.
- `CH0_DOUTA` : donnée d'entrée analogique du premier convertisseur sur le premier

canal (canal A).
- CH0_DOUTB : donnée d'entrée analogique du premier convertisseur sur le deuxième canal (canal B).
- CH1_DOUTA : donnée d'entrée analogique du deuxième convertisseur sur le premier canal.
- CH1_DOUTB : donnée d'entrée analogique du deuxième convertisseur sur le deuxième canal.

Les sorties principales de ce circuit sont les suivantes :
- TX : sortie série du circuit.
- CH0_SCLK : signal qui indique la fréquence de transfert des données du premier canal.
- CH0_CS_N : signal qui passe à la valeur '0' pour indiquer qu'il y un transfert en cours des données du premier canal.
- CH1_SCLK : signal qui indique la fréquence de transfert des données du deuxième canal.
- CH1_CS_N : signal qui passe à la valeur '0' pour indiquer qu'il y un transfert en cours des données du deuxième canal.

Et comme la montre la Figure IV.9, ce circuit est composé de trois composants principaux :
- Instance *Reset* : pour réinitialiser la resynchronisation.
- Instance *Clock* : pour générer le signal d'horloge du circuit.
- Instance *Core* : c'est le composant principal de ce circuit. Il se charge de convertir des données analogiques et de les transformer en des mots binaires de 12 bits, qui seront transmis par la suite via des ports série.

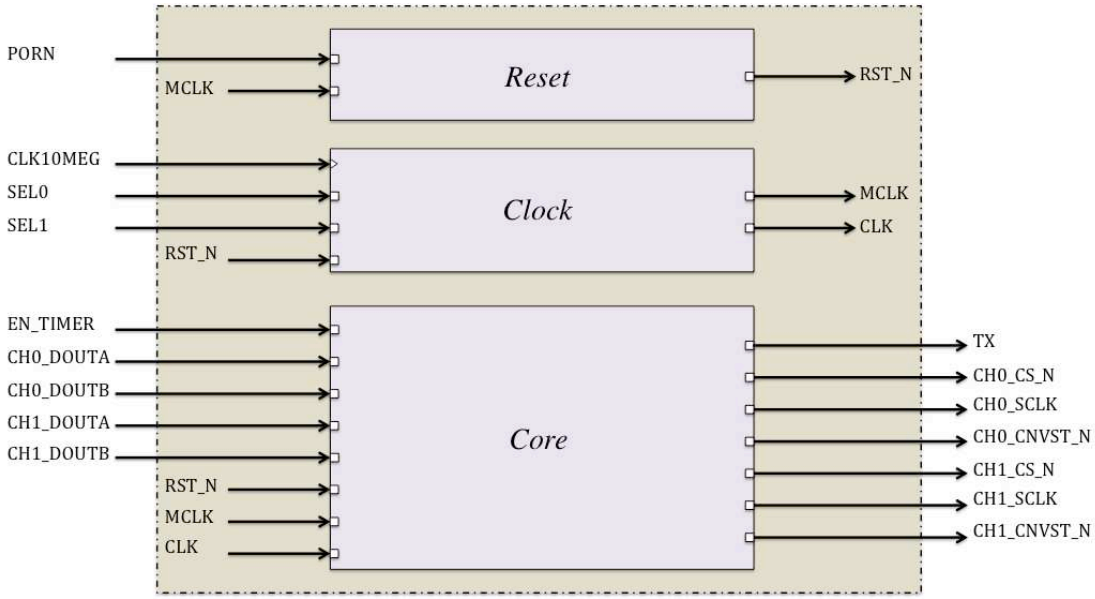


Figure IV.9 : Structure générale du circuit

L'instance *Core* est commandée par deux signaux d'horloge :

- **CLK10MEG** ou **MCLK**: correspond au signal d'horloge fourni par l'utilisateur.
- **CLK** : correspond à une division du signal d'horloge **CLK10MEG** à l'aide du système de génération du signal d'horloge (instance *Clock*). Ce système prend en entrée le signal **SEL** de deux bits, qui désigne un facteur de division du signal d'horloge **CLK10MEG**. Ce facteur de division est utilisé pour définir le débit binaire de la sortie en série **TX** :
  - Si SEL0 = '0' et SEL1 = '0', le facteur est 1
  - Si SEL0 = '1' et SEL1 = '0', le facteur est 2
  - Si SEL0 = '0' et SEL1 = '1', le facteur est 3
  - Si SEL0 = '1' et SEL1 = '1', le facteur est 4

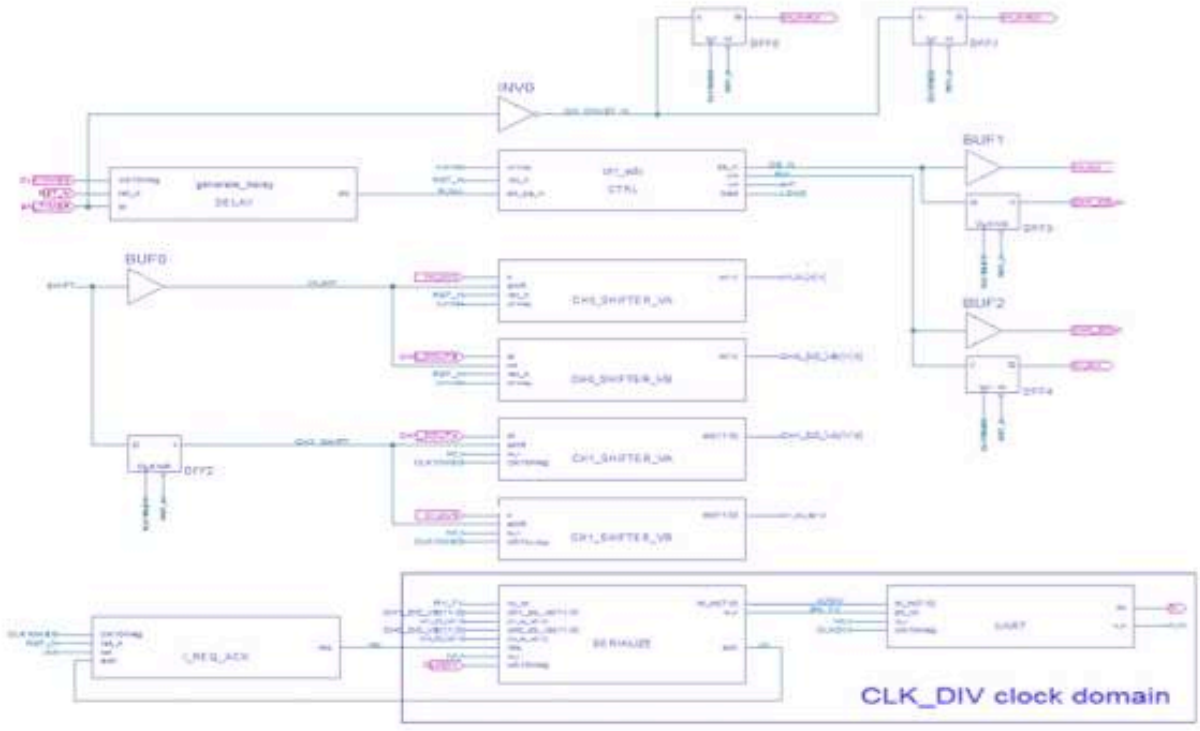


Figure IV.10 : Structure de l'instance Core du circuit Transmetteur

Comme le montre la Figure IV.10, le composant *Core* est à son tour constitué de plusieurs composants :

- Instance *Generate_delay* : produit la valeur vrai sur l'entrée **EN_CS_N** de l'instance *ctrl_adc* 1 ms après la production de la valeur vrai sur l'entrée **EN_TIMER** du circuit.
- Instance *Ctrl_adc* : commande les signaux de début et fin de transfert des données de 12 bits.
- Instance *CH0_SHIFTER_VA* : convertit la donnée présente sur l'entrée CH0_DOUTA en un mot de 12 bits qui sera présent sur sa sortie CH0_DO_VA.
- Instance *CH0_SHIFTER_VB* : convertit la donnée présente sur l'entrée CH0_DOUTB en un mot de 12 bits qui sera présent sur sa sortie CH0_DO_VB.
- Instance *CH1_SHIFTER_VA* : convertit la donnée présente sur l'entrée **CH1_DOUTA** en un mot de 12 bits qui sera présent sur sa sortie **CH1_DO_VA**.
- Instance *CH1_SHIFTER_VB* : convertit la donnée présente sur l'entrée **CH1_DOUTB** en

un mot de 12 bits qui sera présent sur sa sortie **CH1_DO_VB**.

- Instance *I_REQ_ACK* : ce module envoie la valeur '1' sur sa sortie **REQ** quand il détecte la fin de transfert (**LOAD** à '1') et quand le composant *SERIALIZE* envoie la valeur '1' sur l'entrée **ACK** pour indiquer qu'il est libre et peut démarrer le transfert.
- Instance *SERIALIZE* : transforme les données reçues des convertisseurs en des mots de 8 bits pour les transférer.
- Instance *UART* : ce composant permet de sérialiser les données reçues du composant *SERIALIZE* et les envoyer via sa sortie **TX**.

Protocole. Une acquisition est demandée quand le signal **EN_TIMER** est à vrai. Les quatre données de 12 bits sont récupérées des interfaces du convertisseur. Une donnée est disponible quand le signal **LOAD** de l'instance *Ctrl_adc* passe à vrai. Ce signal est reçu par l'instance *I_REQ_ACK*, qui permet la génération de la valeur vrai sur le signal **REQ**. Le signal **REQ** à son tour permet de commander l'instance *SERIALIZE* pour contrôler la transmission à travers l'interface série *UART*. L'instance *SERIALIZE* commence effectivement le transfert quand le signal **ACK** passe à vrai. Le passage de **EN_TIMER** à la valeur vrai permet de déclencher le protocole complet de la transmission des données de 12 bits via la sortie série **TX**. Il est indispensable de fournir suffisamment de temps entre deux **EN_TIMER = '1'** pour ne pas corrompre le transfert. Pendant le transfert des données la sortie **CS_N** de l'instance *Ctrl_adc* (cette sortie est équivalente aux sorties du circuit transmetteur : **CH0_CS_N** et **CH1_CS_N**) se met à la valeur faux pour indiquer que la transmission des données est en cours.

IV.2.2. Définition des assertions de surveillance des fonctionnalités

Ce circuit doit garantir deux exigences principales, qu'on peut formaliser en assertions PSL :

- Les quatre données de 12 bits doivent être reçues en respectant un protocole SPI.
- Il faut suffisamment de temps entre deux acquisitions pour permettre une transmission série complète des données.

On va s'intéresser aux assertions qui surveillent le protocole SPI. Ces assertions s'appliquent sur les deux canaux ADC du circuit. On considère les assertions suivantes :

```
Assertion (A1) :
never ( !CH1_SCLK && CH1_CS_N);
```

L'assertion **(A1)** surveille le comportement du signal **CH1_SCLK**. Ce signal indique la fréquence de transfert des données de 12 bits, autrement, il est toujours maintenu à '1'. Il ne faut jamais avoir le signal **CH1_SCLK** à faux quand il n'y a pas de transfert, ce qui symbolisé par le signal **CH1_CS_N** à '1'. Cette assertion interdit la production en même temps du signal **CH1_SCLK** à '0' et **CH1_CS_N** à '1'.

```
Assertion (A2) :
always ({CH1_CS_N ; !CH1_CS_N} |-> {!CH1_CS_N[*24];CH1_CS_N});
```

L'assertion **(A2)** surveille la durée du passage à '0' du signal **CH1_CS_N** le temps de la transmission des données de 12 bits du premier canal. Un front descendant sur ce signal indique le démarrage du transfert, qui dure le temps équivalent à 24 cycles d'horloge avant le passage de **CH1_CS_N** à '1'.

```
Assertion (A3) :
always ({CH1_CS_N ; !CH1_CS_N } |-> {CH1_SCLK[*2];{!CH1_SCLK;CH1_SCLK}[*11]});
```

L'assertion **(A3)** surveille le comportement du signal **CH1_SCLK** une fois que le transfert est démarré (passage du signal **CH1_CS_N** à '0').

```
Assertion (A4) :
always ({CH1_CS_N ; !CH1_CS_N } |-> {{{CH1_DOUTA[*2]} | {!CH1_DOUTA[*2]}}[*12]});

Assertion (A5) :
always ({CH1_CS_N ; !CH1_CS_N } |-> {{{CH1_DOUTB[*2]} | {!CH1_DOUTB[*2]}}[*12]});
```

Les assertions **(A4)** et **(A5)** surveillent la stabilité des données sur les entrées analogiques **CH1_DOUTA** et **CH1_DOUTB** quand le transfert commence (passage de **CH1_CS_N** à '0').

