Amélioration de performance de la simulation des modèles décrits en langages de description de matériel

A. Morawiec

▶ To cite this version:

A. Morawiec. Amélioration de performance de la simulation des modèles décrits en langages de description de matériel. Micro et nanotechnologies/Microélectronique. Université Joseph-Fourier - Grenoble I, 2000. Français. NNT : . tel-00002983

HAL Id: tel-00002983
https://theses.hal.science/tel-00002983

Submitted on 11 Jun 2003

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Université Joseph Fourier – Grenoble 1

Thèse
présentée par

Adam Jan MORAWIEC

pour obtenir le grade de

Docteur de l'Université Joseph Fourier

Spécialité : Microélectronique

Amélioration des performances de la simulation des modèles décrits en langages de description de matériel.

Date de soutenance: 26 octobre 2000

Composition du Jury:

Président :	M. Gilbert VINCENT
Rapporteurs :	Mme. Anne-Marie TRULLEMANS-ANCKAERT
	M. Przemyslaw BAKOWSKI
Directeur :	M. Jean MERMET
Examinateur :	M. Philippe BUTEL

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

Remerciements

Je tiens avant tout à exprimer ma très profonde reconnaissance à Monsieur Jean Mermet, qui a assuré la direction de ma thèse, pour ses conseils judicieux, mais surtout pour le soutien, l'aide scientifique et morale qu'il m'a toujours apportés au cours de mes années d'études. Qu'il trouve ici l'expression de mon profond respect.

J'adresse également mes remerciements à Madame Dominique Borrione, pour m'avoir accueilli au sein de son équipe et pour m'avoir exprimé son soutien tout au long de mon travail sur cette thèse.

Des remerciements très spéciaux pour Monsieur Raimund Ubar pour toutes les discussions fructueuses, pour les critiques constructives qui ont contribué au contenu de cette thèse, ainsi que pour toute son amitié et son encouragement. Je le remercie très chaleureusement de m'avoir fait part de son expérience. Je remercie également Monsieur Jaan Raik pour le travail que nous avons mené très étroitement ensemble, et pour l'amitié qu'il m'a montrée.

Je remercie Monsieur Gilbert Vincent, qui a bien voulu me faire l'honneur de présider le jury de cette thèse.

Je tiens à remercier Madame Anne-Marie Trullemans-Anckaert, de l'Université Catholique de Louvain, et Monsieur Przemys³aw Bakowski, de l'École Polytechnique de l'Université de Nantes, qui ont eu la gentillesse de bien vouloir être rapporteurs de cette thèse. Je les remercie très sincèrement pour leurs commentaires et remarques précieuses, et également pour le temps qu'ils m'ont accordé.

Je tiens à remercier Monsieur Philippe Butel, de la société Matra BAe Dynamics, pour l'intérêt qu'il a manifesté à l'égard de mon travail, et aussi pour les conseils et le soutien au cours de ce travail.

Je voudrais remercier Monsieur Wojciech Sakowski, de l'Université Technique de Silésie, avec qui j'ai fait mes premières expériences dans le domaine de la recherche. Je remercie pareillement Monsieur Wojciech Dubiel et Monsieur Adam Bitniok pour leur aide et le soutien qu'ils ont manifesté.

Je tiens à remercier Madame Agnès Silvestre, Monsieur Jean-Pierre Moreau, et Mademoiselle Florence Pourchelle pour leur aide dans l'amélioration de la forme de cette thèse.

Je salue aussi tous les membres de l'équipe VDS, présents et anciens, avec qui j'ai eu le plaisir de travailler : Claude Le Faou, Pierre Ostier, Ayman Wahba, Hakim Bouamama, Julia Dushina, Philippe Georgelin, Emil Dumitrescu, Gérard Vitry, Pierre Pomes et tous les membres du laboratoire TIMA pour la sympathie et l'aide qu'ils m'ont apportées ainsi que pour l'ambiance agréable de travail qu'ils ont créée. Merci également à tout le personnel du laboratoire et spécialement à Madame Corinne Durand-Viel, Madame Isabelle Essalhienne, et Madame Patricia Scimone.

Un très grand et très spécial merci à mes Parents Irena et Ernest Morawiec, et à toute ma famille. Sans leur soutien lointain je ne serais pas arrivé là où je suis.

Je tiens à remercier très profondément Kasia, ma femme et ma meilleure amie, pour sa présence à mes cotés au cours de ces nombreuses années.

W Imię Boże

À Kasia, ma femme
À mes parents

Résumé

La complexité des systèmes électroniques, due au progrès de la technologie microélectronique, nécessite une augmentation correspondante de la productivité des méthodes de conception et de vérification. Une faible performance de la simulation est un des obstacles majeurs à une conception rapide et peu coûteuse de produits de haute qualité. Dans cette thèse nous proposons des méthodes pour améliorer la performance d'une simulation dirigée par événements ou par horloge de modèles décrits en langages de description de matériel.

Nous présentons d'abord les méthodes automatisées d'optimisation et de transformation de modèles VHDL, pour l'accélérer la simulation dirigée par événements. Elles sont fondées sur une analyse précise de la performance en simulation de diverses constructions du langage VHDL, et permettent de convertir le modèle initial en un autre modèle plus efficace, tout en garantissant l'invariance de son comportement. D'autres techniques d'accélération utilisent l'abstraction du modèle : abstraction comportementale, de types de données ou d'objets et permettent de supprimer du modèle des détails inutiles dans le cas d'une simulation particulière. Des outils prototype compatibles avec les simulateurs existants sont développés.

Pour améliorer l'efficacité de la simulation dirigée par horloge, nous introduisons une représentation de la fonctionnalité du système par graphes de décision de haut niveau (DDs). Diverses formes de DDs – graphes vectoriels, compressés ou non et graphes orientés registres – sont définis pour optimiser une représentation du système sur plusieurs niveaux d'abstraction. De plus, de nouveaux algorithmes plus rapides d'évaluation des réseaux de DDs sont développés. Ils emploient, seuls ou en combinaison, les deux techniques de simulation : la technique dirigée par événements et l'évaluation rétrogradée. L'ensemble des prototypes fondé sur ces méthodes permet d'obtenir un gain de performances prometteur par rapport aux outils commerciaux.

Abstract

The growing complexity of electronic systems stimulated by IC's technology progress demands a corresponding growth of the productivity of design and verification methods. The low performance of simulation is one of the obstacles preventing a delivery of high quality products in a short time and at a low cost. In this thesis we propose methods aimed at improving the simulation performance of event-driven and cycle-based simulation techniques of HDL models.

Automated optimization and transformation methods of VHDL models, developed to accelerate the event-driven simulation are presented first. These methods, based on the precise measure of simulation performance of VHDL language constructs, convert an initial VHDL model into another functionally equivalent VHDL model offering a better simulation performance. Other acceleration techniques, denoted as abstraction methods, focus on removing from a model all irrelevant details of its behavior or structure. We propose three such methods: behavioral abstraction, data-type abstraction and object abstraction. Prototype tools compatible with currently used simulators are developed to support automatic application of these methods.

For the purpose of improving of the cycle-based simulation efficiency a representation of a digital system by high-level decision diagrams (DDs) is introduced. Some forms of DDs: vector decision diagrams, compressed or not (VDDs and CVDDs) and register-oriented DDs are developed to optimize the representation of a system at different levels of abstraction. In addition, new simulation algorithms of a network of DDs are proposed to further accelerate the simulation execution. These algorithms implement separately or in combination two simulation techniques: the event-driven and back-tracing techniques. The prototype tools are build, based on the DDs simulator, which allow to efficiently simulate various types of decision diagrams with appropriate simulation algorithms.

Table des matières

Abréviations

ADD	**A**lgebraic **D**ecision **D**iagram
AL-FSM	**A**lgorithmic-**L**evel **F**inite **S**tate **M**achine
ASIC	**A**pplication-**S**pecific **I**ntegrated **C**ircuit
BDD	**B**inary **D**ecision **D**iagram
BT	**B**ack-**T**racing DD simulation algorithm
CA	"**C**ompute-**A**ll" DD simulation algorithm
CAD	**C**omputer-**A**ided **D**esign
CAO	**C**onception **A**ssistée par **O**rdinateur
CHDL	**C**omputer **H**ardware **D**escription **L**anguage
CVDD	**C**ompressed **V**ector **D**ecision **D**iagram
DD	High-Level **D**ecision **D**iagram
DSP	**D**igital **S**ignal **P**rocessing
ED-BT	**E**vent-**D**riven **B**ack-**T**racing DD simulation algorithm
ED-FT	**E**vent-**D**riven **F**orward-**T**racing DD simulation algorithm
FPGA	**F**ield **P**rogrammable **G**ate **A**rrays
FSM	**F**inite **S**tate **M**achine
HDL	**H**ardware **D**escription **L**anguage
IP	**I**ntellectual **P**roperty
MDD	**M**ultivalued **D**ecision **D**iagram
MDG	**M**ultiway **D**ecision **G**raph
MTBDD	**M**ulti-**T**erminal **B**inary **D**ecision **D**iagram
PCB	**P**rinted **C**ircuit **B**oard
RODD	**R**egister-**O**riented High-Level **D**ecision **D**iagram
RTL-FSM	**R**egister-**T**ransfer **L**evel **F**inite **S**tate **M**achine
VDD	**V**ector **D**ecision **D**iagram
VHDL	**V**HSIC (**V**ery **H**igh **S**peed **I**ntegrated **C**ircuit) **H**ardware **D**escription **L**anguage
VIF	**V**HDL **I**ntermediate **F**ormat
VITAL	**V**HDL **I**nitiative **T**owards **A**SIC **L**ibraries
VLSI	**V**ery **L**arge **S**cale of **I**ntegration

Chapitre 1

Introduction

1.1 Problèmes étudiés dans la thèse

Le progrès offert par la technologie de fabrication des circuits microélectroniques a ouvert la voie à la conception de systèmes digitaux d'une grande complexité. La productivité des méthodes de conception de ces systèmes, et plus particulièrement celle de la simulation, joue un rôle important pour assurer une haute qualité au produit final. Elle contribue également à la réduction du coût et du temps de conception du système complet.

Des études, rapports et exemples récents de conception des systèmes sur une puce (en anglais : *system-on-chip*) émanant de l'industrie, par exemple [3, 7, 38, 136], prouvent que l'augmentation de la productivité des outils de simulation influence considérablement l'efficacité du flot de conception des systèmes électroniques, en améliorant la probabilité d'avoir une implémentation correcte dès la première mise en œuvre du système, ainsi que la fiabilité et le temps de conception. En outre, le temps de simulation des systèmes complexes, pour lesquels un grand nombre des vecteurs de test est fourni, limite la productivité globale des concepteurs, car plus de 60% de l'effort de conception est consacré à la phase de vérification.

Tout en s'inscrivant dans le cadre décrit ci-dessus, notre travail de recherche a pour objectif majeur d'améliorer des performances de la simulation des descriptions du comportement des systèmes. Pour atteindre cet objectif, nous proposons de suivre deux directions principales : l'une se concentre sur l'optimisation du point de vue de la performance des modèles existants décrits en langage de description de matériel, tandis que l'autre explore le problème de modélisation et d'exécution des modèles représentés sous forme de graphes de décision de haut niveau. La recherche effectuée nous a permis dans le premier cas de développer des méthodes d'accélération de la simulation dirigée par les événements et, dans le deuxième cas de créer une base appropriée pour la simulation dirigée par l'horloge. Ces deux directions de recherche sont présentées dans la thèse.

La première partie du travail est consacrée à l'accélération de la simulation dirigée par les événements. Nous explorons le potentiel d'améliorer l'efficacité de la simulation par l'optimisation du code du modèle et du style de modélisation utilisé lors de sa création. Ces méthodes permettent d'exploiter les outils de simulation disponibles, répandus dans l'industrie et connus par les concepteurs. C'est pourquoi, dans cette partie de travail, nous nous concentrons plutôt sur les méthodes et les techniques de modélisation que sur le développement d'un nouveau type de simulateur.

Pour définir les méthodes d'accélération, nous nous servons d'une analyse des performances en simulation des différentes constructions du langage de description de matériel. Les résultats de cette analyse nous permettent de définir un ensemble de règles d'accélération du code du modèle initial : les règles d'optimisation et de transformation, qui appliquées au modèle, le rendent plus efficace en simulation tout en garantissant l'invariance de son comportement. Nous proposons ensuite d'autres méthodes d'accélération, fondées sur l'abstraction du modèle : abstraction comportementale, abstraction de types de données et l'abstraction d'objets du modèle. L'abstraction comportementale est basée sur la constatation que pendant une simulation particulière, seulement une partie du modèle est activées ou observée, et donc toutes les parties qui ne contribuent pas à l'évaluation des objets observés peuvent être réduites. Le deuxième méthode d'abstraction – l'abstraction des types de données - permet de transformer dans un modèle les types de données détaillés en types plus abstraits, qui offrent une meilleure efficacité en simulation. Étant donné que les signaux sont beaucoup plus coûteux du point de vue du temps de simulation que les variables du même type, la conversion des signaux en variables permet d'obtenir un modèle plus performant. Les conditions et limites de cette conversion sont définies par l'abstraction d'objets.

Dans le cadre d'une simulation dirigée par l'horloge nous proposons de nouvelles formes de graphes de décision de haut niveau, formes utilisées comme un modèle mathématique de représentation de la fonctionnalité du système. Nous définissons, en outre, des algorithmes pour la création et la simulation des réseaux de graphes de décision. Cette représentation alternative permet, tout en offrant la possibilité de perfectionner le processus d'exécution du modèle, d'obtenir une structure de modèle plus compacte et plus appropriée à une mise en œuvre efficace. Nous introduisons plusieurs types de graphes – les graphes vectoriels, compressés ou non, pour la modélisation au niveau algorithmique, ainsi que les graphes orientés registres pour la modélisation de la partie contrôle, pour adapter et optimiser en simulation la représentation du système. La gestion du processus de la simulation est aussi, l'objet d'une étude concernant les perspectives d'amélioration des performances d'exécution. Cette étude nous permet de développer des algorithmes de simulation du réseau de graphes de décision, qui utilisent les techniques dirigées par les événements avec évaluation en avant, et les techniques d'évaluation rétrogradée, éventuellement dirigée par les événements. Ces algorithmes, combinés à une représentation du système par un type approprié de graphes de décision, accélèrent considérablement la simulation, qui dès lors se compare favorablement aux outils commerciaux de simulation dirigée par l'horloge.

1.2 Plan de la thèse

Le premier chapitre contient une introduction générale des principes de simulation, et d'utilisation des langages de description de matériel dans le flot de conception des systèmes électroniques. Cette partie est suivie par une analyse de l'état de l'art de la recherche sur l'amélioration des performances de simulation, qui concerne principalement deux aspects : la gestion du processus de simulation et la représentation des données pour une simulation efficace.

Après cette introduction, le deuxième chapitre est consacré aux méthodes d'amélioration de la performance du code VHDL en simulation dirigée par les événements. Il s'agit plus précisément de trois méthodes : l'optimisation, la transformation et l'abstraction. Toutes ces méthodes servent à transformer un modèle initial afin de le rendre plus efficace en simulation.

Dans la suite (chapitre 3) nous examinons la méthode de l'abstraction d'un modèle. Celle-ci se divise en trois types : l'abstraction comportementale, l'abstraction des types de données, et l'abstraction des objets d'un modèle. Après avoir donné les principes de ces trois types d'abstraction, nous développons les techniques permettant de les mettre en œuvre. Enfin, nous présentons les résultats d'application de la méthode de l'abstraction à des exemples de circuits venant de l'industrie.

Les deux chapitres suivants (chapitres 4 et 5) sont dédiés à la présentation de la méthode d'accélération de la simulation dirigée par l'horloge. Cette méthode est fondée sur la représentation d'un modèle par des graphes de décision de haut niveau. Ceux-ci sont le centre d'intérêt du chapitre 4. Nous y présentons d'abord l'ensemble des définitions et des théorèmes associés à certaines propriétés de ces graphes, pour ensuite montrer leur utilisation dans la modélisation des systèmes au niveau transfert de registres ainsi qu'au niveau algorithmique.

Les méthodes d'exécution de la simulation d'une représentation donnée sous forme de graphes de décision fait l'objet du chapitre 5. Puisque le processus d'évaluation de la fonction représentée par les graphes de décision possède la capacité d'être accéléré, quatre algorithmes ont été développés afin d'exploiter les diverses possibilités de l'optimisation d'exécution de la simulation.

Dans le chapitre 6 nous concluons et présentons les perspectives de notre travail sur le plan théorique aussi bien que sur le plan pratique.

1.3 Position du problème

1.3.1 Conception et vérification des circuits

La conception de systèmes digitaux d'un très haut niveau de complexité est devenue une réalité dans l'industrie électronique grâce à un progrès rapide dans le domaine de la technologie de fabrication des circuits VLSI. L'augmentation de la taille des circuits et la demande croissante de réduction du temps de conception deviennent un problème d'une importance majeure qui exige un agrandissement similaire de la productivité des méthodes de conception. C'est pourquoi, dans la recherche et le développement, un effort remarquable est fourni pour définir de nouveaux paradigmes et des méthodes de conception qui permettent d'améliorer considérablement le rendement du flot de conception (cf. figure 1.1). La transition vers des niveaux plus élevés d'automatisation de la conception obtenue par l'introduction de la synthèse de haut niveau, par la conception et la réutilisation de blocs préconçus, les composants virtuels, comptent parmi les méthodes les plus importantes pour atteindre ce but.

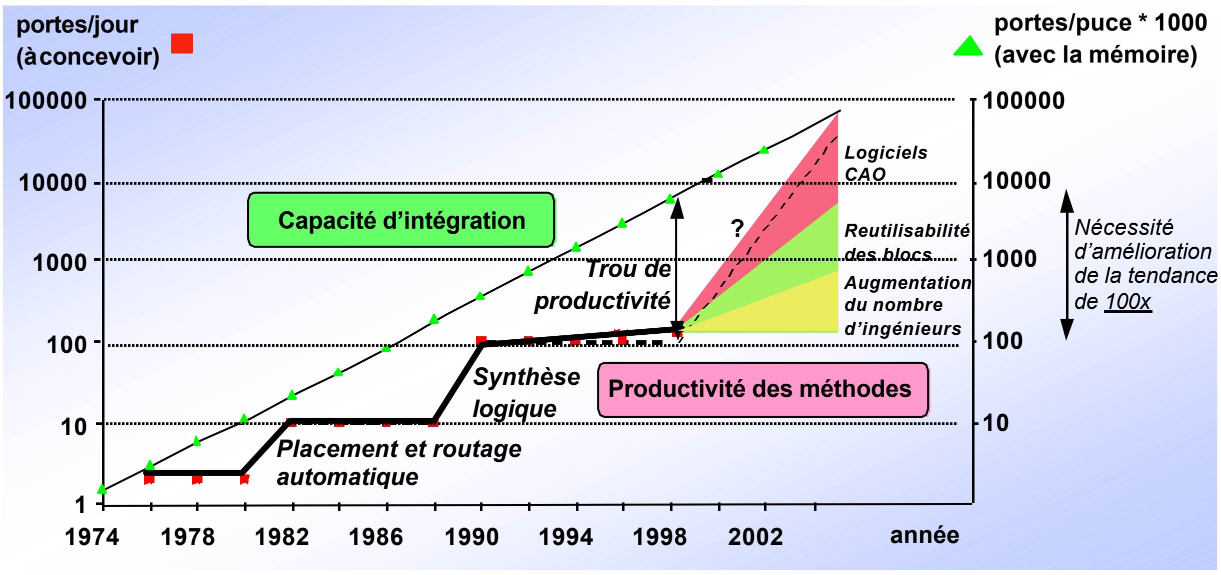


Figure 1.1 : *Divergence de la productivité et de la capacité d'intégration (Source : Alcatel selon McKinsey & Co).*

Étant donné la complexité des circuits, ainsi que celle du flot de conception, il est difficile de s’assurer que les circuits conçus sont corrects. Le temps de vérification augmente de plus en plus pour obtenir un niveau acceptable de confiance en la correction du circuit, et donc la demande de diminution du temps de conception et du coût final ont pour conséquence que la vérification de correction est d’une importance primordiale. La détection tardive d’erreurs de conception est très coûteuse, surtout si le produit est déjà fabriqué, voire commercialisé. Il est donc absolument nécessaire de découvrir toutes les erreurs à partir des toutes premières phases du processus de conception, afin d’éviter la nécessité d’effectuer les itérations supplémentaires des étapes de conception pour corriger ces erreurs. Ce problème prend de l’ampleur dans le contexte de réutilisation des blocs livrés par les fournisseurs des composants de propriété intellectuelle (IP) et intégrés à la suite sur une puce par les concepteurs des systèmes.

Certains problèmes de vérification nécessitent des méthodes spécifiques d’examen. Dans le cas de conception des masques (en anglais : *physical design*) les méthodes qui garantissent l’absence d’erreurs sont la vérification du respect des règles définies de dessin (en anglais : *design rule checking*) ou bien la comparaison des listes d’interconnections.

Aux niveaux de conception plus abstraits que le niveau physique, pour obtenir un fonctionnement correct du circuit final, une démarche appropriée de la vérification devrait être associée à chaque étape de la conception. Deux méthodes complémentaires ont été développées pour répondre à ce besoin : la simulation et la vérification formelle. La simulation permet d’observer le comportement du circuit en développement pour un jeu particulier de vecteurs d’entrée, tandis que la vérification formelle fournit une preuve que certaines propriétés de comportement sont garanties pour tous les vecteurs d’entrée et pour tous les états internes. Il est donc explicable que les deux méthodes soient utilisées dans le processus de conception car elles permettent d’obtenir les objectifs différents de la vérification. Pour la simulation, l’objectif est de fournir une réponse du circuit à des vecteurs d’entrée donnés, qui puisse être observée aux sorties du circuit pour s’assurer que le comportement de celui-ci est correct, c’est-à-dire correspond à sa spécification initiale. L’objectif de la vérification formelle n’est pas d’examiner le comportement particulier du circuit, mais de donner une preuve que la propriété définie dans la spécification est vraie pour toute combinaison d’entrées et dans tout les états internes dans lesquels le circuit peut se trouver.

Dans les deux procédés de vérification, la complexité du circuit à vérifier joue un rôle critique, parce que les méthodes de vérification deviennent inadéquates à la taille des problèmes traités.

Dans le cas de la simulation, pour gagner un niveau suffisant de confiance, et afin de s'assurer que le comportement d'un circuit complexe reflète l'intention du concepteur, il faut d'un côté traiter une structure complexe des données qui représente le modèle du circuit (très souvent avec son environnement) et de l'autre côte exécuter un grand nombre de vecteurs de test pour couvrir le plus grand nombre possible d'états de travail. Cela représente une tâche qui est coûteuse du point de vue de la complexité du calcul et se traduit en conséquence par des temps de simulation très élevés, voire irréalistes dans certains cas extrêmes.

La complexité d'un circuit pouvant être traité par un processus de vérification formelle disponible actuellement est aussi considérablement limitée [102, 103, 106, 107, 159].

La simulation, même si elle ne permet pas la vérification exhaustive du comportement du circuit, n'offrant donc pas une garantie de correction, reste souvent en pratique la seule méthode pour la vérification des circuits complexes.

Quelques exemples industriels, cités ci-dessous, montrent l'importance de la simulation comme moyen de vérification de la fonctionnalité correcte des systèmes :

Pendant le développement des trois circuits ASIC, Alpha (482 mille portes), Beta (824 mille portes) et Gamma (635 mille portes), les plus complexes jamais conçus chez Nortel 63 bogues furent trouvés dans la phase de simulation (6 dans Alpha, 21 dans Beta et 36 dans Gamma) [38]. Un niveau de sévérité fut attribué à chacun de ces bogues : 3 bogues auraient empêché totalement l'utilisation du circuit (même dans la phase du test au laboratoire), 38 bogues exigeaient d'être corrigés avant la fabrication finale (le circuit pouvait être utilisé pour certains tests) et 22 bogues simples ont été découverts. De plus, la phase de simulation était comparée à l'émulation matériel. Selon les résultats, 8 bogues de la totalité des 63 bogues avait une grande probabilité d'être découverts par l'émulation, 21 bogues une probabilité moyenne, et 34 une faible probabilité de découverte dans la phase d'émulation (parmi ces derniers 19 d'une sévérité élevée). Le temps typique de simulation au niveau transfert de registres était 8 jours, au niveau algorithmique 14 heures. La phase de vérification représentait 60% à 70% de l'effort de conception.

Plus de 210 problèmes, dont 32 critiques, ont été détectés par la simulation au niveau algorithmique et la simulation au niveau de cartes pendant la conception d'une série d'ASICs, d'une complexité variant de 20 à 70 mille portes, qui a été développée par Bell Northern Research [135].

Dans le processeur PowerPC conçu par IBM la phase de simulation fonctionnelle a permis de corriger 450 problèmes dans la version 1 du processeur, 480 dans la version 2, et 600 dans la version 3 [3]. Le période de vérification du processeur a duré entre 6 et 15 mois ; pendant ce temps la simulation a été exécutée en continu sur des centaines de machines. Le coût d'une première réalisation en silicium qui varie entre 1 million et 6 millions de francs, montre la nécessité d'obtenir un circuit correct dès la première fabrication prototype.

Siemens a développé un système multiprocesseur qui a été composé de 4 processeurs Pentium et 28 ASICs représentant 2,4 millions de portes logiques. La simulation au niveau algorithmique a permis de trouver et de corriger 320 problèmes, parmi lesquels 150 graves. Les temps de simulation indiqués sont de 23 heures pour chaque 500ì s du temps réel d'opération du système dans le cas de simulation au niveau algorithmique et de 73 heures pour 500ì s du temps réel dans le cas de la simulation au niveau portes logiques [7].

L'industrie de la CAO a une croissance moyenne annuelle de 20.8%. Cependant, les divergences entre les sous-domaines sont importantes : la croissance rapide de la conception au niveau système se traduit par un taux de 27.2%, tandis que la conception traditionnelle au niveau portes logiques représente seulement 8% d'augmentation.

Selon les études menées par DataQuest l'effort consacré à la simulation au niveau algorithmique va croître avec un taux de 26.0% (voir tableau 1.1). La part de la simulation dans la conception système s'élève jusqu'à 25% des ressources globales.

Croissance des étapes de conception au niveau système et algorithmique [1]						
M$	**1997**	**1998**	**1999**	**2000**	**2001**	**Croissance moyenne %**
Niveau système et algorithmique	**84.7**	**107.5**	**137.2**	**174.9**	**223.6**	**27.2**
Conception au niveau système et au niveau algorithmique	42.3	50.8	60.9	73.1	87.8	20.0
Simulation au niveau algorithmique	19.7	24.9	31.4	39.5	49.8	26.0
Synthèse au niveau algorithmique	9.6	13.1	18.0	24.7	33.9	37.5
Vérification formelle	12.5	17.5	24.5	34.3	48.1	40.0
Émulation et accélération au niveau système/algoritmique	0.6	1.2	2.4	3.2	4.1	67.0

Tableau 1.1 : *Croissance des étapes de conception au niveau système et au niveau algorithmique*

[1] Données de DataQuest, Septembre 1997

Les mêmes études montrent (tableau 1.2) que, la croissance venant de la part de la simulation est moins élevée (18.5%) au niveau transfert de registres, mais à ce niveau elle représente un tiers de toutes les ressources dédiées à la conception des systèmes électroniques.

Croissance des étapes de conception au niveau transfert de registres [2]						
M$	**1997**	**1998**	**1999**	**2000**	**2001**	**Croissance moyenne %**
Niveau transfert de registres	**697**	**896**	**992**	**1258**	**1585**	**23.1**
Conception au niveau transfert de registres	28.7	35.8	39.7	44.1	49.0	13.7
Simulation au niveau transfert de registres	303.4	368.0	404.0	484.0	574.0	18.5
Synthèse au niveau transfert de registres	207.8	251.3	271.0	330.2	396.2	18.0
Placement et routage	9.4	16.7	20.9	26.1	32.6	34.1
Conception pour la testabilité	50.2	51.5	57.7	83.7	121.3	28.5
Émulation et accélération	67.5	105.9	108.0	153.4	217.8	34.0
Prototypage virtuel au niveau transfert de registres	5.8	10.6	11.2	23.9	31.0	44.3
Analyse au niveau transfert de registres (timing, consommation, etc.)	39.7	56.3	79.9	113.8	162.7	42.1

Tableau 1.2 : *Croissance des étapes de conception au niveau transfert de registres*

L'augmentation de performance de la simulation répond au besoin de l'industrie concerné par la productivité des méthodes de conception. L'efficacité plus élevée des outils de simulation permet de diminuer le temps global de conception et d'augmenter la qualité finale du produit. Cela peut être atteint grâce à la possibilité de simuler dans un temps de conception restreint un nombre plus élevé de vecteurs de test et ainsi de vérifier le comportement du système dans des cas supplémentaires. Par conséquent la possibilité de repérer les erreurs non découvertes jusqu'à ce moment est augmentée.

1.3.2 Méthodes de vérification par simulation

La simulation est un type de modélisation d'un système qui peut être mis en œuvre sur l'ordinateur. Elle permet de définir la manière dont ce système évolue en fonction du temps, c'est-à-dire le comportement du système, et donc elle peut être comprise comme un processus de modélisation dans lequel une réalité dynamique est imitée grâce à des actions sur ordinateur. Elle sert comme un moyen direct d'observation du comportement hypothétique du système dans des conditions différentes à partir d'un état initial choisi.

La simulation en termes généraux ne permet pas d'établir une preuve de correction du comportement au sens mathématique ou formel, mais elle facilite la compréhension des

[2] Données de DataQuest, Septembre 1997

aspects sélectionnés du comportement du modèle, et aussi des relations entre les composants du système modélisé. L'exécution d'une simulation peut être réitérée plusieurs fois pour gagner en confiance quant au comportement correct du modèle, en examinant différents modes de travail, différentes valeurs des données d'entrée, ou bien différents états initiaux. Par conséquent, la simulation est une technique expérimentale basée sur essais et analyse d'erreurs, qui facilite l'évaluation et la validation des stratégies et des décisions concernant l'architecture ainsi que la fonctionnalité du circuit en cours de développement.

Le comportement réel ou proposé du circuit est modélisé par le biais d'une description du circuit (un modèle) qui peut être exprimée sous la forme de description structurelle du schéma du circuit ou, de plus en plus souvent, en utilisant des langages de description de matériel (en anglais : *computer hardware description languages* – CHDLs, ou simplement *hardware description languages* – HDLs).

La description structurelle sous la forme de schéma électrique est applicable aux circuits imprimés (PCB) ou bien aux circuits ASIC développés de la manière ascendante. Elle consiste en une liste de connexions (en anglais : *netlist*) entre les composants du circuit et elle est créée grâce à l'utilisation de logiciels d'édition de schéma électrique. Deux éléments sont nécessaires pour simuler le circuit : la liste de connections (qui contient l'information sur les entités utilisées dans le circuit et l'information sur la façon dont ces entités sont interconnectées) ainsi que les modèles de simulation des composants référencés dans la liste. Les modèles de simulation sont rassemblés dans les bibliothèques des modèles qui sont fournis par les vendeurs d'ASICs. Cette solution permet de concevoir les circuits au niveau de leur structure et exige que les modèles des composants utilisés soient disponibles.

1.3.3 Modélisation en utilisant des langages de description de matériel

Actuellement, une grande partie des circuits (particulièrement les circuits d'une complexité élevée) est développée en utilisant les langages de description de matériel pour décrire le système au cours d'une ou plusieurs étapes de conception. Les deux langages les plus souvent utilisés sont : VHDL [59] et Verilog [65]. Ils servent de notation formelle à utiliser dans les phases diverses de conception : le développement, la vérification, la synthèse, le test des systèmes électroniques. Cependant, on a développé les deux langages en s'appuyant sur une sémantique de simulation.

Dans le processus de conception il y a plusieurs étapes au cours desquelles les langages de description de matériel peuvent être appliqués :

1. Construction, raffinement et validation des spécifications

Les langages de description de matériel offrent au concepteur un moyen d'exprimer le comportement (compris comme une fonctionnalité avec des relations temporelles) du circuit à la phase de validation des idées ou bien de négociations avec le client. Ici, le concepteur introduit les raffinements apportés à la spécification initiale après avoir pris connaissance de l'information venant de l'examen du code, de résultats de simulation, de résultats de la vérification formelle et d'estimations des caractéristiques de la performance et de la qualité. Ces dernières caractéristiques concernent les mesures telles que le coût, les relations et les dépendances temporelles, la vitesse, la consommation d'énergie, la surface, la productibilité, la testabilité etc., et elles peuvent être considérées et analysées dans le code de la spécification. La spécification du système devrait faciliter la vérification du comportement du système ainsi que la vérification des caractéristiques et des estimations citées ci-dessus. En conséquence, elle devrait contenir les seules informations susceptibles de permettre d'effectuer les analyses correspondantes, tout en favorisant la clarté, la lisibilité et la cohérence du code.

2. Synthèse

A cette phase de conception il est souhaitable, que la description du circuit contienne toute l'information permettant de produire la meilleure réalisation possible à partir d'une spécification donnée. Cette exigence peut être en conflit avec les besoins définis dans le paragraphe précédent, car cela implique l'introduction de détails spécifiques à une des mises en œuvre possibles. Les langages de description de matériel offrent un moyen d'exprimer ou d'introduire ce type d'information et également ils permettent la maintenance d'une documentation complète du projet.

3. Prédiction de fonctionnement

Une fois que la spécification est mise en œuvre sous forme d'une structure d'objets connus interconnectés, il est nécessaire d'avoir la possibilité de prévoir le comportement du résultat. L'application des langages de description de matériel fournissent les constructions qui permettent de décrire le système à ce niveau d'abstraction et de simuler son comportement avec le degré de précision nécessaire.

L'application des langages de description de matériel permet donc de représenter le comportement du circuit sur plusieurs niveaux d'abstraction. Sur chaque niveau le modèle contient un degré souhaité de détails du comportement ou de la structure du circuit modélisé. La transformation entre les différents niveaux d'abstraction peut être réalisée soit par réécriture du modèle, soit par utilisation des outils automatiques de synthèse. Ces derniers devraient donner, comme résultat final, une mise en œuvre optimale pour l'ensemble des conditions données, associées par exemple à la technologie de fabrication. Une des limites de ces outils est l'acceptation d'un sous-ensemble, dit synthétisable, de toutes les constructions du langage de description de matériel. Ce sous-ensemble permet de représenter les concepts et les ressources matérielles, mais il est en même temps privé de mécanismes permettant de représenter certains aspects ou attributs du comportement (par exemple les retards de signaux) ou les concepts abstraits (par exemple certains types de données). Par conséquent, plusieurs modélisations d'un circuit peuvent être développées au cours de la conception pour répondre au besoin de la simulation précise ou de la synthèse automatique.

Langage VHDL

Le langage de description de matériel VHDL permet de décrire le système électronique sous forme d'une hiérarchie de blocs. La racine et les nœuds internes de cette hiérarchie représentent les descriptions structurelles. Les feuilles de la hiérarchie du modèle représentent les descriptions comportementales.

L'unité de conception (en anglais : *design entity*) est l'abstraction de base en terme de matériel dans VHDL. Elle représente une partie du système dotée d'une interface bien définie, par une déclaration d'entité et remplissant une fonction, elle aussi bien définie, par le moyen d'une architecture. Une unité de conception peut représenter un système complet, mais aussi un sous-système, une carte, une puce, un bloc ou une porte logique, quel que soit le niveau d'abstraction intermédiaire. Une unité de conception contient toujours une déclaration d'entité et exactement une architecture. Cependant, une déclaration d'entité peut avoir de multiples architectures spécifiées. Le mécanisme de configuration permet de choisir l'architecture à utiliser parmi celles qui sont définies.

La hiérarchie du système est composée des entités VHDL, qui sont interconnectées, en utilisant le mécanisme d'instanciation d'un composant. Chaque composant peut être associé à une entité définie au niveau inférieur dans le but de définir sa structure ou son propre comportement. Une entité peut être décrite par une hiérarchie de blocs qui contiennent des instructions concurrentes. Un type particulier d'instruction concurrente est l'instruction d'un processus, qui englobe des instructions séquentielles utilisées pour modéliser le comportement du module sous forme d'un algorithme. Parmi les instructions séquentielles, qui sont dans la majorité des cas similaires à celles d'un langage de programmation, on peut en distinguer deux : l'instruction d'affectation de signal et l'instruction **wait**, qui servent à établir une communication entre les processus par un mécanisme d'envoi de messages (en anglais : *passing messages*). Un autre mécanisme de communication concurrente entre les processus, défini par le langage VHDL'93, est l'utilisation de variables partagées.

Le langage VHDL, langage fortement typé (en anglais : *strongly typed*), offre à l'utilisateur des moyens étendus pour définir les types de données. Le mécanisme de paquetage donne la possibilité de réunir des ressources (telles que les déclarations des types ou les sous-programmes) pour qu'elles puissent être partagées par plusieurs entités VHDL.

1.3.4 Simulation

La figure 1.2 présente une vue globale de l'environnement de la simulation dans lequel la description d'un circuit, avec les vecteurs de stimuli, est traitée par le simulateur qui offre un moyen de déboguer le code de la description ainsi que de visualiser et de stocker les résultats de la simulation de ces vecteurs de stimuli.

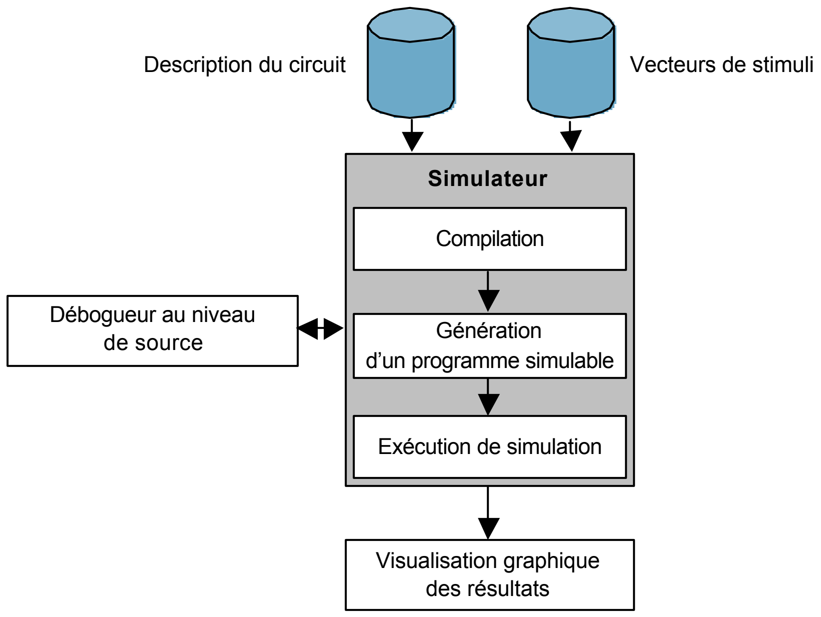

Description du circuit
Vecteurs de stimuli
Simulateur
Compilation
Débogueur au niveau de source
Génération d'un programme simulable
Exécution de simulation
Visualisation graphique des résultats

Figure 1.2 : *Environnement de simulation*

1.3.5 Étapes d'exécution et structure d'un simulateur

L'opération d'un simulateur est constituée de plusieurs étapes logiques qui sont les suivantes :

1. Analyse du code source au niveau syntaxique et sémantique.
2. Génération du code associé à une description donnée en langage de description de matériel.

3. Assemblage du code spécifique à une description donnée et du code du simulateur - noyau de simulateur - qui est une partie commune à toutes les descriptions de circuits dans une technologie particulière de simulation. Cet étape réalise l'association des liens (en anglais : *linkage*) et la lecture des données spécifiques au circuit simulé.
4. Élaboration de la hiérarchie du circuit pendant laquelle est construite une structure de données propre à la description du circuit. Cette structure est utilisée dans l'exécution de la simulation.
5. Initialisation de la structure de données.
6. Exécution de chaque processus de la structure de données jusqu'à son interruption.
7. Exécution des cycles de simulation qui produisent les résultats de simulation.

Les étapes citées ainsi sont censées couvrir différents types de simulateurs acceptant d'utiliser des langages de description de matériel : simulateurs compilés, interprétés ou dirigés par les événements ou par le temps. Ils seront décrits plus en détail dans les paragraphes suivants. Dans certaines mises en œuvre particulières de simulateurs plusieurs de ces étapes peuvent être combinées et effectuées en même temps.

Les opérations décrites dans les étapes 1 à 2 (l'étape 3 est parfois aussi accomplie simultanément avec les étapes précédentes) sont réalisées par un outil dénommé compilateur. Les étapes d'élaboration, effectuées par un élaborateur (l'étape 4), de l'initialisation de la structure de données (l'étape 5) et l'exécution initiale de tous les processus (l'étape 6) sont achevées dans le processus de génération du programme simulable. La structure d'un simulateur spécifiant ces étapes d'opération et ses composants est présentée dans la figure 1.2.

1.3.6 Compilateur

Le premier composant, le compilateur, sert à l'analyse de la syntaxe et de la sémantique d'une description source donnée et, par la suite, à la traduction de cette description exprimée en langage de description de matériel (ici langage de haut niveau) en une représentation d'un niveau plus bas, nommée représentation cible. Cette dernière peut être formulée en utilisant les formats ou les représentations suivantes : du code C, du code assembleur, un code exécutable sur la machine spécifique sur laquelle le code sera directement exécuté, et un format intermédiaire. Le choix du format de la représentation cible dépend de l'ensemble des outils employés dans le simulateur, notamment si le compilateur utilise un outil d'édition de liens (*linker*) ou un assembleur ou s'il génère directement du code exécutable, ou bien si les composants suivants (par exemple l'élaborateur) acceptent un format intermédiaire de données ou du code binaire.

1.3.6.1 Architecture d'un compilateur

Le compilateur comporte cinq composants : un analyseur lexical (*scanner*), un analyseur syntaxique (*parser*), un outil de traitement des contraintes contextuelles, un générateur de code et un outil d'optimisation du code final (démonstration dans la figure 1.3).

Les trois premiers composants forment une phase d'analyse du code source qui a comme objectif de reconnaître si ce code représente une description valable et correcte en langage de description de matériel. La phase finale, dénommée la phase de synthèse, réunit les deux composants restants et se charge de la génération et de l'optimisation du code exécutable. Entre les deux phases de compilation, une structure de données est échangée, nommée format ou représentation intermédiaire. Le format intermédiaire contient toute l'information réunie et déduite pendant la phase d'analyse du code source ainsi que des instructions supplémentaires insérées par le compilateur pour permettre d'accomplir certaines analyses dynamiques de la sémantique. La génération de ce format permet de dissocier la phase d'analyse de la phase de synthèse, et facilite donc la maintenance et le développement indépendant des logiciels pour les deux phases.

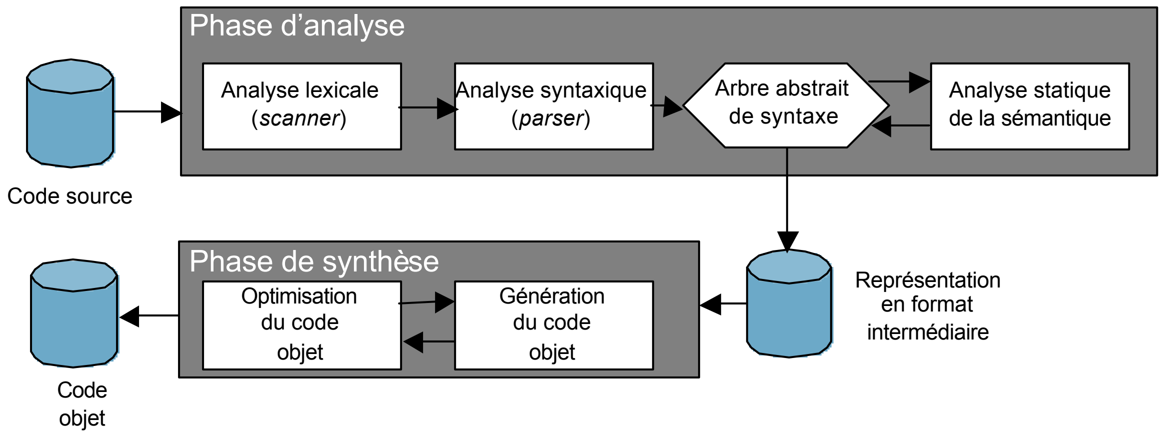

Phase d'analyse
Analyse lexicale (*scanner*)
Analyse syntaxique (*parser*)
Arbre abstrait de syntaxe
Analyse statique de la sémantique
Code source
Phase de synthèse
Optimisation du code objet
Génération du code objet
Représentation en format intermédiaire
Code objet

Figure 1.3 : *Architecture d'un compilateur avec la phase de synthèse pour une simulation compilée*

1.3.6.2 Phase d’analyse

L’analyse lexicale est réalisée par le *scanner* qui parcourt le fichier du code source comme une chaîne de caractères et à partir de cela crée un flux de mots (lexèmes) et de symboles. Le *scanner* emploie des règles lexicales exprimées par un formalisme de la grammaire des expressions régulières pour déterminer les lexèmes qui sont valables dans un langage de description de matériel donné.

Les symboles générés par le *scanner* sont passés au *parser* qui reconnaît la structure des phrases d’un langage source et, utilisant cette information, construit un arbre abstrait de syntaxe. Une partie de la signification d’une phrase dans un langage est associée à la structure de la phrase, c’est-à-dire à l’ordre dans lequel les lexèmes apparaissent dans la phrase. Les règles qui gouvernent l’ordre correct des lexèmes sont connues comme règles de syntaxe. Elles sont analysées par le *parser* qui est aussi appelé *analyseur de syntaxe*. Un ensemble de toutes les règles de syntaxe définit une grammaire du langage.

Un autre type de règles – les règles statiques de sémantique – sert à l’analyse contextuelle de la syntaxe du code. Le but de cette analyse est d’examiner les types et les déclarations pour déterminer leurs portées. Ce type d’analyse introduit les décorations sur l’arbre abstrait de syntaxe.

La dernière étape de cette phase, si elle existe, est la génération d’une représentation du code source en format intermédiaire. Le format intermédiaire est la mise en œuvre d’un arbre cyclique dont les nœuds représentent les objets du langage qui sont utilisés dans le code source. Les nœuds sont définis par les listes des attributs caractéristiques pour un type donné du nœud. Les attributs peuvent contenir une valeur ou peuvent constituer les liens vers les autres nœuds (cf. [75]).

1.3.6.3 Phase de synthèse

Chaque nœud de la description en format intermédiaire est traduit par le générateur du code en une séquence d’instructions du langage cible de la machine sur laquelle le code sera exécuté. L’exécution de ces séquences d’instructions sous contrôle d’une application logicielle nommée simulateur, imite le comportement du modèle. Le matériel sur lequel ce code est exécuté, est censé prendre en compte les différentes contraintes de son architecture et de sa performance. Cela se traduit par les effets importants qui s’imposent sur l’efficacité et le type du code à générer pour le même code source. Pour améliorer le code final, des modifications peuvent être apportées soit au code en format intermédiaire, soit au code d'assemblage avant la génération du code objet.

Pour construire un simulateur pour un circuit donné, il faut associer une partie commune à tous les circuits à simuler, nommée le noyau de simulateur, et une partie propre au circuit. Cette dernière peut être donnée sous trois formes différentes : un code exécutable dans le cas d'un élaborateur compilé, des données interprétées par l'élaborateur ou une combinaison des deux formes précédentes.

1.3.6.4 Code pour l'élaboration

La génération du code, ayant pour but l'élaboration, utilise toute l'information contenue dans le modèle qui est nécessaire pour construire la structure de données, exploitée par la suite dans l'exécution de la simulation. L'information sur chaque objet du langage, déclaré dans le modèle, est introduite dans ce code. Ce code, soit sous une forme exécutable pour un élaborateur compilé, soit sous une forme de données binaires pour un élaborateur interprété, est utilisé dans le processus d'élaboration.

1.3.6.5 Code pour la simulation

Le code pour la simulation contient l'information nécessaire pour exécuter les cycles de simulation conformément à la spécification du modèle. Il s'agit ici par exemple de l'information sur les jeux de stimuli à appliquer : les vecteurs de test qui seront placés aux entrées du circuit en fonction du temps. Cette information sera utilisée pendant l'exécution de la simulation et peut être donnée, comme auparavant, sous deux formes : exécutable ou données binaires.

1.3.7 Génération du programme simulable

1.3.7.1 Processus noyau du simulateur

Le processus noyau (en anglais : *simulator kernel process*) est une partie de simulateur qui est commune à tous les circuits simulés. Le noyau est associé à la partie appropriée du circuit simulé pour construire un programme simulable en utilisant trois méthodes différentes : la première méthode consiste en l'association des liens avec le code objet spécifique au circuit ; dans la deuxième méthode le noyau lit progressivement le fichier contenant l'information sur le circuit, tandis que la troisième méthode est une combinaison des deux précédentes. Cette distinction donne les types compilés ou interprétés de simulateurs.

1.3.7.2 Simulateur de type interprété

Le simulateur interprété contient tout le code exécutable dans le noyau de simulateur. La partie spécifique au circuit n'est pas intégrée au noyau, mais reste sous la forme de données qui sont interprétées dans le processus de simulation.

1.3.7.3 Simulateur de type compilé

Plus le code exécutable est généré uniquement pour le circuit à simuler, plus le simulateur devient un simulateur compilé. Le simulateur entièrement compilé est celui pour lequel on ne peut pas fournir de vecteurs de stimuli après l'étape de sa génération. La construction d'un simulateur de type compilé génère un code exécutable qui reflète le comportement complet du circuit pour le jeu de vecteurs de stimuli fourni.

Les simulateurs de type compilé sont considérés comme plus rapides que les simulateurs de type interprété pour les longues exécutions de simulation car toute l'information sur l'exécution de la simulation est déjà connue au moment de la construction du simulateur. Cependant, chaque changement de vecteurs de stimuli appliqués au circuit simulé déclenche une nouvelle compilation et une reconstruction du simulateur et cela représente un coût supplémentaire considérable, spécialement quand il s'agit de circuits d'une taille importante et d'exécutions courtes de la simulation. Dans ces cas particuliers les simulateurs interprétés offrent le meilleur rendement. Une solution qui peut être un remède à ce problème consiste à appliquer des compilateurs incrémentiels permettant une compilation partielle du code après les modifications mineures, en mettant les vecteurs de test dans une unité « test bench » interfacée avec le module du circuit.

La génération du programme simulable comporte trois phases qui préparent l'exécution de cycles de simulation : l'élaboration, l'initialisation et l'exécution initiale de tous les processus. Ces trois phases sont décrites brièvement ci-dessous :

Phase 1 : Élaboration

L'élaboration est un processus de création, à partir de la description du circuit, de la structure de données exécutée pendant la simulation. Cette structure de donnés sous forme de code exécutable est l'ensemble des processus, créés pour chaque objet de la description initiale, interconnectés en réseaux.

L'élaboration d'une hiérarchie d'entités de conception (en anglais : *design hierarchy*) commence par l'élaboration de l'instruction de bloc externe défini par l'entité et est suivie par l'élaboration de chaque instance de ces composants internes.

Phase 2 : Initialisation de la structure de données

Au début de l'initialisation, le temps courant (le temps simulé) T_c est mis à 0 ns.

Les valeurs initiales sont affectées, à partir des valeurs effectives, à tous les signaux déclarés explicites et implicites (par exemple GUARD ou S'Delayed).

Phase 3 : Exécution initiale des processus

La dernière phase de la construction d'un simulateur est la phase d'exécution de chaque processus de la structure de données une seule fois jusqu'à son interruption. Le temps du premier cycle de simulation est calculé selon les règles des étapes du cycle de simulation (cf. paragraphe 1.1.9.3).

1.3.8 Exécution de la simulation

L'objectif principal de la simulation est la modélisation de systèmes dynamiques, c'est-à-dire les systèmes qui évoluent en fonction du temps. La façon dont l'avancement du temps est modélisé permet de distinguer les principales méthodes d'exécution de la simulation. Nous en distinguons trois : la simulation dirigée par les événements, la simulation dirigée par le temps (en anglais : *time-drive simulation*) et la simulation dirigée par l'horloge (en anglais : *cycle-based simulation*). La première méthode est utilisée dans les simulateurs basés sur les langages de description de matériel VHDL ou Verilog, comme définis par les normes correspondantes [59] et [65]. Certains types de modèles, notamment les modèles des circuits synchrones peuvent être simulés en utilisant la simulation dirigée par l'horloge où l'avance de la simulation est associée au progrès de l'horloge du circuit. La simulation dirigée par le temps est une simulation dans laquelle le temps est incrémenté d'une valeur fixe. Elle est souvent appliquée dans les accélérateurs matériels car elle permet d'activer et d'évaluer simultanément tous les processus et tous les signaux du modèle. Les sections suivantes contiennent de courtes descriptions de chacune de ces méthodes.