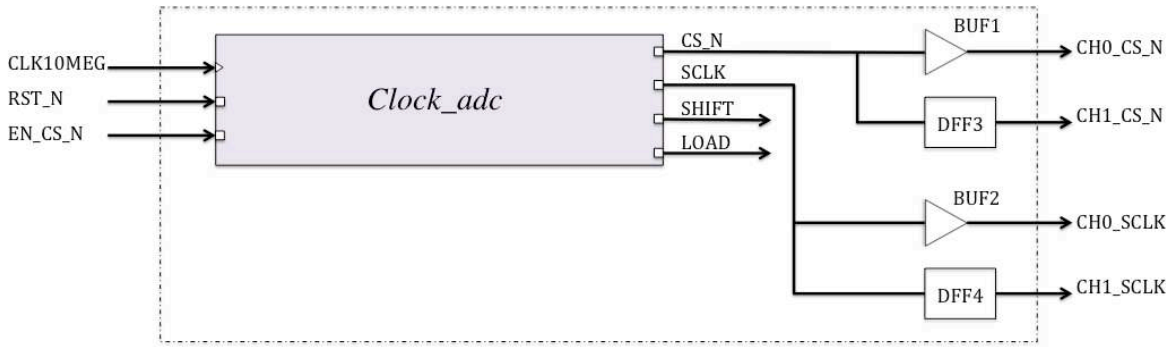


Figure IV.11 : Interface du composant Ctrl_adc et connexion avec les sorties primaires CH0_CS_N, CH1_CS_N, CH0_SCLK et CH1_SCLK

Ces assertions surveillent des parties de fonctionnalités que le circuit doit assurer, spécialement dans la partie du transfert des données de 12 bits et suivant le protocole SPI. Si on reprend le cas des assertions **(A2), (A3), (A4), et (A5)**, on remarque qu'elles sont déjà sous forme d'implications SERE. Les quatre assertions ont la même condition de déclenchement. Le comportement de ces assertions est conditionné par un front descendant sur le signal **CH1_CS_N**, qui est une sortie reliée à la sortie **CS_N** de l'instance *Ctrl_adc*

(Figure IV.11), indiquant le début de la transmission des données. Un front descendant sur le signal `CH1_CS_N` donne deux contraintes à satisfaire : la première `CH1_CS_N (t – 1) = vrai`, et la deuxième `CH1_CS_N (t) = faux`. Dans ce cas les solutions retenues seront des combinaisons des solutions retenues pour chaque contrainte.

IV.2.3. Construction d'arbres de solutions et sélection de solutions

Pour activer les assertions pendant la simulation, il faut s'assurer que des fronts descendants de produisent suffisamment souvent sur le signal `CH1_CS_N`.

Production d'un front descendant avec une solution combinée. L'algorithme va générer un arbre de solutions pour satisfaire chaque contrainte. On doit d'abord faire remonter la contrainte `CH1_CS_N= vrai` jusqu'aux entrées primaires et des états internes du circuit. Ensuite on considère la deuxième contrainte `CH1_CS_N= faux` et l'algorithme génère un arbre de solutions correspondant. À partir des deux listes de solutions sélectionnées, on sera capables de faire des combinaisons des solutions pour pouvoir générer une liste finale qui rassemble les solutions capables de produire un front descendant sur le signal `CH1_CS_N` à l'instant `(t)`.

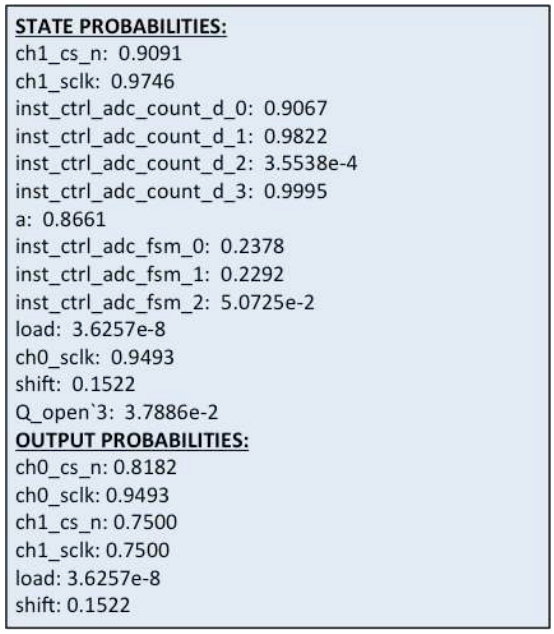


Figure IV.12 : Probabilités des états internes et sorties du circuit ADC

On commence par la contrainte `CH1_CS_N= vrai`. Après extraction des solutions, on a obtenu une première liste qui contient **33 solutions** potentielles. La Figure IV.12 montre les résultats de calculs de probabilités des états du circuit. Cette liste de calculs nous permet de raffiner notre sélection, et choisir les solutions qui contiennent les conditions les plus probables à produire pendant la simulation du circuit. De la liste sélectionnée, on retient la solution S2 pour produire la contrainte `CH1_CS_N= vrai`.

Solution S2 :
Reg_fsm_2 (t - 3) = faux ; Reg_fsm_0 (t - 3) = faux ; Reg_fsm_1 (t - 3) = faux ;
EN_CS_N (t - 3) = faux ;

Les conditions à satisfaire sur les registres internes du circuit portent sur (**Reg_fsm_0**, **Reg_fsm_1** et **Reg_fsm_2**). Lorsque ces registres ont la valeur faux, une valeur vrai est envoyée à l'entrée **EN_CS_N** ; ce qui force la production de la valeur vrai sur la sortie **CH1_CS_N** trois cycles après.

On doit procéder de la même manière pour produire la deuxième contrainte **CH1_CS_N= faux**. L'arbre de solutions est généré pour satisfaire cette contrainte, une liste de **99 solutions** est produite, et suivant les critères de choix une première sélection est faite. Utilisant les résultats des calculs de probabilités d'atteindre les états de la Figure IV.12, on obtient un ensemble raffiné. On va retenir la solution **S3** pour pouvoir produire la contrainte **CH1_CS_N= faux**.

Solution S3 :
Reg_fsm_1 (t - 2) = faux ; Reg_fsm_2 (t - 2) = faux ; EN_CS_N (t - 2) = vrai ;

Ayant les conditions sur **Reg_fsm_1** et **Reg_fsm_2** satisfaites, une valeur vrai est appliquée sur l'entrée **EN_CS_N** ; cela va permettre la production de la contrainte **CH1_CS_N= faux**.

Ces solutions vont nous servir pour construire des séquences de test capables de produire des fronts descendants sur le signal **CH1_CS_N**. Si on reprend le cas des solutions choisies pour produire successivement les contraintes **CH1_CS_N (t – 1) = vrai** et **CH1_CS_N (t) = faux**, on obtient une solution S4 combinée qui satisfait la contrainte globale à l'instant **(t)**.

Solution S4 :
Reg_fsm_2 (t - 4) = faux ; Reg_fsm_0 (t - 4) = faux ; Reg_fsm_1 (t - 4) = faux ;
EN_CS_N (t - 4) = faux ; Reg_fsm_1 (t - 2) = faux ; Reg_fsm_2 (t - 2) = faux ;
EN_CS_N (t - 2) = vrai ;

Prise en compte du contexte. Ce cas d'étude suit un protocole précis. Comme le montre la Figure IV.13, pendant une simulation classique le signal de sortie est majoritairement à '1' et il ne passe à la valeur '0' que pour indiquer le début du transfert des données et reste à '0' jusqu'à la fin du transfert avant de passer de nouveau à '1'. Et suivant ce protocole de fonctionnement, nous allons donc seulement nous concentrer sur la production de la valeur '0' sur la sortie **CH1_CS_N** pour créer un front descendant.

Il suffit dans ce cas de générer les solutions pour la contrainte **CH1_CS_N = faux**, par exemple la solution **S3**. Cette solution va forcer le signal **CH1_CS_N** à '0'.

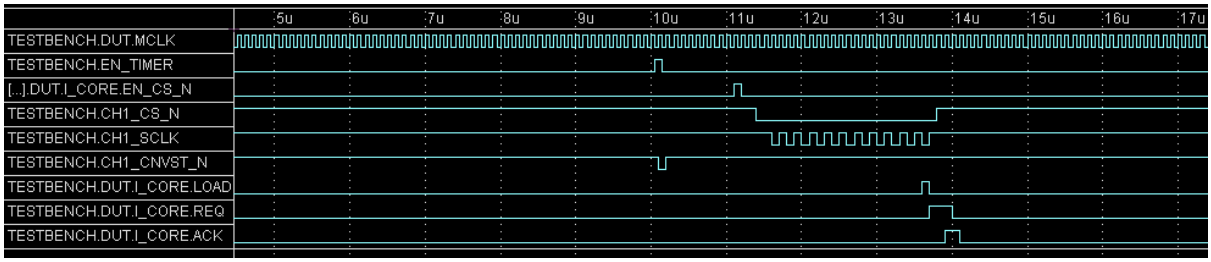


Figure IV.13 : Simulation du circuit ADC pour montrer le protocole suivi

IV.2.4. Construction des générateurs et simulation

A tout, une simulation classique s'impose, nous allons simuler le circuit en appliquant des vecteurs pseudo-aléatoires pour obtenir les premiers résultats des taux de couverture des assertions qu'on surveille. Pour toutes les simulations effectuées dans cette section, on va considérer :
Temps de simulation = 2 ms
Temps cycle d'horloge = 100 ns

Simulation pseudo-aléatoire. Dans cette première partie, on cherche à avoir un aperçu sur le comportement du circuit pendant la simulation avec des séquences pseudo-aléatoirement générées, et aussi obtenir des premiers résultats de taux de couverture des assertions A2, A3, A4 et A5. Ces résultats vont nous servir par la suite comme des références sur quoi on va se baser pour effectuer des comparaisons. Pour pouvoir générer du pseudo-aléatoire, on a utilisé un process qui fait appel à la fonction UNIFORM (Figure IV.14).

```
vecteurs_de_test: process
variable seed1, seed2 : positive := 200;
variable rand : real;
variable int_rand : integer;
variable stim : std_logic_vector(7 downto 0);

begin
  wait until clk10meg'event and clk10meg='0';
   UNIFORM(seed1, seed2, rand);
   int_rand := INTEGER(TRUNC(rand*32.0));
   stim := std_logic_vector(to_unsigned(int_rand, stim'LENGTH));
   en_cs_n <= stim(4);
   wait until falling_edge(CLK10MEG);
   en_cs_n <= '0';
end process vecteurs_de_test;
```

Figure IV.14 : Process Pseudo-aléatoire pour circuit ADC

Simulation et résultat :
En effectuant la simulation avec des vecteurs pseudo-aléatoires on obtient des évolutions des signaux de commande de ADC qui sont présentés dans la Figure IV.15. Une valeur '1' sur le signal **EN_CS_N** donne une commande de démarrage de transfert s'il n'y a pas de transfert

en cours. Un front descendant est produit sur la sortie **CH1_CS_N** pour signaler le début de transfert et reste à '0' durant tout le transfert. Pendant le transfert, la sortie **CH1_SCLK** devient comme une pseudo-horloge pour indiquer la fréquence de transfert des données sur les convertisseurs. Lorsque le transfert est terminé, la sortie **LOAD** passe à la valeur '1' pour l'indiquer, et la sortie **CH1_CS_N** bascule à la valeur '1' et attend une autre acquisition pour démarrer un autre transfert.

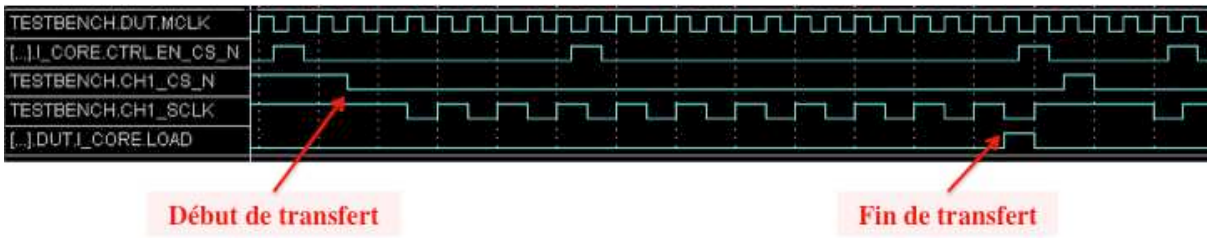


Figure IV.15 : Extrait de simulation du circuit ADC avec vecteurs de test pseudo-aléatoires

On a obtenu des premiers résultats de couverture :

L'assertion A2 était vérifiée 709 fois.
L'assertion A3 était vérifiée 709 fois.
L'assertion A4 était vérifiée 709 fois.
L'assertion A5 était vérifiée 709 fois.

Simulation en utilisant des process avec générateur. Dans cette partie on va reprendre les deux méthodes de choix de solutions à appliquer durant la simulation, et on va transformer les solutions choisies en des processus qu'on va intégrer dans le testbench. On va effectuer les simulations pour chaque solution proposée pour pouvoir comparer.

1. Générateur qui produit un front descendant sur CH1_CS_N : La solution prise en compte dans cette première simulation est celle qui combine les deux solutions qui produisent la valeur '1' suivie de la valeur '0' sur **CH1_CS_N**. La combinaison des deux solutions nous donne la solution **S4**.
Le nouveau process est lié au circuit via les états internes (**Reg_fsm_0**, **Reg_fsm_1** et **Reg_fsm_2**). Si les conditions sur les registres sont satisfaites, c'est le premier bloc du process qui est activé, et des séquences de test déterminées sont envoyées sur l'entrée **EN_CS_N** pour produire le front descendant sur **CH1_CS_N**. Sinon, le deuxième bloc du process est activé, et des séquences pseudo-aléatoires sont envoyées à l'entrée du circuit.

Simulation et résultat :
En effectuant la simulation, on a obtenu les nouveaux résultats du taux d'activation des assertions surveillées suivants :

L'assertion A2 était vérifiée 722 fois.
L'assertion A3 était vérifiée 722 fois.
L'assertion A4 était vérifiée 722 fois.
L'assertion A5 était vérifiée 722 fois.

Les résultats obtenus montrent une légère augmentation par rapport à la simulation avec des vecteurs pseudo-aléatoires. Cela est dû principalement au fait que pendant le transfert toute valeur '1' sur **EN_CS_N** est ignorée, ce qui limite le nombre de productions de fronts descendants sur CN1_CS_N. Les cas de passages à '1' de EN_CS_N sont pris en compte quand il n'y a pas de transfert en cours.

2. Générateur qui produit une valeur '0' sur CH1_CS_N : la deuxième expérimentation est faite avec les mêmes conditions que la première, seul le générateur est changé pour correspondre à la solution S3.

Simulation et résultat :

On a obtenu les résultats suivants concernant les taux de couvertures des assertions observées :

```
L'assertion A2 était vérifiée 769 fois.
L'assertion A3 était vérifiée 769 fois.
L'assertion A4 était vérifiée 769 fois.
L'assertion A5 était vérifiée 769 fois.
```

Avec cette solution on a obtenu des résultats meilleurs qu'avec la solution qui cherche à produire un front descendant. En prenant en compte le protocole suivi par le circuit, on a réussi à améliorer encore plus le taux de couverture par rapport à la simulation pseudo-aléatoire, avec des solutions plus efficaces et moins complexes.

Simulation en utilisant des solutions transformées en générateurs étendus. Dans cette partie on va reprendre les deux approches, et produire des modules HDL qu'on va connecter avec le circuit et effectuer des simulations sur les nouveaux circuits top.

1. Générateur étendu qui produit un front descendant sur **CH1_CS_N**: On va reprendre la solution **S4**. Cette solution doit d'abord être transformée en une spécification logique pour devenir exploitable par *SyntHorus* :

```
Spécification SPC2 :
Always ((not Reg_fsm_0 and not Reg_fsm_1 and not Reg_fsm_2 and not busy) → (not EN_CS_N and next_a[1 :2] (busy)
and next[2] ((not Reg_fsm_1 and not Reg_fsm_2 and busy) → EN_CS_N) and next[3] ( not busy)));
```

La solution **S4** est transformée en spécification **SPC2**. On remarque que la condition de la partie gauche de la deuxième implication est une partie de la condition de la partie gauche de la première implication. Cela risque de créer des conflits pour le générateur. Puisque la spécification est testée dans chaque cycle, si elle est redéclenchée sur le cycle suivant on pourra avoir deux valeurs contradictoires pour le même signal **CH1_CS_N**. la Figure IV.16 montre un schéma simplifié du problème.

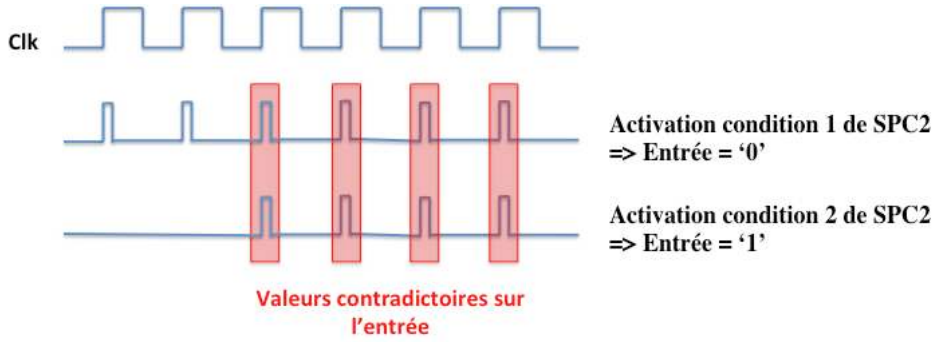


Figure IV.16 : Présence des mêmes conditions successivement dans la même spécification

Pour résoudre ce problème et pouvoir différencier la première condition de la deuxième, on a introduit une nouvelle variable **busy**. Cette variable est initialement à '0', une fois que la spécification est activée par la vérification de la première partie des conditions, cette variable est mise à '1' et est maintenue à '1' jusqu'à la fin de la vérification de la spécification. Cette spécification est utilisée dans *SyntHorus* pour générer le générateur étendu G_SPC2 représenté dans la Figure IV.17.

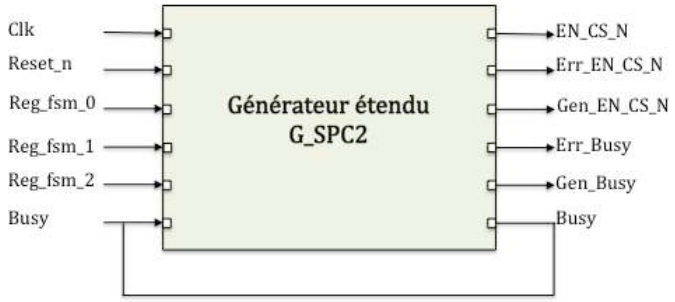


Figure IV.17 : Interface du générateur étendu correspondant à la spécification SPC2

Le générateur étendu de la Figure IV.17 prend comme entrées les signaux internes sur lesquels portent les conditions décrites par la spécification (**Reg_fsm_0**, **not Reg_fsm_1** et **Reg_fsm_2**), et a comme sortie le signal EN_CS_N qui sera connecté à l'entrée EN_CS_N du circuit. Ce module sera connecté de la manière montrée dans la Figure IV.18.

Simulation et résultat :

On a obtenu les résultats suivants de calcul de taux de couverture :

L'assertion A2 était vérifiée 774 fois.
L'assertion A3 était vérifiée 774 fois.
L'assertion A4 était vérifiée 774 fois.
L'assertion A5 était vérifiée 774 fois.

On remarque encore une augmentation de nombre de vérification des assertions observées, par rapport à la simulation pseudo-aléatoire et même en comparaison avec le process vu précédemment. Ici aussi, cette augmentation est probablement due au bloc LSFR du

générateur étendu, qui recalcule les valeurs envoyées aux entrées à chaque cycle, donc la variation sur les valeurs des entrées est plus fréquente.

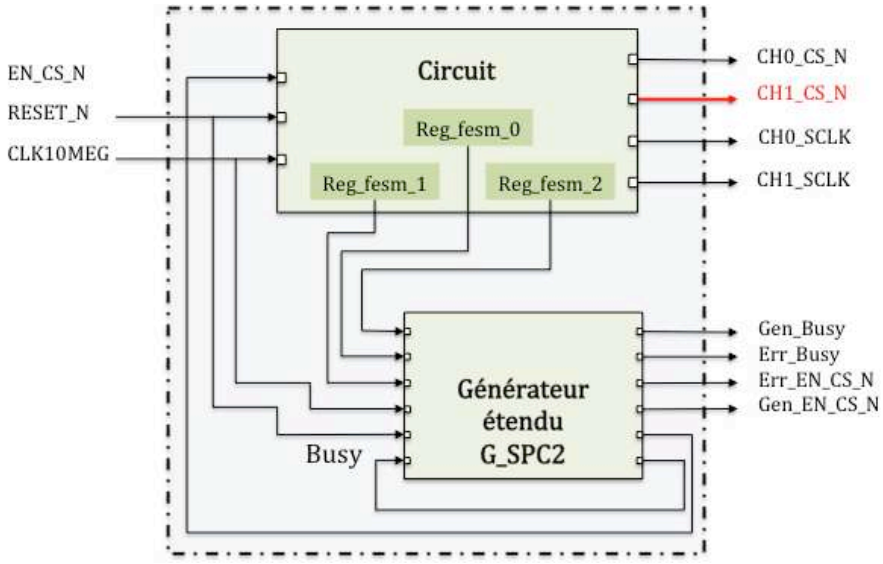


Figure IV.18 : Connexion du module G_SPC2 avec le circuit

2. Générateur étendu qui produit la valeur '0' sur **`CH1_CS_N`**: On reprend ici la deuxième possibilité de solution. On va reprendre la solution S3 et la transformer en une spécification SPC3 afin de pouvoir l'utiliser dans SyntHorus.

Spécification SPC3 :
Always ((not Reg_fsm_1 and not Reg_fsm_2) → EN_CS_N) ;

Cette spécification est transformée par la suite en un générateur étendu.

Figure IV.19 : Interface du générateur étendu G_SPC3 correspondant à la spécification SPC3

Simulation et résultat :

On connecte le générateur G_SPC3 au circuit via ses registres internes et son entrée, et on effectue la simulation avec le nouveau circuit top. On a obtenu les résultats de calcul de taux de couverture suivants :

L'assertion A2 était vérifiée 788 fois.
L'assertion A3 était vérifiée 788 fois.
L'assertion A4 était vérifiée 788 fois.
L'assertion A5 était vérifiée 788 fois.

Ces résultats nous montrent une amélioration de la qualité de la vérification. Aussi, se baser sur le protocole suivi pour le fonctionnement et l'inclure dans les critères de détermination des solutions améliore la vérification et aide à réduire la complexité des générateurs.