1.3.9 Simulation dirigée par les événements

L'élaboration d'une hiérarchie de la description du circuit produit un modèle qui peut être exécuté afin de simuler le concept représenté par le modèle. La simulation implique l'exécution répétitive des processus définis par le modèle qui interagissent, et entre eux et avec l'environnement du circuit. Cette exécution est gérée par le processus noyau du simulateur.

Le processus noyau est une représentation conceptuelle de l'agent qui coordonne l'activité des processus du modèle au cours d'une simulation. Il provoque la propagation des valeurs de signaux explicites ainsi que la mise à jour des valeurs des signaux implicites (tel que GUARD ou S'Quiet). Une activité très importante du processus noyau consiste à détecter les événements qui se produisent quand une simulation évolue. Il est également responsable du déclenchement d'exécution des processus en réponse à ces événements. De plus, le processus noyau maintient le stockage des valeurs des signaux implicites.

1.3.9.1 Pilotes de signaux

Chaque instruction d'affectation de signal dans le modèle génère un pilote (en anglais : *driver*) unique pour ce signal. L'instruction d'une affectation de signal est dite associée au pilote et son exécution au cours de la simulation affecte uniquement le pilote qui est associé à cette instruction.

Un pilote de signal est défini comme une séquence d'une ou plusieurs transactions. Une transaction est constituée d'une composante valeur et d'une composante temps. La composante valeur de chaque transaction définit une valeur qui est supposée être affectée au pilote au moment spécifié par la composante temps de cette transaction. Les transactions dans le pilote sont ordonnées en fonction de leurs composantes temps.

La valeur courante d'un pilote est une valeur de la transaction qui a une composante temps inférieure ou égale au temps courant simulé. A mesure que la simulation avance, le temps simulé peut devenir égal au temps spécifié dans la transaction suivante. Cela provoque une suppression de la transaction courante et la transaction suivante devient la transaction courante du pilote.

Les pilotes de signaux contiennent seulement la prédiction des valeurs possibles des signaux et non pas les valeurs réelles, car ces valeurs peuvent être supprimées au cours de la simulation par le mécanisme de préemption associé au type de retard utilisé dans le modèle : d'inertie ou de transport.

1.3.9.2 Propagation des valeurs de signaux

Au cours de la simulation, le temps simulé avance, en conséquence de quoi les transactions de chaque pilote du signal vont chacune à leur tour devenir les transactions courantes du pilote. Quand un pilote acquiert une nouvelle transaction, qu'elle provoque ou non un changement de la valeur courante de ce pilote, celui-ci devient actif pendant ce cycle de simulation.

Le processus noyau détermine la valeur effective de chaque signal à partir de la valeur courante de son pilote ou bien de plusieurs valeurs courantes, lorsque ce signal a plusieurs pilotes. Les fonctions de résolution, les fonctions de conversion ou les conversions de types peuvent être appliquées afin de calculer la valeur effective. Le mécanisme des fonctions de résolution sert à résoudre les contributions au signal venant de processus différents (pilotes différents) et restant en conflit entre eux. La valeur effective du signal est ensuite utilisée pour mettre à jour la variable contenant la valeur courante du signal. Si cette mise à jour provoque la modification de la valeur courante du signal, un événement est produit sur ce signal. L'apparition d'un événement sur le signal peut provoquer la relance et l'exécution subséquente de certains processus du modèle qui sont sensibles à ce signal, pendant le cycle de simulation dans lequel l'événement se produit. Tous les signaux implicites sont également mis à jour par le processus noyau.

1.3.9.3 Cycles de simulation

L'exécution de la simulation consiste en l'exécution répétitive d'instructions des processus dont le modèle est composé. Chaque exécution de ce type est nommée cycle de simulation ; elle est gérée par le noyau de simulateur. Dans chaque cycle les valeurs de tous les signaux du modèle sont évaluées. Si, comme résultat de cette évaluation, un événement se produit sur un ou plusieurs signaux donnés, tous les processus sensibles à ces signaux, reprennent et sont exécutés en tant que partie du cycle de simulation.

Le cycle de simulation dirigée par les événements se déroule généralement selon les étapes suivantes :

1. Le temps courant T_c est défini comme étant égal au temps du cycle de simulation suivant T_s.
2. Tous les pilotes des signaux sont mis à jour pour le temps courant.
3. Tous les signaux (explicites et implicites) qui sont actifs dans ce cycle de simulation sont mis à jour. De cette opération peuvent résulter des événements sur les signaux.
4. Tous les processus qui sont sensibles aux signaux sur lesquels un événement s'est produit dans l'étape précédente (3) sont exécutés.

5. Le temps du cycle de simulation suivant T_s est déterminé. Il prend la valeur minimale parmi les valeurs suivantes : le temps où un des pilotes devient actif, le temps où un processus reprend ou le temps maximal (cela finit la simulation).

La simulation dirigée par les événements est la méthode la mieux adaptée au simulateurs logiciels car elle permet de traiter de manière séquentielle les processus et les signaux du modèle. Dans cette méthode, on évalue le nombre minimal de signaux, aussi bien que le nombre minimal de processus.

1.3.10 Simulation dirigée par le temps

Dans la simulation dirigée par le temps (en anglais : *time-driven simulation*) le temps simulé est augmenté d'une valeur constante – l'unité d'incrémentation. Au contraire de la simulation dirigée par événements, certains cycles de simulation peuvent ne pas provoquer de changements des valeurs courantes des signaux. La résolution au niveau de temps simulé est égale à l'unité d'incrémentation du temps.

Le cycle de simulation dirigée par le temps se compose des étapes suivantes :

1. Tous les processus sont activés. Pendant leur activation, ils peuvent produire les nouvelles transactions (contributions) aux signaux.
2. Tous les signaux sont évalués pour déterminer leurs valeurs courantes. Les conflits des transactions sont résolus par les fonctions de résolution.
3. Le temps de simulation est incrémenté : $T_s = T_c + 1$ *unité d'incrémentation.*

La simulation dirigée par le temps est une méthode d'exécution de la simulation qui peut être réalisée en tant qu'accélérateur matériel de simulation parce que tous les processus et les signaux de la modélisation peuvent être activés et évalués simultanément (en parallèle). Le point faible des accélérateurs basés sur ce principe est l'incapacité du matériel à détenir simultanément un grand nombre de processus et de signaux du modèle [141].

1.3.11 Simulation dirigée par l'horloge

L'avancement du temps de simulation dans la simulation dirigée par l'horloge est associé à l'avancement de l'horloge du circuit synchrone modélisé. Les valeurs courantes des signaux sont calculées seulement à la fin de chaque cycle de simulation (qui est égal au cycle d'horloge modélisé) et restent inchangées pendant le cycle suivant jusqu'à une nouvelle évaluation. Pour l'évaluation des valeurs courantes des signaux les valeurs des autres signaux sont utilisées et le calcul des retards des signaux est éliminé. Typiquement, la logique utilisée comme une représentation des valeurs des signaux est limitée soit aux deux valeurs : 0 et 1, soit aux quatre valeurs : 0, 1, X (indéfini) et Z (haute impédance).

En comparaison, la simulation dirigée par les événements offre une fonctionnalité très riche en sacrifiant la performance du calcul : chaque signal du modèle est évalué dans chaque module par lequel il se propage. De plus, cette méthode permet d'effectuer un calcul précis du comportement temporel de chaque signal. Différents systèmes logiques peuvent être utilisés pour représenter les valeurs des signaux (par exemple la logique à 9 ou à 28 valeurs). Différents niveaux d'abstraction sont autorisés pour décrire le modèle en commençant par le niveau algorithmique, le niveau de transfert de registres, le niveau portes logiques ou le niveau transistors. La simulation dirigée par l'horloge est limitée à une logique simplifiée des valeurs des signaux et, de ce fait, permet d'optimiser le calcul des fonctions logiques. Par ailleurs, elle ne donne aucune possibilité d'évaluer les retards des signaux.

1.3.12 Mise en œuvre du mécanisme de la simulation

Le simulateur peut être mis en œuvre par :

- un logiciel, qui est exécuté sur une seule station de travail ;
- un logiciel parallèle, qui est exécuté simultanément sur plusieurs stations de travail ou sur un système multiprocesseur ;
- un accélérateur matériel, souvent basé sur un réseau de processeurs spécialisés ;
- une émulation matérielle, dans laquelle le design est mis en œuvre sur un réseau étendu de circuits programmables (FPGA).

Chacune de ces méthodes offre certains avantages et inconvénients du point de vue de la performance, du coût ou de l'applicabilité. La simulation logiciel mono-poste, a été initialement mise en œuvre comme la simulation interprétée, qui a effectué la compilation d'un modèle décrit en VHDL ou Verilog en code des pseudo-instructions. Ce code a été ensuite interprété par le processus noyau du simulateur. Pour augmenter la performance, certains simulateurs accomplissent une compilation du modèle initial en HDL directement sur l'ensemble des instructions du processeur sur lequel la simulation sera exécutée. Les simulateurs compilés sont en général jusqu'à dix fois plus performants que ceux qui travaillent en mode interprété, mais leur efficacité est limitée par le coût de traitement des événements. Ce dernier obstacle est franchi par les simulateurs dirigées par l'horloge ; pourtant leurs application est restreinte aux circuits synchrones définis sous une forme structurelle ou synthétisable. Des simulateurs parallèles, offrent significativement de meilleures performances, que les simulateurs mono-postes (par exemple un gain entre 5 à 7 fois a été signalé en [167]). Leur inconvénient est le coût d'infrastructure de conception. Ce coût est encore plus important en cas de simulation accélérée par matériel, cependant le gain possible peut aller dans cette méthode jusqu'à 100 fois. L'ultime méthode de vérification est l'émulation matérielle, aussi coûteuse, mais permettant d'obtenir une vitesse proche de la vitesse de fonctionnement du matériel réel. Cela se traduit par un gain en performance allant jusqu'à 10000 fois par rapport à la simulation logicielle. Cette méthode ne peut pas être utilisée pour la simulation des modèles algorithmiques temporisés ou des modèles utilisant des constructions autres que synthétisables. Une autre contrainte de l'émulation est liée au fait, que le modèle doit être simulé avec son environnement complet pour assurer une vitesse suffisante d'approvisionnement en vecteurs de stimuli.

1.4 État de l'art

Plusieurs facteurs influencent la performance de la simulation : l'efficacité du code du langage de description de matériel qui représente le modèle du système, le niveau d'abstraction auquel le modèle est décrit, la mise en œvre logicielle du simulateur, le mécanisme de simulation et le modèle mathématique utilisé pour la modélisation et l'exécution du modèle du matériel. Ils comptent parmi les axes principaux sur lesquels la recherche et le développement se focalisent afin d'accélérer le processus de simulation.

1.4.1 Performance du code

Le premier problème, la performance du code HDL, est l'objet de la recherche qui vise à développer des règles de modélisation prenant en compte la performance de simulation. Plusieurs propositions ont été présentées sous forme de guides pour l'écriture de modèles basés sur l'expérience acquise au cours de développement de modèles. VHDL Modeling Guidelines [137] développé par l'ESA/ESTEC (l'Agence Spatiale Européenne) est un document réunissant les règles de modélisation et de documentation uniformes pour le développement, le test, la livraison, la portabilité et la maintenance des modèles. Il définit très brièvement les caractéristiques nécessaires des modèles VHDL développés pour la simulation de composants, la simulation de cartes, la simulation de systèmes et le développement des tests. Dans ce document il est recommandé d'éviter certaines constructions coûteuses du langage VHDL ou de les utiliser avec précaution dans un modèle : ce sont par exemple les instructions de processus, les signaux ou les signaux résolus. Il est aussi conseillé d'utiliser des types abstraits de données tels que les types numériques. L'article [138] propose plus de détails et de conseils sur l'application des méthodes de modélisation décrites ci-dessus.

L'Agence Spatiale Européenne (ESA) a lancé le projet SCADES-2/WP5520 [37] consacré à la comparaison des performances des différents types de simulateurs travaillant en mode interprété ou compilé. La comparaison est fondée sur l'évaluation du temps de simulation et de l'utilisation des constructions du langage de VHDL. Ce travail ne tire pas de conclusions sur la performance des constructions du langage VHDL, mais seulement en ce qui concerne les performances des trois simulateurs spécifiques utilisés dans les expérimentations (Model Techonology V-System v4.3, Synopsys VSS v3.3 en mode interprété et en mode compilé).

Notre travail reprend une idée de base proposée par l'ESA et développe une méthode rigoureuse et automatique d'évaluation des performances des constructions du langage VHDL.

Plusieurs documents ont été proposés, qui contiennent les lignes directrices de la modélisation pour une simulation efficace et qui s'appuient sur des expérimentations spécialisées et effectuées dans un domaine d'application bien précis. Ils comparent des constructions ou des styles de modélisation et démontrent que certains d'entre eux, même s'ils sont fonctionnellement équivalents, n'offrent pas les mêmes performances dans la simulation. Par exemple, Voss et al. [163] démontrent une relation linéaire entre la taille des signaux et les temps de simulation ainsi qu'un désavantage d'utilisation des fonctions de résolution et des fichiers d'entrées et de sorties du point de vue de la performance de simulation. Des méthodes de modélisation des structures répétitives dans un style récursif et par répétition des instructions d'instanciation de composant, ont été analysées par Ashenden [10].

Les suggestions de caractère général, qui concernent la création de modèles performants, sont proposées par Madisetti [82], Levia [79], Hueber [54], Mastretti [86, 87], Balboni [14] et Wicks [165, 166]. Ces travaux réunissent les lignes directrices concernant l'utilisation de variables à la place des signaux, l'application de types de données abstraits, l'expansion en ligne des sous-programmes ou l'ordonnancement des constructions conditionnelles. Un sommaire global de certains de ces travaux est présenté par Pawlak et al. dans [118].

Les travaux cités auparavant définissent les règles de modélisation à suivre au cours du développement du modèle plutôt qu'une méthode de transformation des modèles existants.

Une autre approche, présentée par Z. Navabi et A. Peymandoust [114], définit différentes méthodes de modélisation au niveau transfert de registres pour diminuer le temps de simulation. Ces méthodes permettent de réduire les activités des composants et les buses du chemin de données en remplaçant les interconnections entre les composants par des variables partagées. Le gain obtenu par l'application de cette méthode varie entre 2 et 3 fois. Cette méthode a été proposée seulement comme style de modélisation, et aucune méthode automatique de transformation n'a été proposée.

Peymandoust et Navabi [122, 123] ont proposé un environnement de simulation concurrente qui permet d'exécuter simultanément plusieurs données d'entrée sur le même modèle (le même principe a été déjà utilisé dans la simulation logique). Dans cet environnement, un modèle unique décrit au niveau transfert de registres peut exécuter un programme unique avec les données multiples. Cela permet d'éliminer le coût (compris comme le temps d'opération) associé à la nécessité d'exécuter plusieurs fois le même modèle pour les données qui varient. Particulièrement, l'élaboration des boucles, des tableaux et des listes de données peut bénéficier de cette méthode, car la lecture et l'élaboration individuelle de chaque élément est remplacée par l'élaboration des paquetages de données. Cette méthode utilise le type accès pour modéliser les tailles dynamiques des buses et des interconnections. Elle nécessite la génération du modèle parallèle à partir d'un modèle de départ, qui utilise une bibliothèque des types dynamiques et des composants logiques dynamiques. De plus, elle devrait s'appuyer sur un compilateur concurrent qui génère des données vectorielles pour le modèle concurrent de mémoire.

Une autre méthode pour l'amélioration des performances de la simulation est proposée par Khosravipour [66, 67]. Cette méthode est fondée sur la modélisation hiérarchisée du système afin qu'on ait la possibilité de choisir un composant particulier du système complet à simuler. Pour le composant sélectionné un modèle détaillé est utilisé afin de obtenir un niveau de précision de simulation souhaitable. Par contre, tous les autres composants du système sont représentés sur un niveau d'abstraction plus élevé. Cela permet de supprimer dans leur comportement les détails peu intéressants du point de vue de la simulation du composant sélectionné. Ce travail définit les types d'abstractions qui peuvent être appliqués aux composants : l'abstraction structurelle, l'abstraction des données, l'abstraction temporelle et l'abstraction comportementale, sans se concentrer sur les méthodes automatiques permettant de l'achever.

1.4.2 Performance du simulateur

Une autre direction de recherche est orientée vers l'amélioration de la mise en œuvre des simulateurs. Plusieurs pistes ont été explorés afin de rendre l'exécution de la simulation plus efficace. L'une d'entre elles qui fut expérimentée initialement, se concentre sur la simulation de circuits représentés au niveau structurel. Les primitives des portes logiques, des interrupteurs, ou des mémoires ont été intégrées directement dans le noyau du simulateur, pour diminuer le coût d'exécution de simulation associé à l'évaluation des processus VHDL des portes et des composants de base [139]. Ce principe, commun aux tous les simulateurs logiques, est aussi appliqué dans les simulateurs compatibles avec le standard VITAL [63] et offre une accélération de simulation, tout en gardant la possibilité de modélisation très précise des détails temporels dans une description structurelle de bas niveau.

La simulation distribuée et parallèle est une méthode d'exécution de simulation qui a été étudie par Willis et al. (par exemple [167, 168, 169]). La première approche utilise une idée de partition de la simulation en deux étapes. L'étape initiale débute par l'analyse du code source et se poursuit immédiatement par l'optimisation et l'élaboration de ce code. La deuxième étape est une exécution parallèle de la compilation sur la machine cible, puis de la simulation. Chacune de ces étapes peut être accomplie sur plusieurs nœuds d'un système parallèle, où le nœud peut représenter soit une station de travail, soit un processeur parallèle, soit un système multi-processeur avec la mémoire partagée.

Naroska [112] propose une simulation parallèle qui utilise un algorithme de la simulation discrète pessimiste (conservative discrete simulation), dans lequel les événements de chaque processus sont exécutés selon l'ordonnancement temporel, au contraire de la simulation optimiste, qui exécute tous les événements enregistrés d'un processus sans garantir, que les autres événements (générés par les autres processus) n'ont pas une date antérieure. Un modèle VHDL est divisé en plusieurs partitions, chacune composée de plusieurs processus, qui sont exécutés selon l'ordre des événements produits.

Un simulateur distribué a été développé par Ottens et al. [115], qui réalise une traduction du modèle VHDL en une description en langage C++, description qui est ensuite partagée en plusieurs processeurs en fonction du nombre des connections entre les processus.

Un exemple de l'effort orienté vers une composition de deux méthodes de simulation pour bénéficier des avantages de chacune d'entre elles, est la combinaison de la simulation logicielle dirigée par événements et de l'émulation matérielle. Cette méthode a été proposée par Bauer et al. [15] : elle est basée sur l'utilisation de deux composants : un compilateur, qui réalise la synthèse d'une partie de la description comportementale en description structurelle, et un système reconfigurable de circuits programmables FPGA, dans lequel est située la partie matériel. La communication entre le code comportemental et la partie matériel est assurée par le matériel spécialement synthétisé afin de remplir cette fonction. Une autre méthode, fondée sur l'émulation du système et ensuite, en cas de découverte d'une erreur, sur la simulation, est orientée vers la vérification des systèmes mixtes matériel-logiciel [69].

1.4.3 Méthodes alternatives de représentation et d'exécution des modèles

Une amélioration de la performance de la simulation peut être obtenue par l'utilisation de formes différentes de représentation du comportement du système pour la simulation, qui diffèrent de celles qu'on utilise pour la synthèse. Jusqu'à présent l'application des graphes de décision binaires (en anglais : *binary decision diagrams*, BDDs) et des programmes à embranchements (en anglais : *branching programs*) a été explorée par la recherche pour être utilisée efficacement quant à la performance de simulation. Nous citons par la suite quelques approches proposées.

McGeer et al. [90] proposent une méthode d'évaluation des fonctions discrètes basée sur l'application des graphes de décision à plusieurs valeurs (en anglais : *multi-valued decision diagrams*, MDDs). Dans cette approche, le problème d'évaluation rapide d'une ou de plusieurs fonctions logiques, qui sont représentées par les graphes de décisions à plusieurs valeurs et mises en œuvre par des tableaux hiérarchisés, est étudié. La représentation dans un format MDD est obtenue par une traduction de la représentation par des graphes de décision binaires (BDDs), dans laquelle les variables binaires adjacentes d'un graphe initial sont groupées en variables entières. La nouvelle représentation obtenue de cette manière devient plus compacte, et de ce fait rend plus efficace l'évaluation des valeurs produites par la fonction. Cependant, cette solution peut être appliquée seulement aux fonctions logiques et par suite ne peut être utilisée que pour la modélisation et à la simulation des circuits représentés au niveau des portes logiques.

Ashar et Malik proposent dans [9] une autre approche d'accélération de la simulation dirigée par l'horloge des circuits modélisés au niveau portes logiques. Cette approche utilise une représentation du système par un programme à embranchements (en anglais : *branching program*), qui est isomorphique à un graphe de décision binaires. Pour rendre cette représentation utile à l'évaluation des systèmes à plusieurs sorties, une fonction dite caractéristique est utilisée afin de calculer plusieurs sorties pendant une seule évaluation du graphe. La partition dynamique des sorties parmi plusieurs fonctions caractéristiques permet de représenter le système complet en utilisant un ensemble des graphes de décision à la place d'un seul graphe et de cette façon d'éviter la construction d'un graphe de grande taille. Quoiqu'efficace, cette approche, comme l'approche précédente, ne peut être appliquée qu'aux circuits représentés au niveau des portes logiques.

Une technique hybride d'évaluation rapide de la fonction du système a été développée par Luo et al. [81]. Elle réunit la méthode de simulation par code compilé [1, 47] et la simulation qui utilise le programme d'arborescence [9]. Elle repose sur la structure hiérarchisée du système, qui contient, dans la partie de flot de données, des blocs fonctionnels prédéfinis, tels que multiplicateurs, additionneurs, unités arithmétiques et logiques. Au moment de la création du modèle de simulation du système les blocs fonctionnels sont identifiés et remplacés par leurs fonctions propres (le code compilé à la place de la mise en œuvre au niveau portes logiques). Le reste du système (c'est-à-dire la partie contrôle) est représenté par des graphes de décision binaires, qui maintenant sont plus simples et donc plus compacts. Pendant la simulation, les graphes de décision binaires sont évalués, ainsi que les blocs fonctionnels suivant les valeurs fournies par la partie BDD. Cette technique offre de meilleures résultats que la technique [90], bien qu'elle nécessite l'utilisation d'une bibliothèque prédéfinie de blocs, dont la correction doit être vérifiée avec les autres méthodes de vérification (par exemple la preuve formelle).

1.5 Environnement de vérification

La figure 1.4 présente l'environnement complet de vérification dans lequel sont intégrés les outils d'accélération du modèle et les outils basés sur l'application des graphes de décision de haut niveau proposés dans le cadre de cette thèse. L'ensemble des outils d'accélération (de modification) du modèle contient les outils d'optimisation et de transformation, ainsi que les trois outils d'abstraction, qui mettent en œuvre les méthodes d'abstraction comportementale, d'abstraction de types de données et d'abstraction des objets. A ce niveau, l'environnement offre à l'utilisateur tous les moyens de gérer l'ensemble des modifications qui doit être effectué sur le modèle initial. Ces modifications sont enregistrées afin de garder la trace de tous les changements et de pouvoir reconstituer le modèle au cours des différentes étapes de sa transformation. En s'appuyant sur le format intermédiaire VIF, des outils prototypes ont été développés. La représentation d'un modèle dans ce format est générée à partir du code VHDL après la phase d'analyse de ce code. Les résultats des opérations de modification du modèle sont, eux aussi, données en format VIF. Cela permet de régénérer le modèle accéléré en format VHDL, qui sera ensuite simulé sur un simulateur conforme à la norme du langage VHDL.

La représentation sous la forme de graphes de décision de haut niveau peut être produite, soit en commençant par un format VIF correspondant au modèle initial ①, soit à partir d'un modèle optimisé ②. Cette représentation est simulée dans un simulateur dirigé par l'horloge, qui a été spécialement conçu pour effectuer cette tâche là.

Le choix de graphes de décision de haut niveau comme modèle mathématique pour la simulation, mérite une discussion dans le contexte d'un environnement complet de conception. Cet environnement contient la méthode de conception, ainsi que l'ensemble des logiciels pour toutes les phases de conception, de validation et de vérification, pour la mise en œuvre, mais aussi pour les phases de test et de fabrication de prototypes. Ce choix a donc été motivé par le fait que cette représentation peut être utilisée, non seulement pour la simulation, mais aussi pour la génération automatique de tests, et pour la simulation de défauts de fabrication à partir d'un niveau abstrait de description. De plus, la représentation par des graphes de décision de haut niveau peut être appliquée sur plusieurs niveaux d'abstraction, et ceci à partir du niveau des portes logiques jusqu'au niveau algorithmique. Cela permet de définir un environnement de conception cohérent et complet couvrant plusieurs niveaux d'abstraction et permettant ainsi de développer, en s'appuyant sur le même modèle de représentation, le modèle simulable du système, les vecteurs de stimuli et les vecteurs de test. Cette intégralité de l'environnement de conception répond aux exigences pratiques venant de l'industrie. Les conclusions des travaux de conception de systèmes complexes prouvent [38, 135], que la productivité globale du flot de conception dépend à la fois de l'efficacité de trois méthodes : de la simulation, du développement de vecteurs de stimuli et de test, et de l'application de la méthode de test. La synergie entre ces méthodes, qui peut être obtenue par l'utilisation d'un seul modèle mathématique de représentation du comportement du système au cours de ces différentes phases de conception, répond parfaitement aux besoins d'amélioration de la performance du flot de conception.

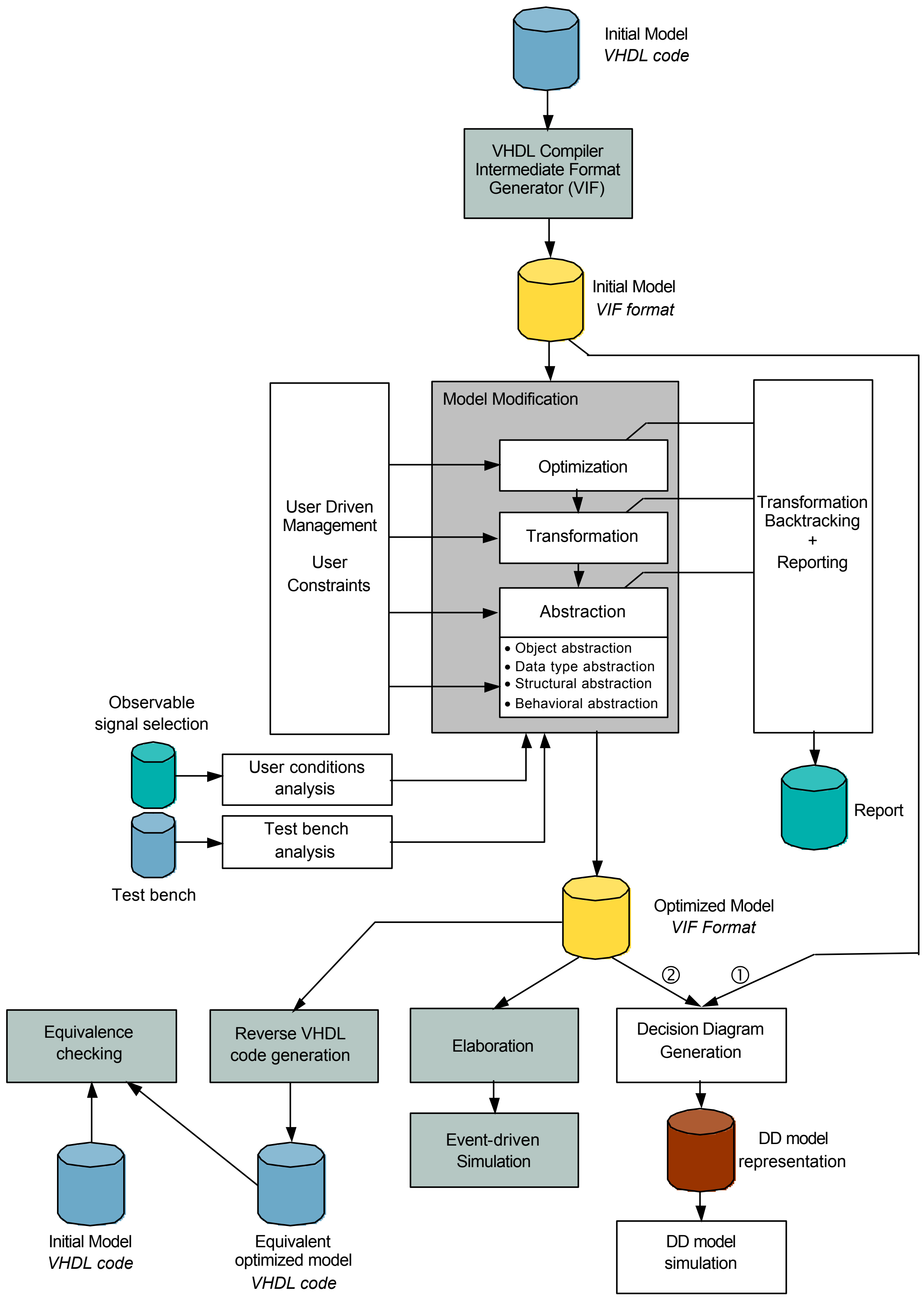


Figure 1.4 : *Environnement complet d'accélération de vérification*

Chapitre 2

Accélération du modèle VHDL

2.1 Résumé

Dans ce chapitre nous introduisons les méthodes d'amélioration de la performance du code VHDL en simulation dirigée par les événements. Trois méthodes ont été développées : l'optimisation, la transformation ainsi que l'abstraction du code VHDL. Ayant utilisé une ou plusieurs de ces méthodes, le modèle initial décrit en VHDL est converti en un autre modèle VHDL. Celui-ci offre une meilleure efficacité en simulation, tout en restant fonctionnellement équivalent au modèle initial.

Les méthodes d'optimisation appliquent des techniques d'accélération des modèles VHDL qui ne changent ni la structure du modèle, ni sa fonctionnalité. Par contre, elles changent ou suppriment toutes les constructions utilisées dans le modèle initial, qui sont inefficaces, voire même superflues.

Par la suite nous proposons les règles de transformation. Celles-ci remplacent certaines instructions ou constructions du langage VHDL utilisées dans un modèle, par les autres instructions qui, sont à la fois fonctionnellement équivalentes, et représentent également de meilleures performances en simulation.

Enfin, l'abstraction d'un modèle repose sur l'élimination de toutes les parties du modèle qui contiennent les détails du comportement ou de la structure du modèle considérés comme inutiles dans une perspective particulière. On peut distinguer quatre axes principaux sur lesquels l'abstraction d'un modèle se déploie : l'abstraction structurelle, l'abstraction de type des données, l'abstraction temporelle et l'abstraction comportementale.

La partie suivante de ce chapitre est consacrée au développement et à l'analyse des styles de modélisation des machines d'états finis du point de vue de la performance de la simulation. Nous décrivons ainsi deux niveaux d'abstraction : le niveau algorithmique et le niveau transfert de registres, tout en proposant également la méthode de modélisation des sorties synchrones (machine de Moore) et asynchrones (machine de Mealy) de la machine d'états finis.

Toutes les méthodes décrites ci-dessus ont été proposées en s'appuyant sur les résultats des mesures de la performance des constructions du langage VHDL. Ces résultats ont été obtenus grâce au développement et à l'application d'une méthode de mesure qui comprend la génération des modèles de test, la mesure précise du temps de simulation ainsi que la gestion automatique de la procédure intégrale de la mesure.

Puisque les méthodes décrites restent indépendantes de l'environnement de simulation, et plus précisément du type particulier de simulateur utilisé, elles peuvent être appliquées automatiquement aux modèles existants, même si ces derniers ont été développés sans tenir compte des problèmes de performance de simulation.

A la fin du chapitre nous décrivons un environnement de la simulation qui permet d'utiliser les méthodes développées. Il s'agit d'un environnement adaptatif qui, d'une part selon les performances de l'outil de simulation utilisé et, d'autre part, selon la configuration de la plate-forme matérielle de la simulation, est capable de sélectionner un sous-ensemble de règles d'optimisation et de transformation les plus efficaces dans les conditions données.

2.2 Acceleration methods [3]

2.2.1 Abstract

The growing complexity of the electronic systems stimulated by the progress in the fabrication technology of integrated circuits requires a corresponding growth of the productivity of the design and verification methods. The low performance of simulation is one of the obstacles preventing a delivery of high quality products in a short time and at a low cost. This section presents three methods developed for improving the simulation performance of VHDL models: the model optimization, the model transformation, and the model abstraction. These methods convert an initial VHDL model into another VHDL model, functionally equivalent, which renders a better simulation performance. The results of simulation time improvement of each method are presented.

2.2.2 Introduction

A faster simulation means a faster detection of bugs present in the design and thus a better final design quality and a lower cost of the development process. The design

[3] Some parts of this section have been presented in [108, 109]

methodology based on the use of hardware description languages depends on a verification methodology involving either formal verification or simulation. Both verification techniques are complementary: the former aims at proving that certain properties of the design behavior hold in each design state, the latter allows one to observe the design behavior under the particular conditions and specified test vectors. Thus, two different design verification objectives can be achieved.

In order to meet the time to market, cost and quality requirements the verification capabilities and performance of a verification process play a crucial role.

In this section three methods for improving the simulation performance are proposed: the model optimization, the model transformation and the model abstraction. The methodology for applying them to the VHDL models have been developed based on the detailed simulation efficiency measures of the VHDL statements and constructs.

All the developed methods allow for the use of existing simulation tools and preserve the initial model functionality as well as the model resolution (however, the latter does not apply to the model abstraction).

The structure of this section is the following: the general approach and the principles of the methods are described in section 2.2.3. This is followed by the presentation of the performance evaluation methodology and the detailed results of efficiency improvement in section 2.2.4. Some conclusions of the work are drawn in section 2.2.5.

2.2.3 Methods description

The methods proposed for improving the simulation efficiency, which are described in this section, convert the initial VHDL model into a corresponding VHDL model which offers better simulation performance.

In the case of optimization and transformation the final optimized model is functionally equivalent to the initial model. However, the model abstraction changes the initial model from the point of view of its behavior and/or structure.

2.2.3.1 Model optimization

Model optimization is a process of transforming the initial model of the circuit in such a way that it does not change either the structure or the functionality of the model. It removes or changes the constructs which are inefficient or unnecessary for the proper behavior of the model. After the optimization process the behavior and the hierarchical structure of the model remain unchanged.

The optimization can benefit from the application of software compilation and optimization techniques:

- input dependency of assignment statements (check if all declared variables and signals are referenced in the model)
- variable lifetime analysis and shortening
- loop unrolling (the number of iterations must be determined at the compile time)
- functions and procedures inline expansion
- branch statement analysis
- common sub-expression elimination

Some results of the simulation efficiency's improvement obtained by using optimization methods are presented in table 2.1.

2.2.3.2 Model transformation

Model transformation consists in a model modification based on the application of transformation rules. These rules define the techniques of transformation of specific VHDL statements or constructs in other corresponding statements/constructs, functionally equivalent, which offer better performances from the point of view of the simulation time. The rules presented in this thesis describe the replacement techniques together with the specific conditions which need to be fulfilled to allow the transformation, as well as the limitations and constraints of their application (cf. [100]). For each rule the anticipated gain obtained trough the replacement is indicated (cf. tables 2.2 and 2.3).

The transformation rules are established based on the theoretical analysis of the VHDL language semantics as well as on the precise measure of the simulation time of each VHDL instruction and construct with the use of two different simulation tools.

Examples of the transformation rules are:

1. Data-structure transformations
   - Array transformation (one- and multidimensional) into the set of variables/signals of the same type
   - Record transformation into the set of variables/signals of the corresponding type
2. Sequential statements transformations

- Replacement of the conditional statement **if-then-else** by the **case** statement
- Loop transformation: **loop/exit**, **while loop**, **for loop**

3. Concurrent statements transformations
   - Replacement of the conditional statement of type "<= **when**" by the statement of type "**with-select**"
   - Transformation of a process with the **wait** statement by a process with the sensitivity list
4. Replacement of the variables and signals by constant values
5. Replacement of generic parameters by constant values

The results presenting the gain in simulation time while applying the transformation rules are summarized in table 2.2.

Some particular modeling techniques have been also evaluated from the point of view of the simulation performance. More precisely, two methods have been exploited:

- The use of shared variables for signal modeling (detailed description can be found in section 3.4)
- The efficient finite state machines representation including the next state and output function representation in Moore and Mealy machines (detailed description can be found in section 2.3)

Some detailed results concerning the gain which can be achieved by the proper modeling of finite state machine are presented in table 2.3.

2.2.3.3 Model abstraction

Model abstraction is a process of withdrawing from a model all irrelevant (from a certain perspective) details of the behavior or structure. It is characterized as a mapping of objects of one class into another, less complex, class.

One can distinguish four principal axes along which the design model abstraction can be achieved: structural abstraction, data-type abstraction, temporal abstraction, and behavioral abstraction [66].

Structural abstraction

The objective of the structural abstraction is to remove the information about the internal design structure. The externally observable behavior of the block remains unchanged.

This type of abstraction can be accomplished by merging the blocs in the internal design structure into a single block for which the external behavior is preserved.

Data-type abstraction

The data-type abstraction is based on the mapping of detailed data types at the implementation level into a more abstract data-type representation. It involves also the mapping of all operations performed on objects of those types. The example of such abstraction is the conversion of the *bit_vector* data type to the *integer* type.

Some specific results of the data-type abstraction are presented in table 2.4.

Temporal abstraction

The temporal abstraction can be achieved at different levels of the model temporal accuracy. It consist of removing the time relevant information from the model by the application of different techniques at various levels:

- detailed delay time modeling removal (e.g. VITAL related data and handling)
- cycle-accurate abstraction (no delay inside clock cycles)
- instruction-accurate abstraction

Behavioral abstraction

The behavioral abstraction allows for partial specification of the entire design while the rest of the model remains undefined (e.g. for certain design states or input values).

The behavioral abstraction of a model is performed by taking into account the requirements of a specific simulation execution. It consists in removing the sections of the model (e.g. blocks, components, signals or variables) which are irrelevant during the present simulation run.

One of the possible ways to practically managing that kind of simulation acceleration is to allow the designer to designate the signals and/or variables which need to be observed in the actual simulation. Using this information all non-observed signals (both internal or external) and corresponding design sections can be removed, which simplifies the model in terms of complexity and memory consumption, and leads to a considerable simulation speed-up. The other way of abstracting the model can be achieved by assigning constant values to the internal variables or signals.

Detailed description of the behavioral abstraction method is presented in section 3.2.

2.2.4 Experimental results

2.2.4.1 Method of experiment

A method has been developed to automatically simulate a comprehensive set of test models and measure the simulation execution time. A C program is used to generate automatically parameterizable VHDL models. Each model generated in that way is devoted to test the performance of a particular VHDL statement or construct. After the phase of generation, the model is compiled, elaborated and simulated. The execution time of each of these phases is precisely measured and stored in a log file. Here, the internal WindowsNT system procedure *GetProcessTimes()* allows to measure the times of the processes launched during the execution of the simulator. The process of generation and execution of the entire suite of models is managed by a specially developed software to enable a full automation, easy extendibility (for new test cases) and portability to different platforms and simulators. The set of scripts enables to launch the entire suite of simulation tests. To give an indication of the complexity of the generation and measurement software: it has more than 19300 lines of C code.

For the purpose of comparison two commercially available simulators have been used: one employing the intermediate input format generated in the compilation phase, the other one generating directly the executable code at a compile time.

The tables presented in this section demonstrate the simulation efficiency ratio for some examples of the methods presented in the previous section: table 2.1 for the optimization methods, table 2.2 for some of the model transformation rules, table 2.3 for the finite state machines modeling techniques and table 2.4 for the data-type abstraction methods. This ratio is a quotient of the time devoted to compile and simulate the considered statement/construct (*column 1*) to the time of the compilation and simulation of the reference statement/construct (*column 2*). The ratio can be understood as a gain, which can be obtained by changing the object indicated in *column 1* to the object indicated in *column 2*. The results are presented for both variables and signals for two simulation tools.

Model optimization methods					
Test	Reference	Simulator 1		Simulator 2	
		Variables	Signals	Variables	Signals
Sequential function/procedure inline expansion					
Function defined in process	Inline statements	9,28	1,23	11,11	1,20
Function defined in architecture	Inline statements	9,46	1,25	11,12	1,20
Function defined in package	Inline statements	8,30	1,23	11,10	1,19
Procedure defined in process	Inline statements	11,70	2,29	13,98	1,14
Procedure defined in architecture	Inline statements	11,99	2,28	13,98	1,14
Procedure defined in package	Inline statements	14,05	2,35	14,13	1,16
Concurrent function/procedure inline expansion					
Function defined in architecture	Inline statements	x	1,10	x	1,09
Function defined in package	Inline statements	x	1,06	x	1,09
Procedure defined in architecture	Inline statements	x	3,78	x	1,10
Procedure defined in package	Inline statements	x	4,51	x	1,39

Table 2.1: *Examples of model optimization methods: function and procedure inline expansion*

The gain which can be obtained by sequential subprogram (function or procedure) inline expansion is important and ranges from 9 to 14 times for variables as parameters and 1.20 (20%) to 2.35 (135%) for signals. For concurrent functions and procedures the gain can be between 1.10 and 4.50. The difference between the gain achieved for the variables and for the signals is caused by the fact that managing the signal objects in the simulation (the creation of a driver, the propagation of signal transactions) is computationally more expensive than the variable management (a simple assignment of a new value). Thus, the computational overhead associated with function/procedure treatment with signals as parameters becomes less significant in the overall computation time.

Moreover, the above results demonstrate also that the simulation tools do not incorporate this kind of optimization into the compilation/elaboration process.

Model transformation rules					
Test	Reference	Simulator 1		Simulator 2	
		Variables	Signals	Variables	Signals
Data types					
One-dimensional Array	Set of variables	1,56	2,36	3,21	1,04
Multidimensional array	Set of arrays	0,92	0,94	7,68	1,20
Record	Set of variables	1,47	2,36	4,92	3,84
Record	Array	1,00	1,00	1,54	3,69
Sequential statements					
If-then-else	Case	1,49	1,09	7,14	10,07
While loop	loop/exit	0,92	0,98	1,08	1,05
For loop	loop/exit	1,41	1,05	1,16	1,11
Concurrent statements					
Conditional assignment (<= when)	with/select	-	1,09	-	8,00
process (wait)	process (sensitivity)	2,40	2,56	2,00	1,16

Table 2.2: *Examples of model transformation rules*

By considering the simulation results of the conditional statements of the type **if-then-else** and **case** one can observe that the former is around 1.5 to 7 times slower for variables and 1.1 to 10.0 times for signals than the latter (depending on the simulator). The semantics of those VHDL constructs is not fully equivalent [59]. Therefore, only some classes of the **if-then-else** statement can be replaced by the **case** statement, which is semantically simpler and can only cover less elaborated behaviors. Conditions and limitations of the transformation rules are presented in detail in [100].

Finite state machine modeling				
Model description	**Simulation**			
	Simulator 1		**Simulator 2**	
	Time [s]	**Ratio**	**Time [s]**	**Ratio**
Moore type FSM				
1 process (both: next state & output)	**2389.3**	**1.00**	**195.3**	**1.00**
2 processes (next state & output)	4879.7	2.04	358.4	1.84
3 processes (synch & asynch out)	7886.7	3.30	839.6	4.30
1 process / 2 tables (next state & output)	2447.5	1.02	220.1	1.13
1 process / 2 tables (next state & output vector)	2807.9	1.18	216.1	1.11
table look-up	3111.4	1.30	1180.4	6.04
Mealy type FSM				
2 processes (clock/state & output)	4910.3	1.82	368.6	1.82
2 processes (next state/output & clock/state)	3023.4	1.12	226.5	1.12
3 processes (clock/state & next state & output)	5654.9	2.10	412.9	2.04
2 processes (next state/output & clk/state) / 2 tables	**2693.6**	**1.00**	203.9	1.01
3 processes (clk/state & nxt state & output) / 1 table	2694.4	1.00	**202.5**	**1.00**

Table 2.3: *Finite state machine modeling comparison*

Several modeling styles used to express the finite state machine behavior have been explored from the point of view of their simulation performance. Those styles differ in the number of VHDL processes used, in the number of choice functions applied (usually **if** or **case** statements) and in the use (or not) of the table look-up technique. The results obtained in the experiments show that for the synchronous sequential circuits the potential gain which can be obtained by the application of the most appropriate modeling style can reach in particular cases 6 times (3.3 times for the other simulator under test). In the case of asynchronous sequential circuits modeling this gain is of around 2 times. The finite state machine modeling methods are presented in detail in section 2.3.

The data-type abstraction (table 2.4) of composite types (e.g. *bit_vector*, *std_logic_vector*) can make simulation of a variable (signal) assignment statement more than 33.0 (12.4) times faster with the use of *simulator 1* and respectively 64.8 (4.9) times with the *simulator 2*.

Assignment statement to object of a given type					
Test	Reference	Simulator 1		Simulator 2	
		Variables	Signals	Variables	Signals
Integer	Bit	1.29	1.11	1.33	1.10
Integer	Boolean	1.28	1.11	1.34	1.11
Integer	Char	1.28	1.11	1.33	1.09
Integer	Enum	1.28	1.11	1.17	1.18
Integer	Std_logic	1.09	0.98	1.06	0.99
Integer	Std_ulogic	1.09	1.01	1.06	1.00
Std_logic	Bit	1.18	1.13	1.25	1.11
Std_ulogic	Bit	1.18	1.10	1.26	1.11
Real	Integer	11.32	1.07	8.90	1.27
Time	Integer	10.78	1.09	9.15	1.03
Bit_vector	Integer	32.13	12.32	64.78	4.94
Std_logic_vector	Integer	32.65	12.34	64.40	4.91
Std_ulogic_vector	Integer	32.30	12.37	64.24	4.88
String	Integer	33.93	12.44	64.71	4.89

Table 2.4: *Examples of the data-type abstraction*

Annex A presents detailed results of the performance measures of VHDL constructs. FSM modeling style descriptions, examples and simulation comparisons are presented in section 2.3 and in Annex B.

2.2.4.2 Application to industrial examples

The above described methods have been applied to some real industrial designs developed by Italtel SpA. The example presented in this section is a HDLC channels link controller design. The results of the simulation of initial model and the models obtained after optimizations, transformations and abstractions are presented in table 2.5.

Acceleration method	Model 1 : fiforx				Model 2 : rxbit			
	Simulator 1		Simulator 2		Simulator 1		Simulator 2	
	Time	Gain	Time	Gain	Time	Gain	Time	Gain
Initial model	2 060	1.000	1 432	1.000	3987	1.000	1 397	1.000
Transformation 1: process merge	2 013	1.023	1 401	1.022	-	-	-	-
Transformation 2: process merge	1 987	1.037	1 367	1.047	-	-	-	-
Object abstraction	1 473	1.399	849	1.685	3290	1.212	1206	1.158
Optimization 1: delete package reference	1 467	1.404	854	1.677	3286	1.213	1206	1.158
Optimization 2: constant instantiation	1 465	1.405	840	1.704	3278	1.216	1168	1.196
Optimization 3: function inline expansion	1 460	1.411	839	1.706	3081	1.294	992	1.408
Transformation 3: if->case	1 458	1.413	832	1.721	-	-	-	-
Optimization 4: delete package reference + const instantiation	1 458	1.413	832	1.721	-	-	-	-
Data-type abstraction 1 (integer type)	1 312	1.570	453	3.160	3042	1.311	968	1.443
Data-type abstraction 2 (integer type)	1 308	1.575	443	3.234	-	-	-	-
Data-type abstraction 3 (bit type)	1 303	1.582	351	4.078	3031	1.315	849	1.646
Data-type abstraction 4 (bit type)	1 308	1.575	354	4.044	2999	1.329	850	1.643
Behavioral abstraction 1: input object invariance	1 214	1.697	342	4.189	2 706	1.473	754	1.853
Behavioral abstraction 2: object observability	1 203	1.712	339	4.226	2 428	1.642	613	2.279
Final result	**1 203**	**1.712**	**339**	**4.226**	**2 428**	**1.642**	**613**	**2.279**

Table 2.5: *Simulation results of transformed and abstracted models of an industrial design.*

The simulation performance improvement obtained by consecutive application of transformation and optimization rules, as well as abstraction methods to the initial models is equal to 71.2% and 322.6% (for *simulator 1* and *2*, respectively) in the case of model 1, and 64.2% and 127.9% in the case of model 2. Note, that some transformations or abstractions do not provide any improvement in the simulation performance of a given model on a specific simulator. This shows the necessity of adapting the set of transformation rules and abstraction methods to each particular simulator (cf. section 2.4).

More details of the application of the model transformation rules and abstraction methods to the real case examples are presented in Annex C.

2.2.5 Concluding remarks

The methods of optimization, transformation and abstraction preserve all the advantages of event-driven simulation. The simulation model, which has been accelerated by means of them, preserves its resolution in the temporal domain, contrarily to other acceleration techniques built upon the cycle-based principle. Moreover, application of acceleration methods is compatible with conventional simulation tools, currently employed by designers. This brings two immediate benefits: the cost of introduction of these methods in the design flow is not high, because it does not require to replace existing simulation tools or to purchase new ones; as a second advantage, the designer can use his experience in writing HDL models and in using the design (simulation) tools jointly with the presented methods. In addition, existing libraries of models (e.g. libraries of reusable IP cores, memories, DSPs or functional units) can also be accelerated.

An important feature of the simulation performance improvement methods, presented in this section, is that they remain complementary to each other, i.e. the most efficient simulation model is obtained if several of these methods are applied consecutively to the model. This is why, in the proposed simulation environment (figure 1.4), all types of accelerations (i.e. optimization, transformation and different forms of abstraction) can be performed collectively on a given model.

The experiments carried out on some industrial designs proved that a significant speed-up of the simulation can be achieved by the application of the methods described in this section, while keeping the event-driven VHDL simulation principle and the existing simulation tools. For example, a speed up of more than 4 times has been obtained by applying some types of optimizations, transformations and data-type abstractions to the HDLC controller design (design *fiforx*).

The systematic observation of simulation performance evaluation results leads to the conclusion that the two simulation tools used in the experiments offer different performances for specific language constructs. Each simulator implements various optimizations for time and memory utilization during the phases of compilation, elaboration and simulation. These optimizations often impose some trade-offs, in the sense that they allow to speed-up some particular types of constructs, while leaving non-optimized or even slowing down the simulation of other constructs. Thus, the fact that particular implementations of the simulation tools provide different performance characteristics, justifies in addition the development of acceleration methods focused on the HDL code improvement. This development remains independent and complementary to the methods devoted to optimize the particular implementation of a simulator.

Furthermore, the work presented in this section can be combined with other efforts aiming at increasing the efficiency of the verification process: e.g. the cycle-based simulation or the exploration of other alternatives proposed for the model representation (e.g. decision-diagram representation – cf. chapters 4 and 5) to further accelerate the simulation execution.

2.3 Finite state machine modeling

2.3.1 Introduction

There exist several different modeling styles to express finite state behavior in a hardware description language model. Those styles have been proposed mainly for the purpose of efficient HDL-driven synthesis process, and as a consequence, they do not necessarily reflect the requirements for efficient simulation modeling, because a model giving correct and efficient synthesis results with the use of a particular synthesis tool, do not automatically provide the best possible simulation performance.

This observation lead to analysis from a simulation performance perspective of some well known finite state machine modeling styles. The results obtained from that analysis permitted the definition of modeling guidelines for efficient simulation as well as a practical method enabling to separate the process of design (definition) of the finite state behavior and creation of simulatable and synthesizable models.

The subsequent sections present the introductory definitions and naming conventions (section 2.3.2), then the modeling styles for the states machines modeled at the algorithmic level (section 2.3.3) and at the register-transfer level (section 2.3.4).

2.3.2 Definitions and terminology

In this section the following definition of a finite state machine will be applied:

***Definition 2.1**: Finite state machine*

A finite state machine is defined as a 5-tuple: $FSM = <I, O, S, \delta, \lambda>$, where:

I is a finite set of input symbols

O is a finite set of output symbols

S is a finite set of states

δ is a transition function defined as $\delta: I \times S \longrightarrow S$

λ is an output function defined as $\lambda: I \times S \longrightarrow O$

◊

A finite state machine that satisfies the conditions of the above definition is referred to as *Mealy machine*. In the design of electronic systems this definition correspond to a sequential circuit with asynchronous type of outputs.

The modification of the output function λ in this definition to $\lambda: S \longrightarrow O$ provides the definition of a *Moore machine*. This type of machine relates to the sequential circuit with synchronous type of outputs.