IV.3. Circuit HDLC

IV.3.1. Présentation du circuit

Généralités. Le cas d'étude traité dans cette section [59] est une IP contrôleur d'interface qui rassemble un module transmetteur et récepteur indépendants qui effectuent des transmissions et réceptions avec un format de trame spécifique, suivant un protocole synchrone série appelé HDLC (ISO/IEC 13239:2002). Ce protocole se situe dans la couche liaison du modèle OSI (Open Systems Interconnection). Cette couche s'occupe de la gestion du transfert des *trames* (paquets de bits).
Ce circuit met en œuvre des fonctionnalités du protocole HDLC tel que la sérialisation/dé-sérialisation, la génération du CRC, la transparence et la génération/détection de l'annulation (abort). Certains des protocoles de niveau supérieur du HDLC ne sont pas traités par l'IP, tels que l'adressage des séquences, la numérotation et le typages des trames. Ces tâches peuvent être effectuées d'une manière logicielle sur des processeurs qui ont ces IPs comme périphériques.

Les trames de données reconnues par le protocole de transmission et de réception HDLC sont organisées comme le montre la Figure IV.20.

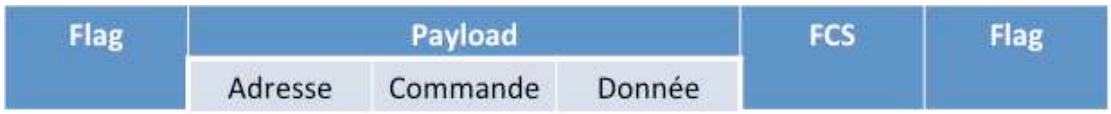


Figure IV.20 : Format général d'une trame de données

Une trame est constituée des champs suivants :

- Flag : c'est une suite de bits qui permettent à l'horloge du récepteur de se synchroniser sur celle de l'émetteur. Il est défini sous la forme d'un octet formé de 6 bits consécutifs à 1, préfixés et suffixés par un bit 0 (0x7E).
- Payload : elle comporte le champ d'adresse du destinataire à qui est envoyée la trame (8 bits), le champ de la commande qui est défini sur 8 ou 16 bits, et donne un numéro de commande qui permet de définir le type de trame, et le champ de la donnée à transmettre, de taille variable.
- FCS (Frame Check Sequence) : ce champ est ajouté à la fin de la trame pour détecter d'éventuelles erreurs de transmission. La séquence contenue dans ce champ correspond au CRC calculé sur le champ Payload.

Structure du HDLC. L'ensemble du système est composé de deux grandes parties : l'émetteur (TX) et le récepteur (RX). L'émetteur est dirigé par une horloge TxClk, toutefois

TxDataEn et TxDataEnOut désignent la fréquence de transmission des données en entrée et sortie. Même chose pour le récepteur, il a une horloge principale RxClk, mais il existe aussi RxDataEn et RxDataOut pour désigner les fréquences de transmission.

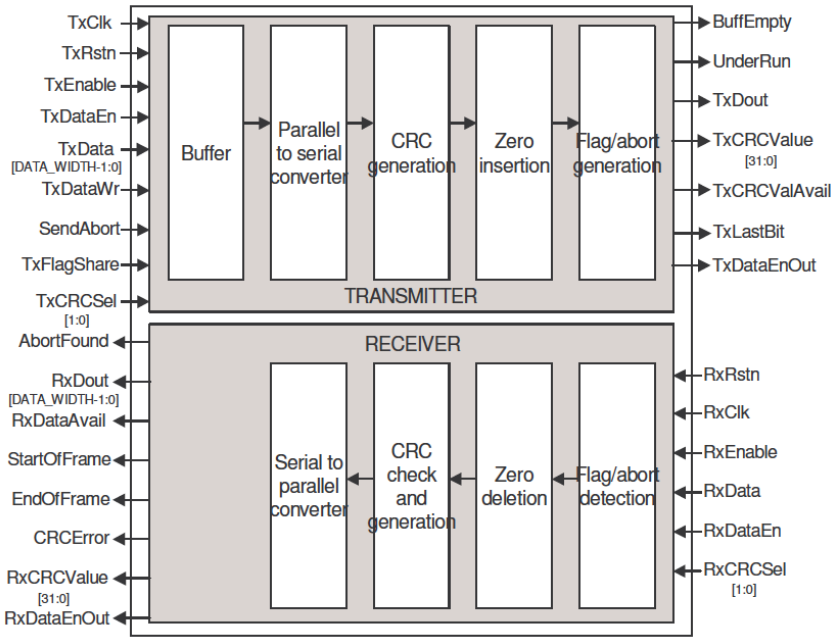


Figure IV.21 : Structure du HDLC

La Figure IV.21 montre la structure générale de l'émetteur et le récepteur du HDLC.

❖ L'émetteur a comme entrées principales :
TxClk : horloge principale de l'émetteur.
TxRstn : signal d'initialisation de l'émetteur.
TxData : bus entrée de données, reçues des processeurs extérieurs.
TxDataWr : signal qui active l'émetteur.
SendAbort : l'émetteur envoie une séquence d'annulation quand ce signal est activé.
TxFlagShare : signal qui donne des informations sur le partage des flags de début et de fin.
TxCRCSel : il permet de sélectionner la taille du CRC (0, 8, 16 ou 32 bits).

Il a comme sorties :
BuffEmpty : signal qui indique que l'octet suivant doit être écrit dans l'émetteur avant 7 cycles de TxDataEn, après que le signal ait été activé pour éviter un UnderRun.
UnderRun : ce signal indique qu'il n'y a plus d'information en entrée de l'émetteur.
TxDout : sortie série des données.
TxCRCValue : signal qui donne la valeur du CRC calculé sur la trame.
TxCRCValAvail : ce signal s'active quand la valeur TxCRCValue est valide.
TxLastBit : ce signal s'active pour signaler que le dernier bit du flag de la transmission est envoyé sur TxDout.
TxDataEnOut : signal actif quand une trame est transmise.

L'émetteur utilise 5 sous-composants :

- ✓ Buffer : composant qui permet le stockage temporaire des données, en attendant que le convertisseur parallèle/série soit disponible.
- ✓ Convertisseur parallèle/série : convertit les données en série pour pouvoir les envoyer via la sortie TxDout de l'émetteur.
- ✓ Générateur CRC : calcule le CRC en fonction des informations contenues dans la trame et le rajoute à la fin de la trame, avant d'envoyer le flag de fermeture.
- ✓ Instance pour l'insertion de Zero : cette instance à comme rôle principal de surveiller les données envoyées et éviter d'envoyer des séquences d'annulation (1111111) en insérant des zéro quand c'est nécessaire.
- ✓ Instance pour la génération flag/abort : il génère les flags d'ouverture et fermeture, et aussi en cas de besoin, les séquences d'annulation.

❖ Tout comme l'émetteur, le récepteur a comme entrées principales:
RxRstn : horloge principale du récepteur.
RxClk : signal d'initialisation du récepteur.
RxData : reçoit les données série.
RxCRCSel : permet la sélection de la taille du CRC (0, 8, 16 ou 32 bits).

Et il a comme sorties :
RxDout : bus de sortie de données du récepteur.
RxDataAvail : signal qui s'active quand la valeur sur le bus RxDout est valide.
StartOfFrame : signal qui indique le début de trame.
EndOfFrame : signal qui indique la fin de trame.
CRCError : signale que la trame reçue contient une ou plusieurs erreurs.
RxCRCValue : indique la valeur du CRC calculée sur la trame (32 bits).
RxDataEnOut : signal actif quand une trame est reçue.
AbortFound : indique que le récepteur a détecté une séquence d'annulation.

Le récepteur est constitué de 4 sous-composants :

- ✓ Convertisseur série/parallèle : convertit les données reçues en série en des données en parallèle.
- ✓ Instance pour la vérification et la génération du CRC : ce bloc calcule le CRC et vérifie s'il n'y a pas eu d'erreur de transmission. En cas d'erreur, le signal CRCError passe à '1'.
- ✓ Instance pour la détection de Zero : permet de détecter et supprimer les zéros insérés par l'émetteur pour éviter d'envoyer des séquences d'annulation.
- ✓ Instance pour la détection du flag/abort : détecte les flags d'ouverture et de fermeture des trames, ainsi que les séquences d'annulation (abort).

Description comportementale. En comportement normal, le processeur externe envoie une donnée valide sur le bus d'entrée de l'émetteur TxData ainsi une validation sur le signal d'entrée TxDataWr qui indique la disponibilité de la donnée sur l'entrée. Une fois que l'émetteur est activé, il démarre la transmission de trame sur TxDout. Le récepteur à son tour

transmet les données reçues sur son bus de sortie RxDout vers les processeurs externes, tandis qu'il signale la disponibilité de la donnée à l'aide de l'activation du signal RxDataAvail. Le début de la réception d'une trame sur le récepteur est indiqué par la génération d'un pulse sur le signal StartOfFrame, et la fin de la réception de la trame est indiqué par la génération d'un pulse sur le signal EndOfFrame.

La transmission entre l'émetteur et le récepteur suit certaines règles pour garantir une bonne réception des données :

- *Activation et désactivation :* l'activation et la désactivation de l'émetteur est reliée à l'état de l'entrée externe TxEnable. Quand cette entrée est à '0' l'émetteur est désactivé, et des séquences de flags partielles ou complètes sont envoyées en permanence vers la sortie TxDout. L'activation et la désactivation du récepteur sont liées à l'état de l'entrée RxEnable, s'il est à '0' le récepteur à son tour est désactivé.
- *Commande d'annulation (abort) :* cette commande est assurée par l'entrée de l'émetteur SendAbort. L'émetteur envoie une séquence (1111111) pour prévenir le récepteur de la fin de la transmission de la trame courante. Cette action est immédiate. Lorsque le récepteur reçoit la séquence d'annulation, il active le signal AbortFound. Ce signal reste actif jusqu'à la réception d'une séquence flag pour démarrer une nouvelle transmission.
- *Transparence :* c'est le processus qui surveille les données transmises ainsi que le CRC accompagnant, pour s'assurer qu'aucune forme de séquence flag ne soit envoyée pendant la transmission ; un bit 0 est inséré par l'émetteur, à chaque fois qu'une séquence de cinq 1 consécutifs est envoyée. Le récepteur à son tour supprime ces bits redondants pour avoir les données d'origine.

IV.3.2. Exigences des fonctionnements décrites sous forme d'assertions

Le composant exécute plusieurs tâches à la fois. En simulation normale, la détection d'une erreur de fonctionnement est quasi impossible. L'introduction d'assertions est primordiale ; ces assertions garantissent une surveillance pendant la simulation des parties de fonctionnement qu'elles décrivent. Nous allons ici nous concentrer sur l'assertion suivante.

Détection fin de trame envoyée. Cette assertion surveille le comportement du récepteur. Pendant la réception des trames le signal EndOfFrame est maintenu à '0'. La fin de transmission sur le récepteur est traduite par le passage à '1' du signal EndOfFrame. Un front montant sur la sortie EndOfFrame du récepteur indique la fin de la réception d'une trame. Aucune donnée valide (RxDataAvail à '1') ne doit alors être transmise avant l'arrivée de la trame suivante (StartOfFrame à '1').

```
ASSERT_EOF :
default clock: posedge(RxClk);
always({! ENDOFFRAME ; ENDOFFRAME} |-> next (! RXDATAAVAIL until STARTOFFRAME));
```

Nous allons décrire ci-dessous l'étude de couverture de l'assertion ASSERT_EOF pendant la simulation [60].

IV.3.3. Identification des conditions d'activation d'assertion et génération des solutions

On commence tout d'abord par identifier les conditions d'activation de l'assertion. Ici on cherche à provoquer un front montant sur le signal de sortie EndOfFrame. L'algorithme est d'abord utilisé avec la première contrainte qui consiste à produire la valeur '0' sur la sortie EndOfFrame ; en sortie on obtient une liste de 2322 solutions générées. Ensuite, l'algorithme génère une deuxième liste de 277 solutions pour produire la contrainte EndOfFrame à '1'.