The abstract syntax of the VHDL used for the definition of modeling styles is shown below [4]:

- Syntactic categories:

$mod \in Models$	$c \in Constants$
$proc \in Processes$	$s \in Signals$
$p \in Non\ Postponed\ Processes$	$S \in Set\ of\ Signals$
$pp \in Postponed\ Processes$	$v \in Variables$
$ss \in Sequential\ Statements$	$sv \in Shared\ Variables$
$e \in Expressions$	$V \in Set\ of\ Variables$
$x \in Values$	

[4] compare [148]

- Definitions:

Model:	mod	$::= \|_{i \in P}\ proc_i$
Process:	$proc_i$	$::= p_i \mid pp_i$
Object:	o_i	$::= x_i \mid v_i \mid sv_i \mid s_i \mid c_i$
Non-postponed process:	p_i	::= while S do ss_i
Postponed process:	pp_i	::= while S do ss_i
Signal assignment:	$sa_i(s)$	::= $s <= e_i$ [after e_i]
Variable assignment:	$va_i(v)$	::= $v := e_i \mid sv := e_i$
Sequential statement:	ss_i	::= null | $sa_i(s)$ | $va_i(v)$ | ss_i ; ss_i | wait on S for e_i until e_i | if e_i then ss_i else ss_i | case e_i is when x_i => ss_i [when x_i => ss_i]
Expression:	$e_i = e_i\ (o_{1,\ldots,}o_n)$ ::= null | o_j | e_i bop e_i | uop e_i where $1 \leq j \leq n$	

A VHDL model *mod* is a collection of processes that communicate with each other by signals and shared variables. In the above definition the operator || indicates the parallel composition of processes, *P* is a finite index set. The operators in expressions are binary (*bop*) and unary (*uop*). The expressions can be either logical or arithmetic.

Using the abstract syntax as proposed above the finite state machine model is represented in terms of:

State variable/signal:	$state ::= s_{state} \mid v_{state}$
Input object:	$in_i ::= o_i,\ in_i \in Inputs$
Output object:	$out_i ::= o_i,\ out_i \in Outputs$
Transition function:	$state = \delta(state, in_1, \ldots, in_n)$
Output function:	$out_1, \ldots, out_m = \lambda(state, in_1, \ldots, in_n)$

In the following sections the algorithmic level and register-transfer level models of finite state machines are presented. These two categories of models are introduced because at different levels of abstraction various modeling styles can be applied with an important impact on the simulation performance.

2.3.3 Algorithmic level finite state machine (AL-FSM)

2.3.3.1 Definition

In the *algorithmic level finite state machine* the transition function is defined as $\delta: I \times S \rightarrow S$, where $state = \delta(state, f(in_1, ..., in_n))$. The transition function derives the value of the next state object as a function of the current state and a function $f = f(in_1, ..., in_n)$, which is a function defined on input objects, and denoted as *input function*. Similarly, the *output function* is defined as $\lambda: I \times S \rightarrow O$, where $h(in_1, ..., in_n, out_1, ..., out_m) = \lambda(state, g(in_1, ..., in_n))$ and the function $h: Inputs \times Outputs \rightarrow Inputs \times Outputs$ is denoted as *action function*.

The output function defines the actions performed on the objects of the model (those actions are denoted by the action function *h*) as a function of current state and of the function $g(in_1, ..., in_n)$ defined on input objects.

Usually, in a finite state machine model the input functions *f* and *g* are the condition checking functions defined on input objects. Thus, in the case of the transition function, for all $o_i \in I$ and $1 \leq i \leq n$, $f = e(o_1, ..., o_n)$.

The action function $h(in_1, ..., in_n, out_1, ..., out_m)$ represents a set of data-path operations performed on objects of the model (both input and output objects): $h(in_1, ..., in_n, out_1, ..., out_m) = ss_i$. Every set of data-path operations to be performed in a specific state of the finite state machine is executed when a particular condition defined on the state object (*state*) and the input objects (function *g*) holds. Thus, the function *h* takes one of the following forms:

h ::= if $g(in_1, ..., in_n)$ then ss_i else ss_i |

| case $g(in_1, ..., in_n)$ is when x_i => ss_i [when x_i => ss_i]

2.3.3.2 AL-FSM modeling styles

In the experiments, 11 different modeling styles of algorithmic level finite state machines have been examined. Based on the results obtained in the simulation, we have selected four modeling styles to be used for the modeling purposes at the algorithmic level. Those styles differ in a number of VHDL processes used in the description as well as in the type of usage (if any) of the table look-up technique.

Below a short description of each style is provided. More details about each modeling style along with an example illustrating the application of the style to a concrete design can be found in Annex B.

Style 1.1

The *style 1.1* description uses a single process sensitive to the clock signal. Both transition (δ) and output (λ) functions are incorporated in one main choice statement of the **case** type that selects the values of the state object. The **if-then-else** statements implement the input function (here $f = g$) for each state value. Each set of sequential statements of the **if-then-else** statement represent the transition and action (output) functions. In this modeling style the table look-up technique has not been applied.

Style 1.2

In the model of a *style 1.2* kind two distinct choice statements are applied in a single process sensitive to the clock signal. The first of those statements implements the input function f, while the second describes the behavior of the action function h. The transition function is realized with the use of a look-up table with the entries of the state object and input function.

Style 1.3

The *style 1.3* differs from the *style 1.2* presented above only in the implementation of the input function f. Here, a second table is used to represent the relation on input objects.

Style 1.4

Two processes are declared in the *style 1.4* description. The first of them, sensitive to the clock signal, describes the action function h. The choice statement allows to define the action function for each value of the state object; the input function g is used to model the actions to be performed in the function of input objects. The second process is sensitive to the state signal and all input signals. Its role is to define the transition function, which is dependent on the input function f.

2.3.3.3 Simulation results

Table 2.6 below presents the comparison of the simulation time of the selected modeling styles. Other styles that were explored in the modeling experiments, presented in Annex B, represent a significant penalty in simulation performance comparing to the *styles 1.1-1.4*.

In this table, the column entitled as *# of processes* indicates the number of process statements used in the given style together with the information (in parenthesis) what functions are combined in each process. The next column shows, besides the number of look-up tables applied in the model, what kind of function has been represented in the table (e.g. *state* indicates the transition function). Finally, the results of simulation of an example model (a *gcd* model, cf. Annex B for details) are presented for two simulation tools. The first tool generates an intermediate format representation and then compiled code, the other one is a direct compiled code simulator. The column *Ratio* presents the ratio to the fastest model for a given simulator.

The difference in the simulation time between various AL-FSM modeling styles do not exceed 19% in the case of the first simulator and 63% in the case of the second one (cf. Annex B for all results).

Note: *Style 1.4* has been presented in the comparison despite its performance disadvantage, since it is used in the modeling for synthesis purposes.

AL-FSM Modeling Style				Simulation results			
Style	# of processes	# of tables	# of choice statements	Simulator 1		Simulator 2	
				Time [s]	Ratio	Time [s]	Ratio
1.1	1 (clk+δ+λ)	0	1	2 093 840	1.05	320 361	1.00
1.2	1 (clk+δ+λ)	1 (state)	2	2 090 697	1.04	338 878	1.06
1.3	1 (clk+δ+λ)	2 (state, in)	1	2 001 598	1.00	351 605	1.10
1.4	2 (clk+λ, δ)	0	2	2 341 828	1.17	469 525	1.47

Table 2.6: *Comparison of AL-FSM modeling styles*

2.3.4 Register-transfer level finite state machine (RTL-FSM)

2.3.4.1 Definition

In the register-transfer level finite state machine the transition function is defined as $\delta: I \times S \longrightarrow S$, where $state = \delta(state, in_1, \ldots, in_n)$. The transition function derives the value of the next state object as a function of the current state and the current values of input objects $in_1, \ldots, in_n$. The output function is defined as $\lambda: I \times S \longrightarrow O$, where

$$out_1, \ldots, out_m = \lambda(state, in_1, \ldots, in_n)$$

The output function defines the values of the output objects of the model as a function of current state and the current values of input objects $in_1, \ldots, in_n$.

2.3.4.2 RTL-FSM modeling styles

The finite state machine behavior is described by a process, in which actions to be performed in each state are declared. This process, modeling the transition function, is activated by a single clock signal and there is a state signal or variable that models the state object (i.e. register in the final implementation). The state transitions are modeled as assignments to the state object of an enumerative type. The output signal values are updated by the signal assignment statements (in the same or in a separate process) and these outputs are either synchronous or asynchronous, what corresponds respectively to the Moore or Mealy type of FSM.

In the modeling experiments, 20 different modeling styles have been proposed and analyzed for the register-transfer level finite state machines.

The proposed modeling styles offer different event-driven simulation performances and differ basically in terms of:

- capacity of modeling Moore or Mealy type of machines (synchronous or asynchronous outputs)
- number of VHDL processes used in the description (between 1 and 3)
- usage of one or two objects (signals or variables) for the state representation
- usage of table look-up technique for the transition and/or output function
- results of synthesis

Almost all of these styles (except those using multidimensional arrays) are synthesizable, accepted by a majority of the synthesis tools (cf. [162]).

The choice of one of those modeling styles for a given problem has a great impact on the synthesis process results. The difference of 20% in the size (number of gates) of the implementation is reported in [162]. The selection of the appropriate modeling style has also an important influence on the efficiency of the code simulation.

The following sections describe each of those styles. Annex B presents details, examples and simulation results for the styles proposed in this section.

2.3.4.3 Moore machine modeling

Style 2.1

In the description *style 2.1* all elements of the finite state machine are described by a single process p_i activated by the clock signal. One main choice statement of the **case** type incorporates both transition (δ) and output (λ) functions. For each value of the case expression (denoting an actual state) an **if-then-else** statement is declared (when required) to implement the FSM behavior as a function of input object values. Each set of sequential statements of the type sa_i or va_i of the **if-then-else** statement represent the transition and output functions.

Style 2.2

A single process statement p_i sensitive to the clock signal is used to describe the FSM behavior in the *style 2.2*. Two look-up tables are declared (as array constants): one for the transition function and the other one for the output function. The current values of the state variable and output signals are derived in two distinct variable assignment statements *va(state)* and *va(result)* from the corresponding tables, for which the entries are the FSM input signals. *result* is the variable of a compound type which acquires the current value of all output signals, and which is in turn used as a source in the signal assignment statements $sa_i(out_i)$ to assign current values to the outputs $out_1, ..., out_n$.

Style 2.3

The *style 2.3* is a modification of the *style 2.2* with the only difference that in the entire model, including the control part modeled as FSM and the data path, the separate signals being the outputs of FSM are combined into one compound signal. Thus, separate signal assignment statements $sa_i(out_i)$ are no more required in the model.

Style 2.4

A single look-up table, which is an array of record objects, is used as a representation of both functions: transition and output. A single evaluation in the variable assignment statement $va_i(result)$, where *result* is of the record type, executed in a single process p_i for every change of the clock signal, is necessary to obtain the current values of the state variable and outputs of the machine. Variable and signal assignment statements are used to assign the current values to the state variable and to output signals respectively.

Style 2.5

The *style 2.5* involves defining the output function in one process p_1 sensitive to the clock signal, and the transition function in a second process p_2 sensitive to state and input objects. Two distinct signals *state* and *next_state* are required to model the states. In p_1 a state transition statement of type *state <= next_state;* is provided to model the state change activated by the clock, while the next state value is evaluated in p_2.

Style 2.6

In this style a unique look-up table is declared, in which each row is composed of two parts: the entry part (1) with the input/current state and the result part (2) containing the output/next state values. This table is a representation of the finite state machine in the form of a transition table as defined further in this section. The next state value and the outputs values are obtained in the process of scanning the entry part of the table. Once the current input values and the present state value match with the entry part, the result part is fetched from the table. The result part contains the next state value along with the output signal values. The scan function is implemented in the model as a *for* loop, which represents a significant drawback in the simulation performance.

2.3.4.4 Mealy machine modeling

Style 3.1

Two processes p_1 and p_2 are used in the *style 3.1*: the process p_1 sensitive to the clock signal defines the state transition function, while p_2 is activated by the state signal and all input signals; it performs the assignment to the outputs with separate signal assignment statements $sa_i(out_i)$.

Style 3.2

The *style 3.2* involves specifying the transition function as well as the output function in one process p_1 sensitive to the state and input signals, whereas the second process p_2 activated by the clock contains only the state transition statement *state <= next_state*. With this partition, the combinational part (p_1) is separated from the sequential part (p_2), which enables to model an asynchronous behavior. As in the *style 2.5*, the state object is modeled by two signals: *state* represents the current state and *next_state* represents the subsequent state.

Style 3.3

Similarly to the *style 3.2*, the transition and output functions are specified in one process p_1, while the second process p_2 contains the state transition statement of the type *state <= next_state*. To represent both transition and output functions two tables are declared. In p_1 current values of state and input objects are employed to evaluate, in the variable assignment statements *va(state)* and *va(result)*, the next state value as well as the outputs from the corresponding tables. Two distinct signals: *state* and *next_state* are declared to model the states.

Style 3.4

Here, the finite state machine is described with three processes, which model the state transition statement (p_1 sensitive to clock), the transition function (p_2 sensitive to state and input signals) and the output function (p_3 again sensitive to state and input signals). As in the previous styles two signals are required to model the state object.

This style can be extended to model asynchronous and synchronous outputs in the same finite state machine model. This extension can be achieved by adding to the process p_1 the synchronous output function, while the process p_3 contains the asynchronous output function.

Style 3.5

This style is based on the *style 3.4* with some difference in the specification of the transition and output functions. Here, these functions are represented by a single table instead of the **case** / **if-then-else** statements. The table is queried twice: in the process p_2 to derive the next state value and in the process p_3 to obtain the output signal values.

2.3.4.5 Simulation results

In table 2.7 a comparison of the simulation results of the modeling styles for register-transfer level finite state machines are presented [5].

From the results obtained in experiments one can conclude that a unique look-up table technique with the scanning function is highly inefficient in the simulation (the difference in the simulation time to the most efficient modeling style is of 30% for the first simulator and more than 500% for the second one). The other modeling styles for Moore and Mealy machines differ up to 27% in terms of simulation time for the *gcd* example. The simulation experiments performed on other examples (presented in Annex B) having up to 10000 states show differences between those styles up to 330% for the Moore machine models and 110% for the Mealy machine model.

[5] The meaning of symbols and notation used are similar to the table 2.6.

RTL-FSM Modeling Styles				Simulation time			
Style	# of processes	# of tables	# of choice statements	Simulator 1		Simulator 2	
				Time [s]	Gain[%]	Time [s]	Gain [%]
Moore machine							
2.1	1 (clk+δ+λ)	0	1	11 304 575	1.00	1 692 934	1.00
2.2	1 (clk+δ+λ)	2 (state, out)	0	11 595 944	1.03	1 907 553	1.13
2.3	1 (clk+δ+λ)	2 (state, out)	0	13 285 113	1.18	1 873 113	1.11
2.4	1 (clk+δ+λ)	1 (state+out)	0	11 609 574	1.03	1 962 121	1.16
2.5	2 (clk+λ, δ)	0	2	11 468 651	1.01	1 740 753	1.03
2.6	1 (clk+δ+λ)	1	1 (in loop)	14 721 125	1.30	10 231 842	6.04
Mealy machine							
3.1	2 (clk+δ, λ)	0	2	5 389 289	1.12	822 923	1.12
3.2	2 (δ+λ, clk)	0	1	5 396 710	1.12	868 059	1.18
3.3	2 (δ+λ, clk)	2 (state, out)	0	4 808 054	1.00	740 745	1.01
3.4	3 (clk, δ, λ)	0	2	5 368 790	1.12	835 832	1.14
3.5	3 (clk, δ, λ)	1 (state+out)	0	4 809 486	1.00	735 698	1.00

Table 2.7: *Comparison of RTL-FSM modeling styles*

2.3.5 Design methodology implementation

We propose a modeling methodology which is based on the approach of detaching the process of defining a finite state machine behavior from the process of creating of models for the purpose of the simulation and synthesis (figure 2.1). This approach will enable to define independently the function of a finite state machine in the general canonical form, which in turns, will serve to generate in the automatic way the corresponding simulation and synthesis models. The generation process of both models can be driven by particular requirements and conditions imposed by the designer and/or by the simulation and synthesis tool(s). In the case of simulation, the selection of the most appropriate modeling style will be determined by the simulation efficiency of the proposed styles established for the given simulation tool and for the specific simulation environment (operating system, hardware configuration, etc.). This solution offers several advantages:

- the possibility to extend the set of modeling styles (a new modeling style can be added together with the appropriate generator from the canonical form),
- the ability to adapt the modeling style to a particular simulator and a particular simulation environment, and
- the efficient management of simulation and synthesis models (the changes introduced in the simulation model can be brought back into the synthesis model in an automatic way via the canonical representation).

The canonical form of the FSM is represented by a transition table as defined in [18]. The transition table representation of a finite state machine displays the next state and output functions in tabular form. The rows of the table correspond to the possible states of the machine, while the columns correspond to the possible input symbols in the case of RT-level FSM (or to the function *f* defined on the input symbols in the case of algorithmic level FSM). The entry found at the intersection of the *i*-th and the *j*-th column is the transition function $\delta(state, in_1,\ldots, in_n)$ and the output function $\lambda(state, in_1,\ldots, in_n)$. The general form of this table is illustrated in table 2.8.

		Present input		
		in_1	□	in_n
Present	s_1	$\delta(s_1, in_1) / \lambda(s_1, in_1)$		$\delta(s_1, in_n) / \lambda(s_1, in_n)$
state	□		next state / output	
	s_S	$\delta(s_S, in_1) / \lambda(s_S, in_1)$		$\delta(s_S, in_n) / \lambda(s_S, in_n)$

Table 2.8: *General form of the transition table*

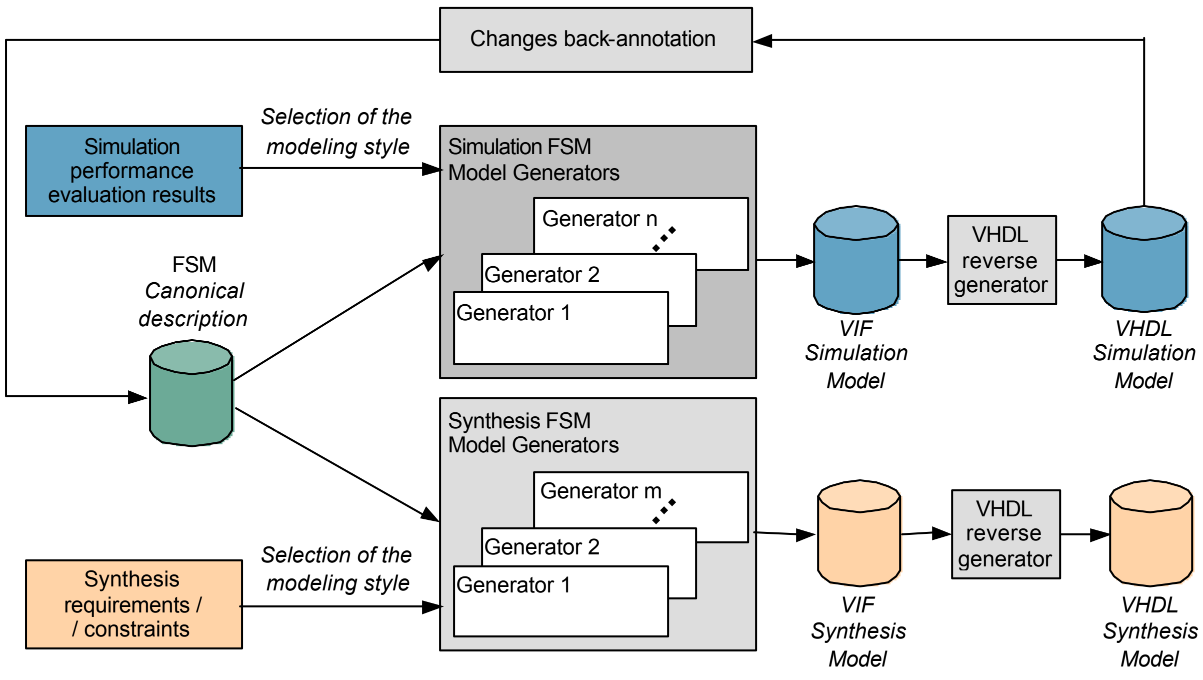

Changes back-annotation
Selection of the modeling style
Simulation performance evaluation results
Simulation FSM Model Generators
Generator n
Generator 2
Generator 1
FSM Canonical description
VHDL reverse generator
VIF Simulation Model
VHDL Simulation Model
Synthesis FSM Model Generators
Generator m
Generator 2
Generator 1
Selection of the modeling style
Synthesis requirements / / constraints
VHDL reverse generator
VIF Synthesis Model
VHDL Synthesis Model

Figure 2.1: *Simulation and synthesis model generation from the finite state machine definition in canonical form*

2.3.6 Concluding remarks

This section proposes and analyses different modeling styles to be applied to model efficiently finite state machines at two levels of abstraction: the algorithmic level and the register-transfer level. The experimental results show that a well-chosen modeling style can increase the simulation performance more than 60% in case of modeling algorithmic FSMs, and more than 300% in modeling of register-transfer level FSMs.

The modeling styles introduced for the improvement of simulation performance are not necessarily compatible with the requirements of HDL-driven synthesis. This is why an automated management method has been proposed to assist the development of models for the purpose of simulation and synthesis. This method enables the definition of finite state behavior in a canonical form, which is then translated into an HDL model for simulation or synthesis, according to the next design action. In order to ensure the consistency of all models, every change made in the simulatable model, as an effect of error corrections in the simulation phase, has to be automatically introduced (via the *changes back-annotation* module, figure 2.1) to the canonical representation of the finite state machine.

2.4 Environment for performance evaluation of simulation tools and for transformation rules adaptation [6]

The particular efficiency improvement results obtained in the evaluation of the simulation time of VHDL constructs are strongly dependent on the compilation and simulation tools used. The reason for this is that different implementations of the compiler and/or simulator are associated with different optimization techniques incorporated in the tools at several steps of the model processing: starting from the parsing and compilation with or without the use of the intermediate format, throughout the data structure used for elaboration, the elaboration process, to the implementation of the simulation algorithm. In order to generalize the performance improvement methods presented here, it is possible to adapt the set of optimization and transformation rules to a particular simulation system.

The procedure to achieve this goal is the following: the set of generic optimization and transformation rules is defined regardless of the simulation system to which it will be applied. For each new compilation/simulation tool a set of test models is executed which determines the particular measures of the compilation and simulation efficiency. Based on the results obtained from those test executions, an adapted set of optimization/transformation rules is selected, which is then applied to simulated models. This general approach enables the saving of model processing time, which does not necessarily imply a significant gain in simulation efficiency on a specific simulator. The entire technique is illustrated in figure 2.2 in which the main components of the tool adaptation environment are depicted.

[6] Some parts of this material has been initially presented in [100, 109]

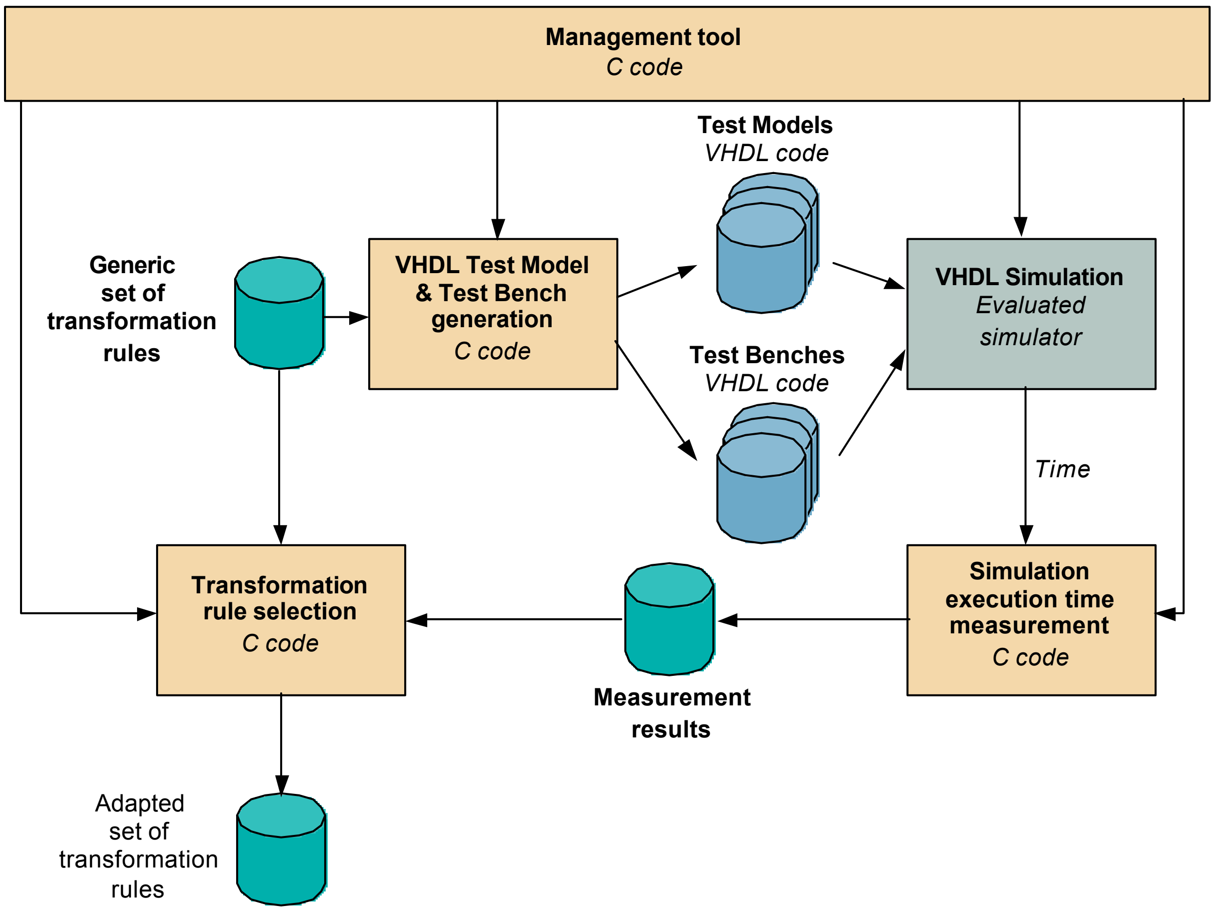

Management tool
C code
Test Models
VHDL code
Generic set of transformation rules
VHDL Test Model & Test Bench generation
C code
VHDL Simulation
Evaluated simulator
Test Benches
VHDL code
Time
Transformation rule selection
C code
Simulation execution time measurement
C code
Measurement results
Adapted set of transformation rules

Figure 2.2: *Tool adaptation environment*

2.5 Prototype tools

The model acceleration methods, presented in this chapter, have been proposed based on results obtained in a process of a detailed analysis of simulation performance of VHDL language constructs. To make this process practically realizable, a set of prototype tools has been designed and developed. Its main objective is to perform in an automatic way the following tasks:

- generation of test models
- execution of a compilation and/or simulation tool for a given model
- precise time measurement of the execution time
- storage of measurement results in a log file

2.5.1 Model generation

A model generation tool is devoted to produce a comprehensive set of test models. Each model generated by this tool is used to test a particular VHDL language construct in different configurations. Thus, the generation tool enables the parameterization of the generated model, in which one or many of the following parameters can be selected (if appropriate):

- the number of tested constructs of a given type
- the number of loops in which a given construct is simulated
- the number and type of objects to be affected (ports, signals, variables and/or constants)
- the value to be assigned in the assignment statement
- the dimension of objects of a compound type (i.e. arrays or records)
- the number of design hierarchies
- the branching conditions in a branching statement

The ability to set some parameters for the model allows to generate the customized models of a specific size, and in this way to influence indirectly the order of magnitude of simulation execution time.

The automatic generation of test models, implemented as a C program, enables an easy extension of the test suites, which can be used to cover, for example, new proposed modeling styles. This extension can be accomplished directly by additions made to the C program.

In some cases, the models generated for the analysis require a large amount of storage space. Thus, it would be impractical or sometimes even impossible to store all, or even some of the models at a time, and then perform the execution and measurement procedure. The approach proposed here allows for the provision of a single parameterized model at a time, which after being generated is tested in compilation, elaboration and/or simulation, and at the end deleted before the generation of the next model.

The generation tool is composed of the following modules and functions that are devoted to generate some particular parts of the VHDL code: program parameters recognition, resource library declaration, package declaration, entity declaration with ports and generics module, architecture generation, array declaration functions, enumerated type declaration functions. More details about the implementation details of the generation tool, including some examples of the source code, can be found in [99].

2.5.2 Time measurement

In the WindowsNT operating system, a program is executed as a single process. Each process can be composed of one or many parallel threads, and can also execute other processes. In order to ensure the multitasking operation mode, the operating system executes sequentially each running thread and allocates to it a specific amount of processor time (usually 20ms). The time measurement software, developed for the purpose of a precise measure of execution time, first executes the measured program (compiler or simulator), then monitors the system for each new launched system process, and then measures its execution time. The WindowsNT system, after finishing the execution of a process, provides via the system procedure *GetProcessTimes()*, the information about the kernel and processor times allocated to that process. This information is used to compute the effective execution time of all processes taking part in the simulation. Such a measurement procedure is more complicated than just measuring the execution start and end time, but it provides more accurate results in a multitasking environment.

The entire procedure of automated generation, execution and measurement allows to launch a series of simulations of test models in a completely automatic way, as well as to store (compute and analyze, if necessary) the results obtained. As such, the tool is thought to be a part of the adaptative simulation environment (as described in section 2.4) used for the test of simulation tools performance.

2.5.3 Model optimization and transformation tool

The second set of prototype tools implements some model optimization and transformation rules. For this purpose the LVS system is used, which is described in detail in [75, 76, 77]. For a given model, the LVS compiler generates a representation in the VHDL Intermediate Format (VIF), which is a graph data structure reflecting all the information appropriate to the VHDL model. The LVS system offers a set of C procedures, which enable to access, scan and modify the VIF representation of a model via the procedural interface (called LPI, LEDA Procedural Interface). The VIF format being extensible, permits to introduce and store new information in the model (this feature has been used to implement the particular attributes and the data structure for object dependency graph or data-type dependency graph – cf. chapter 3).

The LVS system has been applied to develop the prototype tool, performing the following optimization methods: automatic package reference removal, constant instantiation as well as subprogram (function or procedure) inline expansion. The prototype tool implements currently also the transformation method of the **if** statement to the **case** statement.

2.6 Conclusions

In this chapter, the methods of increasing the performance of HDL models in the event-driven simulation have been presented. In general, these methods explore the potential of accelerating the models, which is proper to the VHDL code executed on a particular simulation tool. Thus, they allow to accelerate models which were developed without considering the simulation performance requirements.

The methods presented in this chapter address the requirements which have directly motivated our work. First of all, they are compatible with the simulation tools currently used in the design flow and with the common design practice, so that their application does not require to change the tool support for the simulation. Furthermore, all those methods can be applied to the existing libraries of models, or else to models under development, in which no attention is paid to the simulation performance. This is the consequence of the fact that these methods do not require changes in the simulation paradigm (as it is in the case of e.g. cycle-based simulation). Therefore, the level of model resolution in terms of data types of objects, the level of granularity and precision in time domain, the design hierarchy, etc. are preserved, while at the same time the designer benefits from a higher performance in simulation. Even though the design passes throughout several simulation tools and environments in different phases of the design process, the methods (or directly - the results obtained by applying these methods) can migrate from one tool and environment to another.

Since the methods can operate in an automatic way, they can be introduced easily into the existing design flows. The designer is freed from following some modeling rules during the development of the model, or from modifying by hand already existing models. This clearly addresses, in yet another way, the productivity issue of the design flow, especially when the reuse of previously designed components takes place.

In order to ensure for the designer an appropriate level of control on the type of modifications to be completed on the model, the simulation environment allows the designer to select (enable or disable) the set of actions (i.e. types of optimization, transformation or abstraction), which are to be performed. In addition, the tracking of all modifications which are done in the model, helps the designer to link at any time the optimized model to its original.

Finally, it appeared in the experiments carried out on the available simulators, that the set of generic optimization and transformation rules should be adapted to each particular simulation tool to better exploit the potential of acceleration. As it has been shown, some of the simulation bottlenecks have been optimized in some simulator engines, but not in all of them. Moreover, the optimizations, which are incorporated in the simulation technology, and which focus on acceleration of a particular language construct, often imply drawbacks in the simulation of other constructs. This observation leaded to the development of the adaptative simulation environment, which is capable to tune the optimization and transformation rules to the particular simulation tool (or even to a particular version of it). In addition, even some optimization techniques well-known from the software domain, are not incorporated in the tools currently available on the market. This justifies additionally the necessity to perform that kind of optimizations directly on the source code.

Chapitre 3

Abstraction du modèle

3.1 Résumé

Les méthodes de la modification du code source d'un modèle, parmi lesquelles se situent les méthodes d'abstraction présentées d'une manière générale dans le chapitre précédent, offrent un avantage important d'un point de vue de la performance de la simulation. Cependant, afin de pouvoir utiliser ces méthodes dans le flot de conception des systèmes électroniques, il faut développer les techniques permettant de les appliquer à un modèle d'une manière automatique et convenable pour l'utilisateur. C'est pourquoi ce chapitre est dédié d'abord à la présentation détaillée des trois méthodes d'abstraction : la méthode de l'abstraction comportementale, la méthode de l'abstraction des types de données ainsi que la méthode de l'abstraction des objets d'un modèle, pour ensuite proposer les techniques les mettant en œuvre.

La première méthode - l'abstraction comportementale – permet de réduire, dans un modèle initial, certaines de ses parties (par exemple les blocks, les composants, les signaux ou les variables) en fonction des besoins et des conditions d'une exécution particulière de la simulation. L'interprétation de ces besoins et conditions peut aboutir aux deux types d'abstraction comportementale. Le premier type est fondé sur l'observation des vecteurs de stimuli fournis aux entrées d'un modèle. En effet, dans certains cas, lors d'une exécution complète de la simulation, les valeurs de certains signaux d'entrée restent toujours constantes. Cette information peut être utilisée afin de simplifier le modèle dans les parties qui modélisent le comportement en fonction des valeurs autres que celles déclarées constantes. Ce type d'abstraction comportementale est nommé en anglais : *input object invariance*. Quant au deuxième type d'abstraction comportementale, il permet, avant une exécution de la simulation, de sélectionner les signaux internes et les ports de sortie du modèle qui seront observés au cour de la simulation. Lors du processus de l'abstraction, toutes les parties du modèle, qui décrivent le comportement des objets non-sélectionnés (donc : non-observés) sont éliminés. Ce type d'abstraction comportementale est nommée en anglais *observability of objects.*

Une fois la description des deux types d'abstraction comportementale terminée, nous développons les techniques permettant de l'effectuer d'une manière automatique. Ces techniques utilisent une analyse de dépendance des objets dans un modèle, laquelle a pour but de déterminer les objets et les constructions à abstraire (c'est-à-dire à supprimer). Après la suppression des objets, une optimisation du code est effectuée.

Par la suite, nous présentons une autre méthode de l'abstraction appelée l'abstraction des types de données. A l'origine de cette méthode on trouve une analyse des performances en simulation de différents types de données. Les résultats de cette analyse mettent en évidence le fait que le temps de simulation des objets des types de données abstraits (par exemple le type entier) est beaucoup moins élevé que celui de simulation des objets des types plus détaillés (par exemple *bit_vector*). Par conséquent, une méthode d'abstraction permettant de remplacer d'une manière automatique certains objets des types de données détaillées par les objets des types plus abstraits, peut être conçue. La description de cette méthode, ainsi que des algorithmes de sa mise en œuvre, font l'objet de la partie suivante de ce chapitre.

La troisième méthode d'abstraction présentée dans ce chapitre est celle des objets d'un modèle. Étant donné que le mécanisme de simulation d'un signal est un processus complexe et coûteux en terme de temps de simulation, nous proposons une méthode d'accélération d'un modèle qui met en place une substitution automatique des signaux par des variables du même type. Cette substitution peut être effectuée seulement dans le cas dans lequel certaines conditions d'utilisation d'un signal dans le modèle sont remplies. C'est pour cette raison que cette partie du chapitre est consacrée d'abord à la définition des conditions de la substitution, puis à la description de la technique de modélisation qui permet de l'appliquer au modèle.

Pour les trois méthodes d'abstraction décrites ci-dessus nous montrons les résultats expérimentaux de l'application de chacune d'entre elles aux projets industriels.

3.2 Behavioral abstraction [7]

3.2.1 Abstract

This section focuses on the behavioral abstraction methods of VHDL models for the improvement of event-driven simulation performance. The model abstraction takes into

[7] Some parts of this chapter have been initially presented in [105]

account the specific simulation requirements in a particular simulation execution: the observability of the internal and external signals as well as the test-bench vectors of the input signals. According to those requirements the initial model is abstracted in order to decrease its complexity in terms of the number of VHDL objects used, but also in terms of the elaborated model memory consumption. The practical results of the simulation performance improvement comparing to the conventional simulation are presented.

3.2.2 Introduction

New design paradigms are addressed by the research to increase the efficiency of the design methods and tools. Among them, shift to the higher levels of automation through the introduction of e.g. high level synthesis, or the design reuse belongs to the most important ones. The design and verification of complex systems requires new methods offering much higher productivity than those currently available. Several of such methods shortly discussed below have been proposed to overcome the problem posed by the complexity of the process of simulation. The abstraction mechanisms or the application of equivalent model representations based on decision diagrams or branching programs for fast discrete function evaluation are some of the most significant examples of these methods.

Khosravipour et al. [66] introduced abstraction methods for improving the simulation efficiency while raising the level at which the design is represented. Cycle-based simulation is treated as one of possible timing abstraction. This work provides an extensive classification of abstraction mechanisms but does not provide any technique to apply it in the design flow. The work presented in this section extends the scope of the behavioral abstraction as defined in [66] and provides techniques and prototype tools to implement it.

Other approaches for the improvement of the simulation performance apply different models of computation based on various forms of decision diagrams. Some of them are suitable for logic level only (bit, bit-vector) – they are often based on some forms of binary decision diagrams BDDs (e.g. McGeer et al. [90] applied MDD representation for bit vector discrete function representation and evaluation for logic level cycle-based simulation) or on branching programs (e.g. P. Ashar and S. Malik [9] apply branching programs for efficient logic function evaluation or Y. Luo et al. [81] use a cycle-based simulation technique for synchronous circuits which combines a BDD-based logic level cycle simulator with fast hierarchical direct evaluation of high-level functional units stored in a library). The other representation types based on high-level decision diagrams, e.g. [110, 155, 157] can be applied at more abstract description levels. All the above methods are suited for the cycle-based simulation.

From the point of view of the final user (i.e. the designer) the main difference between the above described techniques and the abstraction method presented in this section, is that the former use different models of computation which are based on the data structures generated from the HDL description and combined with the cycle-based simulation technique, and the latter permits to use the HDL description together with the event-driven simulation paradigm.

3.2.3 Approach

The model behavioral abstraction is performed while taking into account the conditions and requirements of a particular simulation execution. It consists in removing of some sections of the model (e.g. blocks, components, signals or variables), which are irrelevant in the particular simulation run.

One of the possible ways of practically managing that kind of simulation acceleration is to allow the designer to designate the signals and/or variables, which need to be observed in the current simulation. Using this information all non-observed signals (both internal or external) and corresponding design sections can be removed, what simplifies the model in terms of complexity and memory usage and leads to a considerable simulation speed-up. The second type of the solution presented here is based on the application of constant values to the input ports or internal signals, and in simplifying the model according to the values assigned.

The work presented here focuses on the transformation of an existing model or model being under construction, rather than on the model creation mechanisms. This approach allows to treat already existing libraries of models and it remains compliant to the existing widely used design methods and tools. The latter enables to apply the presented method during the model development phase.

In order to fully benefit from the advantages of the behavioral abstraction, this method can be combined with the structural abstraction [66, 109]. The structural abstraction consists in removing from a complex composed and hierarchical models the information about their structure - the existence of separate design blocks as well as the design hierarchy. To the model flattened by the structural abstraction, behavior abstraction method can be applied, and in this case, the abstraction is not limited by the borders of a single block (or component) and can also correspondingly abstract the blocks belonging to the environment of the block under transformation.

The presentation of the behavioral abstraction method is organized as follows: section 3.2.4 describes two types of behavioral abstraction method: abstraction based on the input object invariance and abstraction using the information on observability of design signals. In section 3.2.5 the application of both types of abstraction to an example VHDL code is presented. Details of implementation of the abstraction method are provided in section 3.2.6. Section 3.2.7 summarizes some experimental results of the application of behavioral abstraction to industrial models.

3.2.4 Model abstraction

The model behavioral abstraction (alternatively called model reduction) consists in the analysis of the dependency of objects (signals, variables, ports) used in the model, and on the removal of the sections of the initial model (e.g. blocks, components, processes, signal or variable objects), which are irrelevant in the particular simulation run. The model abstraction takes into account the level of details that needs to be tracked by the designer in the simulation as well as the particular state or function (often localized in large models), which is verified in the current simulation run.

Behavioral abstraction changes the global model structure and/or behavior, but it does not change the localized behavior of the model for the selected set of internal and output signal objects or the behavior of the model for determined constant values of input or intermediate signals/variables.

Two methods of applying the model abstraction are defined based on the input vector analysis and on the specification of the observability of signals.

3.2.4.1 Input object invariance

The input object invariance abstraction method is based on the assignment of constant values to input signals of the model. Taking into account the structure of the model, the application of constant values to input objects allows to reduce those parts of the model, which define its behavior as a function of the selected object for other values than the determined constant value.

The following code using the **case** statement illustrates the application of the method:

```
case func is
    when pass1  => result := operand1;
    when pass2  => result := operand2;
    when add    => result := operand1 + operand2;
    when subtract   => result := operand1 - operand2;
end case;
```

For example, knowing that *func* is always taking only one value e.g. *add* the code can be reduced to the following statement:

```
result := operand1 + operand2;
```

That, in consequence, eliminates the entire **case** statement from the model. Even a partial reduction of the code, e.g. removal of some unreachable choices in the **case** statement can make the code more compact, thus more efficient in terms of the memory usage, which leads to the shorter simulation time.

There are two practical possibilities to manage the input object invariance:

- either the designer specifies explicitly the inputs of the model and their values for the entire simulation run (e.g. the *chip enable* signal always set), or
- the input vector test bench can be analyzed in order to define a set of abstracted models, which correspond to one initial model, and which are to be executed for different parts of the entire test suite; the execution of abstracted models is performed for the corresponding set of the input vectors, for which constant values of the inputs have been determined and for which the model has been generated. After the execution of the simulation, the status of the model (i.e. the values of all signals and variables) is recorded in order to properly initialize the next abstracted model to execute.

3.2.4.2 Observability of signals

Often in the design practice the specific simulation execution is focused to a very local behavior of the entire model. In that case, in order to enable the behavioral abstraction, the designer is allowed to designate, before the execution of the simulation, all the signals (both internal and external, i.e. ports), which must be observed during the simulation run. This information is used to reduce the simulation model by removing from it all non-observable signals and corresponding design sections: statements, processes, components, which are orthogonal to the behavior modeled as a function of the observable signals. Here, *orthogonal* means that the removed signals do not contribute in any way to the behavior of the observable signals. The removal procedure is based on the examination of the dependency of modeled objects. This technique is described in the next section.

3.2.4.3 Abstraction definition

Taking the definition of a deterministic system $D=<T, I, O, S, \Omega, \delta, \lambda>$

where: T is the time set;

$I = i_1 \ldots i_m$ is the set of inputs;

$O = o_1 ,.. , o_n$ is the set of outputs;

$S = s_1 ,.. , s_p$ is the set of states;

Ω is the set of admissible input functions;

δ is the transition function defined as: δ: $S \times T \times \Omega \rightarrow S$ and

λ is the output function defined as: λ: $T \times S \times \Omega \rightarrow O$.

Input object invariance: The constant values of some input or intermediate signals are determined and assigned to those signals for the entire simulation execution:

$$\exists_{0<j\le m} i_j = const$$

The remaining k inputs ($k < m$) form a new set of inputs I' for which holds:

$$\forall_{i \in I'} i \neq const$$

and at the same time:

$$\forall_{i \notin I'} i = const$$

in consequence of which $\Omega \rightarrow \Omega'$ where $\Omega' \subset \Omega$, and $\delta \rightarrow \delta'$, $\lambda \rightarrow \lambda'$.

The behavioral abstraction achieved by the input object invariance can be defined as a following relation $R(D, D')$, which reduces the system D to $D' = < T, I', O', S, \Omega', \delta', \lambda'>$.

Observability of signals: If the designer defines the set of observable signals $O''=o_1''\ldots o_l''$ where $l < n$ and $O'' \subset O$, the model behavioral abstraction, as a function of the observable signals, can be defined as a following relation $R(D, D'')$, in which D is the initial system definition and $D''=<T, I, O'', S, \Omega, \delta, \lambda''>$ is the resulting reduced system in which $\lambda'' \subset \lambda$.

Both those methods, i.e. the input object invariance and the observability of signals can be combined to further reduce the initial model.

3.2.5 Example

3.2.5.1 Input object invariance example

An example of the application of the input object invariance abstraction is presented in figures 3.1 to 3.3. Figure 3.1 shows the initial VHDL code of the model, on which two processes of abstraction will be performed: the first one is based on the assumption, that the input port *a* always takes the value *'1'*, while the second one assigns the value *'0'* to the input port *b*.

Input object invariance abstraction 1: *a* = *'1'*

Assuming that the model should be abstracted for the input *a*, and that the value of *a* is equal to *'1'*, the following steps are undertaken in the abstraction procedure (note: the numbers below indicate parts of the code affected in each step, as shown in figure 3.1):

1. elimination of the declaration of the input port *a*;
2. elimination of *a* from the process sensitivity list;
3. simplification of the expression in the **if** statement condition;
4. removal of the sequence of statements in the **if** statement, corresponding to the condition that will never evaluate to the value *true*, because *a = '1'*;
5. simplification of the expression in the variable assignment statement;
6. simplification of the use of an array object in the variable assignment statement;
7. simplification of the declaration of the type *f_array* taking into account step 6 and knowing that the port *a* will always take the value *'1'*,
8. simplification, as a result of the action performed in step 7, of the declaration of the constant *derive* of the type *f_array*;
9. step 8 leads to the reduction of the enumeration type *functions*, because only two values (*add* and *sub*) of that type are in use;
10. the consequence of the reduction in step 9 is that the variable *func* of the type *functions* can only evaluate to two values, what, in turns, allows to reduce two of the alternatives in the **case** statement.

```
entity E is                  ①
    port (  clock, a, b, c  : in    bit;
            in1, in2        : in    integer;
            result          : out   integer;
            ready           : out   bit);
end E;
```

```
architecture A of E is
    type     functions  is (pass1, pass2, add, sub);          ⑨
    type     f_array    is array (bit, bit) of functions;     ⑦
    constant derive  : f_array :=  ( (pass1, pass2),          ⑧
                                     (add, sub));
    signal     temp  : integer;

begin                          ②
    process (clock, a, b)
        variable  x      : bit;
        variable  func   : functions;
    begin
    (...)                         ③
        if (a='1' and b='0') then
            x := '1';
            temp <= 2 * in1;
        elsif (a='0') then
            x := not c;                   ④
            temp <= in1 + 1;
        elsif (b='1' and c='0') then
            x := '0';
            temp <= in1 - 1;
        else                      ⑤
            x := a and c;
            temp <= in2 / 2;
        end if;
    (...)                         ⑥
        func := derive (a, x);
    (...)
        case func is
            when pass1  => result <= in1;
            when pass2  => result <= in2;      ⑩
            when add    => result <= in1 + in2;
            when sub    => result <= in1 - in2;
        end case;
    (...)
    end process;
end;
```

Figure 3.1: *VHDL code example; the numbers and shading indicate statements affected by the first abstraction for a = '1'*

All the above optimizations are based on the assumption that the types *functions* and *f_array* and the constant *derive* are not used in other parts of the model.

The final result of the abstraction is shown in figure 3.2.

```
entity E is              ↙①↙⑧
    port ( clock, b, c     : in    bit;
           in1, in2        : in    integer;
           result          : out   integer;
           ready           : out   bit);
end E;

architecture A of E is
    type      functions  is  (add, sub);
    type      f_array    is  array (bit) of functions;   ←⑥
    constant  derive  : f_array := ((add, sub));
    signal    temp    : integer;

begin                    ↙②
    process (clock, b)
        variable  x      : bit;
        variable  func   : functions;   ←⑤
    begin
    (...)
        if (b='0') then          ←③
            x := '1';              ↓
            temp <= 2 * in1;
        elsif (b='1' and c='0') then
            x := '0';
            temp <= in1 - 1;
        else
            x := c;
            temp <= in2 / 2;
        end if;
    (...)
        func := derive (x);     ←④
    (...)
        case func is
            when add    => result <= in1 + in2;   ←⑦
            when sub    => result <= in1 - in2;
        end case;
    (...)
    end process;
end;
```

Figure 3.2: *VHDL code obtained as a result of the abstraction of the code from figure 3.1 for a = '1'; the numbers indicate steps of the second abstraction for b = '0', the shading shows affected parts of code*

Input object invariance abstraction 2: *b = '0'*

If the code resulting from the *abstraction 1* is once again abstracted taking into account that the value *'0'* is always assigned to port *b*, the VHDL code presented in figure 3.3 is obtained. The abstraction procedure consists of the following steps:

1. removal of the declaration of the input port *b*;
2. elimination of *b* from the process sensitivity list;
3. elimination of the **if** statement based on the assumption that *b = '0'*;
4. the previous step leads to the assignment of the value *'1'* to the variable *x*; as a consequence, the assignment statement to the variable *func* of the value, which obtained from the constant array *derive* depending on the value of *x* (the statement *func := derive(x);*), can be replaced by the assignment of the constant value *add* to *func*; this implies that each appearance of the variable *func* in the code can be substituted by the constant value *add*, and in consequence of which the variable *func* can be deleted from the code;

5. step 4 follows with the removal of the declaration of the variable *func*;
6. optimizations of steps 4 and 5 enable to remove the declarations of the types *functions* and *f_array*, as well as of the constant *derive* (see note below);
7. in consequence of reductions done in steps 4-6 (substitution of *func* by *add*), the **case** statement can be reduced to a single signal assignment statement.
8. since the part of code, which used the value of the input port *c* in the assignment statement *x := c*, has been removed in step 3, the declaration of the port *c* became superfluous and it can be removed from the model (the removal is admitted only if the note below is valid).

The steps 1 to 8 are illustrated in figure 3.2 with the numbers and shading pointing out the parts of the code affected by the abstraction.

Note: all the above reductions are based on the assumption that the input port *c*, the variable *func*, the types *functions* and *f_array* and the constant *derive* are not used in other parts of the model.