En observant de plus près le protocole du récepteur HDLC on remarque que le signal est majoritairement à '0', il ne passe à '1' que pour signaler la fin de la trame et retourne à '0' pour attendre la fin de la trame suivante. Ce comportement régulier de la sortie EndOfFrame nous aide à réduire la complexité des solutions aptes à produire des fronts montants sur cette sortie ; au lieu d'agencer une solution qui satisfait les deux contraintes (EndOfFrame = faux ensuite EndOfFrame = vrai), il suffit de chercher des solutions qui consiste à forcer uniquement la valeur '1'.

Sélection des solutions. Plusieurs contraintes ont été imposées dû au protocole HDLC et à une forte dépendance entre les signaux des entrées du circuit. Tout cela a rendu le calcul des probabilités d'atteindre les états internes irréaliste puisque le principe de ce calcul se base sur l'absence de toute corrélation entre les entrées primaires, ce qui n'est clairement pas le cas ici. Pour forcer la valeur '1' sur EndOfFrame, on retient une solution (après élimination des signaux inutiles) :

Solution S5:
rxenable(t-5) = true ; rxflagdet_c_reg_ShiftReg_6(t-5) = true ; nx2270(t-5) = false ;
rxflagdet_c_reg_ShiftReg_7(t-5) = true ; nx3211(t-5) = false ; rxdataen(t-5) = true ;
rxflagdet_c_reg_ShiftReg_5(t-5) = true ; rxflagdet_c_reg_ShiftReg_4(t-5) = true ;
rxflagdet_c_reg_ShiftReg_3(t-5) = true ; rxflagdet_c_reg_ShiftReg_2(t-5) = true ;
rxflagdet_c_reg_ShiftReg_1(t-5) = false ; nx20(t-5) = true ; nx3189(t-5) = false ;
rxdata(t-5) = false ;

Cette solution sera mise en œuvre dans la suite de cette section pour orienter la simulation vers une meilleure couverture d'assertion.

IV.3.4. Mise en œuvre de la solution sélectionnée et simulation

L'objectif principal est de vérifier si la solution sélectionnée augmente le taux de la couverture. Pour cela une simulation de référence s'impose.

Simulation avec des vecteurs de test pseudo-aléatoires. La première simulation sera dirigée par des vecteurs générés d'une façon pseudo-aléatoire. Cependant, la forme des trames doit être respectée, en commençant toujours par des séquences de flag d'ouverture et en finissant par des séquences de flag de fermeture. Alors deux process ont été ajoutés dans le

testbench initial et un troisième a été modifié, afin d'envoyer des trames avec des tailles pseudo-aléatoires.

- Un premier process est rajouté pour créer un entier aléatoire int_rand à partir de la fonction UNIFORM, qui sera converti en une variable binaire qu'on nomme stim.
- Un deuxième process définit un compteur cpt. Il est utilisé pour fixer la taille globale des données envoyées sur le récepteur.
- Le troisième process SEND_BITS (Figure IV.22) est un process déjà existant dans le testbench. Il est modifié pour insérer la partie qui sera générée pseudo-aléatoirement dans les trames envoyées, tout en respectant leur forme standard. Ces trames sont envoyées sur le bus d'entrée de récepteur RxData.

```
-------------------------------------------------------------------------
-- Génération des Trames

while (cpt < 15000) loop      -- On crée des trames tant qu'on n'a pas eu
                              -- 15000 cycle de RxDataEn.
  trame1 <= '1';              -- Génération de la 1er partie de la trame.
  UNIFORM(seed1, seed2, rand);
  long_min := INTEGER(TRUNC(rand*4.0))+3; -- long_min aléatoire entre 3 et
                                          -- 7.
  while (counter < long_min * 8) loop     -- Génération de long_min trames
                                          -- de 8 bits.
      wait until rising_edge(RxDataEn);
      RxData <= stim(2);              -- RxData prend une valeur aléatoire
      counter <= counter + 1;
      if (stim(2) = '1') then       -- Incrémentation de Count_Ones si
        Count_Ones <= Count_Ones + 1; -- plusieurs '1' se suivent.
      else
        Count_Ones <= 0;
      end if;
      if (Count_Ones = 5) then      -- On force RxData à '0' si 5 '1' se
        RxData <= '0';              -- suivent.
       Count_Ones <= 0;
      end if;
    end loop;

    counter <= 0;
    Count_Ones <= 0;
    trame1 <= '0';
    trame2 <= '1';              -- Génération de la 2ème partie de la trame

    while (EndOfFrame = '0') loop       -- Cette trame se finira dès qu'un
      wait until rising_edge(RxDataEn); -- flag va apparaitre
      RxData <= stim(2);
      if (stim(2) = '1') then
        Count_Ones <= Count_Ones + 1;   -- Count_Ones permet de compter le
      Else                              -- nombre de '1' qui se suivent.
        Count_Ones <= 0;
      end if;
      if (Count_Ones = 6) then
        Count_Ones <= 0;
      end if;
    end loop;

    trame2 <= '0';

  end loop; -- cpt
```

Figure IV.22 : Process SEND_BITS modifié pour générer des données pseudo-aléatoirement

Comme le montre la Figure IV.22, les trames vont être créées de la façon suivante : on génère un nombre pseudo-aléatoire longueur_min qui est le nombre minimum de données (de 8 bits) qu'on veut mettre dans une trame (CRC compris). Il faut par exemple que longueur_min soit au moins égal à 3 si on souhaite avoir un CRC de 16 bits. Ici, il va être compris entre 3 et 7. La première étape consiste à produire longueur_min données « viables », c'est-à-dire avec les

zéros insérés pour qu'il n'y ait pas de flag dans les données. Puis deuxièmement, on laisse produire des valeurs pseudo aléatoirement jusqu'à ce qu'arrive un flag. Et on recommence de la même façon pour la trame suivante. Dans les simulations, ces deux étapes sont détectées grâce aux signaux Trame1 et Trame2. De plus, elles sont effectuées trois fois. Une fois avec le CRC de 16 bits, une autre fois avec celui de 32 bits et une dernière fois sans CRC.

Simulation et résultat :

On a voulu tout d'abord s'assurer que ces modifications n'affectent pas le protocole de réception du HDLC. On a pu observer le comportement de l'assertion et on a obtenu un premier résultat de taux de couverture : **L'assertion était vérifiée 47 fois**.

Ce premier résultat obtenu va nous servir comme référence pour pouvoir comparer et prouver l'amélioration apportée les solutions proposées par notre méthode.

```
  while (EndOfFrame = '0') loop     -- Génération d'une trame avec forçage
                                    -- des entrées.
    wait until RxClk'event and RxClk='1';
    wait until RxClk'event and RxClk='1';

    if(rxflagdet_c_ShiftReg_1_s = '0'  and rxflagdet_c_ShiftReg_2_s = '1'
and rxflagdet_c_ShiftReg_3_s = '1' and rxflagdet_c_ShiftReg_4_s = '1' and
rxflagdet_c_ShiftReg_5_s = '1' and rxflagdet_c_ShiftReg_6_s = '1' and
rxflagdet_c_ShiftReg_7_s = '1' ) then
      Test_if1 <= '1';
      rxdata <= '0';
      Count_Ones <= 0;
      for i in 0 to 7 loop
        wait until RxClk'event and RxClk='1';
      end loop;
      Test_if1 <= '0';

    else
      Test_else1 <= '1';
      RxData <= stim(2);
      if (stim(2) = '1') then
        Count_Ones <= Count_Ones + 1;
      else
        Count_Ones <= 0;
      end if;
        if (Count_Ones = 6) then
          Count_Ones <= 0;
        end if;
      for i in 0 to 7 loop
        wait until RxClk'event and RxClk='1';
      end loop;
    end if;
  end loop;

  trame2 <= '0';
  Test_if1 <= '0';
  Test_else1 <= '0';


end loop; -- cpt
```

Figure IV.23 : Process SEND_BITS après ajout du générateur

Simulation pseudo-aléatoire avec générateur. La deuxième étape consiste à transformer la solution sélectionnée en générateur décrit en process, ensuite, insérer ce générateur dans le testbench, qui va forcer des valeurs sur les entrées pendant la simulation en cas de détection des conditions d'activation de ce générateur. On a repris le même process SEND_bits modifié et on a rajouté plusieurs conditions dans le but de mettre en œuvre le générateur. Comme le montre la Figure IV.23 les nouvelles modifications ont été ajoutées dans la deuxième partie de la trame. Et pour effectuer la simulation, la Figure IV.24 donne un aperçu sur la façon dont sont agencés les composants, ainsi les connexions du nouveau

process avec le signal interne Rxflagdet_c_ShiftReg pour vérifier les conditions d'activation du générateur, et l'entrée RxData.

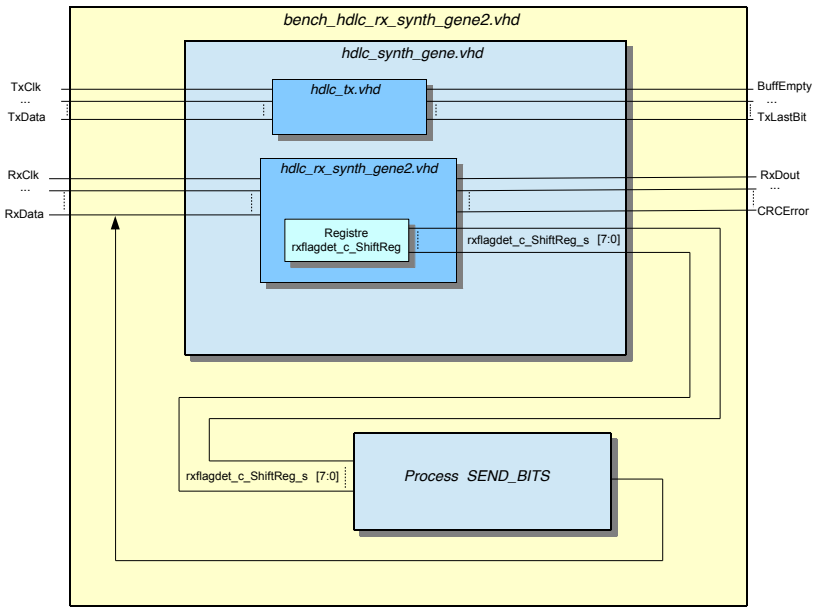


Figure IV.24 : Connexion du process avec le récepteur du HDLC dans le testbench

Simulation et résultat :

En effectuant la simulation avec les nouvelles contraintes mises en œuvre dans le process de génération des trames de données, on a remarqué qu'effectivement les trames sont construites en respectant les formes d'envoi standard tout en ayant des données générées pseudo-aléatoirement de tailles différentes à chaque fois. Comme le montre la Figure IV.25, pendant la simulation du composant avec le nouveau process, des fronts montants sur la sortie EndOfFrame sont produits plus souvent.

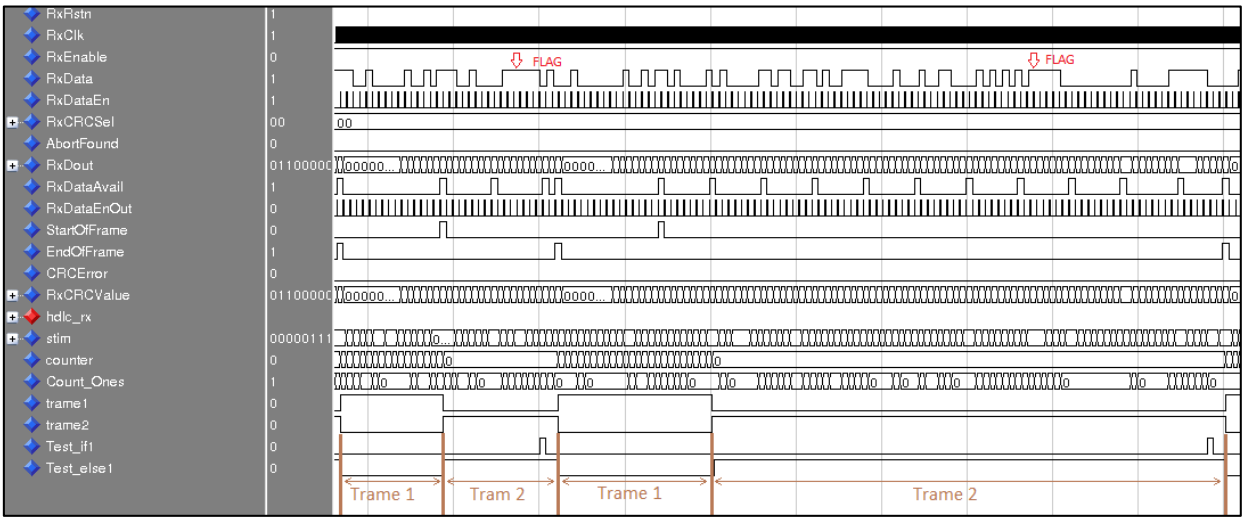
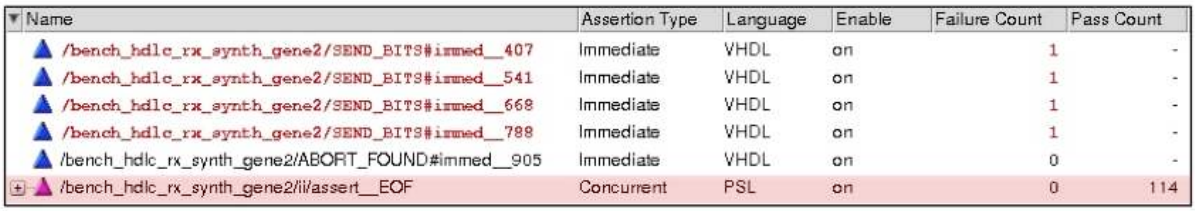


Figure IV.25 : Extrait de simulation des signaux du récepteur du HDLC avec le générateur

Comme le montre la Figure IV.26, qui représente les résultats de la vérification de l'assertion ASSERT_EOF, il y a une amélioration remarquable du taux de couverture d'assertion. Cette dernière **est vérifiée 114 fois**.



Name	Assertion Type	Language	Enable	Failure Count	Pass Count
/bench_hdlc_rx_synth_gene2/SEND_BITS#immed__407	Immediate	VHDL	on	1	-
/bench_hdlc_rx_synth_gene2/SEND_BITS#immed__541	Immediate	VHDL	on	1	-
/bench_hdlc_rx_synth_gene2/SEND_BITS#immed__668	Immediate	VHDL	on	1	-
/bench_hdlc_rx_synth_gene2/SEND_BITS#immed__788	Immediate	VHDL	on	1	-
/bench_hdlc_rx_synth_gene2/ABORT_FOUND#immed__905	Immediate	VHDL	on	0	-
/bench_hdlc_rx_synth_gene2/ii/assert__EOF	Concurrent	PSL	on	0	114

Figure IV.26 : Résultats de vérification de Assert_EOF

Les cas d'études traités dans ce chapitre ont permis de donner un aperçu sur l'efficacité de l'approche développée dans le cadre de cette thèse. Des résultats ont montré clairement l'amélioration apportée en utilisant les solutions générées dans le but précis de satisfaire des contraintes qui portent sur les signaux activant les assertions.

V. Conclusion et perspectives

Les travaux réalisés dans cette thèse entrent dans le contexte de la vérification fonctionnelle. Rappelons que cette vérification consiste à s'assurer que le fonctionnement d'un circuit respecte la spécification fonctionnelle de départ. Plusieurs méthodes de vérification fonctionnelle ont été proposées, on retient principalement la vérification statique à base de Model Checking et la vérification dynamique à base d'assertions. L'introduction des assertions intégrables dans les descriptions des circuits a permis de faire un grand pas en avant dans le domaine de la vérification. Les assertions matérialisées par des composants appelés moniteurs représentent des descriptions des fonctionnements à surveiller. Toutefois, des séquences de test générées aléatoirement peuvent conduire à un faible taux de couverture des conditions d'activation des moniteurs et, de ce fait, peuvent être peu révélatrices de la satisfaction des assertions. Pour résoudre ce problème, plusieurs approches basées sur le Model Checking ont été déjà réalisées pour la génération des séquences de test. Cependant, l'utilisation du Model Checking donne la possibilité de produire une solution à la fois.

La problématique majeure traitée dans le cadre de cette thèse est de s'assurer de l'activation des assertions affectées au circuit à vérifier pour améliorer la qualité de la vérification. Les méthodes à base d'automates finis ont plusieurs inconvénients dont le plus important est l'explosion combinatoire. Dans le contexte de SoCs complexes ces méthodes trouvent rapidement leurs limites. L'objectif principal dans cette thèse était de proposer une approche efficace pour améliorer le taux de couverture des conditions d'activation des assertions PSL et pour résoudre le problème de vacuité.

Nous avons proposé une méthode pour améliorer la qualité de la vérification à base d'assertions. Pour cela, un algorithme a été développé. Cet algorithme prend en entrée le système à vérifier et les contraintes à satisfaire, déduites des opérandes qui activent les assertions. Il fournit en sortie un large choix de vecteurs de test contenant des traces spécifiques qu'on peut appliquer sur les entrées du circuit lorsqu'une séquence de signaux internes particulières est observée. Il est possible ensuite d'intégrer les solutions sélectionnées pour construire des générateurs de séquences de test. Ces derniers permettent de générer des séquences de test bien déterminées, visant à activer d'une manière sûre les assertions à vérifier au cours de la simulation. Deux principaux cas d'étude ont été utilisés pour tester l'intérêt de cette approche. Nous avons pu montrer que l'algorithme développé permet de fournir un grand choix de solutions aptes à orienter la simulation pour mettre en évidence les fonctionnements décrits par des assertions et augmenter le taux de couverture de ces assertions. Avoir plusieurs solutions alternatives pour activer une même assertion donne la possibilité de choisir celle qui est la plus apte (efficace et moins complexe). Améliorer la qualité de la couverture d'assertion garantit le bon déroulement de la phase de la vérification, qui représente un goulot d'étranglement dans le flot de conception des circuits.

Les travaux décrits dans cette thèse présentent un prototype de base vers la vérification au niveau structurel. Les résultats obtenus prouvent l'efficacité de cette approche.

C’est une voie qui promet beaucoup de perspectives d’évolutions pour améliorer et aller jusqu’à concevoir un vrai outil capable de traiter tout cas d’étude. Les travaux de cette thèse se sont concentrés sur le traitement des cas d’assertions moyennement compliquées, et traitent les contraintes cas par cas pour combiner les solutions avant d’obtenir celles qui seraient en mesure d’apporter des améliorations au niveau de la qualité de la vérification. Il serait plus intéressant de pouvoir permettre de combiner les contraintes et générer des solutions globales, ce qui va diminuer les étapes de construction des solutions efficaces, et qui va conduire à un gain de coût et de temps.

Bibliographie

Bibliographie

[1] Edmund M. Clarke, Orna Grumberg and Doron A. Peled: **"Model Checking",** The MIT Press, 2000.

[2] Sheldon B. Akers : **"Binary Decision Diagrams",** *In : IEEE Transactions on Computers, Vol. c-27, No. 6, June 1978.*

[3] A. Biere, A. Cimatti, Edmund M. Clarke, O. Strichman, and Y. Zhu: **"Bounded Model Checking",** *In Vol. 58 of Advances in Computers, 2003.*

[4] **"IEEE Standard VHDL Language Reference Manual",** *The Institute of Electrical and Electronics Engineers, Inc. IEEE Std 1076, 2000 Edition, Published 29 December 2000.*

[5] **"IEEE Standard Verilog ® Hardware Description Language",** *The Institute of Electrical and Electronics Engineers, Inc. IEEE Std 1364-2001, Published 28 September 2001.*

[6] Foster H., Krolnik A., and Lacey D.: **"Assertion-Based Design",** *Springer, Boston, (2004).*

[7] IEEE Std 1850-2005: **"IEEE Standard for Property Specification Language (PSL) ",** *IEEE (2005).*

[8] IEEE Std 1800-2005, IEEE Standard for System Verilog: **Unified Hardware Design, Specification and Verification Language**. *IEEE (2005).*

[9] J. Srouji, S. Mehta, D. Brophy, K. Pieper, S. Sutherland, and IEEE 1800 Work Group: **"IEEE standard for systemverilog - unified hardware design, specification, and verification language",** *Technical report, pub-IEEE-STD:adr, Nov 2005.*

[10] J. Havlicek and Y. Wolfshal: **"PSL and SVA: Two standard assertion languages addressing complementary engineering needs",** *In Proceedings of the Design and Verification Conference: DVCon'05, (2005).*

[11] **http://tima.imag.fr/vds/Horus/**

[12] Y. Oddos, K. Morin-Allory, and D. Borrione : **"Horus: A tool for Assertion-Based Verification and on-line testing",** *In: Proc. MEMOCODE'08 (2008).*

[13] Y. Oddos : **"Vérification semi-formelle et synthèse automatique de PSL vers HDL "**, Thèse de l'Université de Grenoble, *octobre 2009.*

[14] Y. Abarbanel, I. Beer, L. Gluhovsky, S. Keidar, and Y. Wolfsthal: **"FoCs – Automatic Generation of Simulation Checkers from Formal Specifications",** *In : Computer Aided Verification. LNCS. Springer-Verlag, 2000.*

[15] **http://www.research.ibm.com/haifa/projects/verification/focs/present/index.htm**

[16] M. Pellauer, M. Lis, D. Baltus, and R. Nikhil : **"Synthesis of synchronous assertions with guarded atomic actions",** *In Proceedings of the 4th ACM-IEEE International Conference on Formal Methods and Models for Codesign: MEMOCODE'05, pages 15–24, Jul. 2005.*

[17] E. Gascard : **"From sequential extended regular expressions to deterministic automata",** *Technical Report - TIMA Laboratory, Jul. 2005.*

[18] M. Jenihhin, J. Raik, A. Chepurov, and R. Ubar: **"Assertion checking with PSL and high-level decision diagrams",** *In Proc. Workshop on RTL and High Level Testing: WRTLT'07, (2007).*

[19] M. Boulé and Z. Zilic: **"Incorporating efficient assertion checkers into hardware emulation",** *In Proc. International Conference on Computer Design: ICCD'05, pages 221–228. IEEE Computer Society, 2005.*

[20] M. Boulé and Z. Zilic: **"Efficient automata-based assertion-checker synthesis of PSL properties",** *In Proceedings of IEEE International High Level Design Validation and Test Workshop: HLDVT'06, Nov 2006.*

[21] **http://www.dolphin.fr/medal/products/smash/options/smash_SVA.php**

[22] I. Beer, S. Ben-David, C. Eisner, and Y. Rodeh: "**Efficient detection of vacuity in ACTL formulas",** In Proc. *9th Conferences en Computer Aided Verification, volume 1254 of Lecture Notes in Computer Science, pages 279-290, 1997.*

[23] O. Kupferman and M. Y. Vardi: **"Vacuity Detection in Temporal Model Checking",** *CHARME'99, 1999.*

[24] L. Di Guglielmo, F. Fummi and G. Pravadelli: **"Vacuity Analysis for Property Qualification by Mutation of Checkers",** *In: DATE 2010.*

[25] I. Beer, S. Ben-David, C. Eisner and Y. Rodeh: **"Efficient Detection of Vacuity in Temporal Model Checking",** *Formal Methods for Technology Transfer, vol. 4, no. 2, pp. 141-163, 2001.*

[26] **http://iroi.seu.edu.cn/books/asics/Book2/CH14/CH14.5.htm**

[27] Christian Landrault : **"Test, Testabilité et Test Integré Des Circuits Intégrés Logiques",** *Cours de Test, Master Systèmes Microélectroniques, Montpellier 2007.*

[28] Prabhakar Goel : **"An Implicit Enumeration Algorithm to Generate Tests for Combinational Logic Circuits",** *In : IEEE Transactions on Computers, March 1981.*

[29] H. Fujiwara: **"On the Acceleration of Test Generation Algorithms",** *In : IEEE Transactions on Computers, Dec. 1983.*

[30] G. Parthasarathy, Chung-Yang Huang, and Kwang-Ting Cheng : **"An Analysis of ATPG and SAT algorithms for Formal Verification",** *High-Level Design Validation and Test Workshop, 2001.*