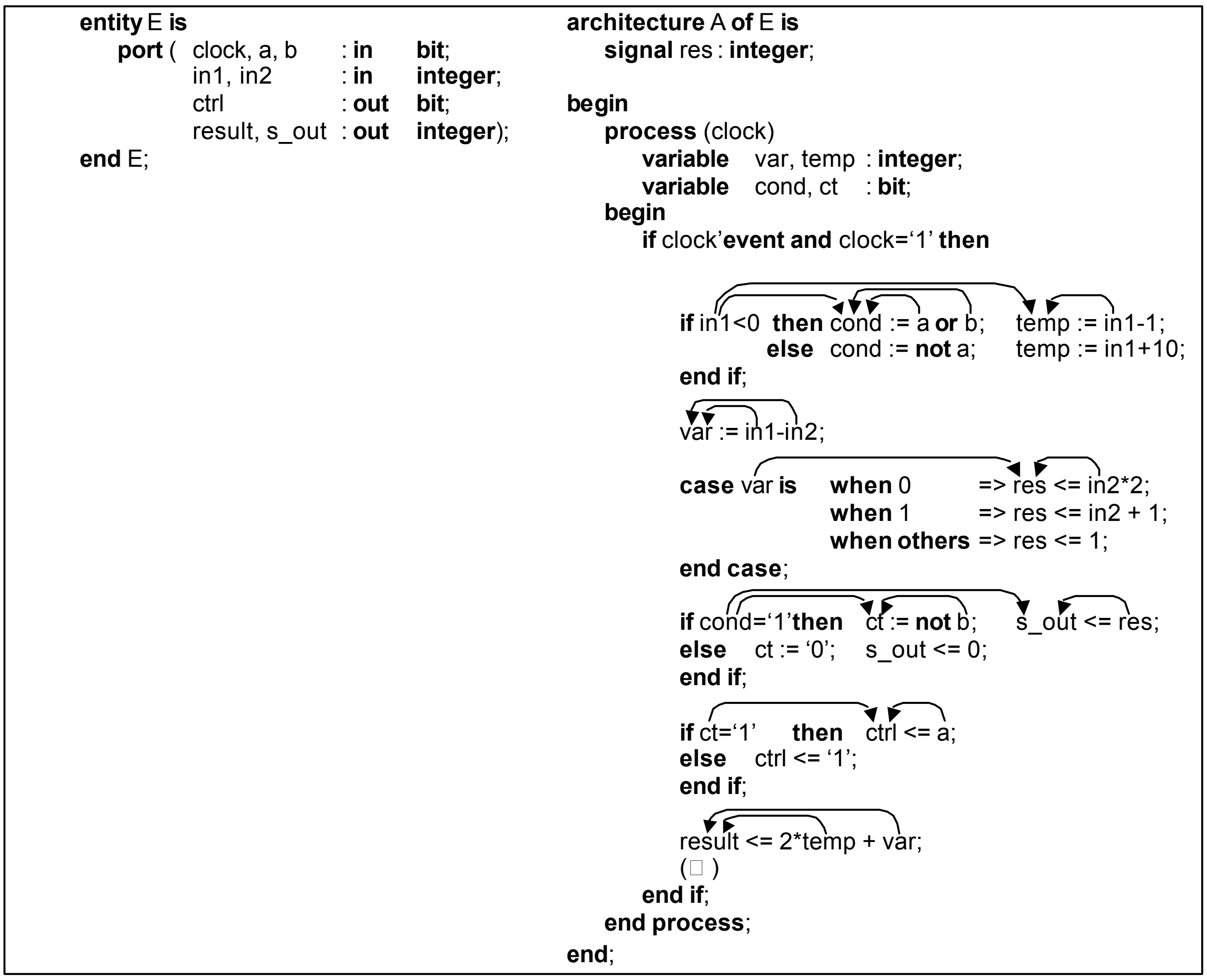
Figure 3.3 presents the code obtained as a result of both abstractions 1 and 2, as described above.

```
entity E is                                architecture A of E is
    port  ( clock      : in    bit;            signal    temp  :   integer;
            in1, in2   : in    integer;
            result     : out   integer;    begin
            ready      : out   bit);           process (clock)
end E;                                             variable   x   : bit;
                                               begin
                                               (...)
                                                   x := '1';
                                                   temp <= 2 * in1;
                                               (...)
                                                   result <= in1 + in2;
                                               (...)
                                               end process;
                                           end;
```

Figure 3.3: *VHDL code obtained by the input object invariance abstraction of the code from figure 3.1 for a = '1' and b = '0'*

3.2.5.2 Observability of objects example

Figures 3.4 to 3.8 present an example of the abstraction based on the method of observability of objects. In the following, two abstraction procedures are described. In the first one, two output objects are observed: *ctrl* and *result*, while the second abstraction provides the observabitity of a single output port *result*. The initial code of the example is shown in figure 3.4.



```
entity E is                                    architecture A of E is
    port (  clock, a, b    : in    bit;            signal res : integer;
            in1, in2       : in    integer;
            ctrl           : out   bit;        begin
            result, s_out  : out   integer);       process (clock)
end E;                                                 variable   var, temp  : integer;
                                                       variable   cond, ct   : bit;
                                                   begin
                                                       if clock'event and clock='1' then

                                                          if in1<0  then cond := a or b;    temp := in1-1;
                                                                    else  cond := not a;    temp := in1+10;
                                                          end if;

                                                          var := in1-in2;

                                                          case var is   when 0        => res <= in2*2;
                                                                        when 1        => res <= in2 + 1;
                                                                        when others => res <= 1;
                                                          end case;

                                                          if cond='1'then   ct := not b;    s_out <= res;
                                                          else    ct := '0';   s_out <= 0;
                                                          end if;

                                                          if ct='1'    then   ctrl <= a;
                                                          else    ctrl <= '1';
                                                          end if;

                                                          result <= 2*temp + var;
                                                          (□ )
                                                       end if;
                                                   end process;
                                               end;
```

Figure 3.4: *VHDL code example used in the abstraction based on the observability of objects; the arcs show the dependency relation between objects*

The abstraction method starts with the creation of the object dependency graph for the model. The analysis performed on the model permits to establish dependency relations between all objects of the model. This dependency is illustrated by arcs in figure 3.4 (note: the dependency on the signal *clock* is omitted in this figure) and is reflected by the object dependency graph depicted in figure 3.5. This figure also shows the groups of objects that have been abstracted in two abstraction procedures presented below.

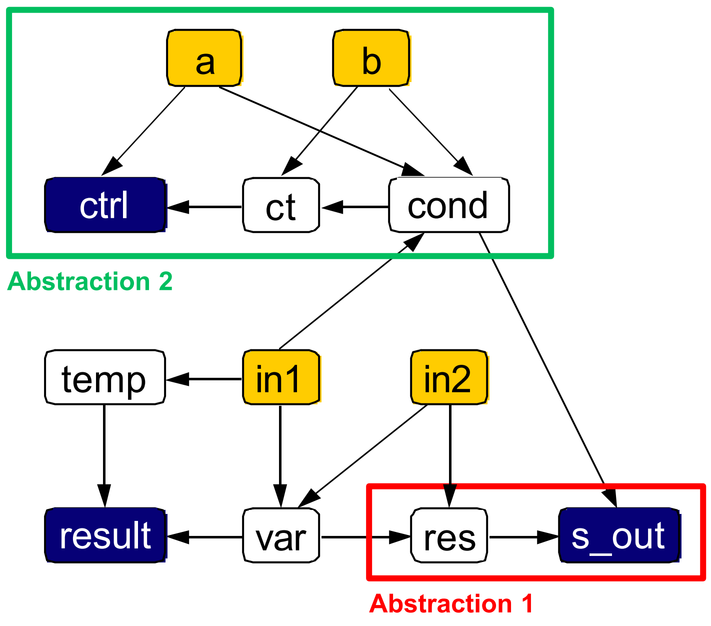


Figure 3.5: *Object dependency graph for the model presented in figure 3.4.*

Objects observability abstraction 1: port *s_out*

In this abstraction the output port *s_out* is removed from the list of observable objects. As the object dependency graph shows, this port is directly dependent on a single object - the signal *res*. Moreover, there are no other objects in the model which are dependent on *res*. Thus, the abstraction procedure will consist in removal of the statements declaring the port *s_out* and the signal *res*, as well as in the elimination of all statements in the model, which produce the assignments to those objects. In detail, the abstraction procedure is composed from the following actions, which are presented in figure 3.6 by the appropriate numbers:

1. elimination of the declaration of the port *s_out*;
2. removal of the declaration of the signal *res*;
3. removal of assignment statements to the signal *res* in the **case** statement; since the **case** statement does not perform any other action, it will be removed entirely from the model in the optimization process;
4. elimination of assignments to the port *s_out* in the **if** statement.

The code which is obtained after performing all the above actions is presented in figure 3.7.

```
entity E is
    port (  clock, a, b : in    bit;
            in1, in2    : in    integer;
            ctrl        : out   bit;
            result      : out   integer;
            s_out       : out   integer );  ←①
end E;

architecture A of E is
    signal     res     : integer;  ←②

begin
    process (clock)
        variable   var, temp  : integer;
        variable   cond, ct   : bit;
    begin
        if clock’event and clock=‘1’ then
            if in1<0 then  cond := a or b;    temp := in1-1;
                    else   cond := not a;     temp := in1+10;
            end if;
            var := in1-in2;
            case var is    when 0       => res <= in2*2;
                           when 1       => res <= in2+1;    ←③
                           when others => res<=1;
            end case;
            if cond=‘1’    then   ct := not b;     s_out <= res;
                           else   ct := ‘0’;       s_out <= 0;
            end if;                                      ↑
            if ct=‘1’    then   ctrl <= a;               ④
                         else   ctrl <= ‘1’;
            end if;
            result <= 2*temp + var;
            (□ )
        end if;
    end process;
end;
```

Figure 3.6: *VHDL code with the indication of actions to be performed in the abstraction of the port* s_out

```
entity E is               ⑥
    port (  clock, a, b : in    bit;
            in1, in2    : in    integer;
            ctrl        : out   bit;      ←①
            result      : out   integer);
end E;

architecture A of E is
begin
    process (clock)
        variable   var, temp    : integer;
        variable   cond, ct     : bit;  ←②
    begin
        if clock’event and clock=‘1’ then
            if in1<0 then  cond := a or b;    temp := in1-1;
                    else   cond := not a;     temp := in1+10;
            end if;                        ③
            var := in1-in2;
            if cond=‘1’    then   ct := not b;
                           else   ct := ‘0’;      ←④
            end if;
            if ct=‘1’    then   ctrl <= a;
                         else   ctrl <= ‘1’;      ←⑤
            end if;
            result <= 2*temp + var;
            (□ )
        end if;
    end process;
end;
```

Figure 3.7: *Result of the abstraction of the output port* s_out*; the numbers indicate the parts of code affected in the abstraction of the port* ctrl

Objects observability abstraction 2: port *ctrl*

The objective of the second abstraction is to preserve the observability of the single output port *result*. Taking into account the outcome of the previous abstraction, in which the port *s_out* has been abstracted, in this abstraction the port *ctrl* can be removed from the model. As it can be inferred from the object dependency graph (figure 3.5), the port *ctrl* depends directly on two objects: on the port *a* and on the variable *ct*. The variable *ct*, in turns, depends on the variable *cond* and on the port *b*. Since no other objects depend neither on *ct* nor on *cond* (the port *s_out*, depending on *cond* has already been deleted in the previous abstraction), both these variables can be removed from the model. The same can apply to the input ports *a* and *b*, because in the object dependency graph they proceed only objects, that have been removed. Note, that the last assumption is valid only if those ports are not used in other parts of the model, than those depicted in figure 3.6. In consequence, the abstraction procedure includes the following tasks:

1. removal of the output port declaration for *ctrl*;
2. removal of the variable declarations for *cond* and *ct*;
3. elimination of the assignment statements to the variable *cond* in the **if** statement;
4. elimination of the **if** statement assigning values to the variable *ct*;
5. elimination of the **if** statement assigning values to the output port *ctrl*;
6. removal of the statements declaring input ports *a* and *b*;

The result of the abstraction described above is demonstrated in figure 3.8.

```
entity E is
    port (  clock       : in    bit;
            in1, in2    : in    integer;
            result      : out   integer);
end E;

architecture A of E is
begin
    process (clock)
        variable   var, temp  : integer;
    begin
        if clock'event and clock='1' then
            if in1<0    then    temp := in1-1;
                        else    temp := in1+10;
            end if;
            var := in1-in2;
            result <= 2*temp + var;
            (□ )
        end if;
    end process;
end;
```

Figure 3.8: *VHDL code obtained as a result of the object observability abstraction in which output ports* s_out *and* ctrl *have been abstracted*

3.2.6 Implementation of the behavioral abstraction method

The implementation of the behavioral abstraction method is based on the static analysis of the VHDL code and comprises three distinct actions: the creation of the *object dependency graph*, the behavioral abstraction algorithm consisting in the *removal of unnecessary objects* and in the *code optimization*.

In order to describe the introduced actions the terminology and definitions are presented below [8] followed by the description of the method.

3.2.6.1 Definitions and terminology

Definition 3.1: *Target*

A variable or signal is said to be a ***target*** of a statement if its value can be re-evaluated by that statement.

In particular, in the following types of statements the target is determined:

- concurrent signal assignment statement;
- sequential signal or variable assignment statement;
- procedure call with the signal/variable as an actual part of the formal parameter of mode **out** or **inout**;
- component instantiation with the port of mode **out**, **inout** or **linkage**;
- block instantiation with the port of mode **out**, **inout** or **linkage**;
- entity instantiation with the port of mode **out**, **inout** or **linkage**.

◊

Definition 3.2: *Source*

A variable or signal is said to be a ***source*** of a statement if one of the following conditions is satisfied for it:

- it appears in the value expression of a waveform element of a sequential or concurrent signal assignment;
- it appears in the sub-element evaluation expression of another signal or variable;
- it appears in the reject expression of a selected signal assignment statement;
- it appears in the time limit expression of a **wait** statement;
- it appears in the expression on the right hand side of a variable assignment;

[8] The definitions for *target*, *source* and *assignment statements* are based on the initial definitions presented in [74]

- it is connected to a port of mode **in**, **inout**, **linkage** or **buffer** of a component or **entity** instantiation statement or **block** statement;
- it is the actual part of formal parameter of mode **in** or **inout** in a subprogram call.

◊

Definition 3.3: *Assignment statement*

An ***assignment statement*** is one of the following:

- sequential assignment;
- concurrent assignment.

◊

Definition 3.4: *Sequential assignment*

A ***sequential assignment*** is a sequential statement that provokes the re-evaluation of a variable or signal. It is one of the following instructions :

- sequential signal assignment;
- sequential variable assignment;
- sequential procedure call with a parameter of mode **out** or **inout**.

◊

Definition 3.5: *Concurrent assignment*

A ***concurrent assignment*** provokes the re-evaluation of a signal and is any one of the following statements:

- concurrent signal assignment;
- concurrent procedure call with a parameter of mode **out** or **inout** connected to the signal;
- component instantiation with a port of mode **out**, **inout** or **linkage** connected to the signal;
- block instantiation with a port of mode **out**, **inout** or **linkage** connected to the signal;
- entity instantiation (VHDL'93) with a port of mode **out**, **inout** or **linkage** connected to the signal.

◊

Definition 3.6: *Dependent signal/variable*

A target signal or variable is said to be ***dependent*** on a given signal or variable if one or more of the following conditions hold:

- the target appears in the assignment statement, in which the given signal or variable is a source;
- the target appears in the assignment statement, which is a part of the sequence of statements of the **if** statement, **case** statement or **loop** statement, in which the given signal or variable appears in the condition or expression of those statements;
- the target appears in the assignment statement, which is a part of the process statement part of a **process** statement with the sensitivity list, for which the given signal is a part of the process sensitivity list;
- the target appears in the assignment statement, which is a part of the process statement part of a **process** statement with the **wait** statements where the given signal or variable is a part of the **wait** statement sensitivity list in sensitivity clause or condition in the condition clause or in the expression in timeout clause;
- the target appears in the assignment statement, which is a part of the **block** statement in which a given signal appears in the guard expression or is a port of mode **in** or **inout** of the block statement;
- the target is the port of mode **out** or **inout** of the component instantiation, for which the given signal is an actual of formal parameter of mode **in** or **inout**. ◊

Note: the word *appears* in the above paragraph can also denote the hierarchical (or nested) appearance.

Definition 3.7: *Control statement*

A ***control statement*** is one of the following:

- sequential control statement:
  - **if** statement;
  - **case** statement;
  - **loop** statement;
  - **next** statement;
  - **exit** statement;
- concurrent control statements:
  - conditional signal assignment;
  - selected signal assignment. ◊

Definition 3.8: *Primary signal/variable*

A signal or variable object is said to be ***primary*** if it is not dependent on other signal or variable object. It might be, however, dependent on a constant object. The primary inputs of the design block are primary signals. ◊

3.2.6.2 Object dependency graph construction

The first step of the model behavioral abstraction is the creation of the object dependency graph of the model. Each statement of the code is analyzed to determine the type (source/target) and dependency of all objects. The object dependency graph built upon this analysis is a directed graph, in which the nodes represent the objects from the initial model: ports, signals, variables, shared variables or constants. All these objects can be both source and targets for other objects. The edges of the graph corresponding to the ordered pairs of nodes represent the dependency between objects in the model. The graph is a cyclic graph if some of objects in a model are mutually dependent. Otherwise, the graph is acyclic. For example, if two statements appear in the model: (1) $x <= y + z;$ and (2) $y <= 1 - x;$ it follows from (1) that the signal x is dependent on y, and from (2) that the signal y is dependent on x; therefore, the object dependency graph of that model is cyclic.

The analysis of the initial model should be done in a hierarchical manner to extract the relevant information. This is achieved in the way described below.

In the first step the following lists of objects are successively created:

- the **unit input list** is created for the design unit: the ports of the mode **in**, **inout** or **buffer** are added to that list;
- the **block input list** is created for every internal **block** statement of the design unit: the ports of the mode **in**, **inout** or **buffer** or the signals appearing in the guard expression of the **block** statement are added to that list;
- the **process signals list** is created for every **process** statement: the signals from the process sensitivity list or from the **wait** statement sensitivity clause or condition clause (cf. LRM [59] rule 8.1) are added to that list;
- the **control list** is created for every sequential and concurrent control statement: the following objects are added to that list:
  - **if** statement condition primaries; for each **elsif** part a separate list is created with the primaries of the actual condition and the primaries of all the conditions appearing before the actual **elsif** in the current statement;

- **case** expression primaries;
- **loop** statement: **while** condition primaries or **for** discrete range primaries;
- **next** statement condition primaries;
- **exit** statement condition primaries;
- conditional signal assignment waveform condition primaries; for each conditional waveform a separate list is created with the primaries of the actual condition and primaries of all the conditions appearing before the actual conditional waveform in the current statement;
- selected signal assignment expression primaries;

- the **component input list** is created with the actual part associated with the formal ports of the mode **in**, **inout** or **linkage**;
- the **concurrent procedure input list** is created with the actual part associated with the formal parameters of the mode **in**, **inout** or **linkage**.

The model is then hierarchically scanned to determine the target objects, starting from the lowest level of the design hierarchy. For each target a unique dependency list is created in the following way:

- the source list of the target is determined and added to the dependency list;
- the objects from the control list are added to the dependency list from the actual scope; this step is repeated recursively if the control statement is in the scope of another control statement;
- the process signal list of the process statement in the scope of which the actual target appears is added to the dependency list;
- The block input list of the block statement (if any) in the scope of which the current target appears is added to the dependency list.

A schema of the dependency analysis is depicted in figure 3.9.

3.2.6.3 Behavioral abstraction algorithm

The behavioral abstraction algorithm consists of two tasks: the removal of unnecessary objects and of the code optimization.

The first task is achieved by the following symbolic algorithm:

1. In the object dependency graph mark all nodes representing the signals and variables designated by the designer as observable;

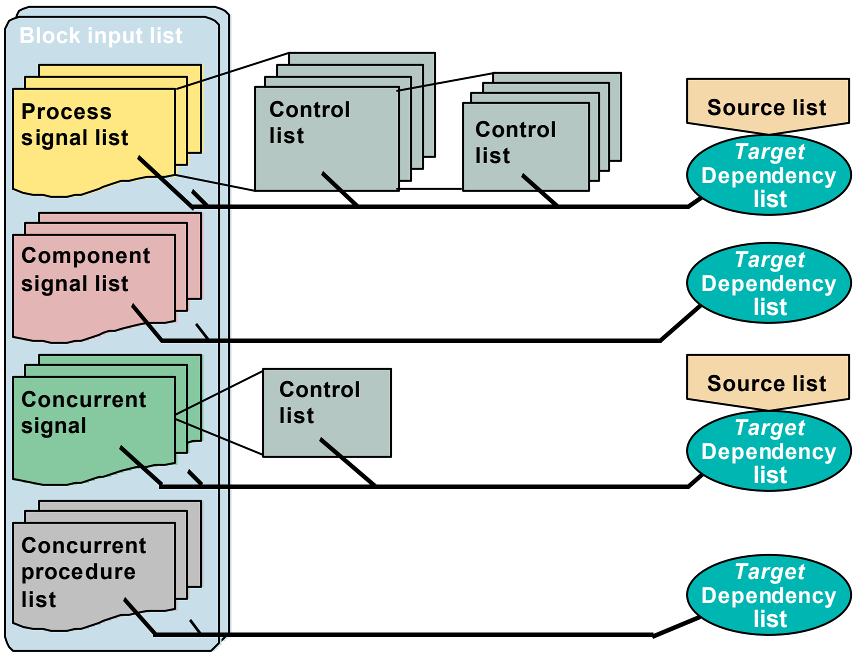


Figure 3.9: *Object dependency analysis*

2. For all marked nodes mark the predecessor nodes;
3. Repeat step 2 until the root nodes of the graph (i.e. nodes without predecessors) are reached;
4. Remove unmarked nodes.

The code optimization is realized after the removal of non-observed objects. It consists in the hierarchical analysis of the reduced code to further remove from it those statements, which do not perform any action. For example, in the situation in which all the sequential statements of an **if** statement have been removed, the **if** statement itself can be removed from the model.

The input object invariance abstraction uses also the object dependency graph to assign and propagate constant values throughout the model. The code optimization is performed next, in order to simplify the model as much as possible without changing its initial behavior (in the example of the **case** statement in section 3.2.4.1: first the constant value for *func* is determined, then the entire **case** statement is reduced to only one assignment statement).

3.2.7 Practical Experiments

3.2.7.1 Tool environment

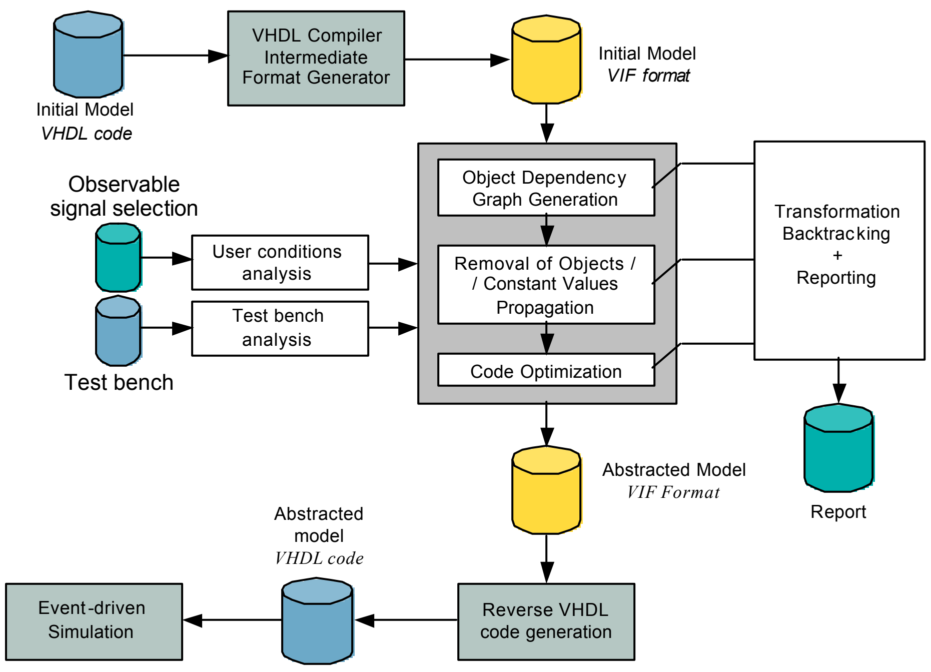

Initial Model
VHDL code
VHDL Compiler Intermediate Format Generator
Initial Model
VIF format
Observable signal selection
User conditions analysis
Test bench analysis
Test bench
Object Dependency Graph Generation
Removal of Objects / / Constant Values Propagation
Code Optimization
Transformation Backtracking + Reporting
Report
Abstracted Model
VIF Format
Abstracted model
VHDL code
Event-driven Simulation
Reverse VHDL code generation

Figure 3.10: *Behavioral abstraction tool environment*

Figure 3.10 presents the tool environment built to support the behavioral abstraction. All the operations necessary to accomplish the behavioral abstraction are performed on the intermediate format representation of the model. Three successive actions of object dependency graph generation, removal of objects, propagation of constant values and the code optimization are performed. Those actions are carried out based on the specification of observable signals and analysis of test-bench vectors. All the operations on the code are reported in a separate file. The resulting abstracted model in the VIF format is transformed into VHDL code by the reverse code generator.

3.2.7.2 Simulation results

Table 3.1 presents the characteristics of circuit examples used in the experiments. Some results of the application of the behavioral abstraction to the real examples are presented in table 3.2.

Taking the *gcd* example, the gain which can be obtained by the input object invariance method applied for both data inputs of the design can be of 75% (1.75 times). The gain recorded for the *diffeq* model is of 2.66 times, when the simulation results of only one output are observed.

Some additional examples of the application of the behavioral abstraction method are presented in Annex C. Moreover, some further details about the experiments and designs under consideration are provided there.

Design	RTL description					
	Inputs	Outputs	Internal Blocks	Internal Signals	Lines of code	Test length [vectors]
gcd	4	2	12	67	526	195200
mult8x8	4	2	21	124	3350	28140
diffeq	7	3	23	149	3370	20810
dlx	5	7	18	252	3872	1624
fiforx	8	3	2	29	1113	50000
rxbit	4	3	2	22	1341	100000
bus_interface	15	10	2	30	1720	100000

Table 3.1: *Characteristics of design examples*

Design	Abstraction	Description	Simulation time [s]	Gain in comparison to the initial model
gcd	Initial model		616.0	1.00
	Input object invariance	1 input abstracted	461.0	1.34
	Object observability	inputs 1 & 2 abstracted	**352.0**	**1.75**
mult8x8	Initial model		68.1	1.00
	Input object invariance	1 input abstracted	**54.4**	**1.25**
diffeq	Initial model		1365.0	1.00
	Input object invariance	1 input abstracted	1103.7	1.24
	Object observability	Observability: 2 outputs	877.0	1.56
	Object observability	Observability: 1 output	**512.8**	**2.66**
dlx	Initial model		2239.2	1.00
	Object observability	Observability: 1 output	**1223.0**	**1.83**
fiforx	Initial model		1432.2	1.00
	Input object invariance	5 inputs abstracted	1108.1	1.29
	Object observability	Observability: 1 output (from 3)	**1044.1**	**1.37**
rxbit	Initial model		1397.0	1.00
	Input object invariance	2 inputs abstracted	986.5	1.42
	Object observability	Observability: 2 outputs (from 4)	**660.3**	**2.12**
bus_interface	Initial model		765.8	1.00
	Input object invariance	7 inputs abstracted	544.8	1.41
	Object observability	Observability: 4 outputs (from10	**384.7**	**1.99**

Table 3.2: *Simulation results for model behavioral abstraction*

3.2.8 Concluding remarks

Section 3.2 of this chapter presented a method for improving the event-driven simulation performance, based on HDL models behavioral abstraction. The behavioral abstraction consist in the removal of some parts of the complete model, the behavior of which is not observed in the specific simulation run.

Two complementary methods are introduced to implement the behavioral abstraction: the input object invariance and observability of signals. As observed in experiments, the first method brings an improvement of simulation performance, which varies from approximately 25% to 42% for the circuit examples to which this method has been applied. An important characteristic of this method is that in the simulation of an accelerated model, the behavior of all objects declared in the model (ports and signals) can be observed.

The application of the second method, allowing the observability of selected signals or ports in the model, leads to the significant acceleration of simulation which, for the models tested in experiments, rises up to 2.7 times[9] (170% of increase) comparing to the initial model. The actual gain that can be obtained by applying this method is a trade-off between simulation efficiency and the number of selected observable objects in the model: if less objects are selected for observation, the gain in performance is higher.

This section presented the description of specific techniques that have been developed (and partially implemented) to perform the behavioral abstraction. The appropriate data structure (the object dependency graph) and the algorithms enabling model analysis, abstraction and optimization have been provided to make feasible the implementation of the method.

The behavioral abstraction can be combined with the other abstraction methods and other model acceleration techniques (i.e. optimization and transformation) to better exploit the potential of increasing HDL models simulation performance.

A possible extension of this work is to apply formal verification methods to prove that the behavior of the reduced model is contained in the behavior of the initial (complete) model. In other words, the objective is to prove that the reduced model implies the initial model for the subset of port and signal objects, or for the constant values assigned to input ports.

[9] The real gain that can be obtained varies in terms of a number of selected objects to be observed.

3.3 Data-type abstraction

3.3.1 Introductory definitions

In a VHDL model the objects can be declared as scalar or composite objects. The latter are used to define either collections of values of homogenous type: these are the arrays of values, or collections of values of potentially heterogeneous types: these are the record types. The data-type abstraction method proposed in this section is aimed at replacing objects of the detailed types (usually composite types, e.g. *bit_vector* or *std_logic_vector* type) by objects of more abstract types (often scalar types, e.g. *integer*).

The potential replacements under consideration are summarized in table 2.4 together with the gain, which is possible to obtain while replacing the objects of one type (either variables or signals) by the same objects of another type in the assignment statement.

The following language objects declared in a design unit are considered in the data-type abstraction method: constants, ports, signals, shared variables and variables. The data-type abstraction method can be applied in an automatic way to a set of objects in the model that are interrelated.

Definition 3.9: *Interrelated objects*

The ***interrelated objects*** of the model are those language objects (i.e. constants, ports, signals or variables) that are one of the following:

- the target and all sources of an assignment statement (signal or variable assignment);
- the objects of the *expression* in the
  - **case** statement;
  - **return** statement;
  - selected signal assignments;
  - guard of a **block** statement;
- the objects of every sub-expression which is a part of the *condition* (and which evaluates to the Boolean value) in the:
  - **wait** statement;
  - **if** statement;
  - **while** iteration scheme of the **loop** statement;
  - **when** condition of the **next** statement;

- **when** condition of the **exit** statement;
- conditional signal assignments;
- generate statement.

◊

The abstraction method starts with setting up for (possibly) every object of the model, a hypothesis that the type of a given object can be replaced by another data type, which is considered as a more abstract (in terms of the granularity, resolution, range of values or the internal representation). The abstraction is subordinated to the simulation performance of the considered data type, i.e. the selection of an abstract data type, to which the currently used type will be transformed, is determined by the simulation efficiency of the objects of a given type. When the data-type abstraction can apply, two categories of analysis are performed on the object: the analysis of the interrelation of that object with other objects of the model and the investigation of the kind of the usage of a given object in the model.

The objective of the first analysis is to determine the group of interrelated objects in the model. If a given object under consideration is interrelated with other object(s) in the model, the change of the type of that object should be followed by the corresponding change of the type of all interrelated objects. As an example, if we consider a simple assignment statement of the type $a <= b + c$, where a, b and c are the signals of the type *bit_vector*, if the abstraction of the type of the signal a from *bit_vector* to *integer* will be performed, the abstraction of signals b and c to *integer* type should also be done.

The second type of analysis is carried out to verify which categories of operations are defined on the object, and whether or not the object is partially used in some statements of the model. For the sake of clarity, the following definitions are introduced:

Definition 3.10: *Compound object*

Every object of a non-scalar type is a ***compound object***. In particular, the arrays of objects (one-dimensional, i.e. vectors, and multidimensional) and the records are compound objects. ◊

Definition 3.11: *Sub-element*

A ***sub-element*** of a compound object is one of the following:

- an index of an array,
- a slice of an array,
- a field of a record.

◊

In the model, the sub-elements of compound objects, although grouped together by the declaration of the object under a single object name, can be accessed or used independently in different parts of the model.

Definition 3.12: *Partial usage of a compound object*

If a sub-element (or sub-elements) of a compound object is (are) used independently in the model, we denote this situation as a ***partial usage*** of the compound object. ◊

Definition 3.13: *Bit-level operators*

A ***bit-level operator*** is either a logical operator, shift operator or concatenation operator (cf. [59] section 7.2 for the definition of logical, shift and concatenation operators). ◊

Definition 3.14: *Type dependency attributes*

For every two interrelated objects the following attributes, denoted as ***type dependency attributes***, can be associated:

- *partial usage attribute*, which indicates if the partial usage of an object in relation with other object takes place; this attribute, if any, is specified in terms of the sub-element(s) accessed (e.g. index or slice);
- *bit-level operation attribute*, which indicates, if any, whether or not objects are interrelated by a bit-level operator.

◊

The purpose of defining the type relation attributes for each pair of objects is to store the information about the kind of usage of the object in the model. This information is further used to determine whether the abstraction of the type of the object can be performed in the model.

3.3.2 Method description

The data-type abstraction can be performed on a group of interrelated objects, only if the following conditions hold:

- there is no partial usage of any interrelated object in the model and
- there are no operations in the model on the interrelated objects as operands, in which bit-level operators are used.

Thus, the procedure of the data-type abstraction consists in (1) creating groups of interrelated objects, in (2) extracting information about the type dependency attributes for each pair of interrelated objects, i.e. the partial usage attributes and the bit-level operation attributes, and in (3) selecting proper groups of objects, for which the type can be abstracted.

The following section gives the overview of the way, in which this procedure operates and how it can be implemented.

3.3.3 Method implementation

In order to make possible the analysis as described above, a type dependency graph is created for the entire model. This graph stores all the information relevant to the objects declared in the model: it reflects the complete picture of the interrelations between objects, as well as the attributes characteristic to each relation. The definition below formalizes the introduction of the type dependency graph.

Definition 3.15: *Type dependency graph*

A ***type dependency graph*** of a given model is a non-directed graph, in which:

- the nodes are the following objects: constants, variables, shared variables, signals and ports declared in the model,
- the edges connect the interrelated objects and
- the edges are labeled with the type dependency attributes: partial usage attributes and bit-level operation attributes.

A well-formed type dependency graph for a given design unit is a graph, in which all the objects declared in the unit are represented as nodes, and all relations between those objects are represented by edges with the appropriate, if any, type dependency attributes. ◊

Algorithm 3.1: *Type dependency graph creation*

Symbolic algorithm of the creation of the type dependency graph can be described as follows:

Initially, for every objects declared in the model, a single unconnected node is created. In the next step every single statement appearing in the model, the expression and condition in the statement, is analyzed in order to:

- connect the nodes labeled with objects that are interrelated;
- label the edges with the type dependency attributes, if the objects are related by the partial usage or bit-level operation.

After analyzing all the statements in the model, the type dependency graph is divided into partitions that collect nodes interconnected by edges, possibly decorated by labels. The partition encloses the nodes, which are interconnected between them. The graph partition that groups the nodes in which all the edges are not labeled by any label is denoted as *total*. Otherwise, the partition is denoted as *partial*. In the total partition all objects have the same data type. In the partial partition the objects can have types derived from the types of other objects in the partition. Note that the graph partition can contain a single node.

Algorithm 3.2: *Type abstraction of design unit objects*

1. Setting up the hypothesis of possible data-type abstraction for every object in the design unit.
2. Creation of type dependency graph for the model
3. For every partition in the graph:
   - 3.1. If the partition contains a single node (i.e. the node is not connected to any other node in the graph): the data type of the object represented by that node can be replaced by any other abstract data type (according to the valid hypothesis of the replacement).
   - 3.2. If the partition collects more that one node (nodes of a partition are connected between them):
     - If the partition is a total partition: the data type of all objects belonging to that partition can be replaced by another abstract data type (according to the valid hypothesis of the replacement).
     - If the partition is a partial partition, i.e. in that partition at least one edge is labeled with the type dependency attribute, the data types of all objects in the partition should remain unchanged.
4. Perform the data-type abstraction for the selected objects according to the previously defined hypothesis.

3.3.4 Experimental results

Table 3.3 presents the results of application of the data-type abstraction method to some industrial examples. Four abstractions have been performed on the *fiforx* model with the final gain in simulation time of 1.34 times for the first simulator and 2.3 times for the second one, three abstractions on *rxbit* model with the gain of 1.08 (1.19) times and ten abstractions on *bus_interface* model with the improvement of simulation performance of 1.72 (1.81) times. More details about the abstractions performed (e.g. types of abstracted objects, number of objects abstracted) and about the results can be found in Annex C.

Data-type abstraction: gain						
Abstraction number	**Simulator 1**			**Simulator 2**		
	Fiforx	**rxbit**	**bus_int**	**fiforx**	**rxbit**	**bus_int**
Initial model	1.000	1.000	1.000	1.000	1.000	1.000
Data-type abstraction 1	1.326	1.025	1.289	2.287	1.054	1.295
Data-type abstraction 2	1.340	1.056	1.589	2.292	1.189	1.587
Data-type abstraction 3	1.361	**1.077**	1.606	2.444	**1.188**	1.593
Data-type abstraction 4	**1.343**	-	1.663	**2.298**	-	1.731
Data-type abstraction 5	-	-	1.664	-	-	1.733
Data-type abstraction 6	-	-	1.667	-	-	1.736
Data-type abstraction 7	-	-	1.701	-	-	1.798
Data-type abstraction 8	-	-	1.708	-	-	1.799
Data-type abstraction 9	-	-	1.717	-	-	1.802
Data-type abstraction 10	-	-	**1.723**	-	-	**1.807**

Table 3.3: *Data-type abstraction method application results*

3.3.5 Concluding remarks

The data-type abstraction method offers a powerful mechanism to increase the simulation performance of HDL models. As shown in the experiments, the improvement in simulation obtained by the application of this method went up to 2.3 times for some industrial circuit models tested.

This section provided the necessary techniques to make possible the implementation of the data-type abstraction method of a given model. In order to perform the data-type abstraction, an analysis of dependency between objects in the model should be completed. This analysis allows to partition all objects declared in the model into groups of interrelated objects of the same type. The second objective of this analysis is to determine the kind of the relation between the interrelated objects. Taking into account some constraints that can prevent the abstraction of a given group of objects, like the partial usage of objects or the bit-level operation on objects, this analysis allows to make the decision, whether the abstraction of a given group of objects is feasible or not. The applicability of the data-type abstraction method depends only on the way in which objects are used in the model, i.e. the particular characteristics of the model imply the feasibility of the data-type abstraction.

The benefits that can be brought by the data-type abstraction depend also on the particular tool on which the model is simulated. More precisely, they depend on the difference in simulation efficiency between objects of a detailed data type and objects of a more abstract data type; this difference in performance is specific to a particular simulator.

3.4 Object abstraction: replacement of signal by variable

A particular type of abstraction of objects used in the VHDL model is the replacement of a signal object by a variable object of the same type. This type of abstraction can be applied to signals with no delay specified.

There exist four different kinds of usage of the signal objects in VHDL:

1. Signal is used locally in one process and a single assignment statement with that signal as a target is declared in the model. In that case no resolution function is required.
2. Signal is used locally in one process and multiple assignment statements with the that signal as a target are declared in the model. In that case a resolution function is required.
3. Signal is used globally, i.e. in more than one process, but it does not appear either in the sensitivity list of any process of the model, or in any wait statement in the model.
4. Signal is used globally, i.e. in more than one process, and at the same time it is used in the sensitivity list of one or more processes of a model, or in one or many wait statements.

The following sections describe in detail the transformation rules for the replacement (abstraction) of the signal object by the variable object, which is called *object abstraction*. The transformation rules are adapted to the kind of usage of the signal in the model (as defined above). Each of these definitions is preceded by a set of conditions determining the particular case to which the given rule applies.

3.4.1 Local signal usage in one process / single signal assignment

3.4.1.1 Conditions

This case covers the signal usage in the following situation:

- signal is used as a source or target inside a single process;
- no multiple assignments to that signal are specified in the model;
- no resolution function is specified;
- if the signal is a target in the signal assignment statement: no delay is specified (no **after** clause specified followed by the time expression in the assignment statement).

3.4.1.2 Transformation rule

The signal can be replaced by the local variable of the same type.

3.4.2 Global signal usage / local assignment

3.4.2.1 Conditions

This case covers the signal usage in the following situation:

- signal is used as a target inside a single process;
- signal is used as a source in one or many processes;
- no multiple assignments to that signal are specified inside the process;
- no resolution function is specified;
- if the signal is a target in the signal assignment statement: no delay is specified (no **after** clause specified followed by the time expression in the assignment statement).

3.4.2.2 Transformation rule

The signal can be replaced by a shared (global) variable of the same type. No resolution function is required.

3.4.3 Global signal usage / multiple assignments

3.4.3.1 Conditions

This case covers the signal usage in the following situation:

- signal is used as a target inside one or many processes;
- signal is used as a source in one or many processes;
- multiple assignments to that signal can be specified inside the process;
- resolution function declared;
- if the signal is a target in the signal assignment statement: no delay is specified (no **after** clause specified followed by the time expression in the assignment statement).

3.4.3.2 Transformation rule

The transformation rule defined for this particular case and described in details below comprises the substitution of a signal object by a set of shared variables as well as the declaration of a specially designed variable resolution function. All the signal assignment statements declared in a model are replaced by the variable assignment statements which apply the variable resolution function.

The signal to be substituted in the transformation procedure is called a *replaced signal*.

For each signal assignment statement a global variable is declared. This variable is called a *value transfer variable*. With each value transfer variable an enumeration number is associated.

An additional global variable is declared to store the actual value of the replaced signal. This variable is called a *result variable*.

Each signal assignment statement with the target of the replaced signal is substituted by the variable assignment statement of the following form:

- the target is a new value transfer variable;
- the source is a call to the variable resolution function (defined below) with two parameters: the value transfer variable number and the value of the expression to be assigned initially to the replaced signal.

The *variable resolution function* is defined as follows:

- Function "pureness":

  The variable resolution function is an impure function. This enables the usage of shared variables defined outside of the declarative part of the function.
- Formal parameter list:

  Parameter 1: the value transfer variable number – the number of the value transfer variable to be assigned in the current variable resolution function call;

  Parameter 2: the value to be assigned in the current variable resolution function call to the shared variable: result variable;
- Function body:

  The function body is divided into two parts:

  1. In the first part a new value is assigned to the value transfer variable corresponding to the *Parameter 1* of the formal parameter list. The value is determined based on the *Parameter 2* of the formal parameter list. The assignment is performed using a **case** statement.

2. The second part is devoted to determine the value of the result variable. It is similar to the signal resolution function.

- Return type and value:

  The returned type/subtype is the same as the type/subtype of the replaced signal.

  The value corresponds to the value obtained in the evaluation of the *Parameter 2* of the formal parameter list, which corresponds to the value assigned to the replaced signal assignment statement.

3.4.3.3 Example

Below in figure 3.11 an example of the replacement of a signal object by a shared variable is presented. The package *PSV* contains the declaration of type *ulogic*, the subtype *logic*, which is a resolved subtype of the type *ulogic*, as well as the shared variables declarations. The function *resolved* is a resolution function for the signals of the type *ulogic*. The function *resolved_SV* is a variable resolution function.

In the process *P1* of the architecture body *ASV* the signal assignment statement *A <= IN1*; is replaced by a variable assignment statement with the call of the variable resolution function. The process *P2* is similar to *P1*, the second variable assignment statement is included here. The last process *P3* presents the usage of the shared variable *A_SV*, which replaces the initial signal *A*.

```
package PSV is
    type ulogic is ( '0', '1', '2', '3');
    shared variable    A_SV  : ulogic;
    shared variable    C_1   : ulogic;
    shared variable    C_2   : ulogic;

    type ulogic_vector is array
        (natural range <> ) of ulogic;
    type logic_table is array ( ulogic, ulogic)
        of ulogic;

    constant resolution_table : logic_table := (
        ( '0', '1', '2', '3'),     -- | 0 |
        ( '1', '1', '2', '3'),     -- | 1 |
        ( '2', '2', '2', '3'),     -- | 2 |
        ( '3', '3', '3', '3') );   -- | 3 |

    function resolved ( s : ulogic_vector )
        return ulogic;
    impure function resolved_SV
        (i : integer; s : ulogic) return ulogic;

    subtype logic is resolved ulogic;
end PSV;

package body PSV is
  -- signal resolution function (traditional)
    function resolved ( s : ulogic_vector )
        return ulogic is
        variable result : ulogic := '0';
    begin
        if (s'Length = 1) then
            return s(s'Low);
        else
            for i in s'range loop
                result := resolution_table(result, s(i));
            end loop;
        end if;
        return result;
    end resolved;

  -- variable resolution function
    impure function resolved_SV
        ( i : integer; s : ulogic ) return ulogic is
    begin
    -- Part 1: value transfer variable assignment
        case i is
            when 1 => C_1 := s;
            when 2 => C_2 := s;
            when others => null;
        end case;
    -- Part 2: result variable value determination
        A_SV := resolution_table(C_1, C_2);
        return s;
    end resolved_SV;
end PSV;
```

```
use WORK.PSV.all;
entity SV is
      port (  IN1    : ulogic;
              IN2    : ulogic;
              SEL1   : in    bit;
              SEL2   : in    bit);
end SV;

architecture ASV of SV is
      signal    A       : logic;
      signal    ASV     : ulogic;

begin
      P1: process (SEL1)
      begin
         --  the following statement is replaced in
         --  the model by the assignment statement
         --  to the shared variable C_1 (below):
         A <= IN1;
         -- variable resolution function called for
         -- the first value transfer variable
         C_1 := resolved_SV(1, IN1);
      end process;

      P2: process (SEL2)
      begin
         --  the following statement is replaced in
         --  the model by the assignment statement
         --  to the shared variable C_2 (below):
         A <= IN2;
         -- variable resolution function called for
         -- the second value transfer variable
         C_2 := resolved_SV(2, IN2);
      end process;

      P3: process (SEL1, SEL2)
      begin
         -- usage of the result variable
         ASV <= A_SV;
      end process;

end ASV;
```

Figure 3.11: *Example of the usage of shared variables in place of signals*

3.4.4 Global signal usage / signal in sensitivity list

No signal to variable transformation is possible due to the nature of the synchronization of the processes by signal objects.

3.4.5 Experimental results

Table 3.4 shows the results of application of the object abstraction method to some industrial examples. In the experiments seven objects have been abstracted in model *fiforx* and *rxbit* with the final gain in simulation performance of 1.39 and 1.23 times respectively (for the second simulator used in experiments: 1.63 and 1.20 times). In Annex C a detailed description of the abstraction application results are presented.

Object abstraction method: gain						
Abstraction number	**Simulator 1**			**Simulator 2**		
	fiforx	**rxbit**	**bus_int**	**fiforx**	**rxbit**	**bus_int**
Initial model	1.000	1.000	1.000	1.000	1.000	1.000
Object abstraction 1	1.036	1.137	1.025	1.027	1.086	1.036
Object abstraction 2	1.040	1.193	1.085	1.031	1.101	1.033
Object abstraction 3	1.047	1.199	**1.104**	1.034	1.105	**1.046**
Object abstraction 4	1.057	1.209	-	1.052	1.139	-
Object abstraction 5	1.073	1.229	-	1.061	1.147	-
Object abstraction 6	1.264	1.238	-	1.609	1.178	-
Object abstraction 7	**1.390**	**1.230**	-	**1.630**	**1.196**	-

Table 3.4: *Object abstraction method application results*

3.4.6 Concluding remarks

The object abstraction method presented in this section allows to achieve an important gain in simulation performance. The previous paragraph of this section demonstrates some results of application of this method to industrial examples (the same examples have been used to prove the viability of other abstraction methods). According to these results, the substitution by variables or shared variables of all signal objects in the model, which are candidate to such an operation, represents a speed-up in simulation going up to 1.6 times in comparison to the simulation of the initial model.

However, this method, even if it offers an important advantage in simulation performance, limits the visibility of internal signals of the model. On one hand, the signals are declared in the model to ensure the synchronization between processes as well as to enable the accessibility to the signal objects from many processes. On the other hand, the objects, even if they are used locally, can be declared in the model as signals to allow the observability of their behavior in simulation. Thus, the fact that substituted signals will not be visible in the simulation, in some cases can reduce the applicability of this method.

3.5 Prototype tools

In order to support both behavioral abstraction methods: the observability of signals and the input object invariance, an object dependency graph generator has been developed. The graph is implemented as a set of lists, each of them collecting objects declared in a model: ports, signals, variables, shared variables and constants. To each object declared in the model a single list of objects, on which that object depends, is attached.

In its current status, the tool performing the data-type abstraction supports the automatic construction of the type dependency graph. The graph generated by the tool is decorated with the appropriate type dependency attributes: the partial usage attributes and the bit-level operation attributes. The tool creates partitions of the graph containing interrelated groups of objects and indicates the kind of relation between objects (expressed as attributes). This information serves currently as a support for the manual replacement of data types of objects in each partition, leaving to the user the selection of the abstract data types as well as the replacement of object declarations. Ultimately, the type dependency graph generation will be an integral part of a fully automated data-type abstraction tool.

At present, no tool support is provided for the object abstraction method. All experiments presented in this chapter have been carried out by manual modification of the VHDL code. However, an automated tool can be developed to perform the object abstraction based on the definitions provided in section 3.4. Conceptually, the tool should start its operation with a process of verification of conditions delimiting the kind of usage of signal objects in VHDL, for each signal declared in a model. If a given signal fulfill the set of conditions, for which one of the transformation rules has been defined, it can be replaced by a variable or a shared variable of the same type, as defined by a specific rule. If required, an appropriate resolution function should be declared. All signal assignment statements declared in a model, such that the abstracted signal is the target of the assignment, should be replaced by variable assignment statements (with the variable resolution function, if necessary).

The LVS system has been used in the development of the prototype tools supporting the abstraction methods (cf. short description of LVS in section 2.5.3, more details can be found in [75, 77]). The generation of object and type dependency graphs is based on an extension of the initial VIF schema definition in order to introduce new types of nodes and attributes. This allows to embody the graph data structure directly in the intermediate format of the model, in a form which is concise and easy to manage. The set of procedures of LPI (LEDA Procedural Interface), provided to handle nodes or attributes, or to perform specific actions on the intermediate format, has been adapted to cover new extensions. The extended browser tool used to visualize the model data structure has been recompiled to accommodate new definitions of nodes and attributes. One of the major advantages of the LVS system was its capability to regenerate the VHDL source code of the model from the extended intermediate format representation by a reverse-analysis process (the appropriate tool is called *reverse VHDL generator*). The process of reverse generation of the VHDL code takes into account modifications done on the VIF format and allows to check the consistency of the modified description. A possible future extension to Verilog Hardware Description Language can be developed in the same environment, since LVS offers a common VHDL*Verilog intermediate format V^2IF, which is a superset of VIF, together with the VHDL and Verilog compilers, reverse generators and browsers.

3.6 Conclusions

Table 3.5 presents a summary of the results obtained by a consecutive application of the optimization and transformation methods (defined in chapter 2) followed by the use of abstraction methods to accelerate the simulation of some industrial models. A comparison of these results with the results presented in sections describing each particular abstraction method, leads to the conclusion that the model the most efficient in simulation can be obtained by successive application of several of these methods to the same model. Taking as an example the *bus_interface* model, one can observe that individually the behavioral abstraction allowed to speed-up this model 1.99 times, the data-type abstraction resulted in 1.81 times acceleration, while the object abstraction brought an improvement in simulation of 1.1 times. All these methods jointly applied to the *bus_interface* model (completed by the optimization and transformation methods) result in a performance enhancement of 2.83 times.

The detailed design characteristics as well as the complete set of simulation results for these designs are presented in Annex C.

Acceleration methods summary: simulation performance improvement						
Acceleration method	**Simulator 1**			**Simulator 2**		
	fiforx	**rxbit**	**bus_int**	**fiforx**	**rxbit**	**bus_int**
Initial model	1.000	1.000	1.000	1.000	1.000	1.000
Transformation 1: process merge	1.023	-	1.017	1.022	-	1.051
Transformation 2: process merge	1.037	-	-	1.047	-	-
Object abstraction	1.399	1.212	1.095	1.685	1.158	1.081
Optimization 1: delete package reference	1.404	1.213	1.105	1.677	1.158	1.084
Optimization 2: constant instantiation	1.405	1.216	1.106	1.704	1.196	1.087
Optimization 3: function inline expansion	1.411	1.294	-	1.706	1.408	-
Transformation 3: if->case	1.413	-	-	1.721	-	-
Optimization 4: delete package reference + constant instantiation	1.413	-	-	1.721	-	-
Data-type abstraction 1	1.570	1.311	1.393	3.160	1.443	1.424
Data-type abstraction 2	1.575	1.315	1.697	3.234	1.646	1.745
Data-type abstraction 3	1.582	1.329	1.779	4.078	1.643	1.897
Data-type abstraction 4	1.575	-	1.790	4.044	-	1.903
Data-type abstraction 5	-	-	1.811	-	-	1.904
Data-type abstraction 6	-	-	1.814	-	-	1.904
Data-type abstraction 7	-	-	1.822	-	-	1.955
Behavioral abstraction 1: input object invariance	1.642	1.398	1.915	4.103	1.748	2.199
Behavioral abstraction 2: input object invariance	1.661	1.473	2.058	4.167	1.853	2.461
Behavioral abstraction 3: input object invariance	1.697		2.061	4.189		2.654
Behavioral abstraction 4: object observability	1.698	1.582	2.141	4.195	2.106	2.698
Behavioral abstraction 5: object observability	1.712	1.642	2.224	4.226	2.279	2.834
Final result	**71.2%**	**64.2%**	**122.4%**	**322.6%**	**127.9%**	**183.4%**

Table 3.5: *Summary of results obtained by the application of abstraction methods to some industrial designs*

All categories of abstraction presented in this chapter remain compatible with the existing simulation tools preserving the advantages of fine event-driven simulation. In this way, the tools developed to support them act as front-end tools. In the global design flow, the model resulting from the application of one of abstraction methods is stored in the library in parallel to the initial model under development. Only the latter is used in the next design steps (e.g. synthesis).

Since the generation of an abstracted model is done statically and automatically, it does not represent a significant additional overhead in computation, making the use of the abstraction methods easy to integrate and manage in the current design practice.

The possible extension of the work presented in this chapter can focus on the development of assessment techniques, which will enable to provide to the designer some estimations of performance improvement that can be obtained by applying the abstraction methods to a given model. These techniques, using the results of simulation performance measurements of HDL constructs on a given simulation tool, should enable to evaluate the possible outcome of performing different categories of abstraction on the objects of a particular model. This evaluation will give to the designer a useful indication about the potential improvement, which can be expected after applying these abstraction methods to a particular object. These estimations combined with the automatic carrying out of abstractions can efficiently support the designer at various stages of the design process in selecting the appropriate types of abstraction, as well as in pointing out the objects to which the given abstraction should apply.

In the work presented, we focused on the use of VHDL language. However, the methods presented are sufficiently general to be applied to other languages (e.g. Verilog).

Chapitre 4

Représentation du modèle par des graphes de décision de haut niveau

4.1 Résumé

Dans les chapitres précédents nous nous sommes penchés sur les méthodes d'amélioration des performances de la simulation dirigée par les événements, qui exploitent le potentiel d'accélération propre au langage de description de matériel.

Quant à ce chapitre et le chapitre suivant, ils sont consacrés à la présentation d'une méthode d'accélération de simulation dirigée par l'horloge. Cette méthode est basée sur une représentation mathématique de la fonctionnalité d'un système électronique par des graphes de décision de haut niveau (dénommé en anglais : *high-level decision diagrams*, DDs). Ce type de graphes offre un moyen compact et efficace de représenter, sur plusieurs niveaux d'abstraction, une fonction réalisée par le système. Par conséquent, nous considérons dans ce chapitre les deux niveaux d'abstraction: le niveau algorithmique et celui du transfert de registres.

Au fur et à mesure de la présentation de l'utilisation des graphes de décision pour la modélisation d'un système, nous donnons les définitions, la terminologie et les règles de la construction correcte de ces graphes. Puis, nous donnons les preuves de certaines propriétés de cette représentation et comparons les DDs avec d'autres types de graphes de décision. Ensuite, nous utilisons quelques exemples pour montrer l'application de cette représentation à la modélisation d'un circuit de chemin de données et d'une machine d'états finis.

Cette partie du chapitre est suivie par la présentation de l'application des graphes de décision à la modélisation des circuits au niveau transfert de registres et au niveau algorithmique. Ainsi, deux formes de graphes y sont introduites : les graphes de décision vectoriels (en anglais : *vector decision diagrams* - VDDs) et les graphes de décision vectoriels compressés (en anglais : *compressed vector decision diagrams* - CVDDs). Leur fonction est d'obtenir une modélisation du circuit qui est, à la fois optimisée en terme de complexité de la représentation, et efficace en simulation.

Chacune de ces formes de graphes a été appliquée afin de modéliser quelques circuits exemples. Ce processus a permis de tester et de comparer les formes de graphes entre elles. Il donne aussi la possibilité de confronter la méthode de simulation fondée sur l'utilisation des graphes avec les méthodes conventionnelles de la simulation, c'est-à-dire la simulation dirigée par les événements et la simulation dirigée par l'horloge.

4.2 High-level decision diagrams [10]

4.2.1 Definitions and terminology

Consider a digital system as a network $N = (Z, F)$ of components where Z is the set of all variables of the system, which are the input variables, output variables, state variables or variables representing the connections between components of the system. The variables can be of scalar type: enumeration type (e.g. Boolean type, multivalued types) or integer type, or else of composite type (e.g. one-dimensional arrays (i.e. vectors) or multidimensional arrays). Denote by $X \subset Z$ and $Y \subset Z$, correspondingly, the subsets of input and output variables. $V(z)$ denotes the possible values for $z \in Z$, which are finite. Let F be the set of digital functions on Z: $z_k = f_k (z_{k,1}, z_{k,2}, \dots , z_{k,p}) = f_k (Z_k)$ where $z_k \in Z$, $f_k \in F$, and $Z_k \subset Z$. Some of the functions $f_k \in F$, for the state variables $z \in Z_{STATE} \subset Z$, are next state functions.

Definition 4.1: *Decision Diagram*

A high level ***decision diagram*** (denoted as DD) is a rooted finite directed acyclic graph $DD = (M, \Gamma, Z)$ where M is a set of nodes, Γ is a relation in M, and $\Gamma(m) \subset M$ denotes the set of successor nodes of $m \in M$. The nodes $m \in M$ are marked by labels $z(m)$. The labels can be: variables $z \in Z$, terms of $z \in Z$, algebraic functions applied to terms of $z \in Z$, algebraic formulas, or constants. ◊

The set of nodes M is a union of three sets $M = I \cup \Phi \cup T$: the set of initial nodes I, the set of internal nodes Φ and the set of terminal nodes T.

The initial node $m_0 \in I$ has a label identifying the function or a set of functions represented by the diagram; its in-degree (the number of edges in-going to the node) is 0, the out-degree (the number of edges out-going from the node) is 1.

[10] Some parts of this chapter have been initially presented in [155, 156, 157, 158]

The internal nodes $m_i \in \Phi$ are labeled with variables, terms or algebraic formulas. Their in-degree is equal or bigger than 1 (i.e. the internal node can be connected to more than one node of the diagram). If the node labeling variable, term or algebraic formula is defined on the carrier denoted by S, the out-degree of the internal node can range between 1 and the cardinality $|S|$ of S.

The terminal node $m_T \in T$ is labeled with the terms defined on the carrier C of the function represented by the diagram. The term can be either a single variable, a constant (a single element of the carrier C) or a function applied to the arguments that are terms. The in-degree of a terminal node is equal or bigger than 1 (i.e. the terminal node can be connected to more than one node), its out-degree is 0.

The edge issuing from an internal node m_i is labeled by the term $e_i^j(m_i, m_i^v)$, where $1 \leq j \leq q_i$ and q_i is the total number of edges issuing from the node m_i. This term is defined on the carrier $C_j \subseteq C_i$, where C_i is the carrier of the term labeling the node of m_i.

For non-terminal node m, where $\Gamma(m) \neq \emptyset$, an onto function exists between the values of $z(m)$ and the successors $m^v \in \Gamma(m)$ of m. By m^v we denote the successor of m for the value $z(m) = v$.

Definition 4.2: *Activated edge*

An edge $e(m, m^v)$ which connects nodes m and m^v is called *activated* iff there exists an assignment $z(m) = v$. ◊

Definition 4.3: *Activated path*

A series of nodes connected by activated edges, which begins with the node m_i and finishes with the node m_j makes up an *activated path* $l(m_i, m_j)$. ◊

Definition 4.4: *Full activated path*

An activated path $l(m_0, m_T)$ from the initial node m_0 to a terminal node m_T is called a *full activated path.* ◊

Definition 4.5:

The decision diagram DD represents a function defined inductively as follows:

i). if DD consists of an initial node m_0 and a single terminal node m_T labeled by the formula P, the diagram DD represents P;

ii). If DD has an initial node m_0 labeled with f with edges labeled e_1, e_2, …, e_n leading to subdiagrams DD_1, DD_2, …, DD_n, and if each DD_i represents a formula P_i, then DD represents the formula:

$$DD = \bigvee_{1 \le i \le n} ((f = e_i) \wedge DD_i) \ . \ \Diamond$$

Definition 4.6: *Representation of a function by a decision diagram*

A decision diagram $DD_k = (M, \Gamma, Z_k)$ represents a function $z_k = f_k\,(z_{k,1}, z_{k,2}, \dots , z_{k,p}) = f_k\,(Z_k\,)$ iff for each value $v(Z_k) = v(z_{k,1}) \times v(z_{k,2}) \times \dots \times v(z_{k,p})$, a full path in DD_k to a terminal node m_T is activated, where $z(m_T) = z_k$ is valid. ◊

Each function $f_k \in F$ in the system network $N = (Z, F)$ is represented by a decision diagram $z_k = DD_k(Z_k)$ [9, 90].

The decision diagrams incorporate variables of abstract types to denote data values as well as function symbols to denote data operations. This allows to model complex digital systems (including sequential systems) in a concise and efficient way, independently of the width of the data path. The DDs can represent relations as well as sets of functions, with the structure sharing (the appropriate technique is presented in section 4.4). They significantly contribute to the decrease of the complexity of the system function representation, and throughout that, to the computation performance increase.

The above definitions only present a framework for the decision diagram construction. They do not provide, however, sufficient details that are necessary to build correctly constructed DDs, which offer a compact and efficient representation of the system functionality, and which can be manipulated by powerful algorithms. The following section addresses those requirements and provides a set of well-formedness conditions for the creation of decision-diagram representation.

4.2.2 Well-formedness conditions

Condition 1: *Kind of nodes*

The decision diagram should at least contain an initial node m_0 and a terminal node m_T. □

Condition 2: *Reachability*

Every node in the diagram is contained in at least one full activated path, i.e. there are no unreachable parts of the diagram. □

Condition 3: *Edge labeling terms*

The edge issuing from an internal node m_i is labeled by term $e_i^j(m_i, m_i^v)$ defined on the carrier $C_j \subseteq C_i$. This term can be one of the following: a single value v for the unique value of the node label $z(m) = v$, the term representing a range of values, the term representing an enumeration of values, or the term which is a combination of the preceding types: an enumeration of single values and/or ranges of values. □

Condition 4: *Mutual exclusivity of labeling terms*

Every two edges issuing from one node m_i should not be labelled by terms that represent the same value, either explicitly or as one of the elements in the range. In other words, the value v, to which the node label will evaluate, should uniquely appear as a labeling in all edges of that node. This condition preserves the determinism of the diagram representation. □

Condition 5: *Completeness of edge labeling*

If the node label $z(m_i)$ is defined on the carrier C_i, the edge labeling terms of all edges issuing from that node should cover all the possible values, to which the label $z(m_i)$ can be evaluated. It means that $C_1 \cup C_2 \cup, \ldots, \cup C_q = C_i$, where q is the total number of edges issuing from the node m_i, and the edge labels enumerate all elements of C_i. □

Definition 4.7: *Redundant node*

In the decision diagram, a *redundant node* is a node labelled by the term defined on the carrier C_i, with all edges labelled by terms that enumerate all elements of the carrier C_i, all leading to the same subdiagram. ◊

Condition 6: *Redundancy of nodes*

A well-formed decision diagram does not contain any redundant node. □

Definition 4.8: *Diagram isomorphism*

Two diagrams DD and DD' are said to be *isomorphic* if there exist a one-to-one mapping σ from all nodes of DD to the nodes of DD' such that for every node m of DD, if $\sigma(m) = m'$, then either:

i). both nodes m and m' are terminal nodes with $z(m) = z(m')$, and the carriers of labeling terms are equal $C_i = C'_i$ or

ii). both nodes m and m' are internal (non-terminal) nodes with $z(m) = z(m')$, the carriers of labeling terms are equal $C_i = C'_i$ and:

$$\forall_{\nu \in C_i} \sigma(m^\nu) = m'^\nu \quad \text{and} \quad \forall_{\nu \in C_i} e(m,m^\nu) = e'(m',m'^\nu)$$

Isomorphic diagrams represent the same function *f*.

◊

Definition 4.9: *Subdiagram*

A *subdiagram* S_R of a decision diagram is a rooted directed acyclic graph, for which the following conditions hold:

i.) subdiagram is rooted by one of the internal nodes of the main decision diagram called a *root node* m_R;

ii.) subdiagram contains all nodes of the main diagram that belong to all activated paths such that they start with the root node and they can be traced in the main diagram from the root node to the terminal nodes:

$$\forall_{m_i \in M} m_i \in S_R \ \text{ iff } \ \forall_{T} m_i \in l(m_R, m_T)$$

◊

Definition 4.10: *Reduced decision diagram*

A decision diagram is reduced iff:

i.) it contains no node m such that for all successor nodes of m denoted by m_i and m_j, where $i \neq j$, the nodes are labeled by the same node labeling term $z(m_i) = z(m_j)$;

ii.) it does not contain two distinct nodes m_i and m_j, such that the subdiagrams rooted by m_i and m_j are isomorphic.

◊

The lemma presented below follows from the definition of the reduced decision diagram.

Lemma 4.1:

For every node m in a reduced decision diagram, the subdiagram S_m rooted by m is itself a reduced decision diagram. □

Condition 7: *Minimality*

A well-formed decision diagram is a reduced decision diagram i.e. a diagram that does not contain any distinct isomorphic subdiagrams. □

Note that in DD representations no constraints are imposed on the appearance or ordering of the variables or terms labeling the nodes. However, it is evident that the ordering of terms in the labeling has an implication on the structure and complexity of the diagram.

Definition 4.11: *Well-formedness of decision diagram*

A decision diagram DD is said to be *well formed* iff it satisfies *conditions 1* through *7* of well-formedness, as defined above. ◊

Definition 4.12: *Ordered decision diagram*

A decision diagram is said to be *ordered* if each path, which can be traced in the diagram, has the same order of node labeling terms. ◊

4.2.3 Reduced decision diagram

A decision diagram can be reduced in size without changing the denoted function by eliminating redundant nodes and duplicate subdiagrams. This section presents the procedure to obtain the reduced form of the decision diagram and it follows with the proof of the concept.

4.2.3.1 Reduction procedure

Any number of distinct isomorphic subdiagrams of a decision diagram can be replaced by only one of those subdiagrams without changing the function represented by the diagram. The procedure is the following:

Two cases are distinguished: the subdiagrams are connected to the unique node (1) or to different nodes (2) of the decision diagram.

Case 1: The isomorphic subdiagrams are connected to the same node.

Consider a decision diagram in which there exist two isomorphic subdiagrams S_1 and S_2 representing the function f. Their root nodes are connected to the same node m_i by edges labeled with terms e_1 and e_2 respectively.

If the label $z(m_i)$ of the node m_i evaluates to the value which is one of those represented by the term e_1, the function f is obtained while activating the edge labeled by e_1; the same function is obtained if the root node label takes the value represented by the term e_2: here the edge labeled by e_2 is activated, what again leads to the subdiagram representing the function f.

In order to achieve a reduced form of a decision diagram it is sufficient to remove from the diagram one of the subdiagrams S_1 or S_2 and replace the edge label connecting the root node of the remaining subdiagram by the label which is the union of terms e_1 and e_2.

Now, if the node label $z(m_i)$ takes either the value represented by e_1 or by e_2 always a unique edge is activated and the same function f is obtained, represented by the unique subdiagram. If the diagram contains more than two isomorphic subdiagrams rooted at the node m_i, this procedure should be repeated for every two of them.

Case 2: The isomorphic subdiagrams are connected to distinct nodes.

Consider a decision diagram in which there exist two isomorphic subdiagrams representing the function f, connected to distinct nodes m_1 and m_2 by edges labeled with terms e_1 and e_2, respectively. Here, it is sufficient to remove from the diagram one of those subdiagrams, and add to the root node of the remaining subdiagram a new edge connecting that node with the node previously connected to the root node of the removed subdiagram. The label of the new edge will be the same as the label of the edge connecting the root node of the deleted subdiagram. The function represented by the decision diagram remains unchanged, since the unique representation of the function f is evaluated either from node m_1 or m_2. If the diagram contains more than two isomorphic subdiagrams rooted at different nodes this procedure should be repeated for every two of them.

For the proof of the procedure cf. the theorems 4.2 and 4.3 presented further in this section.

Note: the procedure described above, if applied to all isomorphic subdiagrams, allows to fulfill the *condition 7* of well-formedness of a decision diagram (cf. definition 4.11).

4.2.3.2 Proofs of properties of the decision-diagram representation

Theorem 4.1:

In the decision diagram every non-terminal node belongs to at least one full activated path. □

Proof: The proof is straightforward since this is the specific case of the *condition 2* on reachability of nodes: every node, also the terminal node, should be contained in at least one full activated path. □

Lemma 4.2:

If a decision diagram DD is isomorphic to DD' by a mapping σ, then for any node m in DD, the subdiagram rooted by m is isomorphic to the subdiagram rooted by $\sigma(m)$. □

Theorem 4.2:

For any function f there is a unique reduced decision diagram (for a given ordering of node labeling terms) representing f. Any other decision diagram (for the given ordering of terms) denoting f contains more nodes. □

Proof[11]*:*The proof is by induction on the number int_f of internal nodes of the diagram. If $int_f = 0$ then f must be defined by the term $z(m_T)$ labeling the unique terminal node m_T. The reduced decision diagram cannot contain terminal nodes other than m_T, since every node in the diagram is reachable (from the *condition 2* of the diagram well-formedness) and that would imply a full activated path corresponding to a set of argument values in the diagram which contains other terminal node than m_T.

Suppose DD contains at least one internal (non-terminal) node m such that every successor node m^v of m is labeled by $z(m_T)$, i.e. $\forall v \in C_m\ z(m^v) = z(m_T)$. Then, either some of those terminal nodes are distinct, in which case they constitute isomorphic subdiagrams or they are identical: $\forall v_i,\ v_j \in C_m$ holds $z(m^{v_i}) = z(m^{v_j})$. In either case, DD would not be a reduced decision diagram. Thus the only reduced decision diagram corresponding to the function $f = z(m_T)$ is the terminal node labeled by $z(m_T)$. Clearly, this is the minimal representation.

Next suppose that the statement of the theorem holds for any function g having $int_g < int_f$, where $int_f > 0$. Let m_i be the initial node of the decision diagram (the node connected to the diagram function node), and f_v be the function represented by the subdiagram for $z(m_i) = v$, $v \in C_{m_i}$. For all v, f_v are represented by unique reduced decision diagrams, since each of them has a number of nodes smaller than int_f.

[11] The proof proceeds along the same lines as the proofs for BDDs provided by Bryant [23] and for MDDs presented by Srinivasan et al. [140].

Let DD and DD' be two reduced decision diagrams for the function f. In the following it will be shown that these two diagrams are isomorphic, consisting of a root node m_i being the first node of the DD, and of the subdiagrams denoting the functions f_v , for all $v \in C_{m_i}$. Let m and m' be root nodes of DD and DD' respectively. The subdiagrams rooted by m and m' both denote the function f. The subdiagrams rooted by successor nodes of m and m' i.e. the nodes m^v and m'^v for all $v \in C_{m_i}$ and $v \in C_{m'_i}$ denote the functions f_v, and thus, by induction must be isomorphic according to some mapping σ_v. The claim is that the subdiagrams rooted by m and m' must be isomorphic according to the mapping σ defined as:

$$\sigma(n) = \begin{cases} m' & \text{if } n = m \\ \sigma_v(n) & \text{if } n \text{ is in the subdiagram rooted by } m^v \end{cases}$$

In order to prove this, one need to show that the function σ is well defined, and that it is an isomorphic mapping.

If a node n is contained in both the subdiagrams rooted by m^p and m^q then the subdiagrams rooted by $\sigma_p(n)$ and $\sigma_q(n)$ must be isomorphic to the one rooted by n and therefore to each other. Since DD' contains no isomorphic subdiagrams, this can hold only if $\sigma_p(n) = \sigma_q(n)$, and hence there is no conflict in the above definition of σ. Therefore, the mapping σ is well defined.

By similar reasoning, one can see that σ must be one-to-one mapping: if there were distinct nodes n_1 and n_2 in DD having $\sigma(n_1) = \sigma(n_2)$, then the subdiagrams rooted by these two nodes would be isomorphic to the subdiagram rooted by $\sigma(n_1)$ and therefore to each other implying that DD is not reduced.

Finally, the properties that σ is onto and is an isomorphic mapping follows directly from its definition and from the fact that each σ_v obeys these properties.

To show that DD and DD' are isomorphic, let DD be a reduced decision diagram representing f. Let m_i be the initial node in the diagram on which f depends. Diagram DD contains exactly one initial node, because if some other node existed, the subdiagrams rooted by it and m_i would be isomorphic. Suppose instead that there is some node n for which the number of internal nodes of the diagram rooted by n is $int_n < int_f$ but there is no other node w having $int_n < int_w < int_f$. The function f does not depend on the node m_i and hence the subdiagrams rooted by n^v, $\forall v \in C_n$ all denote f, but this implies that $n^p = n^q$, $\forall p,\ q \in C_n$, i.e.

DD is not a reduced diagram. Similarly, the node *m'* must be the root of DD', and hence the two diagrams are isomorphic. □

Theorem 4.3:

For all diagrams representing f and for the given ordering of node labeling terms, only the reduced diagram has a minimum number of nodes. □

Proof: Let DD be a decision diagram with the minimum number of nodes. Since the reduced diagram is unique, if DD is not the reduced decision diagram it would imply that DD either has a node m such that for all successor nodes m^p and m^q, $\forall p,\ q \in C_m$ there is $m^p = m^q$, or it contains two distinct nodes m and m' such that the subdiagrams rooted by m and m' are isomorphic. In either case, one can reduce the number of nodes in DD, contradicting that DD has the minimum number of nodes. Therefore, DD must be the unique reduced decision diagram. □

4.2.4 Example of the DD representation of a design

Depending on the class of digital system (or level of its representation), we may have various types of DDs, in which nodes have different interpretations and relationships to the system structure. In register-transfer level (RTL) descriptions, we usually decompose digital system into control and data parts. State and output variables of the control part serve as addresses and control words, and the variables in the data part serve as data words. High-level data word variables describe RTL component functions in data parts.

As an example, a subnetwork of a digital system and its DD are depicted correspondingly in figures 4.1 and 4.2. Here, R_1 and R_2 are registers (R_2 is also an output), M_1, M_2 and M_3 are multiplexers, + and * denote adder and multiplier, *IN* is an input bus, y_1, y_2, y_3 and y_4 serve as input control variables, and a, b, c, d, e denote internal buses. In DD, the control variables y_1, y_2, y_3 and y_4 are labeling internal decision nodes of DD with their values shown at edges. The terminal nodes are labeled by constant *#0* (reset of R_2), by word variables R_1 and R_2 (data transfers to R_2), and by expressions related to data manipulation operations of the network. By bold lines and colored nodes, a full activated path in the DD is shown from $x(m_0) = y_4$ *to* $x(m_T) = R_1 * R_2$, which corresponds to the pattern $y_4 = 2$, $y_3 = 3$, and $y_2 = 0$. By colored boxes, the activated part of the network at this pattern is depicted.

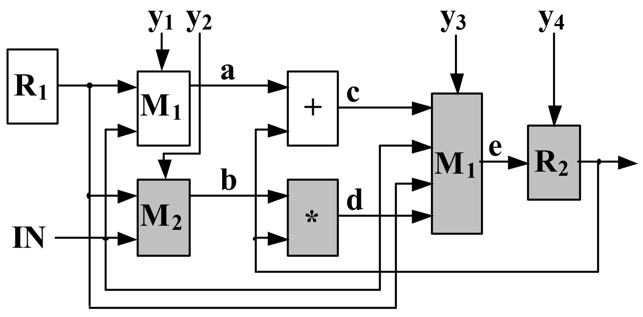


Figure 4.1: *Example of an RTL design*

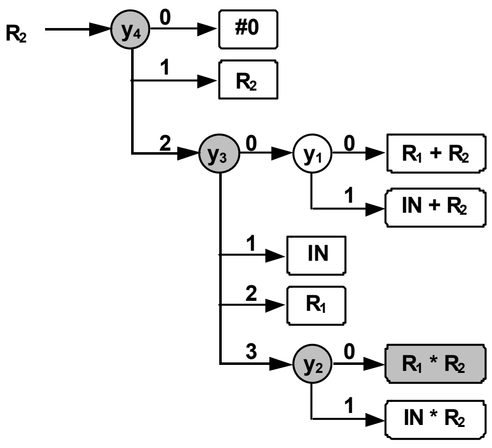


Figure 4.2: *Decision Diagram for the RTL design from figure 4.1*

4.2.5 Comparison between DD and other decision-diagram representations

4.2.5.1 Binary Decision Diagrams

The Binary Decision Diagrams (BDDs) are directed acyclic graphs used for the representation of Boolean functions in a canonical form [23]. The decision diagrams (DDs) can be considered as a generalization of binary decision diagrams (BDDs), while assuming that:

i.) the carrier set C of the function represented by DD is $C = \{0, 1\}$;

ii.) the node labeling terms are the variables defined on the carrier set $C_i = \{0, 1\}$, i.e. nodes have only two issuing edges labeled by constants *0* and *1*;

iii.) the terminal nodes can be labeled with constants *0* or *1*;

4.2.5.2 Edge-Valued Binary Decision Diagrams

In Edge-Valued Binary Decision Diagrams (EVBDDs) originally proposed by Lai et al. [72] all edges have associated weights. The value of a function is determined by additively combine those weights, i.e. by following a path from a root node to terminal node, summing the edge weights encountered. By selecting an appropriate scheme for edge weights the resulting graph provides a canonical and compact form of the function.

4.2.5.3 Binary Moment Diagrams and Multiplicative Binary Moment Diagrams

BMDs, as defined by Bryant and Chen [24, 25], provide a canonical representations for a class of linear functions defined on Boolean, integer, rational, or real variables. Thus BMDs can model the functionality of data path circuits operating over word level data. The node variables in these diagrams are Booleans. In multiplicative BMDs (*BMDs) the edges have weights, although in the evaluation of a function value, the weights are combined multiplicatively rather than additively. The conciseness of *BMDs allows for the representation of square or multiplication/exponentiation functions with, respectively, the polynomial (quadratic) or linear complexity.

4.2.5.4 Multi-Terminal Binary Decision Diagrams / Algebraic Decision Diagrams

Similarly to DDs, in the algebraic decision diagrams (ADDs) introduced by Bahar et al. [12] or Multi-terminal Binary Decision Diagrams (MTBDDs) proposed by Fujita et al. [40], the terminal nodes can be labeled with elements of the carrier set which is not necessarily composed of only two values. However, only constants are allowed as terminal node labels in ADDs/MTBDDs. The internal nodes of those types of graphs are limited to the Boolean variables with two out-going arcs. This type of diagrams is very well suited to model the operations on arithmetic (MTBDDs) and algebraic structures (ADDs): e.g. the matrix multiplications, shortest-path computations or solution of linear equations have been shown.

4.2.5.5 Multivalued Decision Diagrams

The Multivalued Decision Diagrams MDDs introduced by Srinivasan et al. [140] are diagrams for the representation of functions having multiple-valued inputs and multiple-valued outputs. In the diagram the nodes represent variables defined on the multiple-valued carriers, thus the number of edges issuing from a node is equal to the cardinality of the carrier set of the variable labeling the node. Similarly, the terminal nodes are defined on multi-valued carriers.

4.2.5.6 Multiway Decision Graphs

The Multiway Decision Graphs (MDGs) [32] and DDs have in common that any number of edges can issue from a given node. In addition to the features offered by DDs, the function symbols in MDGs can be used as edge labels. Moreover, the edge labels need not denote all the values in a given range (allow for the incomplete representation), and need not be mutually exclusive (allow for the non-determinism). The leaf nodes must be labeled by a Boolean value (*true*, except where the graph has only one node labeled *false*). The MDGs are used for the formal verification algorithms with abstract data types and uninterpreted function symbols as well as for efficient implicit state enumeration.

4.2.5.7 Comparison between diagram representations

Table 4.1 presents a categorization of the mentioned above decision-diagram representations. On one hand, the range of function, which can be represented by a given diagram, delimits the application to Boolean or numeric (i.e. integer, rational or real) functions. On the other hand, the node variable type can be either Boolean or numeric, while in the first case the function can be decomposed either "pointwise" [24] according to the value of a node variable, that can take value *0* or *1*; or according to "moments", i.e. how the function value will change as the node variable changes. Moreover, the values of a numeric function can be expressed in terms of values associated with the edges or with the terminal nodes.

Decision-Diagram Representation							
Node variable		Function range					
		Boolean	Numeric				
			Edge-weighted	Terminal			
Boolean	Pointwise	BDD	EVBDD	MTBDD, ADD			
	Moment	FDD	*BMD	BMD			
Numeric						Edge labels	
						constant	variable / function
				Terminal nodes	constant	MDD	MDG
					variable/function	DD	-

Table 4.1: Comparison between decision-diagram representations with regard to the range of represented function and node variable type

4.2.6 High-level DD representation of sequential circuits

4.2.6.1 FSM definition

A finite state machine of a sequential circuit is described using a finite set X of input variables, a finite set Y of output variables and a finite set of Z of state variables, which are pairwise disjoint.

The behavior of a state machine is defined by its transition and output relations, together with its set of initial states. Thus an abstract description of a state machine is a tuple $FSM = (X, Y, Z, Z', F_I, F_T, F_O)$, where:

i.) X, Y, Z are pairwise disjoint vectors of input, output and state variables; to allow for observable state variables (state variables that are also output variables), X and Z are the sets of all input and state variables respectively, while Y is composed from the output variables other than the observable state variables;

ii.) Z' is a set of next state variables; usually each next state variable is obtained by priming the corresponding state variable;

iii.) F_I is a DD representing the function $Z_0 \rightarrow Z$, where Z_0 is a set of initial state variables. F_I is a diagram representing the set of initial states;

iv.) F_T is a DD of type $(X \cup Z) \rightarrow Z'$, F_T is the transition relation;

v.) F_O is a DD of type $(X \cup Z) \rightarrow Y$, F_O is the output relation.

4.2.6.2 Example

As an example of the state machine representation we use a simple version of the MinMax circuit [32]. The MinMax state machine has 2 input variables $X = \{x, r\}$, two output variables $Y = \{m, M\}$ and one state variable $Z = \{c\}$. The variables r and c are defined on the Boolean carrier B, which is the enumeration $\{0, 1\}$, the variables x, m and M are of an abstract type, defined on the carrier C (a particular interpretation of it can be the set of integers).

The state machine assigns to the outputs m and M, respectively, the smallest and the greatest values appearing at the input x since the last active reset variable $(r = 1)$. When the reset variable is set, the variable m takes the maximal value max of the carrier C, and the variable M takes the minimal value min.

The graphical representation of the MinMax state machine is presented in figure 4.3, where the circles correspond to the values of the state variable c and the arcs correspond to the transitions of the machine. The labels on the transition arcs denote the conditions under which each transition is taken and an assignment of values to the output variables *m* and *M*.

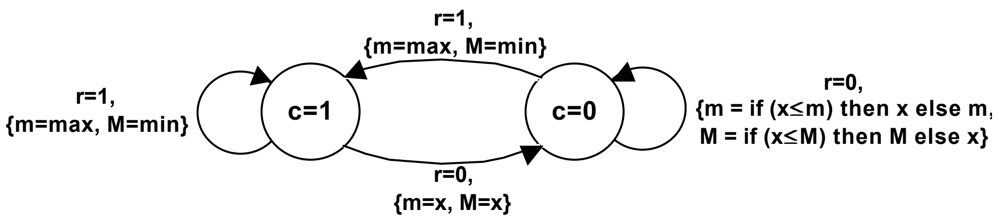


Figure 4.3: *MinMax example finite state machine representation*

In the DD representation of the MinMax state machine, a function "less-than-or-equal" (denoted by "≤") of type $C \times C \rightarrow B$ is used. Furthermore, the constant symbols *min* and *max* of a type *C* are used. The transition relation F_T associated to the next state variable is represented by DD of type $X \cup Z \rightarrow \{c'\}$ depicted in figure 4.4 (note that the next state variable depends only on the value of the input variable).

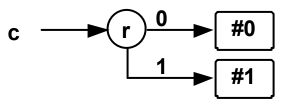


Figure 4.4: *Next state relation of the MinMax state machine*

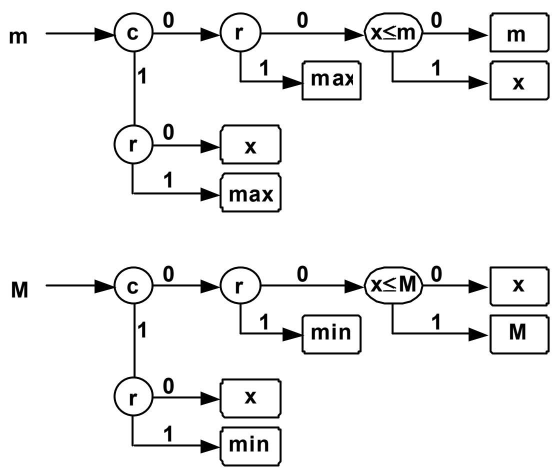


Figure 4.5 : *Individual output relations of the MinMax state machine*

The output relation F_O can be described by the set of individual output relations, one associated with each output variable. The DDs of these relations are shown in figure 4.5 and represent the assignments to the variable *m* of type F_m: $(X \cup Z) \rightarrow \{m\}$ and to the variable *M* of type F_M: $(X \cup Z) \rightarrow \{M\}$.

The output relation F_O can be represented by a single decision diagram jointly for both individual output relations F_m and F_M. This diagram can be obtained by conjunction of the respective individual output relations F_O: $(X \cup Z) \rightarrow \{m, M\}$. In the same way the entire finite state machine can be represented by a single decision diagram *FSM*: $(X \cup Z) \rightarrow \{c', m, M\}$, which is a conjunction of next state and output individual relations. The resulting decision diagram is presented in figure 4.6. In this diagram a traversal from the initial node to terminal nodes allows to determine the values of state and output variables *c*, *m* and *M* (here denoted as a compound variable *c.m.M*). A specific kind of node is introduced in this representation, which is called *addressing node* (cf. definition 4.16 in the reminder of this chapter), labeled with the addressing variable *i*. The labels of edges issuing from an addressing node correspond to elements of the compound variable *c.m.M*. Starting from the addressing node all subdiagrams rooted at that node should be traversed in order to evaluate current values of all state and output variables represented by the diagram. The appropriate method of creating this kind of representation is presented in details in section 4.4.

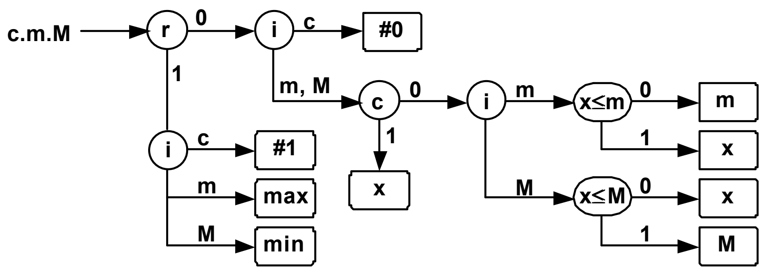


Figure 4.6: *MinMax finite state machine represented as a single high-level decision diagram*

4.3 Application of high-level decision diagrams at the register-transfer level[12]

4.3.1 Abstract

This section addresses the problem of efficient functional simulation of synchronous digital systems. A technique based on the use of decision diagrams (DD) for representing the

[12] Some parts of this chapter has been initially presented in [155]

functions of a design at RT and algorithmic level is introduced. The DD evaluation technique is combined with cycle based simulation mechanism to achieve a significant speed up of the simulation execution. Experimental results are provided for demonstrating the efficiency gain of this method in comparison to the event-driven simulation.

4.3.2 Introduction

This section focuses on the application of decision diagrams (DDs) also called alternative graphs to represent the functionality of a synchronous system at the RT or algorithmic level and combine them with the cycle-based simulation paradigm to improve the simulation speed. DDs are built separately for the data path and the control section of the design. In the simulation step DDs are evaluated using the input and previous state values in order to determine the next state and output function of the design.

Several methods, based on the application of various types of decision diagrams, have been proposed to overcome the problem of simulation performance improvement posed by the circuit complexity: the application of BDDs [81], branching programs [9] or MDDs [90] for fast discrete function evaluation are some of the examples of a research done in this domain. However, all of those methods are devoted to accelerate logic level simulation, and none of them has applied decision diagrams as a representation of design functionality at the higher level of abstraction (from the RT-level to the algorithmic level).

This part of chapter 4 is organized as follows: in section 4.3.3 decision diagrams are presented together with their application to the representation of sequential circuits. The cycle-based simulation mechanism using DDs is described in section 4.3.4. Some results from practical experiments are presented in section 4.3.5 followed by conclusions in section 4.3.6.

4.3.3 Design representation

Consider a system $S = (F, N)$ as a set F of components (or subnetworks) represented by functions $y = f(x)$ and a network N connecting these components. The system is represented by a set of variables $Z = \{IN, OUT, INT, REG\}$ defined by relationships of component functions $f \in F$. Here IN, OUT, INT and REG represent correspondingly the sets of primary input, primary output, internal bus and system state (register) variables. The set of components can be divided into a control part F_C and a data path F_D, $F = F_C \cup F_D$.

Definition 4.13: *System representation by decision diagrams*

A set of DDs $\{G_y\}$ represent a digital system $S = (F, N)$ if for each function $y = f(x)$ in F there exists a diagram $G_y = (M, G, x)$. The set $\{G_y\}$ is called DD-model for the system S. ◊

Note, in the DD-model we do not have the network N explicitly given. In the DD-model we assume that two variables connected through the network N have the same name. In other words, the set $\{G_y\}$ of the DD-model represents a set of diagrams connected by variables.

When one generates DDs for the components of the data path, at first, the DD for the control logic of the component should be created. If the binary level is implemented in the DD-model, the methods for creating structural DDs [154] can be used. In that case, each node in the diagram will represent a signal path in the gate-level control logic. Hence, the structure of the control circuit will be represented in terms of signal paths in the DD-model. If RT-level diagrams are to be created, we consider higher level (integer) variables which represent control fields of instructions, microinstructions or control buses. By using control variables, a DD is built up with one or more decision nodes in each path of the diagram, and, in general, with more than two output edges from each decision node. A multiplexer or a decoder corresponds to each decision node as a structural part of the module. After creating the decision part of the diagram, all the paths in the model will terminate with nodes labeled by expressions for data transfer, data manipulation or constants. As an example, a data path and its corresponding DD-model are depicted in figures 4.7 and 4.8.

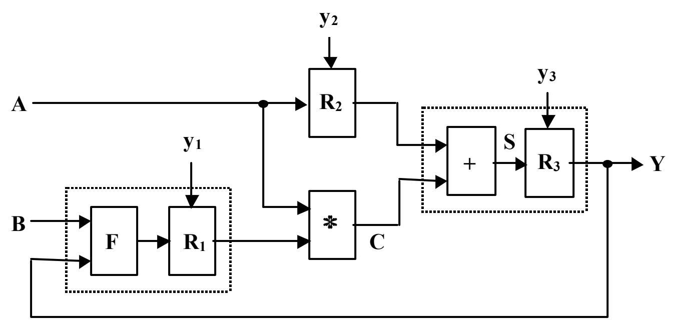


Figure 4.7: *Data path of a digital system*

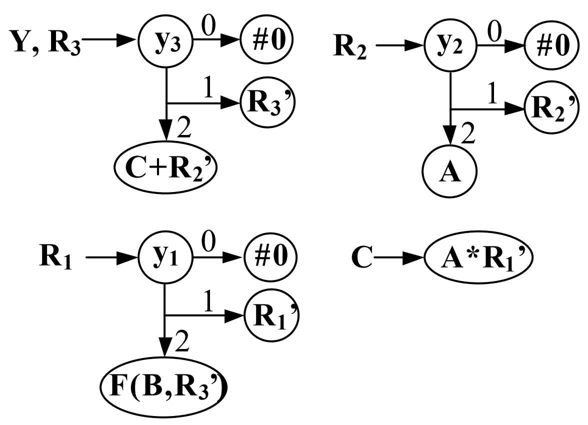


Figure 4.8: *DDs for the data path of a digital system*

The DD-model consists of diagrams G_{R2}, G_C, G_{R1}, G_{R3} for representing the functions of register R_2, multiplier C and two sub-networks R_1 and R_3, surrounded by dotted lines. In this example, y_1, y_2, and y_3 serve as control inputs, A and B are data inputs, R_1, R_2, and R_3 serve as data register variables (by apostrophe the previous state is denoted), C is the output variable of the multiplier and input variable of the adder, and Y is the primary output of the data path. In non-terminal nodes, only control variables are used as labels. Terminal nodes are labeled by data transfer, data manipulation expressions or constants (reset). Each node has a strong relation to the structure of the data path: non-terminal nodes represent the control logic in modules (subnetworks), the nodes labeled by data variables represent buses, and the nodes labeled by expressions represent data manipulation logic. As to the control part of the system, we generate DDs for the output and the next-state behavior for each finite state machine (FSM). In the case of Moore automata, both DDs can be joined. In the same way, as labels for the decision nodes, input and previous state variables of the FSM are used. Each pattern for these variables prescribes a path through the decision tree which would terminate with a node labeled by expression (or constant) to define the next state and the output behavior of the FSM. In figure 4.9 an FSM for controlling the data path presented in figure 4.7 is depicted. To represent the FSM defined by the next state and output function as shown in table 4.2, a DD is created for the vector function: $q, y_1, y_2, y_3 = f(q', R'_2 = 0)$. The predicate $R'_2 = 0$ is used here to represent a flag variable for reporting the state of the data path. We have two decision nodes in the diagram for analyzing the previous state q' of the FSM and the flag $R'_2 = 0$. The terminal nodes are labeled by constants (the values to be assigned to the vector variable q, y_1, y_2, y_3).

q'	x	q	$y_1\ y_2\ y_3$	Activities in the data path
0		1	1 1 0	y_3:R_3 := 0
1		2	2 2 1	y_1:R_1 := B, y_2:R_2 := A
2	R'$_2$=0	0	1 1 2	y_3:R_3 := A∗B
2	R'$_2$≠0	2	1 2 1	y_2:R_2 := A

Table 4.2: *Next state and output functions of the control part*

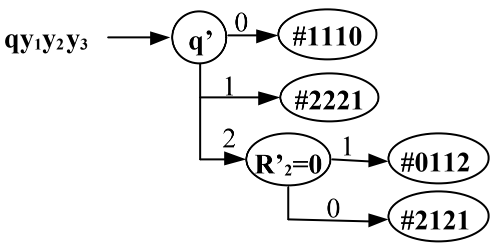


Figure 4.9: *DD-model for the control part*

In order to compress the DD representation of a model, the diagram superposition procedure proposed in [154] for gate-level structural diagram synthesis can be generalized to the RT-level case. Since control signals are usually either primary inputs for the data path or outputs from the control part, no superposition for non-terminal nodes is needed. The control part will be represented separately, and should not be mixed in the model with the data path. On the other hand, terminal nodes that represent data buses can be replaced by diagrams, which describe the components whose outputs are connected to the bus (in the example below *Y* and *R_3*). In this way, the complexity of the model will be reduced. An example of creating the DD for the subnetwork R_3 in figure 4.7 (surrounded by dotted lines) is shown in figure 4.10. The register with reset, hold and load functions (figure 4.10.a) and the adder (figure 4.10.b) connected to the register are represented by a single diagram (figure 4.10.c).

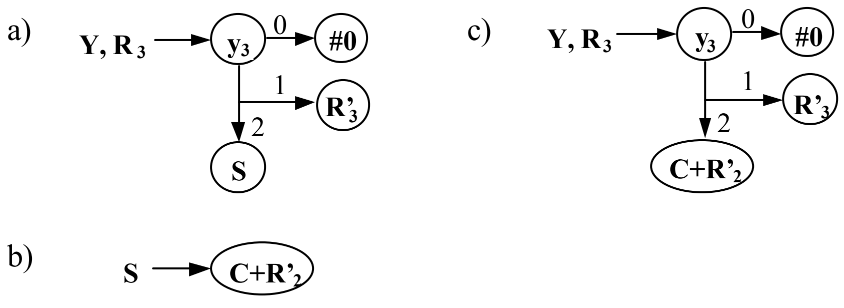


Figure 4.10: *Superposition of two DDs*

Another example of a superpositioned DD for the given subnetwork (figure 4.1) is shown in figure 4.2. Since the bus variables *a, b, c, d, e* disappeared from the DD, the complexity of the model is reduced, what helps to accelerate the cycle-based simulation.

4.3.4 Cycle-based simulation

Cycle-based simulation of synchronous digital systems is performed on a cycle-by-cycle basis. It assumes that there exist (one or many) clock signals in the circuit and all inputs of the systems remain unchanged during the evaluation of their values in the simulation cycle. The results of simulation report only the final values of the output signals in the current simulation cycle.

In the event-driven simulation each computation is proportional to the number of signals both internal and external in the design (number of drivers and updates). In DDs, only the changes of signals on which depend the outputs are used for evaluation, which allows to compute only the necessary information without the overhead associated with evaluation of intermediate values.

The idea of the cycle-based simulation on DDs is the following. The DDs are ranked in such a way that, when a DD is simulated, his arguments should be all either specified or computed. So, the simulation starts with DDs that depend only on input or state variables, i.e. they are specified either by the input vector or by the previous state (from the previous clock cycle). Afterwards, also these DDs that depend on the internal variables which, however, have been already computed in the current cycle, are simulated. In such a way there is a need to assess the output value of each DD, but the evaluation is performed only one time during each cycle. In the event-driven simulation, each change of a signal necessitates the re-evaluation of other signals dependent on it. So, in one simulation cycle there might be several consecutive re-evaluations of the same signal, which represent a significant cost in terms of execution time. In DDs during simulation, not all nodes are traced. Only the arguments of traversed nodes are required for counting. From that aspect, additional gain in simulation speed is achieved with DDs in comparison to other simulation models. The example in figures 4.1 and 4.2 shows that during the simulation for the control pattern $y_4 = 2,\ y_3 = 3,\ y_2 = 0,$ only one path through nodes $y_4,\ y_3,\ y_2$ should be traced with the computation $R_2 = R_1 * R_2$. This is the maximum length $(L = 3)$ of the path traversed in the DD. The shortest one is $L = 1$, and the average (for the case when the probabilities of values of y_i are equal) $L = 2/3 * 1 + 1/3 * (1/2 * 2 + 1/2 * 3) = 1.5.$ In the traditional case of network simulation, always the following sequence of operations should be computed: $a = f_1(y_1, R_1, IN)$, $b = f_2(y_2, R_1, IN)$, $c = a + R_2$, $d = b * R_2$, $e = f_3(y_3, c, d, R_1, IN)$, $R_2 = f_4(y_4, e, R_2)$.

In [78] a method was presented for the synthesis of DDs from VHDL where the fine-grained timing is replaced by a coarse timing, which helps to get rid of unnecessary details from the model not needed in cycle based simulation.

An example of a VHDL description and its cycle-based DD-model is represented in figures 4.11 and 4.12 [78] to illustrate the simulation procedure. For the first three processes of the VHDL description, DDs for computing *state, enable_in, reg_cp* are created, whereas for the last process, DDs for *next-state, outreg, fin, reg_cp_com* and *reg* are created. After superpositioning [78] of the model, only two DDs remain.

The simulation results of 6 clock cycles on these DDs are depicted in table 4.3. The paths traced on DDs for the first cycle are shown by colored nodes in figure 4.12. Only a part of the whole model (two decision nodes instead of eight) was processed, and no timing data (clock event variables like falling and rising edges) were used in computation, which in general results in a higher speed of simulation.

```
entity rd_pc is
    port (  clk, rst, rb0, enable       : in    bit;
            reg, reg_cp, outreg, fin    : out   bit);
end rd_pc ;

architecture archi_rd_pc of rd_pc is
    type StateType is (state1, state2);
    signal state, nstate   : StateType;
    signal enable_in       : bit;
    signal reg_cp_comb: bit ;
begin
    -- process P(state)
    process(clk, rst)
    begin
        if rst='1' then
            state <= state1;
        elsif (clk'event and clk='1') then
            state <= nstate;
        end if;
    end process ;

    -- process P(enable_in)
    process (clk, enable)
        begin
            if clk='1' then
                enable_in <= enable;
            end if;
    end process;

    – proc. P(reg_cp)
    process (clk, reg_cp_comb)
    begin
        if clk='0' then
                reg_cp <= reg_cp_comb;
        end if;
    end process;

    -- process P(nstate, out)
    comb: process (state, rb0, enable_in)
    begin
        case state is
            when state1 =>
                outreg <= '0' ; fin <= '0';
                if (enable_in='0') then
                    nstate <= state1;
                    reg <= '1'; reg_cp_comb <= '0';
                else nstate <= state2; reg <= '1';
                    reg_cp_comb <= '1';
                end if;
            when state2 =>
                if (rb0='1') then
                    nstate <= state2; reg <= '0';
                    reg_cp_comb <= '1';
                    outreg <= '0'; fin <= '0';
                elsif (enable_in='0') then
                    nstate <= state1;
                    reg <= '0'; reg_cp_comb <= '0';
                    outreg <= '1'; fin <= '1';
                else nstate <= state2;
                    reg <= '0'; reg_cp_comb <= '0';
                    outreg <= '0'; fin <= '1';
                end if;
        end case;
    end process;
end archi_rd_pc;
```

Figure 4.11: *VHDL description of a control block*

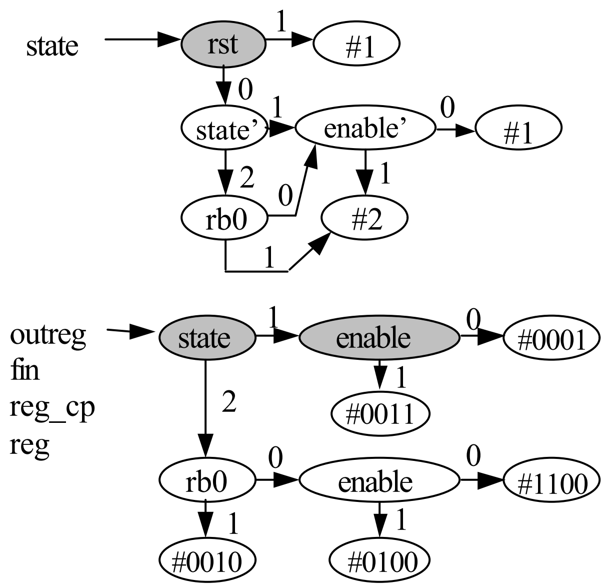


Figure 4.12: *DD-model for the VHDL description presented in figure 4.11.*

To fully exploit the advantage provided by the separation of the combinational part of the design from the sequential part, a dependency analysis can be performed on the combinatorial blocks. This is done in order to find the best ordering of the evaluation of blocks to ensure that the outputs of the blocks are evaluated only when necessary to avoid an overhead in computation.

cycle	1	2	3	4	5	6
rst	1	0	0	0	0	0
enable	0	1	1	1	0	0
rb0	x	x	1	0	0	0
state	1	1	2	2	2	1
outreg	0	0	0	0	1	0
fin	0	0	0	1	1	0
reg_cp	0	1	1	0	0	0
reg	1	1	0	0	0	1

Table 4.3: *Simulation results for 6 clock cycles*

4.3.5 Experimental results

A prototype of a simulator based on the decision-diagram representation and the cycle-based paradigm has been tested on benchmark circuits described in table 4.4. *gcd* and *diffeq* are the HLSynth benchmarks, *mult8x8* is an 8-bit multiplier using Robertson's algorithm, *huff* is a Huffman encoder circuit, *circ1* is a control dominated circuit example and *dat* is a data-path dominated circuit example. The experiments have been performed on Pentium II 200MHz computer with 128MB RAM running Windows NT. In the experiments each circuit is simulated for 200000 random input vectors. The execution time is measured using the internal system procedure *GetProcessTimes()*. The simulators used for experiments were ModelSim PE/Plus 4.7b (Model Technology, Inc) and VeriBest VHDL SysSim 98.0 (VeriBest, Inc.).

The detailed results obtained in experiments are presented in table 4.5. All results are in seconds; *Ratio* is the ratio between the event-driven simulation time and the DD-based simulation time. The improvement in the simulation performance while using DD based approach over event-driven simulators ranges between 2 to 9 times for the faster simulator and 4.6 to 37 times in the case of the slower simulator for the measured benchmark designs [13].

Circuit	Inputs	Outputs	Control states	Complexity after synthesis	
				Gates	F/F
gcd	10	4	8	227	15
diffeq	82	32	6	4195	115
mult8x8	18	16	8	1058	95
huff	5	5	42	1827	139
circ1	11	10	25	1683	345
dat	17	4	25	790	89

Table 4.4: *Benchmark circuits*

Circuit	DD-based simulator	VHDL Simulator 1			VHDL Simulator 2		
		Compilation	Simulation	Ratio	Compilation	Simulation	Ratio
gcd	3.89	0.63	13.13	**3.4**	3.58	33.98	**8.7**
diffeq	8.96	0.69	81.51	**9.1**	3.81	331.62	**37.0**
mult8x8	5.79	0.72	25.38	**4.4**	4.18	64.81	**11.2**
huff	21.71	1.93	44.05	**2.1**	56.06	42.94	**4.6**
circ1	209.22	1.39	242.35	**1.2**	43.96	499.01	**2.6**
dat	388.99	1.44	710.74	**1.8**	42.29	1815.66	**4.8**

Table 4.5: *Comparison between DD-based simulation and HDL event-driven simulation*

4.3.6 Concluding remarks

In this section the application of decision-diagram representation to the register-transfer level design has been presented. Based on this representation, a cycle-based

[13] The exception is the *circ1* design, for which the DD simulation was only 20% faster than the corresponding event-driven simulation. *circ1* is a control dominated design, and for this kind of circuits a specific DD representation, called register-oriented DD - RODD is introduced in chapter 5; RODD representation combined together with an appropriate simulation algorithm enables to efficiently manage the control dominated circuits in DD-based simulation.

simulation technique has been introduced as a simulation method of the network of DDs, which represents the function realized by the system.