[31] Haluk Konuk, Tracy Larrabee : **"Explorations of Sequential ATPG Using Boolean Satisfiability",** *Proc. 11th VLSI Test Symposium, 1993.*

[32] Nikhil S. Saluja, Sunil P. Khatri : **"Efficient SAT-based Combinational ATPG using Multi-level Don't-Cares"** *In Proc. International Test Conference (2005).*

[33] Prasad, M., Hsiao, M., Jain, J.: **"Can SAT be used to improve sequential ATPG methods? ",** *In: Proc. International Conference on VLSI Design (2004).*

[34] Micheal S. Hsiao, Elizabeth M. Rudnick, and Janak H. Patel : **"Sequential Circuit Test Generation Using Dynamic State Traversal",** *In : ED&TC 1997.*

[35] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, Sharad Malik : **"Chaff: Engineering an Efficient SAT Solver",** *In : DAC 2001.*

[36] T. Niermann and J. H. Patel : **"HITEC: A test generation package for sequential circuits"**. *In Proc. of EDAC, pages 214–218, 1991.*

[37] Stephan Eggersglüß, Rolf Drechsler : **"Improving Test Pattern Compactness in SAT-based ATPG",** *In : ATS 2007.*

[38] S. Tasiran and K. Keutzer : **" Coverage metrics for functional validation of hardware",** *In : designs. IEEE Design and Test of Computers, 18(4):36–45, 2001.*

[39] J. Sordoillet, and S. Davey : **"Integrated, comprehensive assertion-based coverage",** *In : EDA Tech Forum 3(1):22–25, (2006).*

[40] **http://www.transeda.com/**

[41] Geist, D., Farkas, M., Landver, A., Lichtenstein, Y., Ur, S., Wolfsthal, Y.: **"Coverage- Directed Test Generation Using Symbolic Techniques***", In: Proc. International Conference on Formal Methods in Computer-Aided Design (FMCAD) (1996).*

[42] G. Ratsaby, B. Sterin, S. Ur: **"Improvements in Coverability Analysis",** *In : Proc. International Symposium of Formal Methods Europe, FME'02. LNCS 2391.*

[43] L. Arditi, H. Boufaïed, A. Cavanié and V. Stehlé: **"Coverage Directed Generation of System-Level Test Cases for the Validation of a DSP System",** *Texas Instruments France. MS 21. BP 5. 06270 Villeneuve Loubet. France. FME 2001. LNCS 2021.*

[44] Oberko¨nig, M., Schickel, M., Eveking, H.: **"Improving Testbench Evaluation using Normalized Formal Properties",** *In: Proc. International Workshop on Verification and Evaluation of Computer and Communication Systems (VECoS'2009) (2009).*

[45] Onur Guzey, and Li-C. Wang : **"Coverage-directed test generation through automatic contraint extraction",** *In : High Level Design Validation and Test Workshop (HLVDT'07), 2007.*

[46] Jason G. Tong, and M. Boulé : **"Definig and Providing Coverage for Assertion-Based Dynamic Verification",** *Journal of Electronic Testing 26, (2010).*

[47] Heon-Mo Koo, and P. Mishra : **"Functional Test Generartion Using Design ans Property Decomposition Techniques",** *In ACM Transactions on Embedded Computing Systems, Vol. 8, No. 4, Article 32, (July 2009).*

[48] O. Kupferman, and Y. Lustig : **"What Triggers a Behavior ?",** *In : Proc. International Conference on Formal Methods in Computer-Aided Design (FMCAD'07), (2007).*

[49] L.Pierre, L.Damri: **"Improvement of Assertion-Based Verification through the Generation of Proper Test Sequences",** *Proc. Forum on specification & Design Languages (FDL'11), September 2011.*

[50] K. Morin-Allory, M. Boulé, D. Borrione and Z. Zilic: **"Validating Assertion Language Rewrite Rules and Sementics with Automated Theorem Provers"***, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems,* 2010.

[51] M. Boulé and Z. Zilic: **"Automata-Based Assertion-Checkers Synthesis of Properties",** *In ACM Transactions on Design Automation of Electronic Systems, 2008.*

[52] V. Singh, and T. Garg : **"Transformation of Simple Subset of PSL into SERE Implication Formulas for Verification with Model Checking and Simulation Engines using Semantic Preserving Rewrite Rules",** *In : United States Patent 7386813 (2008).*

[53] Dietmeyer, D.: **Logic Design of Digital Systems,** *Allyn and Bacon (1978).*

[54] **http://vlsi.colorado.edu/~fabio/CUDD/**

[55] A.Guerre-Chaley: **"Développement d'un outil d'aide à la sélection de solutions pour l'accélération d'assertions",** *Stage de L1 Université Joseph Fourier, Juillet 2012.*

[56] **http://users-tima.imag.fr/vds/ouchet/vsyml.html**

[57] F.Ouchet, D.Borrione, K.Morin-Allory, L.Pierre: **"High-level symbolic simulation for automatic model extraction",** *In : Proc. IEEE Symposium on Design and Diagnostics of Electronic Systems, Liberec (Czech Republic), April 2009.*

[58] Dolphin Integration: **"SLED & SMASH - PSL Detectors Tutorial",** *June 2010.*

[59] Thales: **"Single channel HDLC controller IP - User and Integrator guide",** *2008.*

[60] P.Ferris: **"Évaluation de solutions visant à optimiser le processus de vérification de propriétés fonctionnelles de circuits numériques",** *Mémoire de stage de Master1 EEATS Université Joseph Fourier, Juillet 2012.*

Annexes

Annexe 1.

Cette annexe illustre le déroulement de l'algorithme de la Figure III.4 sur le circuit reconnaisseur de code BCD.

L'algorithme démarre avec $ens\_sig = \{Out\}$, un ensemble *ens_dep* est généré à l'aide de la fonction *traverse_porte*. Cet ensemble contient la liste des signaux d'entrées du composant traversé. Dans le cas de cet exemple on aura $ens\_dep = \{C_2, C_3\}$ (Figure A1.1). La fonction *traverse_porte* construit en parallèle l'arbre des solutions et les signaux dans *ens_sig* avec leurs dépendances suivant le type de composant traversé (disjonctif ou conjonctif), en propageant aussi les contraintes sur les nouveaux signaux.

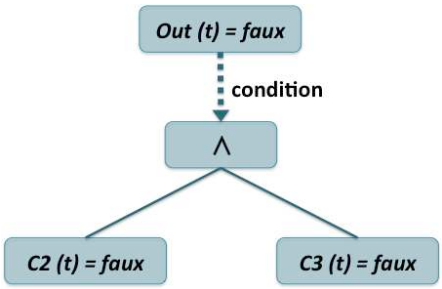


Figure A1.1 : Condition de propagation en arrière de la contrainte Out (t) = faux

L'ensemble *ens_sig* devient $ens\_sig = \{C_2, C_3\}$, puisque C_2 et C_3 ne sont ni des entrées primaires, ni des états déjà traversés par l'algorithme. Cet ensemble est sauvegardé dans une liste servant à vérifier si l'algorithme a déjà traversé un état interne du circuit. On vérifie les signaux de *ens_sig* l'un après l'autre. On commence par le signal C_2, l'algorithme le considère comme une nouvelle contrainte à faire monter vers les entrées et le fait passer dans la fonction *traverse_porte*. Cette fonction génère un nouvel ensemble $ens\_dep = \{S_3, S_4\}$ et reprend l'arbre pour ajouter cette nouvelle branche avec de nouvelles contraintes à propager. Ensuite, l'algorithme vérifie si cet *ens_dep* ne contient pas des entrées primaires pour les supprimer, et aussi, si les signaux dans cet ensemble étaient traversés par l'algorithme une première fois, sinon il considère que c'est la première fois et il ajoute *ens_dep* dans la liste des ensembles déjà traversés pour éviter que l'algorithme passe une deuxième fois par le même signal. La même chose est faite pour le deuxième signal C_3 de *ens_sig*, et la fonction *traverse_porte* produit $ens\_dep = \{C_4, In\}$. Une fois que tous les signaux dans *ens_sig* sont parcourus, l'algorithme crée le nouvel ensemble *ens_sig* qui rassemble tous les signaux dans *ens_dep* qui ne sont ni des entrées primaires, ni des signaux déjà traversés, pour pouvoir remonter d'un pas de plus les contraintes vers les entrées. Dans cet exemple, *ens_sig* est l'union des ensembles *ens_dep* de C_2 et C_3, $ens\_sig = \{S_3, S_4, C_4\}$. L'algorithme continue de traiter tous les signaux sur le cône d'influence pour pouvoir propager les contraintes jusqu'aux entrées primaires où s'arrêter sur des états internes déjà traversés.

La Figure A1.2 montre la propagation de la contrainte $Out\ (t) = faux$ dans le cône d'influence du signal *Out* jusqu'à l'entrée *In* ou s'arrêter sur des états internes du circuit en cas de détection d'une boucle structurelle.

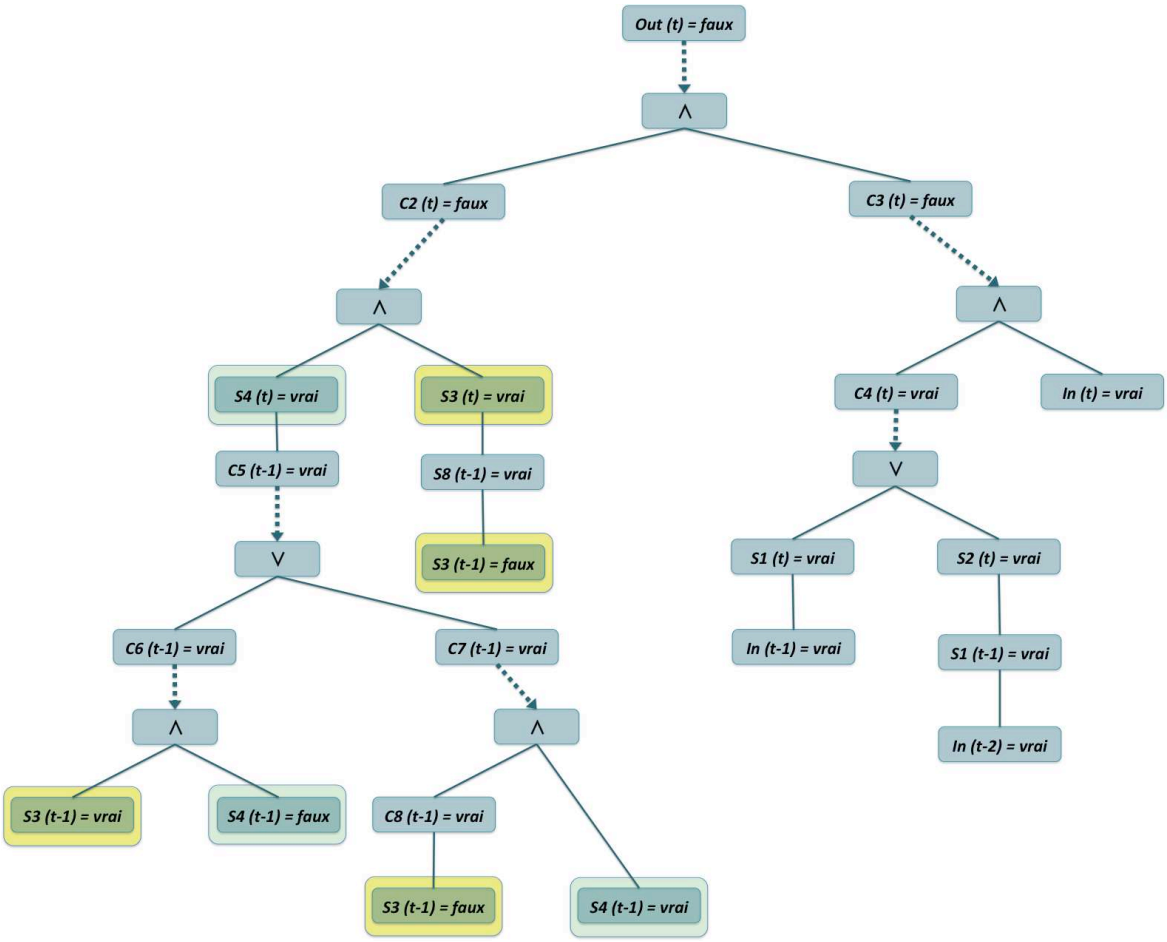


Figure A1.2 : Application de l'algorithme sur le reconnaisseur de code BCD

Annexe 2.