Two effects contribute to the radical improvement of simulation performance, which can be achieved by applying the decision-diagram-based representation. The first effect, related to the DD simulation algorithm, is associated with the evaluation mechanism of the output values of the decision diagram. This evaluation is achieved by tracing, in each simulation cycle, a path in the diagram, which contains only a subset of all the diagram's nodes (i.e. the nodes labeled by some of the input variables). In consequence, this allows in several cases to save unnecessary evaluations of the intermediate variables' values (represented by parts of the diagrams), which do not contribute to the ultimate output values of the diagram. The second effect, inherent to the cycle-based simulation principle, allows to neglect the time specific information in the initial HDL description, as well as it permits to dismiss the computationally expensive scheduling mechanism of future values in the signal driver evaluation.

An initial comparison of the DD-based cycle simulation technique to the event-driven HDL simulation shows very promising results. The improvement of performance of the simulation, which has been observed in experiments on some example circuits, can reach the level between 2 and 9 times (respectively 4.6 and 37 times for other event-driven simulator).

4.4 Application of high-level decision diagrams at the algorithmic level [14]

4.4.1 Abstract

Decision diagrams (DDs) present a suitable way for the digital system representation efficiently used in several steps of the entire design cycle: for design verification, test generation and fault simulation. They offer a competitive simulation performance in comparison to the event-driven HDL-based simulation; however, their initial complexity does not allow to favorably compare their performance with the algorithmic functional description of the circuit's behavior. This section presents the optimization of the DD representation aimed at improving their simulation efficiency. Two types of DDs are introduced: vector

[14] Some parts of this section have been presented in [156, 158]

decision diagrams (VDDs) and compressed vector decision diagrams (CVDDs) as an efficient and concise model of system behavior, which preserves all the advantages of the decision diagrams. Their computational complexity is similar to the algorithmic description. The results of the comparisons between the different types of DDs and the algorithmic representation are included. This comparison is based on a symbolic complexity measure, defined for that purpose, as well as on the measure of the simulation time.

4.4.2 Introduction

Within the last decade *binary decision diagrams* (BDDs) have become the state-of-the–art data structure in VLSI CAD for representation and manipulation of Boolean functions [34, 96]. Several application of various forms of decision diagrams were devoted to accelerate the simulation performance. For example, McGeer et al. [90] applied a type of decision diagrams called multivalued decision diagrams (MDDs) for bit-vector discrete function representation and evaluation for logic level cycle-based simulation. Another approach based on BDDs and branching programs was presented by P. Ashar and S. Malik [9] for fast evaluation (simulation) of logic functions. Luo et al. [81] presented a cycle-based simulation technique for synchronous circuits which combines a BDD-based logic level cycle simulator with fast hierarchical evaluation of functional units stored in a library. All those approaches focus uniquely on the logic level simulation and do not address higher levels of abstraction.

There have been several approaches to broaden the use of *high-level decision diagrams* (DDs) for representing digital systems at higher levels of abstraction. The use of DDs as a model at the RT-level, presented in the previous section, is the first promising example of such an application. It has been shown that the performance of the simulation of the system represented by the DD model at the RT-level is significantly higher than the simulation of the hardware description language model (section 4.3 and [155]). Moreover, DDs have been successfully introduced as a uniform mathematical model of the system behavior not only for the purpose of simulation, but also in other domains of application in the design process: the design error diagnosis, the test generation and the fault simulation [153, 154, 155].

The purpose of this section is to propose the suitable representation based on the decision diagram mathematical model, which offers a comparable expressive power and the computational complexity to the functional (algorithmic) description of the system's behavior. Two types of the DDs are formally defined to represent the system behavior: vector decision diagrams (VDD) and compressed vector decision diagrams (CVDD). The first model is built by the superposition of the DDs representing system variables, which allows to combine several diagrams into one diagram representation. In the evaluation of the variable values only one traversal of the diagram nodes is necessary instead of multiple executions of diagrams for each variable. The compressed version of the VDD is constructed to implement directly the traversal of each path of the graph representing the algorithm. This type of VDDs regroups into clusters the control nodes and the data processing nodes separately, offering the possibility to optimize and reduce the operations performed inside each of those clusters.

Further presentation in this chapter is organized in the following way: section 4.4.3 introduces the system representation by means of DDs. In the two next sections the VDDs and the CVDDs are defined. The experimental results of the comparison between the decision-diagram representations of some representative algorithms are shown in section 4.4.6.

4.4.3 Decision diagrams for system representation at the algorithmic level

Consider a digital system in figure 4.13 with an algorithmic description in figure 4.14. The system consists of control and data parts. The finite state machine of the control part of the system is given by the output function $y = \lambda(q', x)$ and the transition (next-state) function $q = \delta(q', x)$, where y is an integer output vector variable, which represents a microinstruction with four control fields $y = (y_M, y_z, y_{z1}, y_{z2})$, $x = (x_A, x_C)$ is a Boolean input vector variable, and q is the integer state variable. The value j of the state variable corresponds to the state s_j of the FSM.

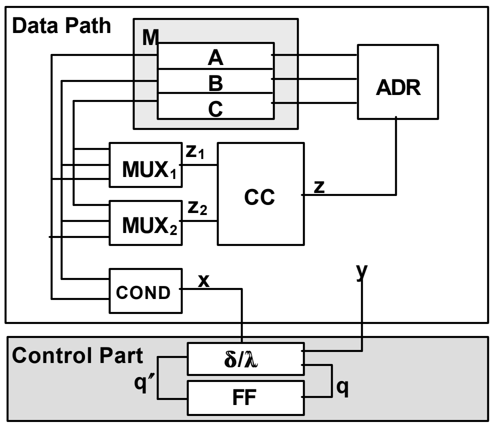


Figure 4.13: *Example of a digital system*

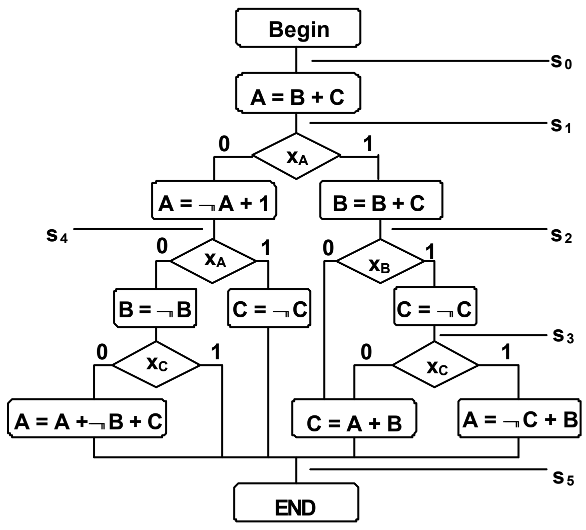


Figure 4.14: *Algorithmic representation of the digital system in figure 4.13*

The data path consists of the memory block M with three registers A, B, C together with the addressing block ADR, represented by three DDs: $A = G_A (y_M, z)$, $B = G_B (y_M, z)$, $C = G_C (y_M, z)$; of the data manipulation block CC where $z = G_z (y_z, z_1, z_2)$; and of two multiplexers $z_1 = G_{z,1} (y_{z,1}, M)$ and $z_2 = G_{z,2} (y_{z,2}, M)$. The block $COND$ performs the computation of the condition function $x = G_x (A, C)$. The component level model of the system consists of the following set of DDs: $N_1 = \{G_q, G_y, G_A, G_B, G_C, G_z, G_{z,1}, G_{z,2}, G_x\}$.

Note that because of a significant number of fanouts, the event-driven simulation paradigm applied for the evaluation of DD output values for this representation will not present any advantages in comparison to the conventional cycle-based simulation (cf. chapter 5 for details).

Using now the following chain of superposition of DDs:

$$A = G_A (y_M, z) = G_A (y_M, G_z (y_z, z_1, z_2)) =$$

$$= G_A (y_M, G_z (y_z, G_{z1} (y_{z1}, M), f_4 (y_{z2}, M))) =$$

$$= G_A (y_M, y_z, y_{z1}, y_{z2}, M) = G_A (y, M) = G_A (G_y (q', x), M) = G'_A (q', A, B, C)$$

we create a new *compact* DD model of the system:

$$N_2 = \{G_q, G'_A, G'_B, G'_C\}.$$

The part of the model related to the data path is represented in figure 4.15 by three diagrams G'_A, G'_B, G'_C . For simplicity, in these diagrams, the terminals nodes for the cases where the value of the function variable does not change, are omitted.

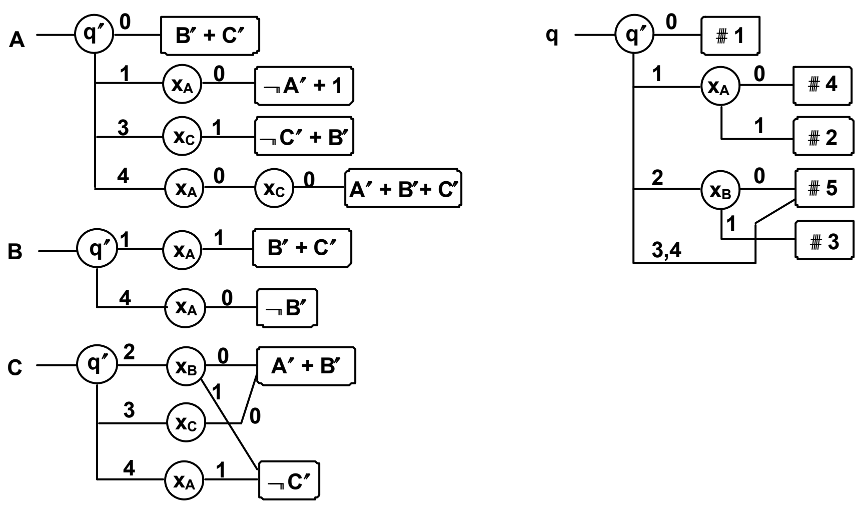


Figure 4.15: *The DD model N_2*

Due to the fanouts, even in the case of the compact DD-model, the event-driven simulation will not provide a significant gain in simulation speed. It is easy to see that in each simulation cycle the changing value of *q'* imposes the simulation of all the functions in the data path.

4.4.4 Vector Decision Diagrams

Consider now a method of a superposition of all DDs: G_A, G_B and G_C of the data path into a single diagram: $M = A.B.C = G_M(q', A, B, C, i)$ which produces a new concise model of the system $N_3 = \{G_q, G_M\}$ represented in figure 4.16 (the next state diagram for N_3 is the same as in figure 4.15, thus, it is not shown).

In order to formalize the above process of merging the decision diagrams to form a new type of decision diagrams denoted as vector decision diagram, the following definitions are introduced:

Definition 4.14: *Vector variable*

A *vector variable M* is a compound variable composed of the system variables $z_i \in Z$. We denote the vector variable M composed of n variables by $M = z_1.z_2. \ldots .z_n$. ◊

Definition 4.15: *Vector function*

A *vector function* is a mapping that assigns values to the vector variable *M*. ◊

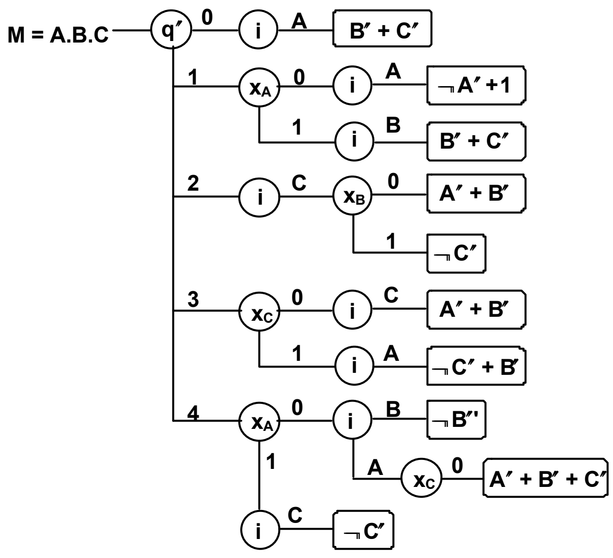


Figure 4.16: *The DD model of N_3*

For computing and assigning new values to different components of the vector variable *M*, we introduce a new type of a DD node called *addressing node a* labeled by an addressing variable *i*. Let have an array *M* with *n* memory locations (registers). The variable *i* may have values from the domain of addresses *i = 1, 2, ... , n* pointing to the memory location *M(i)*.

Definition 4.16: Addressing node

The addressing node *a* is a node of the diagram *G* having *n* successor nodes connected by edges labeled by an addressing variable $i \in (1, 2, \dots, n)$ (it means $z(a) = i$), for which all successor nodes m_i are traversed. ◊

All the successor nodes of the addressing node form the complex output value of the vector variable.

When entering, during the simulation, into the addressing node *a*, one has to traverse all the subdiagrams in the DD for which *a* is the root node. The output edges *i* of the node *a* are omitted when the value of *M(i)* does not change during the current simulation cycle.

Definition 4.17: Node labeling custom order

The order in which appear the labels of the nodes along a path of the decision diagram is called *node labeling custom order.* ◊

Definition 4.18: *Isomorphic paths*

Two paths in the diagram are *isomorphic* if they have the same number of nodes, the labels of the nodes of both paths appear in the same node labeling custom order, and the corresponding edge labels of edges connecting the nodes of a path are equal. ◊

Definition 4.19: *Partially isomorphic diagrams*

Partially isomorphic diagrams are diagrams for which one can distinguish one or more activated paths which are isomorphic. ◊

Definition 4.20: *Path remainder, path terminal node*

Consider the decision diagram, in which there exist a path $l(m_i, m_j)$. The *path remainder* is the subdiagram of the main decision diagram for which the root node is connected by an edge to the node m_j of the path l. The node m_j is denoted as *path terminal node*. ◊

Definition 4.21: *Vector decision diagram*

The diagram G called a *vector decision diagram* (VDD) is a reduced decision diagram representing a vector function $M = f(z_1, \dots, z_k)$, where $M = y_1.y_2. \dots y_n$ is a vector variable. ◊

The vector decision diagram is formed by merging the decision diagrams G_i representing the functions $f_i : y_i = f_i(z_1, \dots, z_n)$ defined for single system variables y_i.

In the DD-model of the system, the functions f_i are represented by separate decision diagrams among which some of them are partially isomorphic. Thus, it is possible to define a merging procedure, which produces a vector decision diagram from the separate diagrams for the system function. This merging procedure performs as follows:

Algorithm 4.1: *DD merging*

1. The vector variable for the output and state variables of the system is created, which will be represented by a single vector decision diagram.
2. All isomorphic paths of the partially isomorphic decision diagrams are merged together to form so called *unique path.*
3. The edges issuing from the terminal node of the unique path are formed from the edges issuing from the terminal nodes of the isomorphic paths, no redundant edges are introduced.
4. The addressing nodes are introduced at the end of each edge issuing from the unique path terminal node.

5. The edges issuing from the addressing nodes are labeled with the variable y_i if the edge connecting the isomorphic path terminal node and the given addressing node existed already in the initial decision diagram G_i.
6. The successor of the addressing node connected by edge labeled by y_i is the isomorphic path remainder of the merged path of the initial diagram G_i.

The vector decision diagrams offer the capability to efficiently represent the array variables (corresponding to register blocks and memories) for computing and updating their values. VDDs are particularly effective for representing functional memories with complex input logic – with shared and dedicated parts for different memory locations. In a general case, all the registers of the data path can be combined in the model as a single memory block.

For the example considered so far, a vector decision diagram is created to compute the values of the vector variable $M = A.B.C$ (cf. figure 4.16). In the new VDD the decision nodes for evaluating new values of the components of *M*, are traversed only once. This property will give the advantage to the new model N_3 over N_2. For example, in the diagram G_M in figure 4.16, for the input vector $q' = 4$, $x_A = 0$, $x_C = 0$, the decision nodes q' and x_A, are traversed for evaluating both new values of *A* and *B* only once, whereas in the model N_2 in figure 4.15 they should be traversed three times, separately for *A*, *B* and *C*.

The introduction of addressing nodes allows also to merge the decision diagram of the vector variable *M* with the next-state decision diagram. This integration forms a single vector decision diagram with the new vector variable $M = A.B.C.q = G_M(q', A, B, C, i)$ for which $N_4 = \{G_M\}$. The result of this operation is presented in figure 4.17.

As an example, a comparison of simulation results for all these four models N_1, N_2, N_3 and N_4 is given in table 4.6. A sequence through states s_0, s_1, s_2, s_3, s_5 is considered. The left entry in the table indicates the number of decision nodes (*if* operations) traversed during the simulation. The right entry denotes the number of data manipulation or transfer operations.

Model	$s_0 \rightarrow s_1$	$s_1 \rightarrow s_2$	$s_2 \rightarrow s_3$	$s_3 \rightarrow s_5$	Total
N_1	7/8	8/8	7/8	8/8	30/32
N_2	4/2	6/2	6/2	5/2	21/8
N_3	3/2	5/2	5/2	4/2	17/8
N_4	2/2	3/2	3/2	3/2	11/8

Table 4.6: Simulation with N_1, N_2, N_3 and N_4

From this example one can see that the superposition of DDs in the model will significantly speed up the simulation of the data manipulation procedures (transfer from the model N_1 to the model N_2). On the other hand, merging DDs (the models N_3 and N_4) by introducing the addressing nodes into the DD-model will reduce the total number of decision nodes to be traversed in diagrams.

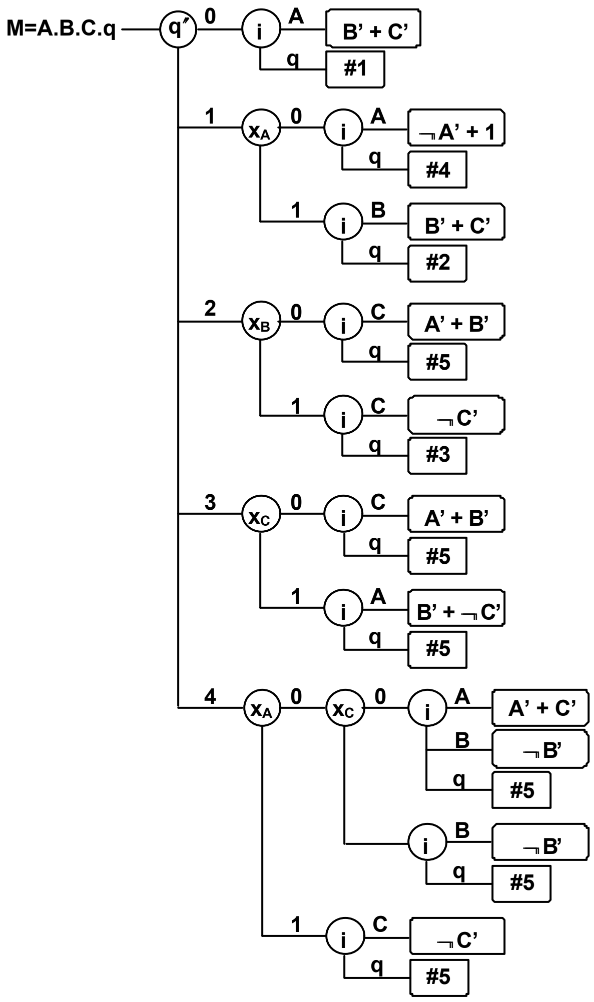


Figure 4.17: The DD model N_4

A symbolic heuristic algorithm can be proposed for the creation of the vector decision diagram.

Algorithm 4.2: VDD Creation

```
VDD creation:
{
   create the set $D$ of the decision diagrams for each output and state variable ;
   select the common node labeling custom order :
   {
      create a single set of all node labeling terms appearing in all decision diagrams of $D$ ;
      order the set in the way that it will start with those node labeling terms, which appear
         in the maximum number of decision diagrams of $D$. This set is called a common
         node labeling custom order ;
   }
   use the common node labeling custom order to rebuild all decision diagrams of the
   set $D$ ;
   integrate isomorphic paths ($D$, $VDD$) ;
}


integrate isomorphic paths (in $S$: set of DDs, out $DD$: decision diagram):
{
if $S \neq \emptyset$ then
   {
      create an empty decision diagram $DD$ ;
      create a set $P$ of all isomorphic paths in $S$ ;
      order the set $P$ according to the length of the paths ;
      for all isomorphic paths $p_i$ in $P$ :
      {
         include the path $p_i$ to $DD$ ;
         add the addressing node $a_i$ at the end of the path $p_i$ ;
         create the set $PR$ of path remainders for the path $p_i$ ;
         integrate isomorphic paths ($PR$, $DD_r$) ; ①
         for all state/output variables $v_i$ :
         {
            create an edge for $a_i$ labeled with the variable $v_i$ ;
            if in the phase of integration of isomorphic paths ① the path remainder
               corresponding to the variable $v_i$ has been merged with the remainder of the
               variable $v_j$ :
            {
               merge the edge labeled with $v_i$ with the edge $v_j$ ;
               label the edge with the label $v_i$, $v_j$ ;
               connect $DD_r$ to the new edge ;
            }
            else
            {
               connect the path remainder corresponding to the variable $v_i$ to the edge
                  labeled with $v_i$ ;
            }
         }
         eliminate the redundant isomorphic subdiagrams ;
         eliminate the redundant nodes ;
      }
   }
}
```

4.4.5 Compressed Vector Decision Diagrams

The decision diagram models presented above offer a suitable and efficient mechanism to implement the algorithmic description of the system. However, all of them require to introduce the next state function *q* to realize the division of the algorithm into states. This necessity implies that the data-path diagram should be traversed several times (according to different values of the variable *q*) in order to complete the execution of the algorithm. As a consequence, the execution performance significantly decreases.

The solution to that problem relies on the construction of the decision diagram in such a way, that it will implement separately the traversal of each path of the initial algorithm. The paths in the algorithm reflect distinct values of the control nodes (different results of the condition evaluation). All the conditions along the path are expressed in terms of initial values of the input variables. Moreover, the condition nodes of the entire path are grouped together and executed first, before the data manipulations. The data processing nodes, which are expressed as assignments to the output variables, are also clustered and performed after the condition checking.

A decision diagram constructed in that way is called *compressed vector decision diagram* (CVDD). It represents the clustering of control and data nodes in the algorithm. The compressed VDD takes all the advantages of the VDD: the compact representation of vector variables and the use of addressing nodes.

For our example the resulting compressed vector decision diagram is presented in figure 4.18.

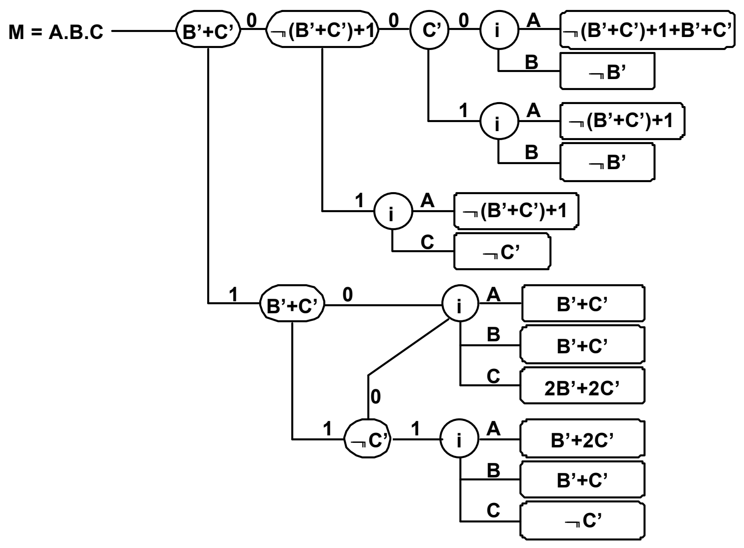


Figure 4.18: *Compressed VDD (CVDD)*

Since, during the condition nodes evaluation in the control part of the path some operations on the input variables are performed, the results of them can be potentially used in the computation of the output values of terminal nodes. In order to make this possible, additional intermediate variables are introduced to store temporarily the partial results that were already calculated. Then, during the evaluation of the terminal nodes, the partial results are reused in the computation of the final values of the output signals.

The application of this technique to the considered example is shown in figure 4.19. Three temporal variables: *x*, *y* and *z* have been introduced and their values are computed in the condition nodes during each path evaluation. These variables are then reused several times in the subsequent condition nodes or terminal nodes as partial results, thus reducing the complexity of the computation.

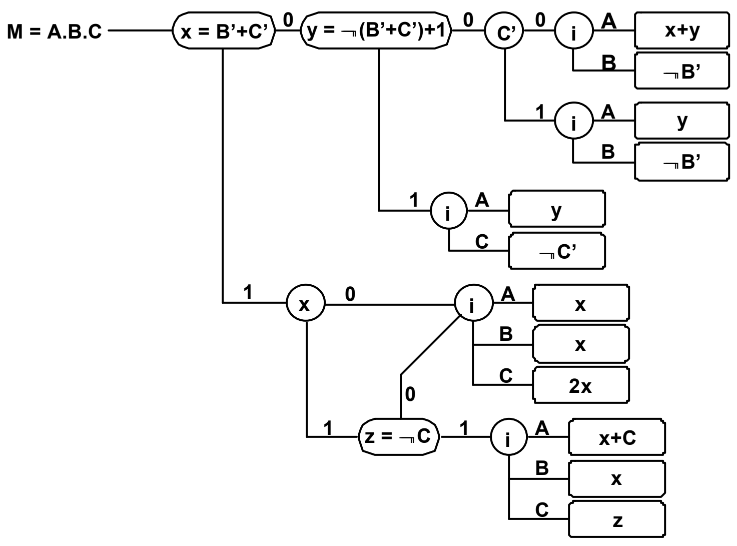


Figure 4.19: *CVDD with intermediate variables*

The construction of CVDD is based on the concept of execution path. An execution path is defined as a sequence of operations in the graph of the initial algorithm, which is executed under certain input conditions (i.e. for specific values of input variables). Paths represent distinct and mutually exclusive executions of the algorithm, which are mapped as separate subdiagrams in the CVDD model.

A sequence of operations beginning at a given starting operation and ending at a terminating operation constitutes an execution path.

The starting operation in a path can be:

- an entry operation in the algorithm graph
- a decision node in the algorithm graph

The terminating operation can be one of the following:

- an operation with no successors
- an operation succeeded by a feedback loop
- an operation succeeded only by a decision node

A decision node in the algorithm graph is considered as a starting node only if, in the paths terminating before this node, the operands of the decision expression can change their values (there exist an assignment operation having on the left hand side at least one of the operands), and these values cannot be computed statically.

In the case of loops in the algorithm represented by a CVDD we still need to introduce the state variables to break the loops and represent in the CVDD the state transition functions as well.

4.4.6 Experimental data

In order to enable the comparison of the initial algorithm with the decision diagrams, as well as different decision diagram implementations between them, a symbolic complexity measure has to be introduced. The complexity measure for the purpose of that application is based on the counting of the elementary operations, which are used in the representation (algorithm or DD) to complete the computation. The elementary operations are: read variable value, assign variable, arithmetic operation, logic operation and condition checking. For the sake of simplicity, the weights of all operations are assumed to be equal, although for the precise measure purposes the weight of various types of operations should be differentiated according to the real computational cost.

Table 4.7 shows the complexity measure for the example presented in this section. The overall computational complexity of the algorithmic description is equal to the CVDD complexity (in the example: 99). The execution complexity of the DD is more than 2.2 times higher than the one of CVDD in this example.

Model	Paths						Total
	P1	P2	P3	P4	P5	P6	P1-P6
Algorithm	21	14	13	14	18	19	**99**
DD	39	33	32	32	42	43	**221**
VDD	30	24	23	25	32	33	**167**
VDD with next state	26	20	19	20	26	27	**138**
Compressed VDD	16	16	15	14	18	20	**99**

Table 4.7: *Symbolic complexity measure of different representation types for the algorithm example presented in this section*

Table 4.8 presents a comparison of the symbolic computational complexity for some examples of algorithms. *Ex1* is the example presented in this section, *Ex2* is an algorithm of computation of a mean value, *Ex3* is a greatest common divisor, *Ex4* is an algorithm of computation of a probability measure in a sorting algorithm, *Ex5* is a numerical calculus of the function *sin x* by the Taylor interpolation and *Ex6* is a fast sorting algorithm.

The compressed VDD representation is, in general, more than two times more efficient than the corresponding DD model (the highest ratio is equal to 3.3 times for *Ex5*). The VDD with the next state function offers in some cases similar performances to CVDD. This is the case of algorithms for which no additional optimizations and reductions are possible. The computational complexity of the algorithmic description is comparable to the one of the compressed VDD. In the situations where several operations along the path of the graph can be combined together, the compressed VDD offers even a better efficiency than the algorithmic representation (e.g. *Ex5*).

Model	Examples					
	Ex1	Ex2	Ex3	Ex4	Ex5	Ex6
Number of paths	*6*	*3*	*4*	*5*	*7*	*7*
Algorithm	99	37	31	57	83	70
DD	221	72	95	139	240	192
VDD	167	56	50	78	124	100
VDD with next state	138	46	37	61	96	86
Compressed VDD	99	33	37	61	73	82

Table 4.8: *Comparison of the proposed DDs representations according to the symbolic complexity measure*

In table 4.9 the results of the simulation of the proposed representations are presented. These representations have been implemented as an executable C code, which has been simulated with the same set of test-bench vectors for all types of representation. The row *Time* shows the global execution time expressed in seconds; the row *Ratio* contains the ratio between the execution time of a given DD or algorithmic representation and the simulation time of a conventional DD, the row *Gain* presents an inverse of the value in *Ratio*, which signifies the simulation speed-up. The results of simulation presented in table 4.9 are well correlated with the symbolic computational complexity estimation given in table 4.8.

The results obtained in these experiments show that, for the linear algorithms which do not contain loops (*Ex1* and *Ex2*), the algorithmic representation and the compressed VDD representation are equal from the point of view of the execution time. However, these two representations allow to achieve 1.4 to 2.1 times better performance in simulation than the corresponding VDD representation. The gain in simulation time of the CVDD representation in comparison to the conventional DD is in between 2.2 and 3.9 times.

The algorithms containing loops (*Ex3-6*) are executed around 50% faster in the case of algorithm representation than if they are modeled as CVDDs. However, for this type of algorithms, the CVDD representation is 3.1 to 4.6 times faster than the initial DD model.

Simulation time comparison						
Example		**DD representation**				
		DD	**VDD**	**VDD with next-state**	**CVDD**	**Algorithm**
Ex1	Time [s]	494.0	343.7	271.9	129.2	127.2
	Ratio	1.000	0.696	0.550	0.261	0.258
	Gain	**1.000**	**1.437**	**1.817**	**3.824**	**3.884**
Ex2	Time [s]	234.6	191.1	147.0	104.5	106.7
	Ratio	1.000	0.815	0.627	0.446	0.455
	Gain	**1.000**	**1.228**	**1.596**	**2.245**	**2.199**
Ex3	Time [s]	1998.6	1173.0	652.5	652.5	439.5
	Ratio	1.000	0.587	0.326	0.326	0.220
	Gain	**1.000**	**1.704**	**3.063**	**3.063**	**4.547**
Ex4	Time [s]	206.9	98.9	49.5	49.5	31.5
	Ratio	1.000	0.478	0.239	0.239	0.152
	Gain	**1.000**	**2.092**	**4.180**	**4.180**	**6.568**
Ex5	Time [s]	868.7	489.6	232.5	190.2	121.5
	Ratio	1.000	0.564	0.268	0.219	0.140
	Gain	**1.000**	**1.774**	**3.736**	**4.567**	**7.150**
Ex6	Time [s]	1547.4	923.8	571.9	393.3	258.8
	Ratio	1.000	0.597	0.370	0.254	0.167
	Gain	**1.000**	**1.675**	**2.706**	**3.934**	**5.979**

Table 4.9: *Comparison of the DD representations according to the simulation execution time*

4.4.7 Concluding remarks

The main objective of this section was to develop the forms of representation based on decision diagrams which, applied in order to model the digital system at algorithmic level of abstraction, allow to achieve better performances in simulation. Various optimization factors have been introduced into the initial decision-diagram representation to obtain a compact and efficient (from the computational complexity point of view) representation of the system.

The first of these optimizations applied to the initial form of diagrams was the introduction of vector variable representation, which enables to merge several decision diagrams of the system into a single diagram. The process of simulation, initially consisting in simulation of a number of decision diagrams in the model, is reduced to simulation of one diagram representing at once several output variables in the model. As the results of practical experiments show, this representation referred to as a vector decision diagram (VDD), simulates 1.2 to 2.1 times faster than the initial DD-based model.

Further optimization of the VDD representation, by integrating the diagram corresponding to the next-state function with the diagram representing the vector variable, resulted in the improvement of the simulation performance from 1.6 to 4.2 times comparing to DD representation (this representation is designated as *VDD with next-state function*).

The next form of the DD representation, the compressed vector decision diagram (CVDD), is obtained by representing each path, which can be traced in the initial algorithm as a separate part of the diagram while clustering the control and data nodes on the path. In this representation terms denoting functions are used as node labels. This allows to optimize the set of operations performed during the simulation of the entire path in a diagram, for given values of input variables. The CVDD representation can be simulated significantly faster than the equivalent DD-model, with the performance gain between 2.2 and 4.6 times.

The compressed VDDs exhibit performances that are almost similar to the performance of the algorithmic description execution. It has been demonstrated that, in some cases, depending on particular optimizations and reductions which can be introduced in the decision nodes and terminal nodes, they even allow to achieve better performances. There is a possible room for additional improvements of the compressed VDD construction method, by applying more advanced optimization techniques to the operations to be performed at terminal nodes, and in the scheduling of decision nodes in the diagram.

4.5 Conclusions

A new conceptual approach based on high-level decision diagrams was developed for the purpose of the simulation of digital systems. This approach allows to use a uniform diagram-based description of VLSI designs on different levels of abstraction starting from the algorithmic level, throughout register-transfer level, up to the gate level, together with uniform simulation procedures based on path tracing on DDs. DDs enable to describe, using the same mathematical concept, a wide class of digital systems on mixed logical and functional levels. This class contains random logic, traditionally treated at the gate level, as well as complex digital circuits like microprocessors, controllers etc., traditionally described at the procedural or RTL levels.

Moreover, as shown in the first section of this chapter, the decision-diagram representation presents a sound mathematical concept, which opens an excellent possibility to apply it as a formal basis for other purposes in the design flow: the fault analysis, testing, automatic test pattern generation or design error diagnosis of digital systems. It allows to create more efficient CAD tools than existing event-driven HDL-based simulators for functional simulation, as well as tools for fault analysis and testing purposes. It is advantageous that several actions in the design flow can be accomplished on the same design model.

The initial comparison of the simulation execution times of the DD-based representation with the conventional event-driven simulation, which has been established for a typical set of design examples, demonstrates already an important improvement in simulation performance, ranging between 2 and 9 times for faster simulator used in the experiments and between 4.6 and 37 for slower one. However, the initial DD-representation used in the experiments still presents a potential to further optimization.

This is why two specific forms of DD representation, the vector decision diagrams (VDDs) and the compressed VDDs (CVDDs), have been developed to further optimize the DD-based representation of the system function at the algorithmic level of abstraction. The diverse improvements, which were introduced in decision-diagram representation, resulted in the achievement of a considerably higher simulation performance of these new representations compared to the initial DD-based model. A gain from 2 to almost 5 times has been recorded in simulation of VDDs or CVDDs models in place of the initial DD representation.

Chapitre 5

Algorithmes de simulation sur les graphes de décision de haut niveau

5.1 Résumé

Comme nous l’avons démontré dans le chapitre précédent, seule l'application des graphes de décision de haut niveau à la modélisation des systèmes électroniques permet d'obtenir une augmentation considérable de la performance de simulation. En revanche, nous n'avons pas encore considéré le problème de la performance d'exécution d'un simulateur de graphes de décision.

C’est pourquoi, dans ce chapitre, tout en considérant que le système complet est représenté par un réseau de graphes de décision, nous décrivons quatre algorithmes différents de la simulation de ce réseau.

Le premier algorithme, dit « *évaluer tout* », est fondé sur le principe de base selon lequel dans chaque cycle de simulation tous les graphes du réseau sont évalués. La simulation utilisant cet algorithme n'atteint évidement pas la meilleure performance possible. C'est pour cette raison que nous proposons l’application de deux techniques de gestion de simulation permettant d’obtenir une performance bien supérieure. Une de ces techniques est basée sur l’observation des événements aux entrées des graphes, tandis que l’autre effectue l’évaluation d’un réseau de graphes en commençant par les sorties du modèle. Ces deux techniques, appliquées soit séparément, soit l’une avec l’autre, permettent de développer les algorithmes de simulation décrits par la suite.

La première de ces techniques, qui a été mise en œuvre dans un algorithme dénommé *l'algorithme dirigé par les événements avec l'évaluation en avant* (en anglais: *event-driven forward-tracing algorithm*), permet d'évaluer le réseau de graphes selon les activités des variables d'entrée de chaque graphe. Il en résulte que, dans le processus de simulation, seulement les graphes pour lesquels les entrées changent sont évalués. Quant aux autres, puisque ces entrées restent inchangées, ils fournissent les mêmes valeurs aux sorties et, par conséquent, leur évaluation est inutile.

La deuxième technique est réalisée dans un algorithme nommé *l'algorithme d'évaluation rétrogradée* (en anglais: *back-tracing algorithm*). Dans chaque cycle de simulation, cette technique commence l'évaluation d’un réseau de graphes par les variables des sorties primaires, et exécute le calcul des graphes pour lesquels les valeurs de sorties sont propagées jusqu'aux sorties primaires du modèle seulement.

Pour modéliser les systèmes au niveau transfert de registres, une nouvelle forme de graphes de décision, appelée en anglais *register-oriented decision diagrams* (RODDs), a été introduite dans le cadre de ce travail. Dans une modélisation d’un système par les graphes RODDs pour chaque registre du système (et seulement pour le registre lui-même) un graphe de décision est créé. La partie asynchrone du circuit connectée aux entrées d’un registre est représentée par les nœuds internes du graphe constitué pour ce registre.

La représentation du système sous forme de graphes RODDs, associée aux deux techniques de gestion de la simulation décrites ci-dessus, permet de développer un nouvel algorithme de simulation, dénommé *l'algorithme dirigé par les événements avec l'évaluation rétrogradée* (en anglais : *event-driven back-tracing algorithm*). Cet algorithme exécute l'évaluation des graphes d'un modèle exclusivement dans le cas où un événement se produit dans la partie contrôle du modèle.

Au fur et à mesure de la présentation de chaque algorithme, nous discutons ses avantages et ses inconvénients par rapport aux autres algorithmes utilisés dans la simulation.

A la fin du chapitre, nous donnons quelques résultats de la simulation des exemples des modèles sur les simulateurs de graphes de décision dans lesquels les algorithmes décrits ont été mis en œuvre. Nous comparons également les méthodes de simulation développées par nous-mêmes avec deux types de simulateurs commerciaux : avec un simulateur dirigé par les événements, ainsi qu’avec un simulateur dirigé par l'horloge.

5.2 Introduction

It has been shown in the previous chapter that the application of the high-level decision diagrams for the system representation and the cycle-based simulation of the DD-model offers an important gain in the simulation speed in comparison to the event-driven HDL-based simulation approach.

The objective of this chapter is to present novel simulation algorithms for efficient functional cycle-based simulation of the DD network. Two techniques have been considered to improve the performance of the simulation execution: the event-driven and back-tracing techniques. The former implemented as an *event-driven forward-tracing algorithm (ED-FT)*, performs the evaluation of the network of DDs according to the activities on the input variables of the diagrams. The latter, implemented by the *back-tracing algorithm (BT)* starts the evaluation of the diagrams in the network from the output variables and executes the computation of only those DDs, for which the output variables propagate to the primary outputs of the model. The application of the event-driven approach to the back-tracing technique resulted in algorithm referred to as the *event-driven back-tracing (ED-BT)*, which executes the DDs in the data path according to the events produced by the control part of the model.

In addition, a new class of the decision-diagram representation for the sequential systems called *register-oriented decision diagrams* (RODD) is introduced and successfully applied in combination with the event-driven back-tracing algorithm to further optimize the simulation execution.

Chapter 5 provides the definition of the decision diagrams and register-oriented decision diagrams for the system representation (section 5.3), the description of the proposed simulation algorithms (section 5.4) and presents the experimental results of the simulation execution performed on real design examples, as well as a comparison of the DD-based cycle simulation techniques and the event-driven as well as the cycle-based hardware description language simulation (section 5.5).

5.3 Decision diagrams for design representation [15]

5.3.1 High-level decision diagram model

High-level decision diagrams have been introduced in the previous chapter in section 4.2 (definitions 4.1 to 4.12) as a representation of a digital system given at various levels of abstraction. Further in this chapter we will denote these diagrams as decision diagrams with the abbreviation: DDs.

[15] Some parts of this chapter have been initially presented in [110, 157, 158]

5.3.2 Register-oriented decision diagrams

A new class of decision diagrams (DDs) referred to as register-oriented DDs (RODDs) is introduced for describing the systems presented at the register-transfer level. A RODD model is a special case of DD models where, for each register, only one decision diagram is created in the model and the number of diagrams in the network is equal to the number of registers in the circuit. Non-terminal nodes of a RODD are labeled by control signals (e.g. register enable signals, multiplexer addresses, reset) (c_1 to c_4 in the example below, figure 5.3), while the terminal nodes are labeled by registers, primary inputs, constants or operations on them (X_1, X_2, R_1, R_2, R_3 in the example). Thus, for each register in a RODD model there exists a corresponding decision diagram, which computes its value as a function of control signals, registers and primary inputs.

An example of a data-path fragment of a digital circuit is presented in figure 5.1.

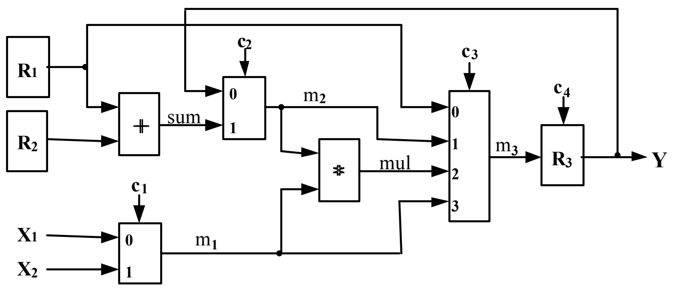


Figure 5.1: *Data path of a digital system*

Figure 5.2 shows the corresponding component level DD representation for that circuit. Every component in the circuit is separately modeled by a single decision diagram. The internal variables are used to interconnect the diagrams composing the complete model. In this model we have: $X = X_1, X_2, R_1, R_2$; $Y = R_3$ and $Z = m_1, m_2, m_3, sum, mul$.

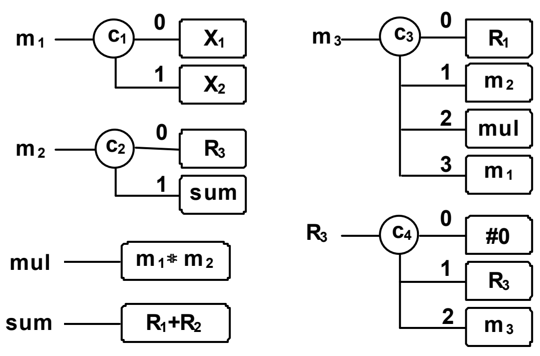


Figure 5.2: *Component level DD model*

Register-oriented DD model is created by a superposition of component-level decision diagrams. The general superposition procedure has been presented in [78]. In the case of RODD models the superposition starts from registers and is traced back to subsequent registers, constants or primary inputs of the model. Figure 5.3 presents the RODD for the data-path output register R_3 of the circuit shown in figure 5.1.

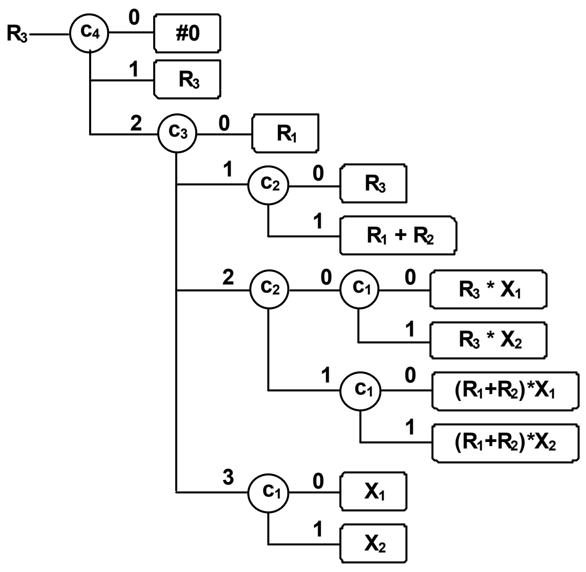


Figure 5.3: Register-Oriented DD model

5.3.3 System representation

The entire system is represented as a levelized network of decision diagrams interconnected by variables. We denote by $X = X_1, X_2, ..., X_n$ the set of all primary inputs of the entire system and by $Y = Y_1, Y_2, ..., Y_m$ the set of primary outputs of the system. Additionally the set $Z = Z_1, Z_2, ..., Z_p$ represents the system internal variables.

The naming convention for the decision diagram to be further used in this chapter is presented in figure 5.4.

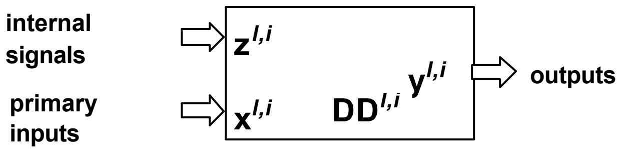


Figure 5.4: Decision diagram naming convention

The input vector variable of the decision diagram DD is divided into two sections: $x \in X$ which represents the system primary inputs (a subset of the primary inputs of the entire system that are used in DD) and $z \in Z$, $z \notin X$ which represents the internal signals as inputs to the diagram (the inputs coming from previous levels or from the feedback). The output variables y of DDs can be primary outputs of the system (in that case $y \in Y$), feedback signals ($y \in Z$) or internal signals for the forthcoming levels (again $y \in Z$).

In order to label the decision diagrams in the network two types of indexes are introduced. The first upper index l indicates the level on which the decision diagram is placed in the network of the system; $l \in (0, \dots, L)$, where L is the total number of levels in the network. The second upper index i is an index of the diagram at the level; $i \in (0, \dots, I^l)$, where I^l is a number of DDs at the level l. The representation of a system as an interconnected network of DDs is shown in figure 5.5.

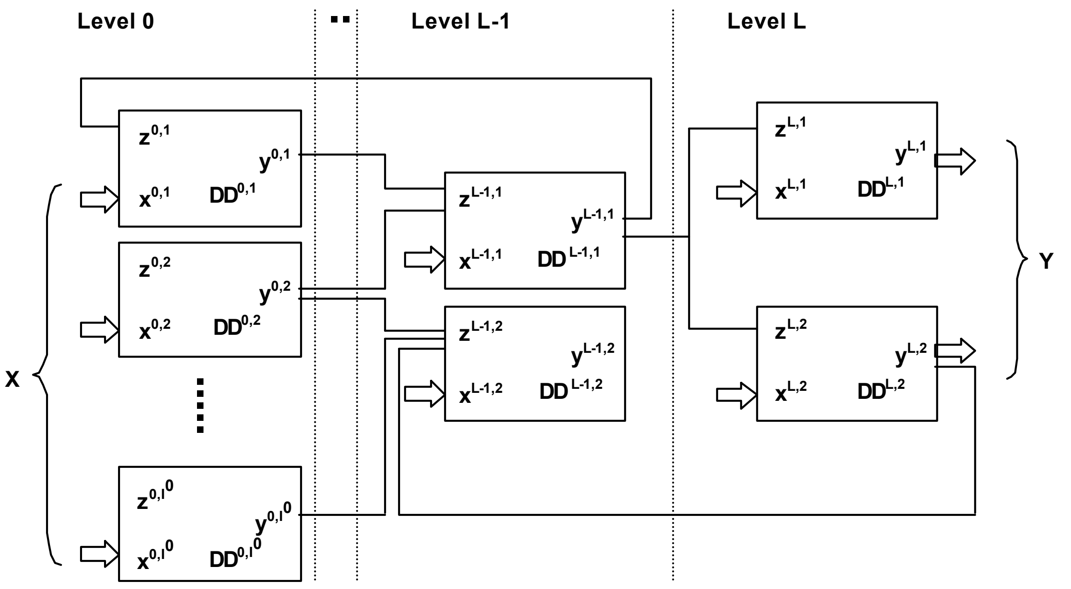

Level 0
Level L-1
Level L
X
Y
$z^{0,1}$ $y^{0,1}$ $x^{0,1}$ $DD^{0,1}$
$z^{0,2}$ $y^{0,2}$ $x^{0,2}$ $DD^{0,2}$
z^{0,I^0} y^{0,I^0} x^{0,I^0} DD^{0,I^0}
$z^{L-1,1}$ $y^{L-1,1}$ $x^{L-1,1}$ $DD^{L-1,1}$
$z^{L-1,2}$ $y^{L-1,2}$ $x^{L-1,2}$ $DD^{L-1,2}$
$z^{L,1}$ $y^{L,1}$ $x^{L,1}$ $DD^{L,1}$
$z^{L,2}$ $y^{L,2}$ $x^{L,2}$ $DD^{L,2}$

Figure 5.5: *System representation with the use of DDs*

For the sake of simplicity, in the above definitions as well as in the following descriptions of simulation algorithms, the timing issues are not considered. It is important to note that the values of the variables belonging to the input vector of a DD that correspond to registers are those values evaluated in the preceding clock cycle. During the forward event-driven simulation, evaluation of a DD corresponding to a register can possibly create events at the following clock cycle. Similarly, if during the back-tracing simulation, a node is traversed whose variable corresponds to a register, a recursion for calculating the value for this variable will be entered at the preceding clock cycle.

5.4 Simulation algorithms

Several approaches can be adapted to manage the evaluation of the decision diagrams in a cycle-based simulation. The order of evaluation of the diagrams, the observation of the events at their inputs or the management of the memorizing elements in the data path have a critical influence on the performance of the simulation execution. The following section introduces four different algorithms elaborated for efficient RODD-model simulation. The event-driven paradigm is adapted in two of them: in the forward-tracing algorithm (ED-FT) and back-tracing algorithm (ED-BT), in order to minimize the number of evaluated DDs in the simulation cycle according to the activity of the DD input variables. Another approach focusing on the minimization of the number of the DD evaluation is applied in the back-tracing algorithm (BT), in which the evaluation of a DD is performed according to the values which are propagated to the outputs of the DD-model.

The advantages and drawbacks of those algorithms are discussed and compared in this section.

5.4.1 "Compute-all" simulation algorithm (CA)

The first approach to evaluate the network of interconnected decision diagrams consists in the evaluation of all diagrams composing the network. The evaluation starts from the primary inputs X at level 0 and propagates to level L producing the primary outputs of the model.

The corresponding symbolic algorithm is presented below:

Algorithm 5.1: *"Compute All" simulation algorithm (CA)*

```
if (new vector X) then
    for (level = 0 to level = L)
        for (index = 0 to index = $I^{level}$)
            Evaluate $DD^{level, index}$;
        end for;
    end for;
end if;
```

The algorithm for evaluating the DD_y that computes the value of variable y is the following:

Evaluate DD_y:

```
m = $m^0$;
while ($\tilde{A}(m) \neq \emptyset$)
    if (z(m) = 'X') then
```

```
            y = 'X';
            exit;
        end if;
        m = Ã(m);
    end while;
    y = z(m);
```

Although the above algorithm seems to be simple and straightforward to implement (no additional computational effort is necessary to run more sophisticated simulation management, as it is the case of other simulation algorithms presented below) and to execute for the simulation of the DD model, it does not offer a satisfactory performance. This is the consequence of the fact that the CA simulation necessitates the evaluation of all the diagrams in the network at each simulation cycle even though for some DDs all the inputs, and in consequence the outputs, do not change in the current cycle. Thus, for these DDs the produced output variable values of the decision diagram remain unchanged. This represents globally an important computational overhead associated to the (useless) evaluation of diagrams, which in case of large systems, modeled by a complex network of diagrams and a localized activity in the network, can be significant in comparison to the useful evaluations actually necessary to compute new internal or output variable values.

5.4.2 Event-driven forward-tracing simulation algorithm (ED-FT)

In order to minimize the number of evaluated decision diagrams in the simulation cycle, by avoiding unnecessary evaluation of those DDs in the network which do not provide any new values at their outputs, an event-driven paradigm is adapted for the simulation execution.

In this approach, during each simulation cycle the activity of the input variables of every diagram in the network is observed. The evaluation starts from the primary inputs and finishes at the level providing the primary output values. For every decision diagram at a given level the activity of its inputs is checked. If an event (it means a change of a value) of at least one of the input variables occurrs, the diagram is evaluated. Otherwise, the previous value calculated in the preceding simulation cycle is taken as the actual value for the output variable(s).

The algorithm performs in the following way: for each new input vector X the DDs for which the inputs X or Z changed, are evaluated, starting from level 0 and propagating to level L, what can be written symbolically:

Algorithm 5.2: *Event-driven forward-tracing simulation algorithm (ED-FT)*

```
if (changes in X) then
    for (level = 0 to level = L)
        for (index = 0 to index = $I^{level}$)
            if ($X^{level, index}$ or $Z^{level, index}$ changed) then
                evaluate $DD^{level,index}$;
            end if;
        end for;
    end for;
end if;
```

The above algorithm presents a better performance in terms of simulation execution time in comparison to the "compute-all" algorithm, since it saves the computation of those DDs, for which the inputs do not change. However, the overall performance is decreased by the additional computational work devoted to keeping track of the input vector events. The conceptual drawback of that algorithm is that it recalculates the new values of DDs even if their outputs do not propagate to the primary outputs of the entire system.

5.4.3 Back-tracing simulation algorithm (BT)

Another approach focused on the minimization of the number of the DD evaluations is applied in the back-tracing algorithm, in which the evaluation of DD is performed according to the propagation of variable values to the outputs of the DD-model.

The back-tracing simulation algorithm evaluates the decision diagrams of the network starting from the output variables. In other words, the decision diagrams belonging to the terminal level *L* of the network of DDs are considered first. Initially, all the internal variables are set to *don't care* values. During the simulation of a decision diagram, if a node with a variable holding the *don't care* value is traversed, the variable value is computed in a recursive way by evaluating the respective decision diagram(s).

If several diagrams evaluated in the same simulation cycle have nodes labeled with the variable produced by a unique decision diagram, only one computation of that diagram is performed to avoid the re-evaluation of the same DD.

Algorithm 5.3: *Back-tracing simulation algorithm (BT)*

```
set all variables to 'X';
if (new vector X) then
    level = L;
    for (index = 0 to index = $I^L$)
        Recursively Evaluate $DD^{L, index}$;
    end for;
end if;
```

The algorithm for recursive evaluation of the DD_y is the following:

Recursively Evaluate DD_y:

```
m = $m^0$;
while ($\tilde{A}(m) \neq \emptyset$)
    if (x(m) = 'X') then
        recursively evaluate $DD_x$;
        if (x(m) = 'X') then
            y = 'X';
            exit;
        end if;
    end if;
    m = $\tilde{A}(m)$;
end while;
y = x(m);
```

The disadvantage of the approach presented in this section lies in the fact that the recursive evaluation of DDs is executed even if none of the inputs of the successive DDs changed its value.

5.4.4 Event-driven back-tracing simulation algorithm (ED-BT)

The event-driven back-tracing simulation algorithm starts with the evaluation of the network of RODDs from the diagram of the control part. The evaluation of the control part allows determining the activity of the variables, which activate the remaining parts of the network. According to the events recorded at the control part outputs, in the next step the register diagrams are evaluated in the back-tracing way. First the registers providing the primary outputs are considered. They are evaluated only if an event occurred at the enable input of those registers and the registers become activated. If no activating event was recorded, the output value(s) of the register is the value computed in the preceding simulation cycle. The back-tracing technique is applied to the evaluation of the remaining diagrams in the network to arrive to the primary inputs, register variables or constant values.

The main improvement of the event-driven back-tracing algorithm proposed here is that it further minimizes the number of evaluated diagrams per simulation cycle according to the activities produced by the control part of the model. In this way, even if an event occurred at the inputs of a DD corresponding to a register, no evaluation of that diagram is performed (as well as no evaluations of diagrams at the input of the registers) unless the register is enabled by the control part. This is an important improvement over the previous algorithms, which allows to dramatically reduce the simulation time.

The *compute-all* and *event-driven forward-tracing* algorithms compute or provide the values for all variables of the DD model starting from the initial level and moving forward to the terminal level (producing the primary output values) of a network at every clock cycle. In contrast, both versions of the back-tracing algorithms at some clock cycles do not provide the values of the internal variables in the case, when they do not influence any primary outputs. This means that a simulation performance increase is associated with the reduction of the observability of the internal signal behavior of the model.

5.5 Experimental results

Experiments have been carried out to compare the simulation algorithms proposed in previous sections. For the experiments the following benchmark circuits have been used: *gcd* is a greatest common divisor circuit [52], *mult8x8* is a 8-bit multiplier [43], *diffeq* is a circuit implementing the differential equation calculation method [52], *huff_enc* is a Huffman encoder [33], *circ1* is a control dominated circuit and *dat* is a data path dominated circuit. Table 5.1 presents the main characteristics and the complexity of the benchmark circuits together with the details about the DD implementation.

The most relevant way of comparing the DD simulation algorithms between them is to compare the number of evaluated decision diagrams for the same set of simulation vectors. In table 5.2, such comparison of the four considered algorithms is given. The total number of decision diagram simulations for each of the algorithms as well as the percentage of the diagrams evaluated is shown. Minimal results are denoted by bold numbers. During the experiments, real test stimuli generated by test generator DECIDER [126] were used in order to activate all possible states of the design behavior (in contrast to random simulation vectors, which do not allow to simulate realistic design behavior).

As table 5.2 shows, in all of the cases the event-driven back-tracing simulation algorithm necessitates to perform the lowest number of diagram evaluations. The difference in terms of evaluated diagrams between the conventional simulation algorithm (referred to as "compute-all") and the event-driven cycle-based simulation algorithm ranges between 18%

and 39% for the tested examples. For more complex models having several levels in the network (e.g. *huff_enc*) this difference becomes more important due to the larger number of DDs which were not evaluated.

The comparison between the simulation algorithms in terms of the run time is presented in table 5.3. *Ratio* indicates the comparison of the simulation time of a given algorithm to the event-driven back-tracing algorithm. The increase up to 60% of the simulation performance is observed in comparison to the initial CA algorithm.

In addition, table 5.4 presents the run-time results of the simulation of the benchmark DD-models performed with the use of a DD-based event-driven cycle simulator and the corresponding VHDL models simulation run on commercial HDL simulators: on an event-driven simulator (VSS, Synopsys) as well as on a cycle-based simulator (Cyclone, Synopsys). The experiment was run on a 366MHz SUN UltraSPARC 60 workstation with 512MB RAM under Solaris 2.5.1 operating system. In order to achieve a better timing resolution all the test sets were multiplied by ten. For optimal performance, Synopsys tools *cylab* and *cysim* were run with *-perf* and *-2state* options.

The increase of the simulation performance obtained by the use of the DD-based simulator in comparison to the event-driven simulator ranges between 1.8 and 32 times, while the comparison to the HDL-driven cycle-based simulation shows the gain from 1.6 to 6.7 times.

Circuit	**RTL description**					**DD description**		
	Inputs	**Outputs**	**Registers**	**FU**	**MUX**	**Graphs**	**Nodes**	**Variables**
gcd	3	1	3	3	2	4	30	13
mult8x8	3	1	7	9	4	8	52	25
diffeq	6	3	7	5	9	8	65	31
huff_enc	5	5	13	22	17	14	202	54
circ1	11	10	20	0	30	31	237	53
dat	17	4	4	16	12	17	91	68

Table 5.1: *Characteristics of circuit examples and the corresponding decision diagrams*

Circuit	**Test length [vectors]**	**Number of diagram evaluations Register-Oriented DD model**							
		CA		**ED-FT**		**BT**		**ED-BT**	
gcd	2949	11792	100%	8109	69%	8714	74%	**4422**	**37%**
mult8x8	2814	22504	100%	10894	48%	15349	68%	**6385**	**28%**
diffeq	2081	16640	100%	10912	66%	14169	85%	**6468**	**39%**
huff_enc	42001	588000	100%	186020	32%	451990	77%	**106000**	**18%**
circ1	41520	1287089	100%	419816	33%	634778	49%	**363987**	**28%**
dat	10000	1699983	100%	1507692	87%	N/A	N/A	**462137**	**27%**

Table 5.2: *Comparison of DD simulation algorithms*

Circuit	Simulation time (10X test length)							
	CA		ED-FT		BT		ED-BT	
	Time [ms]	Ratio	Time [ms]	Ratio	Time [ms]	Ratio	Time [ms]	Ratio
gcd	150	1.36	190	1.73	180	1.64	**110**	**1.00**
mult8x8	290	1.32	320	1.45	350	1.59	**220**	**1.00**
diffeq	260	1.24	330	1.57	330	1.57	**210**	**1.00**
huff_enc	330	1.57	310	1.48	1100	5.24	**210**	**1.00**
circ1	35641	1.02	35120	1.01	35111	1.01	**34790**	**1.00**
dat	406515	1.02	400816	1.01	N/A	N/A	**398603**	**1.00**

Table 5.3: *Experimental results for DD-based simulation algorithms*

Circuit	Simulation time (10X test length)				
	ED-BT	HDL event-driven		HDL cycle-based	
	Time [ms]	Time [ms]	Ratio HDL ED / / ED-BT	Time [ms]	Ratio HDL cycle / / ED-BT
gcd	110	1700	15.45	510	4.64
mult8x8	220	2800	12.73	1000	4.55
diffeq	210	6740	32.10	1260	6.00
huff_enc	210	3860	18.38	1400	6.67
circ1	34790	242350	6.97	108545	3.12
dat	398603	710740	1.78	649723	1.63

Table 5.4: *Comparison between DD and HDL-driven cycle-based simulation algorithms*

5.6 Conclusions

The objective of the work presented in this chapter was to improve the cycle-based simulation performance of the high-level decision-diagram-based system representation. This aim has been achieved by introducing separately and in combination two simulation techniques: the event-driven and back-tracing techniques. Consequently, four different algorithms have been developed to simulate the network of decision diagrams. The first one, referred to as “compute all” algorithm, performs the evaluation of all DDs in the network of a model. To improve the execution performance of this algorithm, the event-driven technique has been introduced, resulting in the development of the event-driven forward-tracing algorithm, which executes the evaluation of diagrams in the model as a function of events occurring on their inputs. This technique permits to save the evaluation of those diagrams in the model, which do not provide new values to the outputs in a given simulation cycle. Another simulation technique has been applied in the development of two other algorithms. This technique, known as back-tracing technique, starts the evaluation of the DD network from the output diagrams, and performs the evaluation only of those diagrams, for which the output values propagate to the outputs of the entire model in the current simulation cycle. The combination of that technique with the event-driven technique provide the most efficient algorithm for the simulation of the DD representation called the event-driven back-tracing algorithm.

The simulation algorithms implementing the above techniques, in addition to the efficient design function representation by the register-oriented DDs (RODDs), represent a significant improvement of the simulation performance in comparison to the conventional simulation technique (i.e. "compute-all" technique). The main contribution to the improvement of the simulation speed comes from the use of a specific type of decision diagrams – RODDs, which allows to efficiently extract only these parts of the system that are active in each simulation cycle.

The efficiency of DD-based simulation can be increased by the superposition of DDs in the RODD model. However, as the number of variables will be reduced by the superposition, the amount of the data produced in the simulation will also be reduced, thus the observability of the internal variables of the system will decrease. Hence, the superposition of DDs is a trade-off problem between the efficiency of simulation and the observability of the internal behavior.

The event-driven forward-tracing simulation algorithm provides an enhancement from 32% to 87% (depending on the model), in terms of a number of DDs evaluated in the simulation cycle, in comparison to the conventional simulation by evaluation of all diagrams in the network. However, due to the computational overhead of keeping track of the events occurring at the DD inputs, the simulation run times are similar to those of the "compute-all" approach. The other algorithm – the event-driven back-tracing algorithm - offers much higher gain in comparison to the "compute-all" approach, which represents an improvement of approximately 25 to 57% in simulation run times.

The DD-based event-driven cycle simulation engine offers a significantly better performance than the cycle-based hardware description language (HDL) simulator. The improvement in simulation time of the DD-based simulator over the HDL-based cycle simulator ranges between 1.63 and 6.67 times for the examples tested in experiments. The difference in simulation performance of the DD-based simulator and the event-driven HDL-based simulator is much higher, and for the given examples was between 1.8 and 32.1 times.

Chapitre 6

Conclusions et perspectives

6.1 Travaux accomplis

L'un des problèmes majeurs relevant du domaine de CAO est la question de l'amélioration des performances des méthodes de vérification de la correction du comportement du système conçu. Cette amélioration a pour but, à la fois d'assurer une qualité supérieure du produit final, et d'augmenter la productivité du flot de conception. C'est à ce titre que dans cette thèse nous avons proposé des méthodes d'amélioration des performances de la simulation des modèles décrits en langage de description de matériel. Ainsi, nous nous sommes penchés sur les problèmes d'accélération de deux types de simulation : d'une part de la simulation dirigée par les événements, et d'autre part de celle dirigée par l'horloge.

Afin d'améliorer les performances de la simulation dirigée par les événements, il existe deux approches possibles pouvant être mis en place. La première approche se fonde sur une modification du code d'un modèle simulable du système. Quant à la deuxième, ayant comme objectif l'optimisation du processus d'exécution de la simulation, elle peut soit perfectionner les simulateurs existants, soit même aboutir à la création d'un nouveau type de simulateur. Dans cette thèse nous proposons les méthodes qui sont basées exclusivement sur les modifications apportées dans le code d'un modèle. De façon générale, ces modifications exploitent le potentiel d'accélération d'un modèle, et sont liées aux types de constructions utilisées pour décrire un comportement du système. Selon l'analyse de la performance des constructions du langage VHDL, certaines d'entre elles, même équivalentes au niveau sémantique, ne possèdent pas des performances identiques en simulation. C'est ce phénomène qui a permis de développer un ensemble de règles d'optimisation et de transformation du code d'un modèle. L'application de ces règles à un modèle donné ne change pas son comportement modélisé, mais permet de rendre celui-ci plus rapide en simulation.

Les autres méthodes d'accélération de la simulation dirigée par les événements, présentées dans cette thèse, sont fondées sur l'abstraction d'un modèle initial. Nous avons proposé trois de ces méthodes. Elles permettent, en fonction de l'objectif spécifique de la simulation, de modifier ou de simplifier le modèle initial dans ses parties non-critiques ou non-observées en simulation. Ce processus a pour but de rendre plus performante la simulation des certaines parties du modèle ou de certains aspects de son comportement, qui sont au centre d'intérêt de cette simulation.

Tout en offrant de meilleures performances en exécution de la simulation, les méthodes proposées permettent de conserver tous les avantages de la simulation dirigée par les événements, c'est-à-dire le même niveau de détails et la même résolution du comportement d'un circuit modélisé sur le plan fonctionnel, temporel et structurel.

Les deux facteurs : la méthode d'accélération appliquée au modèle, ainsi que l'ensemble des constructions du langage et des objets utilisés dans la description initiale de son comportement, portent sur l'augmentation globale de la performance de la simulation. Comme l'ont montré les résultats expérimentaux, dans le cas de certains circuits, cette augmentation peut même aller jusqu'à 4.2 fois (c'est le cas par exemple de l'abstraction comportementale). On peut donc considérer le taux de l'augmentation obtenue comme étant très prometteur, dans la mesure où il justifie l'effort supplémentaire lié à l'application de la méthode proposée dans la pratique de la conception.

En ce qui concerne leur compatibilité avec le flot de conception basé sur l'utilisation des langages de description de matériel, et avec les outils de simulation actuellement disponibles sur le marché, la mise en place des méthodes décrites dans la pratique de conception semble être naturelle et aisée.

En prenant en compte tous les avantages montrés ci-dessus, il est évident que les méthodes fournies présentent un intérêt certain pour l'industrie, et cela même si d'autres méthodes d'augmentation de l'efficacité de la simulation et de la productivité du processus de la vérification, parmi lesquelles la simulation dirigée par l'horloge, sont disponibles.

L'augmentation de la performance de la simulation dirigée par l'horloge fait l'objet de la deuxième partie de la thèse. Les méthodes qui permettent d'améliorer la performance de ce type de simulation sont entièrement fondées sur l'utilisation d'une représentation mathématique de la fonctionnalité d'un système par un réseau des graphes de décision de haut niveau. Déjà, la seule application de cette représentation à la modélisation d'un système a montré son intérêt du point de vue de la performance de simulation, sans même que soient pris en compte l'aspect efficacité d'exécution de la simulation.

Afin de modéliser d'une manière optimale différents types de circuits, et cela sur plusieurs niveaux d'abstraction, diverses formes de graphes ont été introduites. Le développement de ces formes spécifiques, tels que par exemple les graphes vectoriels VDDs, les graphes vectoriels compressés CVDDs, ou les graphes RODDs, découlait de la nécessité de s'adapter aux attributs particuliers des systèmes modélisés : certaines de ces formes sont donc dédiées à la modélisation au niveau algorithmique (les VDDs et les CVDDs) et les autres à la modélisation de la partie contrôle du système (les RODDs). Les résultats expérimentaux obtenus suite à l'application de nouveaux types de graphes, grâce à laquelle la simulation peut être accélérée 2 à 4.6 fois, montrent l'avantage de ces graphes par rapport aux graphes conventionnels DDs.

Pour simuler efficacement un réseau de graphes de décision, un simulateur dirigé par l'horloge a été développé. Dans ce simulateur, quatre algorithmes ont été mis en place dans le but de gérer l'exécution de la simulation. Trois parmi eux, qui sont développés en vue de l'efficacité du processus de simulation, permettent d'accélérer l'exécution de la simulation jusqu'à 60% par rapport à la simulation dirigée par l'algorithme de base.

Une fois tous les facteurs de l'accélération de la simulation décrits ci-dessus appliqués, la simulation d'un modèle du système sous forme de graphes de décision offre un avantage considérable en terme de performance par rapport aux outils commerciaux. Les résultats des expériences montrent que le gain obtenu, grâce à l'application de la méthode proposée dans ce travail, s'élève de 1.6 à 6.7 fois en comparaison de la simulation VHDL dirigée par l'horloge, et de 2 à 32 fois si on la compare avec la simulation VHDL dirigée par les événements.

6.2 Perspectives

Les perspectives de notre travail portent aussi bien sur l'aspect théorique que sur l'aspect pratique.

Sur le plan théorique, plusieurs axes de recherche peuvent être envisagés. Dans la partie du travail concernant l'amélioration des performances du code VHDL (chapitres 2 et 3), plusieurs méthodes de l'optimisation, de la transformation et de l'abstraction ont été proposées. Ces méthodes ont été élaborées suite à l'étude approfondie de la sémantique de simulation du langage VHDL, telle qu'elle est décrite dans le manuel du langage VHDL (la norme IEEE Std 1076-1996 [59]). Pour s'assurer, que l'application de ces méthodes au modèle ne modifie pas son comportement initial, il est souhaitable de prouver formellement que le comportement du modèle accéléré est équivalent à celui du modèle de départ. C'est à ce titre que deux approches formelles peuvent être explorées. La première, basée sur l'application des méthodes et des outils de vérification d'équivalence, permettra de prouver l'équivalence d'un modèle initial donné et de sa forme accélérée. Quant à la deuxième, plus complète, elle peut se reposer sur la preuve formelle qui, en utilisant un démonstrateur de théorèmes, permet de prouver la correction des règles de l'optimisation ou de la transformation. Une fois les règles prouvées, elles peuvent être employées pour toutes les formes possibles dans lesquelles les constructions du langage, étant l'objet de l'optimisation ou de la transformation, peuvent être utilisées. Dans cette thèse nous avons commencé une étude qui, en utilisant le démonstrateur de théorèmes ACL2 [20, 42], vise à prouver la correction de certaines règles de la transformation. Par manque de temps, ce travail a été suspendu.

Un autre axe de recherche concerne les méthodes d'abstraction comportementale (chapitre 3). Dans cette thèse, nous avons proposé une méthode d'abstraction comportementale qui s'applique à une seule unité VHDL (un bloc du projet). Une généralisation de cette méthode, qui permettra de traiter, non seulement un seul bloc, mais également tous les autres blocs dans un modèle hiérarchisé composé de plusieurs blocs, peut être envisagée. Cette approche rendra efficace la simulation, dont l'objectif est la vérification précise d'un bloc sélectionné du circuit dans son environnement complet. En vue de cet objectif, il est possible que cet environnement soit abstrait (c'est-à-dire réduit). Celui-ci pourra donc conserver uniquement les parties de son comportement qui sont nécessaires à modéliser l'interface avec le bloc sélectionné. Toutes les autres parties de l'environnement, celles qui n'interagissent pas directement avec le bloc simulé, pourront alors être réduites.

Une autre application de la méthode d'abstraction comportementale, basée sur la technique d'invariance des entrées du modèle (chapitre 3), peut aboutir au développement d'une méthode spécifique de gestion du processus de simulation. Dans cette méthode, une analyse des jeux de vecteurs d'entrée devrait être effectuée. Cette analyse permettra de partager l'ensemble des tous les vecteurs en partitions, dans lesquelles les valeurs de certains signaux seront constantes. A partir d'un modèle initial, un modèle abstrait sera généré pour chacune de ces partitions. Le processus de génération des modèles abstraits sera dirigé par l'information propre à chaque partition des vecteurs d'entrée. Afin d'optimiser le processus de simulation, chaque modèle abstrait correspondant au jeu de vecteurs d'entrée simulé à un moment donné, devrait être utilisé à son tour. Quant à la méthode de partage, qui s'avère indispensable pour mettre en œuvre la méthode de gestion, elle pourra reposer sur l'estimation d'efficacité en simulation des modèles abstraits correspondants aux partitions. Le problème qui entre autres devrait être étudié, est la question d'un partage optimal qui trouvera le compromis entre le nombre des partitions et l'efficacité globale de la méthode de gestion.

La future recherche dans le domaine d'application des graphes de décision peut aller dans deux directions. La première direction devrait se préoccuper du développement des méthodes de création des graphes de décision de haut niveau. Comme nous l'avons observé au cours de la création de la représentation d'un système sous forme des graphes, la complexité de cette représentation dépend du choix de l'ordre des termes qui marquent les nœuds d'un graphe (*node labeling custom order*). Ce problème est d'ailleurs connu dans la création des autres types de graphes, en commençant par les différentes formes de BDDs. Même si le type de graphes proposé permet de représenter un système sous forme compacte, la taille de cette représentation influencera certainement la performance de la simulation, ainsi que la maniabilité des représentations des systèmes d'une grande complexité. Nous présumons que les méthodes de création des représentations compactes, similaires à celles proposées pour les autres types de graphes, peuvent être développées ou bien adaptées pour les graphes de décision de haut niveau. Ces méthodes pourront exploiter, entre autres, les différents moyens d'optimisation des termes marquant les nœuds d'un graphe et, par conséquent, les techniques de simulation permettant de bénéficier de ce genre d'optimisations, pourront être proposées.

Une autre direction de la recherche à explorer, est celle qui s'intéresse aux problèmes de la génération de la représentation des types particuliers de circuits sous forme de graphes. A la suite d'une analyse des résultats d'expériences, nous avons proposé les graphes de décision RODDs, qui semblent être les mieux adaptés pour modéliser les circuits avec une partie contrôle dominante (voir chapitre 5). Cette forme de représentation, associée avec un algorithme de simulation exploitant les avantages offerts par les RODDs, permet d'obtenir une accélération de la simulation qui est supérieure de 60% par rapport à la représentation utilisant les DDs conventionnels. Une analyse plus profonde de ce phénomène va probablement aboutir au développement de formes de graphes ou d'algorithmes de simulation, lesquels prendront en compte les diverses classes de circuits (par exemple les circuits de la partie contrôle ou du chemin de données) et adapteront une représentation optimale à la base de DDs, ainsi que la technique de simulation la plus performante à cette forme de représentation.

Sur le plan pratique, nous pouvons envisager des améliorations des outils prototypes, ainsi que le développement de leurs parties, jusqu'alors inachevées. Dans la partie de ce travail, qui met en œuvre les méthodes d’optimisation et de transformation du code VHDL nous avons développé un prototype d’outil, qui permet d’appliquer quelques-unes de ces méthodes, afin de démontrer la faisabilité de l’implémentation de celles-ci. Les travaux à accomplir peuvent viser l’implémentation des autres méthodes, ainsi que l’intégration des logiciels mettant en pratique dans un environnement uniforme chacune de ces méthodes. Toutes les méthodes d’abstraction doivent être intégrées dans le même environnement.

L’objectif final de cette intégration est d’obtenir un environnement complet de simulation, permettant à la fois de gérer toutes les opérations d’accélération d’un modèle, ainsi que d’adapter les règles d’optimisation et de transformation au simulateur spécifique, et à la plate-forme de simulation donnée (chapitre 2.4).

Une autre partie du travail pratique devrait être consacrée à l’adaptation des méthodes d’accélération, développées initialement pour le langage VHDL, au langage de description de matériel Verilog. Etant donné les ressemblances entre les deux langages : VHDL et Verilog, la réalisation de cette tâche ne doit pas poser de problème, surtout dans la mesure où les deux langages sont basés sur le même principe de simulation, et leurs constructions ont une sémantique similaire.

La mise en œuvre d’un outil de génération automatique d’une représentation des graphes de décision de haut niveau à partir d’un modèle décrit en VHDL (ou bien en Verilog), qui a fait l’objet de la partie de la thèse concernant l’application des graphes de décision (chapitres 4 et 5), constitue une perspective immédiate dans les travaux pratiques. Si les futurs travaux de recherche dans le domaine des graphes de décision aboutissent, la génération de la représentation pourra, en fonction des besoins de la performance de simulation, incorporer l’adaptation d’une forme particulière de graphes propre à une certaine classe des circuits.

Bibliographie

Références concernant les techniques de simulation :

[1, 2, 9, 15, 29, 46, 47, 48, 51, 53, 70, 80, 81, 84, 85, 88, 90, 110, 111, 112, 115, 122, 123, 125, 128, 129, 130, 134, 139, 141, 146, 147, 151, 155, 156, 157, 160, 164, 167, 168, 169, 173, 174]

Références concernant la performance de la simulation :

[9, 10, 28, 31, 36, 37, 45, 53, 54, 66, 67, 79, 81, 82, 86, 87, 90, 99, 100, 105, 108, 109, 110, 113, 114, 117, 118, 121, 122, 123, 128, 137, 138, 163, 167, 168, 169]

Références concernant les exemples de l'application de simulation :

[3, 7, 8, 38, 55, 135, 136]

Références concernant les graphes de décision :

[5, 9, 12, 23, 24, 25, 32, 34, 40, 72, 78, 83, 90, 96, 110, 127, 133, 140, 152, 153, 154, 155, 156, 158]

Références concernant la vérification formelle :

[13, 19, 20, 21, 22, 26, 27, 42, 44, 71, 73, 91, 92, 98, 102, 103, 106, 107, 116, 132, 142, 143, 148, 159, 170, 171, 172, 174]

Références concernant les langages de description de matériel :

[4, 11, 14, 17, 30, 31, 36, 45, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 74, 82, 86, 87, 93, 94, 95, 118, 120, 131, 137, 138, 148, 149, 163]

Références concernant les mesures de qualité du code HDL :

[14, 39, 86, 87, 144, 165, 166]

Références diverses :

[6, 16, 18, 21, 33, 35, 41, 43, 49, 50, 52, 68, 69, 73, 74, 75, 76, 77, 89, 97, 101, 104, 119, 124, 126, 127, 145, 150, 161, 162, 175]

[1] R.D. Acosta, S.P. Smith, J. Larson: "*Mixed-Mode Simulation of Compiled VHDL Programs*", IEEE International Conference on Computer-Aided Design ICCAD'89, November 5-9, 1989, pp. 176-179

[2] V.D. Agrawal, S.T. Chakradhar: "*Logic Simulation and Parallel Processing*", IEEE International Conference on Computer-Aided Design ICCAD'90, Santa Clara, CA, USA, November 11-15, 1990, pp. 496-499

[3] A. Aharon, D. Goodman, M. Levinger, Y. Lichtenstein, Y. Malka, C. Metzger, M. Molcho, G. Shurek: "*Test Program Generation for Functional Verification of PowerPC Processors in IBM*", IEEE/ACM Design Automation Conference DAC'95, San Francisco, CA, USA, June 12-16, 1995, pp. 279-285

[4] R. Airiau, J.-M. Bergé, V. Olive, J. Rouillard: "*VHDL: langage, modélisation, synthèse*", Presses Polytechniques et Universitaires Romandes et CNET-ENST, Deuxième édition, 1998

[5] S.B. Akers: "*Binary Decision Diagrams*", IEEE Transactions on Computers, vol. C-27, June 1978, pp. 509-516

[6] H. Al-Asaad, J.P. Hayes: “*Design Verification via Simulation and Automatic Test Pattern Generation*”, IEEE/ACM International Conference on Computer-Aided Design ICCAD’95, San Jose, CA, USA, November 5-9, 1995, pp. 174-180

[7] T.W. Albrecht: “*Concurrent Design Methodology and Configuration Management of the SIEMENS EWSD - CCS7E Processor System Simulation*”, IEEE/ACM Design Automation Conference DAC’95, San Francisco, CA, USA, June 12-16, 1995, pp. 222-227

[8] A. Allara: “*ILC16 – 16 HDLC Channels Italtel Link Controller*”, Italtel internal report, 1997

[9] P. Ashar, S. Malik: “*Fast Functional Simulation Using Branching Programs*”, IEEE/ACM International Conference on Computer-Aided Design ICCAD’95, San Jose, CA, USA, November 5-9, 1995, pp. 408-412

[10] P. Ashenden: “*A Comparison of Recursive and Repetitive Models of Recursive Hardware Structures*”, VHDL International User Forum Spring 1994 VIUF’94, Oakland, CA, May 1-4, 1994, pp. 80-89

[11] P. Ashenden: “*The Designer’s Guide to VHDL*”, Morgan Kaufmann Publishers, Inc., San Francisco, CA, 1996

[12] R.I. Bahar, E.A. Frohm, C.M. Gaona, G.D. Hachtel, E. Macii, A. Pardo, F. Somenzi: “*Algebraic Decision Diagrams and Their Application*”, Formal Methods in System Design, vol. 10, Kluwer Academic Publishers, Dordrecht, The Netherlands, 1997, pp. 171-206

[13] F. Balarin, K. Sajid: “*Simplifying Data Operations for Formal Verification*”, IFIP TC10 WG10.5 International Conference on Computer Hardware Description Languages and their Applications, Chapman & Hall, Toledo, Spain, April 20-25, 1997, pp. 27-39

[14] A. Balboni, M. Mastretti, M. Stefanoni: “*Static Analysis for VHDL Model Evaluation*”, Euro-DAC’94 with Euro-VHDL’94 Conference, Grenoble, France, September 19-23, 1994, pp. 586-591

[15] J. Bauer, M. Bershteyn, I. Kaplan, P. Vyedin: “*A Reconfigurable Logic Machine for Fast Event-Driven Simulation*”, IEEE/ACM Design Automation Conference DAC’98, San Francisco, CA, USA, June 15-19, 1998, p. 668-671

[16] R.A. Bergamaschi: “*High-Level Synthesis in a Production Environment*”, in Fundamentals and Standards in Hardware Description Languages, NATO ASI Series, Series E: Applied Science, edited by J. Mermet, vol. 249, Kluwer Academic Publishers, Dordrecht, The Netherlands, 1993, pp. 195-230

[17] J.-M. Bergé, A. Fonkoua, S. Maginot, J. Rouillard: “*VHDL Designer’s Reference*”, Kluwer Academic Publishers, Dordrecht, The Netherlands, 1992

[18] T.L. Booth: “*Sequential Machines and Automata Theory*”, John Wiley & Sons, Inc., New York-London-Sydney, 1967

[19] D. Borrione, F. Corno, P. Prinetto: “*A Tutorial: Formal Methods for VHDL*”, Euro-DAC'94 with Euro-VHDL’94 Conference, Grenoble, France, September 1994

[20] D. Borrione, P. Georgelin, V. Moraes Rodrigues: “*Symbolic Simulation and Verification of VHDL with ACL2*”, International HDL Conference HDLCon 2000, San Jose, CA, USA, March 8-10, 2000, pp. 59-65

[21] D. Borrione, F. Vestman, H.Bouamama: “*An Approach to Verilog-VHDL Interoperability for Synchronous Designs*”, IFIP WG 10.5 Advanced Research Working Conference on Correct Hardware Design and Verification Methods "CHARME'97", Chapman & Hall Publishers, Montreal, Canada, October 16-18, 1997

[22] R.S. Boyer, J.S. Moore: “*A Computational Logic Handbook*”, Academic Press, Inc., 1988

[23] R. Bryant: “*Graph-Based Algorithms for Boolean Function Manipulation*”, IEEE Transactions on Computers, vol. C-35, No. 8, August 1986, pp. 677-691

[24] R. Bryant, Y.-A. Chen: “*Verification of Arithmetic Functions with Binary Moment Diagrams*”, Carnegie Mellon University report CMU-CS-94-160, Pittsburgh, USA, May 1994

[25] R. Bryant, Y.-A. Chen: “*Verification of Arithmetic Circuits with Binary Moment Diagrams*”, IEEE/ACM Design Automation Conference DAC’95, San Francisco, CA, USA, June 12-16, 1995, pp. 535-541

[26] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill: “*Sequential Circuit Verification Using Symbolic Model Checking*”, IEEE/ACM Design Automation Conference DAC’90, Orlando, USA, June 24-28, 1990, pp. 46-51

[27] P. Camurati, P. Prinetto: “*Formal Verification of Hardware Correctness: Introduction and Survey of Current Research*”, IEEE Computer, July 1989, pp. 8-19

[28] CENELEC Report: “*VHDL Modeling Guidelines. Simulation and Documentation Aspects.*”, Comité Européen de Normalisation Électrotechnique CENELEC TC217/WG2 report 2.14, second draft, 1997

[29] S.J. Chandra, J.H. Patel: “*Accurate Logic Simulation in the Presence of Unknowns*”, IEEE International Conference on Computer-Aided Design ICCAD’89, November 5-9, 1989, pp. 34-37

[30] D.R. Coelho: “*The VHDL Handbook*”, Kluwer Academic Publishers, Dordrecht, The Netherlands, 1989

[31] B. Cohen: “*VHDL Coding Styles and Methodologies… and In-Depth Tutorial*”, Kluver Academic Publisher, 1995

[32] F. Corella, Z. Zhou, X. Song, M. Langevin, E. Cerny: “*Multiway Decision Graphs for Automated Hardware Verification*”, Formal Methods in System Design, vol. 10, Kluwer Academic Publishers, Dordrecht, The Netherlands, 1997, pp. 7-46

[33] K.H. Diener, G. Elst, E. Ivask, J. Raik, R. Ubar: “*FPGA Design Flow with Automated Test Generation*”, 11th Workshop on Test Technology and Reliability of Circuits and Systems, 1999, pp. 120-123

[34] R. Drechsler, B. Becker: “*Binary Decision Diagrams. Theory and Implementation*” Kluwer Academic Publishers, Dordrecht, The Netherlands, 1998

[35] K. Duda, A. Morawiec, W. Sakowski: “*Elements of an Environment For Experiments with High-Level Synthesis*”, XVII Krajowa Konferencja Teorii Obwodow i Ukladow Elektronicznych (XVII National Conference on Circuit Theory and Electronic Systems), Karpacz, Poland, October 1994

[36] Electronic Industries Association EIA: “*EIA-567A. VHDL Hardware Component Modeling and Interface Standard*”, EIA Standard, Washington DC, 1989

[37] European Space Agency ESA/ESTEC and E2S n.v.: “*Project SCADES-2/WP5520*”, document reference ES2/SCADES2/VHDL/WP5520/1.0, 1996

[38] A. Evans, A. Silburt, G. Vrckovnik, T. Brown, M. Dufresne, G. Hall, T. Ho, Y. Liu: “*Functional Verification of Large ASICs*”, IEEE/ACM Design Automation Conference DAC’98, San Francisco, CA, USA, June 15-19, 1998, pp. 650-655

[39] F. Fallah, S. Devadas, K. Keutzer: “*OCCOM: Efficient Computation of Observability-Based Code Coverage Metrics for Functional Verification*”, IEEE/ACM Design Automation Conference DAC’98, San Francisco, CA, USA, June 15-19, 1998, pp. 152-157

[40] M. Fujita, P.C. McGeer, J.C.-Y. Yang: “*Multi-Terminal Binary Decision Diagrams: An Efficient Data Structure for Matrix Representation*”, Formal Methods in System Design, vol. 10, Kluwer Academic Publishers, Dordrecht, The Netherlands, 1997, pp. 149-169

[41] L.T.F. Gamut: “*Logic, Language, and Meaning*”, The University of Chicago Press, Chicago and London, 1991

[42] P. Georgelin: “*Formalisation et validation de paquetages standards VHDL dédiés à la synthèse à l’aide du démonstrateur de théorèmes ACL2*”, DEA de Microélectronique, Université Joseph Fourier, Grenoble, France, Juin 1998

[43] E. Gramatova et al.: "*FUTEG Benchmarks*", Technical Report COPERNICUS JEP 9624 FUTEG No9/1995, 1995

[44] A. Gupta: "*Formal Hardware Verification Methods: A Survey*", Formal Methods in System Design, 1, 1992, pp. 151 - 238

[45] S. Habinc: "*VHDL Models for Board-level Simulation*", European Space Agency ESA/ESTEC Report WSM/SH/010, February 1996, ftp://ftp.estec.esa.nl/pub/vhdl/doc/BoardLevel.ps

[46] W. Hahn, A. Hagerer, C. Herrmann: "*Compiled-Code-Based Simulation with Timing Verification*", Euro-DAC'94 with Euro-VHDL'94 Conference, Grenoble, France, September 19-23, 1994, pp. 362-367

[47] C. Hansen: "*Hardware Logic Simulation by Compilation*", IEEE/ACM Design Automation Conference DAC'88, Anaheim, CA, USA, June 12-15, 1988, pp. 712-715

[48] C. Hansen, A. Kunzmann, W. Rosenstiel: "*Verification by Simulation Comparison Using Interface Synthesis*", IEEE Conference on Design Automation and Test in Europe DATE'98, Paris, France, February 23-26, 1998, p. 436-443

[49] J.L. Hein: "*Discrete Structures, Logic, and Computability*", Jones and Bartlett Publishers, Boston, 1995

[50] F.C. Hennie: "*Finite-State Models for Logical Machines*", John Wiley & Sons, Inc., New York-London-Sydney, 1968

[51] M. Heydemann, D. Dure: "*The Logic Automation Approach to Accurate and Efficient Gate and Functional Level Simulation*", IEEE International Conference on Computer-Aided Design ICCAD'88, Santa Clara, CA, USA, November 7-10, 1988, pp. 250-253

[52] HLSynth92 benchmark directory at URL: http://www.cbl.ncsu.edu/pub/Benchmark_dirs/HLSynth92/, 1992

[53] S. Hodgson, Z. Shaar, A. Smith: "*A High Performance VHDL Simulator for Large Systems Design*", ", Euro-DAC'95 with Euro-VHDL'95 Conference, Brighton, UK, September 18-22, 1995

[54] M. Hueber, S. Lasserre, A. de Monteville: "*VHDL Experiments on Performance*", Euro-VHDL'91 Conference, Stockholm, Sweden, September 8-11, 1991, pp. 282-292

[55] D. Humblot, "*Naissance d'un Ordinateur : Bull DPS 7000, à livre ouvert*", Œuvre collective coordonnée par Dan Humblot, Réseaux et Système d'Information Bull S.A., BULL, 1992

[56] IEEE Standard Interface for Hardware Description Models of Electronic Components, IEEE Std 1499-1998, The Institute of Electrical and Electronics Engineers, Inc., New York, NY, USA, November 24, 1999

[57] IEEE Standard Draft of the Shared Variable Language Change Specification (PAR 1076A), Design Automation Standards Committee of the Institute of Electrical and Electronics Engineers Computer Society, 1999

[58] IEEE Standard Multivalue Logic System for VHDL Model Interoperability (Std_logic_1164), IEEE Std 1164-1993, The Institute of Electrical and Electronics Engineers, Inc., New York, NY, USA, May 26, 1993

[59] IEEE Standard VHDL Language Reference Manual, IEEE Std 1076-1993, The Institute of Electrical and Electronics Engineers, Inc., New York, NY, USA, June 6, 1994

[60] IEEE Standard VHDL Analog and Mixed-Signal Extensions, IEEE Std 1076.1-1999, The Institute of Electrical and Electronics Engineers, Inc., New York, NY, USA, 1999

[61] IEEE Standard VHDL Language Math Packages, IEEE Std 1076.2-1996, The Institute of Electrical and Electronics Engineers, Inc., New York, NY, USA, 1996

[62] IEEE Standard VHDL Synthesis Packages, IEEE Std 1076.3-1997, The Institute of Electrical and Electronics Engineers, Inc., New York, NY, USA, 1997

[63] IEEE Standard for VITAL Application-Specific Integrated Circuit (ASIC) Modeling Specification, IEEE Std 1076.4-1995, The Institute of Electrical and Electronics Engineers, Inc., New York, NY, USA, 1995

[64] IEEE Standard for VHDL Register-Transfer Level (RTL) Synthesis, IEEE Std 1076.6-1999, The Institute of Electrical and Electronics Engineers, Inc., New York, NY, USA, 1999

[65] IEEE Standard Hardware Description Language Based on the Verilog Hardware Description Language, IEEE Std 1364 –1995, The Institute of Electrical and Electronics Engineers, Inc., New York, NY, USA, 1995

[66] M. Khosravipour, G. Gridling, H. Grünbacher: "*Improving Simulation Efficiency by Hierarchical Abstraction Transformations*", Forum on Design Languages FDL'98, Lausanne, Switzerland, September 6-11, 1998, vol.2, pp 85-92

[67] M. Khosravipour: "*Modeling and Simulation of Computer Systems Based on Abstraction Networks*", Ph.D. Thesis, Vienna University of Technology, 1996

[68] N. Kim, H. Choi, S. Lee, S. Lee, I. Park, C.-M. Kyung: "*Virtual Chip: Making Functional Models Work On Real Target Systems*", IEEE/ACM Design Automation Conference DAC'98, San Francisco, CA, USA, June 15-19, 1998, pp. 170-173

[69] D. Kirovski, M. Potkonjak, L. Guerra: "*Functional Debugging of Systems-on-Chip*", IEEE/ACM International Conference on Computer-Aided Design, ICCAD'98, San Jose, CA, USA, 1998, pp. 525-528

[70] S. Kumar, F. Rose: "*Integrated Simulation of Performance Models and Behavioral Models*", VHDL International User Forum VIUF Fall 1996, Durham, USA, October 1996, pp. 185-194

[71] R. Kurshan: "*Automata-Theoric Verification*", An Advanced Study Institute of the NATO Science Committee on Verification of Digital and Hybrid Systems, Antalya Turkey, May-June 1997

[72] Y.-T. Lai, S. Sastry: "*Edge-Valued Binary Decision Diagrams for Multi-Level Hierarchical Verification*", IEEE/ACM Design Automation Conference DAC'92, USA, June 1992, pp. 608-613

[73] L. Lavagno, A. Sangiovanni-Vincentelli, E. Sentovich: "*Models of Computation for Embedded System Design*", in System Level Synthesis, NATO Science Series, Series E: Applied Science edited by A. Jerraya and J. Mermet, vol. 357, Kluwer Academic Publishers, Dordrecht, The Netherlands, 1999, pp. 45-102

[74] LEDA: "*APEX Atomic Property EXtractor for full VHDL. Implementor's Guide, Version 1.2*", LEDA S.A., 1994-1997

[75] LEDA: "*LVS Implementor's Guide: VHDL*Verilog Intermediate Format (VIF), Version 4.1*", LEDA S.A., 1997

[76] LEDA: "*VHDL*Verilog System. VHDL Compiler. User's Manual, Version 4.1*", LEDA S.A., 1997

[77] LEDA: "*VHDL*Verilog System. LEDA Procedural Interface (LPI). Implementor's Guide, Version 4.1*", LEDA S.A., 1997

[78] R. Leveugle, R. Ubar: "*Synthesis of Decision Diagrams from Clock-Driven Multi-Process VHDL Descriptions for Test Generation*", 5th International Conference on Mixed Design of Integrated Circuits and Systems, £odz, Poland, June 18-20, 1998, pp. 353-358.

[79] O. Levia: "*Writing High Performance VHDL Models*", Euro-VHDL'91 Conference, Stockholm, Sweden, September 8-11, 1991, pp. 119-127

[80] D.M. Lewis: "*Hierarchical Compiled Event-Driven Logic Simulation*", IEEE International Conference on Computer-Aided Design ICCAD'89, November 5-9, 1989, pp. 498-501

[81] Y. Luo, T. Wongsonegoro, A. Aziz: "*Hybrid Techniques for Fast Functional Simulation*", IEEE/ACM Design Automation Conference DAC'98, San Francisco, CA, USA, June 15-19, 1998, pp. 664-667

[82] V. Madisetti: "*VHDL Simulations – Tips for Speeding*", http://rassp.scra.org/public/tb/gt/vhdl-tips.txt

[83] S. Malik, A. Wang, R. Brayton, A. Sangiovanni-Vincentelli: "*Logic Verification Using Binary Decision Diagrams in a Logic Synthesis Environment*", IEEE International Conference on Computer-Aided Design ICCAD'88, Santa Clara, CA, USA, November 7-10, 1988, pp. 6-9

[84] R.T. Maniwa: "*Focus Report: Formal Verification, Cycle-Based Simulation, Timing Analysis, and ESL Entry*", EE Design, http://www.eedesign.com/Eedesign/ FocusReport9606.htm, June 1996

[85] J. Marantz: "*Enhanced Visibility and Performance in Functional Verification by Reconstruction*", IEEE/ACM Design Automation Conference DAC'98, San Francisco, CA, USA, June 15-19, 1998, pp. 164-169

[86] M. Mastretti: "*VHDL Quality: Synthesizability, Complexity and Efficiency Evaluation*", Euro-DAC'95 with Euro-VHDL'95 Conference, Brighton, UK, September 18-22, 1995

[87] M. Mastretti, M. Sturlesi, S. Tomasello: "*Static Analysis of VHDL Code: Simulation Efficiency and Complexity*", VHDL International Users' Forum Conference Spring 1995 VIUF'95, San Diego, CA, April 2-6, 1995, pp. 7.1-7.9

[88] P.M. Maurer: "*Optimization of the Parallel Technique for Compiled Unit-Delay Simulation*", IEEE International Conference on Computer-Aided Design ICCAD'90, Santa Clara, CA, USA, November 11-15, 1990, pp. 70-73

[89] E.J. McCluskey: "*Logic Design Principles with Emphasis on Testable Semicustom Circuits*", Prentice-Hall, New Jersey, 1986

[90] P. McGeer, K. McMillan, A. Saldanha, A. Sangiovanni-Vincentelli, P. Scaglia: "*Fast Discrete Function Evaluation Using Decision Diagrams*", IEEE/ACM International Conference on Computer-Aided Design ICCAD'95, San Jose, CA, USA, November 5-9, 1995, pp. 402-407

[91] K. McMillan: "*CTL Model Checking*", An Advanced Study Institute of the NATO Science Committee on Verification of Digital and Hybrid Systems, Antalya Turkey, May-June 1997

[92] K. McMillan: "*Symbolic Model Checking*", An Advanced Study Institute of the NATO Science Committee on Verification of Digital and Hybrid Systems, Antalya Turkey, May-June 1997

[93] P. Mencini: "*Introduction to Hardware Description Languages Implemented in the 80's: VHDL*", in Fundamentals and Standards in Hardware Description Languages, NATO ASI Series, Series E: Applied Science, edited by J. Mermet, vol. 249, Kluwer Academic Publishers, Dordrecht, The Netherlands, 1993, pp. 359-384

[94] P. Mencini, S. Olcoz: "*HDL Interoperability: A Compiler Technology Perspective*", VHDL International Users' Forum Conference Fall 1996, Durham, USA, October 27-30, 1996, pp. 51-58

[95] J.P. Mermet (ed.): "*Fundamentals and Standards in Hardware Description Languages*", NATO ASI Series, Series E: Applied Science, vol. 249, Kluwer Academic Publishers, Dordrecht, The Netherlands, 1993

[96] S. Minato: "*Binary Decision Diagrams and Applications for VLSI CAD*", Kluwer Academic Publishers, Dordrecht, The Netherlands, 1996

[97] Model Technology, Inc.: "*ModelSim PE/PLUS VHDL, Verilog, and Mixed-HDL Simulation for PCs Running Windows 95 & Windows NT. User's Guide. Version 4.7*", Model Technology Incorporated, Beaverton, USA, October 1997

[98] J. Moore: "*Theorem Proving*", An Advanced Study Institute of the NATO Science Committee on Verification of Digital and Hybrid Systems, Antalya Turkey, May-June 1997

[99] A. Morawiec: "*VHDL Simulation Performance Evaluation*", Technical Report, Part 1, TIMA Laboratory, 1998

[100] A. Morawiec: "*VHDL Simulation Performance Evaluation*", Technical Report, Part 2-3, TIMA Laboratory, 1999

[101] A. Morawiec: "*The Use of Hardware Description Language VHDL in Design of Electronic Integrated Circuits on the Example of a Graphic Processor Design*" - Master of Science degree dissertation, Silesian Technical University Gliwice, Poland, November 1993

[102] A. Morawiec: "*Inventaire des outils de preuve formelle de circuits développés en Europe. Test d'utilisabilité de deux d'entre eux sur un exemple d'origine industrielle*", Diplôme des Études Approfondies de Microélectronique (DEA), Laboratoire TIMA, Université Joseph Fourier, Grenoble, France, September 1996

[103] A. Morawiec, "*Inventaire des outils de preuve formelle de circuits développés en Europe*", Rapport de recherche DRET N° 96-050, Laboratoire TIMA, Juin 1997

[104] A. Morawiec, T. Gore: "*Emerging Standards for IP Exchange*", Intellectual Property Conference IP97, Santa Clara, CA, USA, March 17-18, 1997, pp. 374-387

[105] A. Morawiec, J. Mermet: "*Behavioral Abstraction of HDL Models for Simulation Performance*", Asia-Pacific Conference on Chip Design Languages APChDL2000 at 16th IFIP World Computer Congress 2000, Beijing, China, August 21-24, 2000

[106] A. Morawiec, J. Mermet: "*European Formal Verification Tools for Model Correctness*", Workshop on Libraries, Component Modeling, and Quality Assurance LCM&QA'97, Toledo, Spain, April 1997, pp. 181-192

[107] A. Morawiec, J. Mermet: "*A Survey of Formal Hardware Verification Tools Developed in Europe*", Asia-Pacific Conference on Hardware Description Languages APCHDL'97, Hsin-Chu, Taiwan, August 18-20, 1997

[108] A. Morawiec, J. Mermet: "*Méthodes pour l'amélioration de la performance de simulation*", Colloque CAO de Circuits Intégrés et Systèmes, Fuveau, Aix-en-Provence, France, May 10-12, 1999, pp. 262-265

[109] A. Morawiec, J. Mermet: "*Techniques for Improving the HDL Simulation Performance*", Forum on Design Languages FDL'99, Lyon, France, August 30-September 3, 1999, pp. 91-100 (Best Poster Award)

[110] A. Morawiec, R. Ubar, J. Raik: "*Cycle-Based Simulation Algorithms for Digital Systems Using High-Level Decision Diagrams*" IEEE Conference on Design, Automation and Test in Europe DATE 2000, Paris, France, March 27-30, 2000, p. 743

[111] R. Murgai, F. Hirose, M. Fujita: "*Speeding Up Look-up-Table Driven Logic Simulation*", IFIP TC10 WG10.5 Tenth International Conference on Very Large Scale Integration VLSI'99, Kluwer Academic Publisher, Lisboa, Portugal, December 1-4, 1999, pp. 385-397

[112] E. Naroska: "*Parallel VHDL Simulation*", IEEE Conference on Design, Automation and Test in Europe DATE'98, Paris, France, February 23-26, 1998, pp. 159-163

[113] Z. Navabi: "*Optimizing RTL Simulation Performance. Methodology and Modeling. Utility Modeling*", RASSP-Sanders Project: Simulation Acceleration, http://www.ece.neu.edu/info/vhdl/Sanders/memory/report.html

[114] Z. Navabi, A. Peymandoust: "*Optimizing RTL Simulation Performance by Reducing Simulation Activities*", RASSP-Sanders Project: Simulation Acceleration, http://www.ece.neu.edu/info/vhdl/Sanders/actsup.html

[115] A. Ottens, W. van Hoogstraeten, H. Corporaal: "*A New Flexible VHDL Simulator*", Euro-DAC'94 with Euro-VHDL'94 Conference, Grenoble, France, September 19-23, 1994, pp. 604-609

[116] S. L. Pandey, K. R. Subramanian, and P. A. Wilsey: "*A Semantic Model of VHDL for Validating Rewriting Algebras*", The 22nd Euromicro Conference "Beyond 2000: Hardware and Software Design Strategies", September 1996, pp. 167-176