Cette annexe explicite la construction de BDD dans CUDD [54].

CUDD Package (CU Decision Diagram)

Le package CUDD (Colorado University Decision Diagram) fait partie de la librairie C/C++ [CUDD]. C'est un ensemble de fonctions qui permet à l'utilisateur la création et la manipulation des diagrammes de Décisions binaires (BDD), de ZDD (Zero-suppressed Decision Diagrams) et de ADD (Algebraic Decision Diagrams).

Garbage Collection. Dans CUDD, un BDD est construit du bas vers le haut, alors plusieurs petits BDDs sont construits dans ce processus qui sont inclus quand on traverse l'arbre. Une fois inclus dans l'arbre, la mémoire de ces petits BDDs peut être récupérée. Alors pour faciliter la tache du *garbage collector*, on a besoin de *référencer* et *déréférencer* chaque nœud dans le BDD.
Pour *référencer* un nœud on utilise la fonction

`Cudd_Ref` (`DdNode`*)

Pour *déréférencer* un nœud, ainsi que tous ses descendants, on utilise la fonction :

`Cudd_RecursiveDeref` (`DdNode`*)

Tableau Unique. C'est dans ce tableau que tous les nœuds crées pour la construction du BDD sont mis. Il est utilisé pour garantir qu'un nœud spécifique est unique, c'est-à-dire que si deux nœuds contiennent les mêmes fils et représentent la même variable, ils doivent être fusionnés en un seul nœud.

Les compléments. On fait réfèrence aux deux fils d'un nœud par le fils *then* qui correspond à la branche 1, et le fils *else* correspondant à la branche 0. Cependant, les fils *else* peuvent être complémentés.

Structures de donnée. CUDD contient deux structures de données qui représentent le cœur de ce paquet, qui sont le `DdManager` et le `DdNode`.

- `DdManager` : la création de cette structure est la première chose faite lors de l'écriture d'un programme avec CUDD et elle doit être transmise à presque toutes les fonctions CUDD. Cependant, il n'est jamais nécessaire de manipuler ou d'inspecter cette structure directement, par contre il faut obligatoirement l'initialiser à l'aide de la macro ci-dessous (Figure A2.1).

```
DdManager * Cudd Init (
  unsigned int  numVars,          // nombre initial des variables BDD
  unsigned int  numVarsZ,         // nombre initial des variables ZDD
  unsigned int  numSlots,         // taille initiale du tableau unique
  unsigned int cacheSize ,        // taille initiale du cache
  unsigned long maxMemory         // occupation maximale ciblée de la mémoire
                                  // (0 veut dire illimitée)
);
```

Figure A2.1 : Fonction d'initialisation de la structure DdManager

- DdNode : cette structure représente la brique de base pour la construction des BDDs. Elle est définie comme suit (Figure A2.2) :

```
struct DdNode {
  DdHalfWord index ;          // indexe de la variable représentée par ce nœud
  DdHalfWord ref ;            // comptage de référence
  DdNode *next ;              // prochain pointeur sur le tableau unique
  union {
    CUDDVALUETYPE value;      // pour les nœuds constants (feuilles)
    DdChildren kids ;         // pour les nœuds internes
  } type;
};
```

Figure A2.2 : Structure de nœud du diagramme de décision

Cette structure contient :

- Index : il symbolise l'unicité d'une variable représentée par ce nœud. Les nœuds sont numérotés dans l'ordre de leur création (commencer par 0). Ces indices sont permanents indépendamment de l'ordre pris en compte dans le BDD.
- Ref : il sauvegarde le comptage référence de cette variable. le comptage référence est incrémenté de 1 à chaque fois que la fonction Cudd_Ref est appelé sur ce nœud, et décrémenté de 1 pour chaque appel de la fonction Cudd_RecursiveDeref. Si ref = 0, CUDD reconnaît que ce nœud n'a plus aucune utilité et peut être supprimé (par le garbage collector).
- Next : contient un pointeur vers le prochain nœud qui représente la même variable que ce nœud. Tous les nœuds qui représentent la même variable sont liés ensembles pour le tableau unique.

Chaque nœud peut être soit une feuille et contient sa valeur (CUDDVALUETYPE), ou un pointeur vers les nœuds fils (Kids).

Ces valeurs sont stockées dans le champ de type. Les macros suivantes aident à définir le type du nœud :

Cudd_IsConstant (DdNode * nœud) : retourne 1 si le nœud est une feuille, sinon 0.
Cudd_V (DdNode * nœud) : retourne la valeur du nœud s'il est une constante.
Cudd_T (DdNode * nœud) : retourne un pointeur sur le fils *then* du nœud.

Cudd_E (DdNode * nœud) : retourne un pointeur sur le fils *else* du nœud. Cette valeur peut être complémentée, et pour vérifier on a :
Cudd_IsComplement (DdNode * nœud) : retourne 1 si le nœud est complémenté, sinon 0. Si ce nœud est complémenté on peut utiliser :
Cudd_Regular (DdNode * nœud) : pour retourner la version régulière du nœud complémenté.

La construction du BDD est faite du bas vers le haut. On commence par la création de deux variables pour chaque entrée. Ensuite, on combine ces variables avec les opérateurs ET/ OU jusqu'à avoir le BDD. Il existe des fonctions qui prennent en charge ces tâches et automatisent la construction du BDD :
Cudd_bddIthVar (DdManager * manager, int * i) : cette fonctions crée une variable avec un indice i donné par le programmeur s'il n'existe pas, sinon retourne un pointeur vers la variable déjà existante.
Cudd_bddNewVar (DdManger * manager) : cette fonction crée une variable avec un indice attribuée automatiquement suivant l'ordre de la création dans le tableau unique.
Cudd_bddAnd (DdManger * manager, DdNode * nœud1, DdNode * nœud2) : cette fonction permet de combiner nœud1 et nœud2 avec une opération conjonctive.
Cudd_bddOr (DdManger * manager, DdNode * nœud1, DdNode * nœud2) : cette fonction retourne une nouvelle représentation du BDD qui combine nœud1 et nœud2 avec une opération disjonctive.

Fonctions de construction de BDD :

À l'aide d'une fonction de transformation de l'arbre de solutions en fonctions CUDD (Figure A2.3), on crée une fonction qui va servir par la suite à la construction du BDD. Les appels de fonctions du paquet CUDD sont mis en ordre pour créer une fonction capable de construire le BDD. On commence par la création de deux variables pour chaque entrée. Ensuite, on combine ces variables avec les opérateurs ET/OU jusqu'à avoir le BDD. Pour cela, une fonction est créée pour la transformation de l'arbre des solutions en des fonctions CUDD, pour créer les variables de BDD qui correspondent à celles dans l'arbre de solutions et les opérateurs ET/OU pour combiner deux variables suivant les mêmes liens que dans l'arbre de solutions.

La Figure A2.3 présente un exemple de fonction pour la construction du BDD. Ce BDD construit correspond à l'arbre de solutions généré par l'algorithme du circuit BCD pour satisfaire la contrainte sur sa sortie : $Out\,(t) = faux$.

```
DdNode * fonction_construction_BDD(DdManager * manager)
{
// Nœuds
DdNode * x_42 = Cudd_bddNewVar(manager);                    // S4 1 v
DdNode * x_39 = Cudd_bddNewVar(manager);                    // S3 1 v
DdNode * x_36 = Cudd_Not(x_39);                             // S3 1 f
DdNode * x_38 = Cudd_Not(x_42);                             // S4 1 f
DdNode * x_7 = Cudd_Not(Cudd_bddNewVar(manager));           // In1 0 f
DdNode * x_21 = Cudd_Not(Cudd_bddNewVar(manager));          // In1 1 f
DdNode * x_32 = Cudd_Not(Cudd_bddNewVar(manager));          // In1 2 f

// Operateurs
DdNode * x_13 = Cudd_bddAnd(manager, x_21 , x_32 );
Cudd_Ref(x_13);
 ⋮
DdNode * x_3 = Cudd_bddOr(manager, x_16 , x_39 );
Cudd_Ref(x_3);
DdNode * x_0 = Cudd_bddOr(manager, x_3 , x_6 );
Cudd_Ref(x_0);

// Déréférencer les nœuds déjà inclus dans le BDD
Cudd_RecursiveDeref(manager,x_3);
 ⋮
Cudd_RecursiveDeref(manager,x_13);

// Retourner la racine de BDD
DdNode * outputs = new DdNode[1];
outputs = x_0;
return outputs;
}
```

Figure A2.3: Exemple de fonction de construction de BDD

Résumé :

Avec la complexité croissante des systèmes sur puce, le processus de vérification devient une tâche de plus en plus cruciale à tous les niveaux du cycle de conception, et monopolise une part importante du temps de développement. Dans ce contexte, l'assertion-based verification (ABV) a considérablement gagné en popularité ces dernières années. Il s'agit de spécifier le comportement attendu du système par l'intermédiaire de propriétés logico-temporelles, et de vérifier ces propriétés par des méthodes semi-formelles ou formelles. Des langages de spécification comme PSL ou SVA (standards IEEE) sont couramment utilisés pour exprimer ces propriétés. Des techniques de vérification statiques (model checking) ou dynamiques (validation en cours de simulation) peuvent être mises en œuvre.

Nous nous plaçons dans le contexte de la vérification dynamique. A partir d'assertions exprimées en PSL ou SVA, des descriptions VHDL ou Verilog synthétisables de moniteurs matériels de surveillance peuvent être produites (outil Horus). Ces composants peuvent être utilisés pendant la conception (en simulation et/ou émulation pour le débug et la validation de circuits), ou comme composants embarqués, pour la surveillance du comportement de systèmes critiques.

Pour l'analyse en phase de conception, que ce soit en simulation ou en émulation, le problème de la génération des séquences de test se pose. En effet, des séquences de test générées aléatoirement peuvent conduire à un faible taux de couverture des conditions d'activation des moniteurs et, de ce fait, peuvent être peu révélatrices de la satisfaction des assertions. Les méthodes de génération de séquences de test sous contraintes n'apportent pas de réelle solution car les contraintes ne peuvent pas être liées à des conditions temporelles. De nouvelles méthodes doivent être spécifiées et implémentées, c'est ce que nous nous proposons d'étudier dans cette thèse.

Abstract :

With the increasing complexity of SoC, the verification process becomes a task more crucial at all levels of the design cycle, and monopolize a large share of development time. In this context, the assertion-based verification (ABV) has gained considerable popularity in recent years. This is to specify the behavior of the system through logico-temporal properties and check these properties by semiformal or formal methods. Specification languages such as PSL or SVA (IEEE) are commonly used to express these properties. Static verification techniques (model checking) or dynamic (during simulation) can be implemented.

We are placed in the context of dynamic verification. Our assertions are expressed in PSL or SVA, and synthesizable descriptions VHDL or Verilog hardware surveillance monitors can be produced (Horus tool). These components can be used for design (simulation and/or emulation for circuit debug and validation) or as embedded components for monitoring the behavior of critical systems.

For analysis in the design phase, either in simulation or emulation, the problem of generating test sequences arises. In effect, sequences of randomly generated test can lead to a low coverage conditions of activation monitors and, therefore, may be indicative of little satisfaction assertions. The methods of generation of test sequences under constraints do not provide real solution because the constraints can not be linked to temporal conditions. New methods must be specified and implemented, this's what we propose to study in this thesis.

Thèse préparée au laboratoire TIMA (Techniques de l'Informatique et de la Microélectronique pour l'Architecture des ordinateurs)

ISBN : 978-2-11-129171-3