[117] B. Paulsen, O. Levia: "*Techniques for Writing High Performance and High Quality VHDL Models*", Euro-DAC'92 with Euro-VHDL'92 Conference, Hamburg, Germany, September 7-10, 1992

[118] A. Pawlak, F. Bouchard, P. Bakowski: "*Survey on VHDL Modeling Guidelines*", Workshop on Libraries, Component Modeling, and Quality Assurance LCM&QA'97, Toledo, Spain, April 1997, pp. 117-128

[119] D. Perrin: "*Finite Automata*", in "Handbook of Theoretical Computer Science" edited by J. van Leeuwen, vol. B "Formal Models and Semantics", Elsevier Science Publishers B.V., 1990, pp. 1-58

[120] D.L. Perry: "*VHDL*", McGraw Hill, 1991

[121] G. Peterson: "*Performance Tradeoffs for Emulation, Hardware Acceleration, and Simulation*", International HDL Conference HDLCon 2000, San Jose, CA, USA, March 8-10, 2000, pp. 222-229

[122] A. Peymandoust, Z. Navabi: "*VHDL Concurrent Simulation of RT Level Components*", RASSP-Sanders Project: Simulation Acceleration, http://www.ece.neu.edu/info/vhdl/Sanders/conabs.html

[123] A. Peymandoust, Z. Navabi: "*VHDL Concurrent Simulation of RT Level Components*", VHDL International Users' Forum Conference Fall 1996, Durham, USA, October 27-30, 1996, pp. 353-357

[124] A. Postula: "*VHDL Specific Issues in High Level Synthesis*", in "VHDL for Simulation, Synthesis and Formal Proofs of Hardware" edited by J. Mermet, Kluwer Academic Publishers, Dordrecht, The Netherlands, 1992, pp. 117-134

[125] R. Raghavan, J.P. Hayes, W.R. Martin: "*Logic Simulation on Vector Processors*", IEEE International Conference on Computer-Aided Design ICCAD'88, November 7-10, 1988, pp. 268-271

[126] J. Raik, R. Ubar: "*High-Level Path Activation Technique to Speed Up Sequential Circuit Test Generation*", European Test Workshop, 1999

[127] J. Raik, R. Ubar: "*Sequential Circuit Test Generation Using Decision Diagram Models*", IEEE Conference on Design, Automation and Test in Europe DATE'99, Munich, Germany, March 9-12, 1999, pp. 736-740

[128] E. Röhm: "*Latest Benchmark Results of VHDL Simulation Systems*", Euro-DAC'95 with Euro-VHDL'95 Conference, Brighton, UK, September 18-22, 1995, pp. 406-411

[129] W.J. Schilp, P. Maurer: "*Unit Delay Simulation with the Inversion Algorithm*", IEEE/ACM International Conference on Computer-Aided Design ICCAD'96, San Jose, CA, USA, November 10-14, 1995, pp. 412-417

[130] S. Schmerler, Y. Tanurhan, K.D. Müller-Glaser: "*Advanced Optimistic Approaches in Logic Simulation*", IEEE Conference on Design, Automation and Test in Europe DATE'98, Paris, France, February 23-26, 1998, pp. 362-368

[131] S.E. Schulz: "*A Tutorial Introduction to VITAL*", Mentor Users' Group Conference, Portland, October 1995

[132] C.-J. Seger: "*An Introduction to Formal Hardware Verification*", Technical Report 92-13, Department of Computer Science, University of British Columbia, June 1992

[133] E.M. Sentovich: "*A Brief Study of BDD Package Performance*", International Conference on Formal Methods in Computer Aided Design, Lecture Notes in Computer Science, No. 1166, Springer Verlag, November 1996, pp. 389-403

[134] E.J. Shriver, K.A. Sakallah: "*Ravel: Assigned-Delay Compiled Code Logic Simulation*", IEEE/ACM International Conference on Computer-Aided Design ICCAD'92, Santa Clara, CA, USA, November 8-12, 1992, pp. 364-368

[135] A. Silburt, A. Evans, G. Vrckovnik, M. Dufresne, T. Brown: "*Functional Verification of ASICs in Silicon-Intensive Systems*", DesignCon98 Conference, Santa Clara, CA, USA, January 1998

[136] A. Silburt, I. Perryman, J. Bergeron, S. Nichols, M. Dufresne, G. Ward: "*Accelerating Concurrent Hardware Design with Behavioural Modelling and System Simulation*",

IEEE/ACM Design Automation Conference DAC'95, San Francisco, CA, USA, June 12-16, 1995, pp. 528-533

[137] P. Sinander: "*VHDL Modelling Guidelines*", European Space Agency ESA/ESTEC Report ASIC/001, September 1994, ftp://ftp.estec.esa.nl/pub/vhdl/doc/ModelGuide.ps

[138] P. Sinander, S. Habinc: "*Using VHDL for Board Level Simulation*", IEEE Design & Test of Computers, Fall 1996

[139] S.P. Smith, J. Larson: "*A High Performance VHDL Simulator With Integrated Switch and Primitive Modeling*", Computer Hardware Description Languages and Their Applications, IFIP, Elsevier Science Publishers B.V., North-Holland, 1990, pp. 299-313

[140] A. Srinivasan, T. Kam, S. Malik, R.K. Brayton: "*Algorithms for Discrete Function Manipulation*", IEEE International Conference on Computer-Aided Design ICCAD'90, Santa Clara, CA, USA, November 11-15, 1990, pp. 92-95

[141] A. Stanculescu: "*HDL-Driven Digital Simulation*", ", in Fundamentals and Standards in Hardware Description Languages, NATO ASI Series, Series E: Applied Science, edited by J. Mermet, vol. 249, Kluwer Academic Publishers, Dordrecht, The Netherlands, 1993, pp. 263-280

[142] J. Staunstrup: "*A Formal Approach to Hardware Design*", Kluwer Academic Publishers, Dordrecht, The Netherlands, ISBN 0-7923-9427-5, January 1994

[143] V. Stavridou: "*Formal Methods in Circuit Design*", Cambridge University Press, 1993

[144] N.S. Stollon, J.D. Provence: "*Measures of Syntactic Complexity for Modeling Behavioral VHDL*", IEEE/ACM Design Automation Conference DAC'95, San Francisco, CA, USA, June 12-16, 1995, pp. 684-689

[145] Synopsys: "*Cyclone VHDL Datasheet*", www.synopsys.com/products/simulation/cyclone_ds.htm, 1999

[146] S. Switzer, D. Landoll, T. Anderson: "*Functional Verification with Embedded Checkers*", International HDL Conference HDLCon 2000, San Jose, CA, USA, March 8-10, 2000, pp. 174-178

[147] S.A. Szygenda: "*Digital Systems Simulation. Digital Logic Simulation In a Time-Based, Table-Driven Environment. Part 1. Design Verification*", Computer, March 1975, pp. 23-36

[148] K. Thirunarayan, R. Ewing: "*Characterizing a Portable Subset of Behavioral VHDL-93*", IFIP TC10 WG10.5 International Conference on Computer Hardware Description Languages and their Applications, Chapman & Hall, Toledo, Spain, April 20-25, 1997, pp. 97-113

[149] D.E. Thomas, P.R. Moorby: "*The Verilog Hardware Description Language*", Second Edition, Kluwer Academic Publishers, Dordrecht, The Netherlands, 1995

[150] W. Thomas: "*Automata on Infinite Objects*", in "Handbook of Theoretical Computer Science" edited by J. van Leeuwen, vol. B "Formal Models and Semantics", Elsevier Science Publishers B.V., 1990, pp. 133-164

[151] B. Tuck: "*After Hard Knocks, Cycle-Based Simulators Stand Their Ground*", ASIC & Tool Review, Computer Design, October 1996, pp. 76-80

[152] R. Ubar: "*Multi-Valued Simulation of Digital Circuits with Structurally Synthesized Binary Decision Diagrams*", OPA (Overseas Publishers Association) N.V. Gordon and Breach Publishers, Multiple Valued Logic, vol.4, 1998, pp. 141-157

[153] R. Ubar: "*Vektorielle Alternative Graphen und Fehlerdiagnose für digitale Systeme*", Nachrichtentechnik/Elektronik, (31) 1981, H.1, pp.25-29.

[154] R. Ubar: "*Test Synthesis with Alternative Graphs*", IEEE Design and Test of Computers, Spring 1996, pp.48-59.

[155] R. Ubar, A. Morawiec, J. Raik: "*Cycle-based Simulation with Decision Diagrams*", IEEE Conference on Design, Automation and Test in Europe DATE'99, Munich, Germany, March 9-12, 1999, pp.454-458.

[156] R. Ubar, A. Morawiec, J. Raik: "*Vector Decision Diagrams for Simulation of Digital Systems*", Design and Diagnostics of Electronic Circuits and Systems Workshop DDECS2000, Smolenice, Slovakia, April 5-7, 2000, pp. 44-51

[157] R. Ubar, A. Morawiec, J. Raik: "*Back-Tracing and Event-Driven Techniques in High-Level Simulation with Decision Diagrams*", 2000 IEEE International Symposium on Circuits and Systems ISCAS'2000, Geneva, Switzerland, May 28-31, 2000

[158] R. Ubar, J. Raik, A. Morawiec: "*High-Level Decision Diagrams for Improving Simulation Performance of Digital Systems*", The 4th World Multiconference on Systemics, Cybernetics and Informatics SCI'2000, Orlando, Florida, USA, July, 23-26, 2000

[159] C.A.J. van Eijk: "*Formal Methods for the Verification of Digital Circuits*", Ph.D. thesis, Technische Universiteit Eindhoven, The Netherlands, 1997

[160] J. Varghese, M. Butts, J. Batcheller; "*An Efficient Logic Emulation System*", IEEE Transactions on VLSI Systems, vol. 1, No 2, June 1993, pp. 171-174

[161] VeriBest, Inc.: "*VeriBest Online Documentation. VB98.0*", VeriBest Incorporated, Boulder, Colorado 80301, USA, 1998

[162] E. Villar, P. Sanchez: "*Synthesis Application of VHDL*", in Fundamentals and Standards in Hardware Description Languages, NATO ASI Series, Series E: Applied Science, edited by J. Mermet, vol. 249, Kluwer Academic Publishers, Dordrecht, The Netherlands, 1993, pp. 231-262

[163] A.P. Voss, R.H. Klenke, J.H. Aylor: "*The Analysis of Modeling Styles for System Level VHDL Simulations*", VHDL International Users' Forum Conference Fall 1995, Boston, MA, October 15-18, 1995

[164] Z. Wang, P. Maurer: "*LECSIM: A Levelized Event Driven Compiled Logic Simulator*", IEEE/ACM Design Automation Conference DAC'90, Orlando, Florida, USA, June 24-28, 1990, pp. 491-496

[165] J.A. Wicks, J.R. Armstrong: "*Rating the Efficiency of VHDL Behavioral Models*", VHDL International Users' Forum Conference Fall 1996, Durham, USA, October 27-30, 1996, pp 345-351

[166] J.A. Wicks, J.R. Armstrong, R. James: "*VHDL Model Efficiency*", Asian-Pacific Conference on Hardware Description Languages APCHDL'96, Bangalore, India, January 8-10, 1996, pp. 150-154

[167] J. Willis, Z. Li, T. Lin "*Use of Embedded Scheduling to Compile VHDL for Effective Parallel Simulation*", Euro-DAC'95 with Euro-VHDL'95 Conference, Brighton, UK, September 18-22, 1995, pp. 400-405

[168] J. Willis, R. Newshutz: "*Auriga: A Compiler that Addresses NUMA Architectures*", FTL Systems, Inc, 1996

[169] J. Willis, D. Siewiorek: "*Optimizing VHDL Compilation for Parallel Simulation*", IEEE Design & Test of Computers, September 1992

[170] P.A. Wilsey: "*Developing a Formal Semantic Definition of VHDL*", in VHDL for Simulation, Synthesis and Formal Proofs of Hardware, J. Mermet (ed), Kluwer Academic Publishers, Dordrecht, The Netherlands, 1992, pp. 245-256

[171] P.A. Wilsey, D.M. Benz, and S.L. Pandey: "*A Model of VHDL for the Analysis, Transformation, and Optimization of Digital System Designs*", Conference on Hardware Description Languages CHDL'95, August 1995, pp. 611-616

[172] P.A. Wilsey, S.L. Pandey, and K. Umamageswaran: "*A Formal Model of Digital Systems Compatible with VHDL*", RASSP Digest, vol. 3, September 1996, pp. 46-48

[173] S. Woods, G. Casinovi: "*Gate-Level Simulation of Digital Circuits Using Multi-Valued Boolean Algebras*", IEEE/ACM International Conference on Computer-Aided Design ICCAD'95, San Jose, CA, USA, November 5-9, 1995, pp. 413-419

[174] J. Yuan, J. Shen, J. Abraham, A. Aziz: "*On Combining Formal and Informal Verification*", Conference on Computer-Aided Verification CAV'97, Haifa, Israel, July 1997

[175] A. Zamfirescu: "*Logic and Arithmetic in Hardware Description Languages*", in Fundamentals and Standards in Hardware Description Languages, NATO ASI Series, Series E: Applied Science, edited by J. Mermet, vol. 249, Kluwer Academic Publishers, Dordrecht, The Netherlands, 1993, pp. 79-108

Annex A

Optimization and transformation rules

A.1 Description of rules

Table A.1 below presents the description of the optimization and transformation rules: the column *Initial state* describes the initial model conditions before the application of a rule; in the column *Resulting state* a description of the model state after applying of a specific rule is provided. More information about the optimization and transformation rules, in particular a description of the rule limitations, scope and applicability can be found in [100].

Rule	Description	Initial state	Resulting state
Subprograms and packages			
1	Procedure declaration statement transfer to architecture declarative part: 1.1 sequential procedure 1.2 concurrent procedure	Procedure is declared in package declarative part and the package body	Procedure is declared in architecture declarative part
2	Procedure declaration statement transfer to process declarative part	Procedure is declared in package declarative part and package body	Procedure is declared in process declarative part
3	Function declaration statement transfer to architecture declarative part: 3.1 function call from the sequential statement 3.2 function call from the concurrent statement	Function declared in package declarative part and package body	Function declared in architecture declarative part
4	Function declaration statement transfer to process declarative part	Function declared in package declarative part and package body	Function declared in process declarative part
5	Inline expansion of sequential procedure call: 5.1 procedure declared in process 5.2 procedure declared in architecture 5.3 procedure declared in package	Sequential procedure declared in package declarative part / package body or in architecture declarative part or in process declarative part	Procedure expanded as a set of sequential statements
6	Inline expansion of sequential function call: 6.1 function declared in process 6.2 procedure declared in architecture 6.3 procedure declared in package	Function declared in package declarative part / package body or in architecture declarative part or in process declarative part	Function expanded as a set of sequential statements
7	Inline expansion of concurrent procedure call: 7.1 function declared in process 7.2 procedure declared in architecture	Procedure declared in package declarative part / package body or in architecture declarative part and called from the concurrent procedure call	Procedure introduced as an equivalent process statement in the place of the concurrent procedure call

Table A.1.1: *Optimization and transformation rules description (part 1)*

Rule	Description	Initial state	Resulting state
Types			
8	Replacement of integer-type object by a bit-type object	Object of an integer type	Object of a predefined bit type
9	Replacement of integer type object by a Boolean type object	Object of an integer type	Object of a predefined Boolean type
10	Replacement of integer type object by a character type object	Object of an integer type	Object of a predefined character type
11	Replacement of integer type object by an enumerated type object	Object of an integer type	Object of an enumerated type
12	Replacement of integer type without a range constraint by integer type with a range constraint	Object of an unconstrained integer type	Object of an constrained integer type
13	Replacement of a physical type by an integer type	Object of a physical type	Object of an integer type
14	Replacement of a predefined Time type by an integer type	Object of a predefined Time type	Object of an integer type
15	Replacement of real type by an integer type	Object of a real type	Object of an integer type
16	Replacement of constrained real type by: 16.1 integer type 16.2 constrained integer type	Object of a constrained real type	Object of an integer type or constrained integer type
17	Replacement of bit_vector type to integer type	Object of a bit_vector type	Object of an integer type
18	Replacement of std_logic type to integer type	Object of a std_logic type	Object of an integer type
19	Replacement of std_ulogic type to integer type	Object of a std_ulogic type	Object of an integer type
20	Redefinition of object type from std_logic to std_ulogic	Object of a std_logic type	Object of a std_ulogic type
21	Replacement of string type to integer type	Object of a string type	Object of an integer type
22	Replacement of a one-dimensional array by a set of variables/signals	Object declared as a one-dimensional array of a given type	Set of variables/signals of a given type
23	Replacement of multidimensional array by a set of variables/signals	Object declared as a multidimensional array of a given type	Set of variables/signals of a given type
24	Replacement of multidimensional array by a set of one-dimensional arrays	Object declared as a multidimensional array of a given type	Object declared as a one-dimensional array of a given type
25	Replacement of the array (record) positional association list by named association list	Positional association list specified	Named association list specified
26	Replacement of a record type by an array type	Object of a record type	Object is an array type
27	Replacement of the record type by a set of variables/signals	Object of a record type	Set of signals of corresponding subtypes
Sequential statements			
28	Replacement of a variable/signal declaration by a constant declaration	Object declared as a variable/signal of a given type	Object declared as a constant of a given type
29	Replacement of a source of a signal assignment statement from variable to constant	In the signal assignment statement the source is a variable of a given type	In the signal assignment statement the source is a constant of a given type
30	Replacement of an **if-then-else** statement by a **case** statement	**if-then-else** statement	**case** statement
31	Replacement of **loop**/**exit** statement by **while** statement	**Loop** statement with **exit** statement (without iteration scheme)	**Loop** statement with **while** condition iteration scheme
32	Replacement of **while** statement by **loop**/**exit** statement	**Loop** statement w ith **while** condition iteration scheme	**Loop** statement with **exit** statement (without iteration scheme)
33	Replacement of **for** statement by **while** statement	**Loop** statement with **for** iteration scheme	**Loop** statement with **while** condition iteration scheme
34	Replacement of **for** statement by **loop**/**exit** statement	**Loop** statement with **for** iteration scheme	**Loop** statement with **exit** statement (without iteration scheme)

Table A.1.2: *Optimization and transformation rules description (part 2)*

Rule	Description	Initial state	Resulting state
Concurrent statements			
35	Replacement of process statement with the **wait** statement by the process statement with the sensitivity list	Process statement with the **wait** statement	Process statement with the sensitivity list
36	Replacement of the conditional signal assignment statement by a selected signal assignment statement	Conditional signal assignment statement	Selected signal assignment statement
Language constructs			
37	Replacement of the generic parameters by the constants	Generic parameters specified for the block	Generic constants are replaced by the corresponding actuals
38	Replacement of the variable/signal of a constant value by the constant object	Variable/signal object declared, that holds a constant value	Constant object declared
39	Design hierarchy flattening	Subcomponents defined separately and instantiated in the design architecture body by component instantiation statements	Component architecture (set of concurrent statements) directly instantiated in the design architecture body
FSM modeling styles			
40	Moore machine: Style 2.5 transformation to Style 2.1	Style 2.5 description	Style 2.1 description
41	Moore machine: Style 3.1 transformation to Style 2.1	Style 3.1 description	Style 2.1 description
42	Moore machine: Style 3.2 transformation to Style 2.1	Style 3.2 description	Style 2.1 description
43	Moore machine: Style 3.4 transformation to Style 2.1	Style 3.4 description	Style 2.1 description
44	Moore machine: synchronous/asynchronous machine transformation to Style 2.1	Synchronous/asynchronous machine description	Style 2.1 description
45	Mealy machine: Style 3.2 transformation to Style 3.1	Style 3.2 description	Style 3.1 description
46	Mealy machine: Style 3.4 transformation to Style 3.1	Style 3.4 description	Style 3.1 description
47	Mealy machine: synchronous/asynchronous machine transformation to Style 3.1	Synchronous/asynchronous machine description	Style 3.1 description
Object abstraction			
48	Signal to variable replacement: local signal usage in one process / single signal assignment	Signal objects used in the assignment statement both as a target and a source	Variable objects used in the assignment statement both as a target and a source
49	Signal to variable replacement: local signal usage in one process / single signal assignment	Signal objects used in the assignment statement as a target, source is a constant	Variable objects used in the assignment statement as a target, source is a constant
50	Signal to variable replacement: local signal usage in one process / single signal assignment	Signal objects used in the assignment statement as a target, source is a variable	Variable objects used in the assignment statement as a target, source is a variable

Table A.1.3: *Optimization and transformation rules description (part 3)*

A.2 Simulation results

The following tables A.2 and A.3 present detailed results of the simulation time measurement of the test models devoted to evaluate the simulation performance improvement of each rule. Table A.2 shows these results for the variable objects and table A.3 for signal objects. Both tables are divided in two main parts: the first one contains results for large models (1E07 statements) and shorter simulation time (1s), while the second one for small model (10 statements of a given type) and longer simulation time (1000s). Moreover, the simulation experiments have been carried out on two simulation tools: on Veribest VB VHDL SysSim 98.0, VeriBest, Inc (indicated as *Simulator 1*) and on ModelSim PE/Plus 4.7b, Model Technology, Inc (indicated as *Simulator 2*). The column *Test* presents the simulation time of the test model containing the initial statement before application of the acceleration method, the column *Ref* the simulation time of the model after application of a given acceleration method. *Gain* is a ratio between *Test* and *Ref*.

Model acceleration rules: variables												
Rule	**Simulation time: 1E06ns Statements: 1E07**						**Simulation time: 1E09ns Statements: 10**					
	Simulator 1			**Simulator 2**			**Simulator 1**			**Simulator 2**		
	Test	**Ref**	**Gain**	**Test**	**Ref**	**Gain**	**Test**	**Ref**	**Gain**	**Test**	**Ref**	**Gain**
1.1	10992.7	12873.8	**0.854**	7325.5	7405.6	**0.989**	2814.4	2782.8	**1.011**	955.0	979.1	**0.975**
2	10724.3	12873.8	**0.833**	7324.6	7405.6	**0.989**	2825.5	2782.8	**1.015**	955.1	979.1	**0.976**
3.1	8667.6	7610.0	**1.139**	5830.3	5819.0	**1.002**	2279.9	2242.2	**1.017**	754.8	769.5	**0.981**
4	8505.0	7610.0	**1.118**	5824.1	5819.0	**1.001**	2282.8	2242.2	**1.018**	754.8	769.5	**0.981**
5.1	10724.3	916.6	**11.700**	7324.6	524.1	**13.975**	2825.5	1992.0	**1.418**	955.1	510.8	**1.870**
5.2	10992.7	916.6	**11.992**	7325.5	524.1	**13.977**	2814.4	1992.0	**1.413**	955.0	510.8	**1.870**
5.3	12873.8	916.6	**14.045**	7405.6	524.1	**14.130**	2782.8	1992.0	**1.397**	979.1	510.8	**1.917**
6.1	53.2	27.9	**1.910**	31.9	7.8	**4.064**	2282.8	2062.4	**1.107**	754.8	502.0	**1.504**
6.2	53.4	27.9	**1.917**	31.8	7.8	**4.060**	2279.9	2062.4	**1.105**	754.8	502.0	**1.503**
6.3	47.9	27.9	**1.721**	32.3	7.8	**4.115**	2242.2	2062.4	**1.087**	769.5	502.0	**1.533**
8	294.0	378.0	**0.778**	8347.9	11097.3	**0.752**	2299.1	2330.9	**0.986**	496.7	508.6	**0.976**
9	294.8	378.0	**0.780**	8311.5	11097.3	**0.749**	2285.6	2330.9	**0.981**	496.7	508.6	**0.977**
10	295.2	378.0	**0.781**	8323.8	11097.3	**0.750**	2285.3	2330.9	**0.980**	496.7	508.6	**0.976**
11	297.9	381.8	**0.780**	14029.0	16390.3	**0.856**	2301.7	2322.7	**0.991**	504.1	501.2	**1.006**
12	380.5	381.8	**0.997**	16291.6	16390.3	**0.994**	2282.6	2322.7	**0.983**	501.6	501.2	**1.001**
13	4322.7	381.8	**11.321**	2329.7	261.7	**8.902**	2323.2	2322.7	**1.000**	521.4	501.2	**1.040**
14	4116.6	381.8	**10.782**	2393.2	261.7	**9.145**	2337.9	2322.7	**1.007**	514.9	501.2	**1.027**
15	4322.7	381.8	**11.321**	2329.7	261.7	**8.902**	2323.2	2322.7	**1.000**	521.4	501.2	**1.040**
16.1	4347.7	381.8	**11.387**	1930.5	261.7	**7.377**	2330.6	2322.7	**1.003**	522.8	501.2	**1.043**
16.2	4347.7	380.5	**11.425**	1930.5	263.2	**7.333**	2330.6	2282.6	**1.021**	522.8	501.6	**1.042**
17	12146.1	378.0	**32.132**	11291.5	174.3	**64.778**	2555.6	2330.9	**1.096**	876.7	508.6	**1.724**
18	12342.1	378.0	**32.651**	11225.6	174.3	**64.400**	2544.2	2330.9	**1.092**	925.7	508.6	**1.820**
19	12209.7	378.0	**32.300**	11197.3	174.3	**64.238**	2554.1	2330.9	**1.096**	919.8	508.6	**1.808**
20	12342.1	12209.7	**1.011**	11225.6	11197.3	**1.003**	2544.2	2554.1	**0.996**	925.7	919.8	**1.006**
21	12826.8	378.0	**33.933**	11278.8	174.3	**64.705**	2552.7	2330.9	**1.095**	876.9	508.6	**1.724**
22	594.6	381.8	**1.557**	838.9	261.7	**3.205**	2060.9	2322.7	**0.887**	493.7	501.2	**0.985**
23	22109.7	54.4	**406.742**	4633.3	6.8	**677.475**	6155.4	2322.7	**2.650**	21199.9	501.2	**42.302**
24	22109.7	24103.3	**0.917**	4633.3	603.5	**7.677**	6155.4	6335.5	**0.972**	21199.9	3828.4	**5.538**
25	20355.0	20374.0	**0.999**	8697.7	8893.1	**0.978**	5995.9	6006.1	**0.998**	28147.9	55383.0	**0.508**
26	24437.6	24336.5	**1.004**	1287.4	838.9	**1.535**	2043.6	2060.9	**0.992**	490.6	493.7	**0.994**
27	560.2	381.8	**1.467**	1287.4	261.7	**4.919**	2043.6	2322.7	**0.880**	490.6	501.2	**0.979**
28	21121.9	12142.5	**1.739**	11602.2	8574.5	**1.353**	1942.7	1938.7	**1.002**	502.2	499.5	**1.005**
30	1900.0	1271.6	**1.494**	1296.8	181.6	**7.139**	1954.0	1954.6	**1.000**	521.8	516.1	**1.011**
31	3842.5	3536.7	**1.086**	6095.3	6592.8	**0.925**	2121.1	2128.7	**0.996**	559.6	571.2	**0.980**
32	3536.7	3842.5	**0.920**	6592.8	6095.3	**1.082**	2128.7	2121.1	**1.004**	571.2	559.6	**1.021**
33	5432.1	3536.7	**1.536**	7096.9	6592.8	**1.076**	2139.3	2128.7	**1.005**	566.3	571.2	**0.991**
34	5432.1	3842.5	**1.414**	7096.9	6095.3	**1.164**	2139.3	2121.1	**1.009**	566.3	559.6	**1.012**
35	36411.5	15155.2	**2.403**	1435.3	719.1	**1.996**	2943.5	2238.3	**1.315**	1119.1	1017.2	**1.100**
37	25290.1	20223.5	**1.251**	14152.0	16294.5	**0.869**	1964.0	1940.1	**1.012**	505.6	501.1	**1.009**
38	21121.9	12142.5	**1.739**	11602.2	8574.5	**1.353**	1942.7	1938.7	**1.002**	502.2	499.5	**1.005**

Table A.2: *Model acceleration results: variables*

Model acceleration rules: signals												
Rule	Simulation time: 1E06ns Statements: 1E07						Simulation time: 1E09ns Statements: 10					
	Simulator 1			Simulator 2			Simulator 1			Simulator 2		
	Test	Ref	Gain	Test	Ref	Gain	Test	Ref	Gain	Test	Ref	Gain
1.1	2586.0	2657.6	**0.973**	2688.7	2733.1	**0.984**	4527.7	4620.3	**0.980**	2399.5	2408.0	**0.996**
1.2	5982.5	7130.9	**0.839**	5521.0	6954.9	**0.794**	5825.0	5973.4	**0.975**	2813.3	3410.9	**0.825**
2	2594.6	2657.6	**0.976**	2680.7	2733.1	**0.981**	4536.4	4620.3	**0.982**	2399.3	2408.0	**0.996**
3.1	1413.3	1388.1	**1.018**	2831.3	2790.0	**1.015**	3353.8	3312.0	**1.013**	2375.1	2387.1	**0.995**
3.2	1900.9	1817.7	**1.046**	5479.2	5445.7	**1.006**	3599.4	3573.1	**1.007**	2841.1	2859.7	**0.994**
4	1391.7	1388.1	**1.003**	2812.5	2790.0	**1.008**	3332.3	3312.0	**1.006**	2375.0	2387.1	**0.995**
5.1	2594.6	1132.7	**2.291**	2680.7	2354.2	**1.139**	4536.4	3070.2	**1.478**	2399.3	2141.2	**1.121**
5.2	2586.0	1132.7	**2.283**	2688.7	2354.2	**1.142**	4527.7	3070.2	**1.475**	2399.5	2141.2	**1.121**
5.3	2657.6	1132.7	**2.346**	2733.1	2354.2	**1.161**	4620.3	3070.2	**1.505**	2408.0	2141.2	**1.125**
6.1	151.7	125.7	**1.206**	211.7	193.2	**1.096**	3332.3	3117.8	**1.069**	2375.0	2143.5	**1.108**
6.2	152.3	125.7	**1.211**	211.8	193.2	**1.096**	3353.8	3117.8	**1.076**	2375.1	2143.5	**1.108**
6.3	149.2	125.7	**1.186**	210.9	193.2	**1.092**	3312.0	3117.8	**1.062**	2387.1	2143.5	**1.114**
7.1	5982.5	1581.3	**3.783**	5521.0	5011.9	**1.102**	5825.0	3269.6	**1.782**	2813.3	2631.7	**1.069**
7.2	7130.9	1581.3	**4.509**	6954.9	5011.9	**1.388**	5973.4	3269.6	**1.827**	3410.9	2631.7	**1.296**
8	998.3	1105.2	**0.903**	229.6	252.4	**0.910**	3261.5	3367.8	**0.968**	631.8	660.6	**0.956**
9	996.1	1105.2	**0.901**	227.9	252.4	**0.903**	3263.0	3367.8	**0.969**	631.7	660.6	**0.956**
10	996.2	1105.2	**0.901**	232.7	252.4	**0.922**	3259.2	3367.8	**0.968**	631.7	660.6	**0.956**
11	1003.5	1111.5	**0.903**	228.2	269.4	**0.847**	3170.4	3359.0	**0.944**	631.7	660.6	**0.956**
12	1102.6	1111.5	**0.992**	317.7	269.4	**1.179**	3380.5	3359.0	**1.006**	680.0	660.6	**1.029**
13	1187.6	1111.5	**1.069**	341.7	269.4	**1.268**	3455.9	3359.0	**1.029**	699.8	660.6	**1.059**
14	1215.5	1111.5	**1.094**	277.0	269.4	**1.028**	3461.6	3359.0	**1.031**	697.8	660.6	**1.056**
15	1187.6	1111.5	**1.069**	341.7	269.4	**1.268**	3455.9	3359.0	**1.029**	699.8	660.6	**1.059**
16.1	1190.2	1111.5	**1.071**	282.3	269.4	**1.048**	3458.7	3359.0	**1.030**	710.7	660.6	**1.076**
16.2	1190.2	1102.6	**1.079**	282.3	317.7	**0.889**	3458.7	3380.5	**1.023**	710.7	680.0	**1.045**
17	13619.9	1105.2	**12.324**	1247.2	252.4	**4.941**	9289.3	3367.8	**2.758**	1506.3	660.6	**2.280**
18	13641.5	1105.2	**12.343**	1239.0	252.4	**4.909**	9730.2	3367.8	**2.889**	1552.3	660.6	**2.350**
19	13675.1	1105.2	**12.373**	1232.0	252.4	**4.881**	9719.7	3367.8	**2.886**	1552.4	660.6	**2.350**
20	13641.5	13675.1	**0.998**	1239.0	1232.0	**1.006**	9730.2	9719.7	**1.001**	1552.3	1552.4	**1.000**
21	13749.6	1105.2	**12.441**	1233.1	252.4	**4.885**	9279.8	3367.8	**2.755**	1506.3	660.6	**2.280**
22	2620.4	1111.5	**2.358**	11157.2	10714.7	**1.041**	2062.8	3359.0	**0.614**	493.8	660.6	**0.748**
23	54912.3	120.1	**457.362**	12924.0	27.4	**471.179**	25260.5	336.7	**75.016**	4571.1	65.0	**70.363**
24	54912.3	58690.8	**0.936**	12924.0	10787.4	**1.198**	25260.5	26403.2	**0.957**	4571.1	2545.4	**1.796**
25	4584.7	4511.3	**1.016**	9159.4	9156.6	**1.000**	3305.5	3300.4	**1.002**	2197.4	2158.9	**1.018**
26	2627.6	2620.4	**1.003**	41157.6	11157.2	**3.689**	2035.0	2062.8	**0.987**	490.5	493.8	**0.993**
27	2627.6	1111.5	**2.364**	41157.6	10714.7	**3.841**	2035.0	3359.0	**0.606**	490.5	660.6	**0.742**
28	8013.0	5694.1	**1.407**	2316.0	2132.1	**1.086**	2470.2	2453.8	**1.007**	805.0	846.4	**0.951**
29	6907.4	5694.1	**1.213**	2327.0	2132.1	**1.091**	2473.8	2453.8	**1.008**	843.7	846.4	**0.997**
30	3353.3	3068.3	**1.093**	2030.2	201.6	**10.069**	2240.2	2233.9	**1.003**	710.7	710.2	**1.001**
31	3019.9	2971.1	**1.016**	23049.8	24181.5	**0.953**	2615.1	2649.2	**0.987**	842.0	885.8	**0.951**
32	2971.1	3019.9	**0.984**	24181.5	23049.8	**1.049**	2649.2	2615.1	**1.013**	885.8	842.0	**1.052**
33	3173.4	2971.1	**1.068**	25508.9	24181.5	**1.055**	2667.4	2649.2	**1.007**	898.9	885.8	**1.015**
34	3173.4	3019.9	**1.051**	25508.9	23049.8	**1.107**	2667.4	2615.1	**1.020**	898.9	842.0	**1.067**
35	3817.5	1493.1	**2.557**	5891.3	5059.8	**1.164**	4107.3	3271.4	**1.256**	2762.1	2622.0	**1.053**
36	2985.7	2731.1	**1.093**	1556.6	194.6	**8.000**	2135.8	2116.5	**1.009**	713.8	710.9	**1.004**
37	29023.2	28018.8	**1.036**	10819.0	10831.2	**0.999**	3031.9	3020.3	**1.004**	646.5	660.6	**0.979**
38	8013.0	5694.1	**1.407**	2316.0	2132.1	**1.086**	2470.2	2453.8	**1.007**	805.0	846.4	**0.951**
39.1	454.2	367.4	**1.236**	7170.4	5805.5	**1.235**	4938.3	4993.8	**0.989**	3083.8	3091.7	**0.997**
39.2	470.4	365.8	**1.286**	7014.8	5723.2	**1.226**	4963.7	5011.9	**0.990**	3085.7	3092.2	**0.998**
40	4879.7	2389.3	**2.042**	358.4	195.3	**1.835**	2456.7	2221.0	**1.106**	812.5	663.7	**1.224**
41	4910.3	2389.3	**2.055**	368.6	195.3	**1.887**	2351.7	2221.0	**1.059**	727.2	663.7	**1.096**
42	3023.4	2389.3	**1.265**	226.5	195.3	**1.159**	2443.7	2221.0	**1.100**	785.3	663.7	**1.183**
43	5654.9	2389.3	**2.367**	412.9	195.3	**2.114**	2539.4	2221.0	**1.143**	859.3	663.7	**1.295**
44	7886.7	2389.3	**3.301**	839.6	195.3	**4.298**	2605.6	2221.0	**1.173**	945.7	663.7	**1.425**
45	3023.4	4910.3	**0.616**	226.5	368.6	**0.614**	2443.7	2351.7	**1.039**	785.3	727.2	**1.080**
46	5654.9	4910.3	**1.152**	412.9	368.6	**1.120**	2539.4	2351.7	**1.080**	859.3	727.2	**1.182**
47	7886.7	4910.3	**1.606**	839.6	368.6	**2.278**	2605.6	2351.7	**1.108**	945.7	727.2	**1.300**
48	8013.0	395.4	**20.267**	2316.0	177.4	**13.058**	2470.2	1942.7	**1.271**	805.0	502.2	**1.603**
49	5694.1	288.5	**19.735**	2132.1	154.6	**13.794**	2453.8	1938.7	**1.266**	846.4	499.5	**1.695**
50	6907.4	395.4	**17.470**	2327.0	177.4	**13.120**	2473.8	1942.7	**1.273**	843.7	502.2	**1.680**

Table A.3: *Model acceleration results: signals*

Annex B

Finite state machine modeling

B.1 GCD example description

The Greatest Common Divider *GCD* is a circuit realizing the following function (e.g. [16]):

$$pgcd(x, y) = \begin{cases} pgcd(x-y, y) & \text{if } x > y \\ pgcd(x, y-x) & \text{if } x < y \\ x = y & \text{if } x = y \end{cases}$$

The circuit has four inputs: *in1*, *in2*, *start* and *clock*, and two outputs: *pgcd* and *ready*. The finite state machine of the *GCD* circuit is presented in figure B.1.

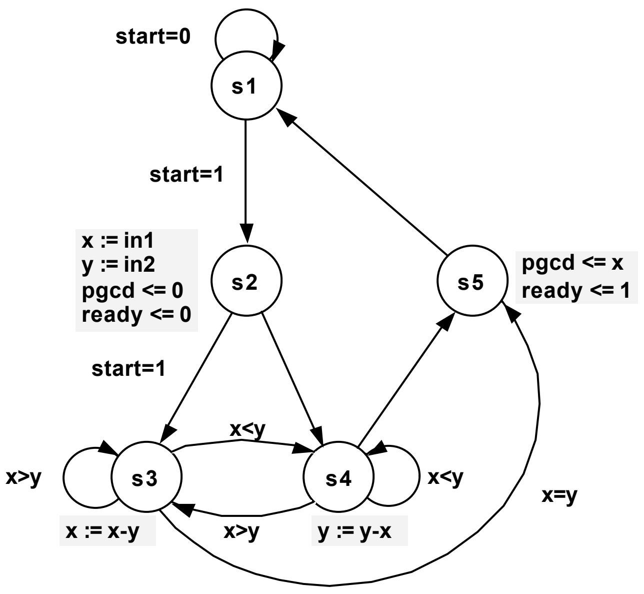


Figure B.1: *GCD finite state machine*

In the following two sections the algorithmic and register-transfer level finite state machine models are presented. Those models illustrate different FSM modeling styles, as introduced in section 2.3.

B.2 Algorithmic level model

B.2.1 Description of the modeling styles

The entity declaration for the algorithmic level model is the following:

```
entity gcd is
    port (  clock   : in    bit;
            start   : in    bit;
            in1     : in    integer;
            in2     : in    integer;
            pgcd    : out   integer;
            ready   : out   bit);
end gcd;
```

The modeling styles 1.1-1.11 are described in detail in section 2.3.3.2. Below the description of the *GCD* example in each of those styles is presented.

Style 1.1

```
architecture style_1_1 of gcd is
    type    states is (s1, s2, s3, s4, s5);
begin
SM: process (clock)
    variable x, y : integer;
    variable state    : states := s1;
begin
    if clock'event and clock='1' then
        case state is
        when s1 =>
            if start='1' then   state := s2;
                                x := in1; y := in2;
                                pgcd <= 0; ready <= '0'; end if;
        when s2 =>
            if      x>y then    state := s3;
                                x := x-y;
            elsif   x<y then    state := s4;
                                y := y-x;
            else                state := s5;
                                pgcd <= x; ready <= '1';
            end if;
        when s3 =>
            if      x>y then    x := x-y;
            elsif   x<y then    state := s4;
                                y := y-x;
            else                state := s5;
                                pgcd <= x; ready <= '1';
            end if;
        when s4 =>
            if      x>y then    state := s3;
                                x := x-y;
            elsif   x<y then    y := y-x;
            else                state := s5;
                                pgcd <= x; ready <= '1';
            end if;
        when s5 =>  state := s1;
        end case;
    end if;
end process;
end;
```

Style 1.2

```
architecture style_1_2 of gcd is
    type        states  is (s1, s2, s3, s4, s5);
    subtype     atype   is integer range 0 to 2;
    type        fsm   is array(states, bit, atype) of states;
    constant    fsms : fsm := (
                ((s1, s1, s1), (s2, s2, s2)), --s1
                ((s5, s4, s3), (s5, s4, s3)), --s2
                ((s5, s4, s3), (s5, s4, s3)), --s3
                ((s5, s4, s3), (s5, s4, s3)), --s4
                ((s1, s1, s1), (s1, s1, s1)));--s5
begin
SM: process (clock)
    variable x, y       : integer;
    variable a          : atype;
    variable state      : states := s1;
begin
if clock'event and clock='1' then
    if      x>y then    a := 2;
    elsif   x<y then    a := 1;
    else                a := 0; end if;
    state := fsms(state, start, a);
    case state is
        when s1 =>  null;
        when s2 =>  x := in1; y := in2;
                    pgcd <= 0; ready <= '0';
        when s3 =>  x := x-y;
        when s4 =>  y := y-x;
        when s5 =>  pgcd <= x; ready <= '1';
    end case;
end if;
end process;
end;
```

Style 1.3

```
architecture style_1_3 of gcd is
    type        states   is (s1, s2, s3, s4, s5);
    subtype     atype    is integer range 0 to 2;
    type        fsm      is array(states, bit, atype) of states;
    constant    fsms : fsm := (
                ((s1, s1, s1), (s2, s2, s2)), --s1
                ((s5, s4, s3), (s5, s4, s3)), --s2
                ((s5, s4, s3), (s5, s4, s3)), --s3
                ((s5, s4, s3), (s5, s4, s3)), --s4
                ((s1, s1, s1), (s1, s1, s1)));--s5
    type        at   is array(boolean, boolean) of atype;
    constant    atable : at := (
                ((0), (1)),--0
                ((2), (2)));    --1
begin
SM: process (clock)
    variable x, y       : integer;
    variable a          : atype;
    variable state      : states := s1;
begin
if clock'event and clock='1' then
    a := atable(x>y, x<y);
    state := fsms(state, start, a);
    case state is
        when s1 =>  null;
        when s2 =>  x := in1; y := in2;
                    pgcd <= 0; ready <= '0';
        when s3 =>  x := x-y;
        when s4 =>  y := y-x;
        when s5 =>  pgcd <= x; ready <= '1';
    end case;
end if;
end process;
end;
```

Style 1.4

```
architecture style_1_4 of gcd is
    type    states is (s1, s2, s3, s4, s5);
    signal      x, y : integer;
    signal      state, next_state : states := s1;
begin
OUTP: process (clock)
begin
    if clock'event and clock='1' then
        state <= next_state;
        case state is
        when s1 =>
            if start='1' then   x <= in1; y <= in2;
                                pgcd <= 0; ready <= '0';
            end if;
        when s2 =>
            if      x>y then    x <= x-y;
            elsif   x<y then    y <= y-x;
            else                pgcd <= x; ready <= '1';
            end if;
        when s3 =>
            if      x>y then    x <= x-y;
            elsif   x<y then    y <= y-x;
            else                pgcd <= x; ready <= '1';
            end if;
        when s4 =>
            if      x>y then    x <= x-y;
            elsif   x<y then    y <= y-x;
            else                pgcd <= x; ready <= '1';
            end if;
        when s5 =>
                ready <= '0';
        end case;
    end if;
end process;
SM: process (state, start, x, y)
begin
    next_state <= state;
    case state is
        when s1 =>
            if start='1' then   next_state <= s2; end if;
        when s2 =>
            if      x>y then    next_state <= s3;
            elsif   x<y then    next_state <= s4;
            else                next_state <= s5;
            end if;
        when s3 =>
            if      x>y then    null;
            elsif   x<y then    next_state <= s4;
            else                next_state <= s5;
            end if;
        when s4 =>
            if      x>y then    next_state <= s3;
            elsif   x<y then    null;
            else                next_state <= s5;
            end if;
        when s5 =>      next_state <= s1;
    end case;
end process;
end;
```

Style 1.5

```
architecture style_1_5 of gcd is
    type    states is (s1, s2, s3, s4, s5);
begin
SM: process (clock)
    variable x, y       : integer;
    variable state      : states := s1;
begin
    if clock'event and clock='1' then
        case state is
            when s1 =>
                if start='1'    then    state := s2; end if;
            when s2 =>
                if      x>y then        state := s3;
                elsif   x<y then        state := s4;
                else                    state := s5; end if;
            when s3 =>
                if      x>y then        null;
                elsif   x<y then        state := s4;
                else                    state := s5; end if;
            when s4 =>
                if      x>y then        state := s3;
                elsif   x<y then        null;
                else                    state := s5; end if;
            when s5 =>  state := s1;
        end case;
        case state is
            when s1 =>  null;
            when s2 =>  x := in1; y := in2;
                        pgcd <= 0; ready <= '0';
            when s3 =>  x := x-y;
            when s4 =>  y := y-x;
            when s5 =>  pgcd <= x; ready <= '1';
        end case;
    end if;
end process;
end;
```

Style 1.6

```
architecture test1_str of test1 is
    type        states   is (s1, s2, s3, s4, s5);
    subtype     atype    is integer range 0 to 2;
    type        fsm is array(states, bit, atype) of states;
    constant    fsms : fsm := (
                ((s1, s1, s1), (s2, s2, s2)), --s1
                ((s5, s4, s3), (s5, s4, s3)), --s2
                ((s5, s4, s3), (s5, s4, s3)), --s3
                ((s5, s4, s3), (s5, s4, s3)), --s4
                ((s1, s1, s1), (s1, s1, s1)));--s5
    type at is array(boolean, boolean, boolean) of atype;
    constant    atable : at := (
                ((0,0), (1,1)), --0
                ((2,2), (2,2)));--1
begin
SM: process (clock)
    variable x, y       : integer;
    variable a          : atype;
    variable state      : states := s1;
begin
    if clock'event and clock='1' then
        a := atable(x>y, x<y, x=y);
        state := fsms(state, start, a);
        case state is
            when s1 =>  null;
            when s2 =>  x := in1; y := in2;
                        pgcd <= 0; ready <= '0';
            when s3 =>  x := x-y;
            when s4 =>  y := y-x;
            when s5 =>  pgcd <= x; ready <= '1';
        end case;
    end if;
end process;
end;
```

Style 1.7

```
architecture style_1_7 of gcd is
    type        states   is (s1, s2, s3, s4, s5);
    subtype     atype    is integer range 0 to 2;
    type        fsm is array (states, bit, boolean,
        boolean) of states;
    constant    fsms : fsm := (
                (((s1, s1), (s1, s1)), ((s2, s2), (s2, s2))), --s1
                (((s5, s4), (s3, s3)), ((s5, s4), (s3, s3))), --s2
                (((s5, s4), (s3, s3)), ((s5, s4), (s3, s3))), --s3
                (((s5, s4), (s3, s3)), ((s5, s4), (s3, s3))), --s4
                (((s1, s1), (s1, s1)), ((s1, s1), (s1, s1))));--s5
begin
SM: process (clock)
    variable x, y       : integer;
    variable a          : atype;
    variable state      : states := s1;
begin
    if clock'event and clock='1' then
        state := fsms(state, start, x>y, x<y);
        case state is
            when s1 =>  null;
            when s2 =>  x := in1; y := in2;
                        pgcd <= 0; ready <= '0';
            when s3 =>  x := x-y;
            when s4 =>  y := y-x;
            when s5 =>  pgcd <= x; ready <= '1';
        end case;
    end if;
end process;
end;
```

Style 1.8

```
architecture style_1_8 of gcd is
    subtype     states   is integer range 1 to 5;
    subtype          atype    is integer range 0 to 2;
    type        fsm is array(states, bit, atype) of states;
    constant    fsms : fsm := (
                ((1, 1, 1), (2, 2, 2)),    --s1
                ((5, 4, 3), (5, 4, 3)),    --s2
                ((5, 4, 3), (5, 4, 3)),    --s3
                ((5, 4, 3), (5, 4, 3)),    --s4
                ((1, 1, 1), (1, 1, 1)));   --s
begin
SM: process (clock)
    variable x, y       : integer;
    variable a          : atype;
    variable state      : states := 1;
begin
    if clock'event and clock='1' then
        if      x>y then      a := 2;
        elsif   x<y then      a := 1;
        else                  a := 0; end if;
        state := fsms(state, start, a);
        case state is
            when 1 =>   null;
            when 2 =>   x := in1; y := in2;
                        pgcd <= 0; ready <= '0';
            when 3 =>   x := x-y;
            when 4 =>   y := y-x;
            when 5 =>   pgcd <= x; ready <= '1';
        end case;
    end if;
end process;
end;
```

Style 1.9

```
architecture style_1_9 of gcd is
    type        states is (s1, s2, s3, s4, s5);
    signal      x, y : integer;
    signal      state, next_state : states := s1;
    subtype     atype    is integer range 0 to 2;
    type        fsm is array(states, bit, atype) of states;
    constant    fsms : fsm := (
                ((s1, s1, s1), (s2, s2, s2)), --s1
                ((s5, s4, s3), (s5, s4, s3)), --s2
                ((s5, s4, s3), (s5, s4, s3)), --s3
                ((s5, s4, s3), (s5, s4, s3)), --s4
                ((s1, s1, s1), (s1, s1, s1)));--s5
begin
OUTP: process (clock)
begin
    if clock'event and clock='1' then
        state <= next_state;
        case state is
            when s1 =>
                if start='1' then   x <= in1; y <= in2;
                                    pgcd <= 0; ready <= '0';
                end if;
            when s2 =>
                if      x>y then    x <= x-y;
                elsif   x<y then    y <= y-x;
                else                pgcd <= x; ready <= '1';
                end if;
            when s3 =>
                if      x>y then    x <= x-y;
                elsif   x<y then    y <= y-x;
                else                pgcd <= x; ready <= '1';
                end if;
            when s4 =>
                if      x>y then    x <= x-y;
                elsif   x<y then    y <= y-x;
                else                pgcd <= x; ready <= '1';
                end if;
            when s5 =>  ready <= '0';
        end case;
    end if;
end process;

SM: process (state, start, x, y)
    variable a    : atype;
begin
    next_state <= state;
    if      x>y then    a := 2;
    elsif   x<y then    a := 1;
    else                a := 0; end if;
    next_state <= fsms(state, start, a);
end process;
end;
```

Style 1.10

```
architecture style_1_10 of gcd is
    type      states is (s1, s2, s3, s4, s5);
    signal    x, y : integer;
    signal    state, next_state : states := s1;
    subtype   atype    is integer range 0 to 2;
    type      fsm is array(states, bit, atype) of states;
    constant  fsms : fsm := (
              ((s1, s1, s1), (s2, s2, s2)), --s1
              ((s5, s4, s3), (s5, s4, s3)), --s2
              ((s5, s4, s3), (s5, s4, s3)), --s3
              ((s5, s4, s3), (s5, s4, s3)), --s4
              ((s1, s1, s1), (s1, s1, s1)));--s5
    type      at   is array(boolean, boolean,
              boolean) of atype;
    constant  atable : at := (
              ((0,0), (1,1)), --0
              ((2,2), (2,2)));--1
begin
OUTP: process (clock)
begin
    if clock'event and clock='1' then
        state <= next_state;
        case state is
            when s1 =>
                if start='1' then  x <= in1; y <= in2;
                                   pgcd <= 0; ready <= '0';
                end if;
            when s2 =>
                if      x>y then    x <= x-y;
                elsif   x<y then    y <= y-x;
                else                pgcd <= x; ready <= '1';
                end if;
            when s3 =>
                if      x>y then    x <= x-y;
                elsif   x<y then    y <= y-x;
                else                pgcd <= x; ready <= '1';
                end if;
            when s4 =>
                if      x>y then    x <= x-y;
                elsif   x<y then    y <= y-x;
                else                pgcd <= x; ready <= '1';
                end if;
            when s5 =>  ready <= '0';
        end case;
    end if;
end process;
SM: process (state, start, x, y)
    variable  a    : atype;
begin
    next_state <= state;
    a := atable(x>y, x<y, x=y);
    next_state <= fsms(state, start, a);
end process;
end;
```

Style 1.11

```
architecture style_1_11 of gcd is
    subtype   states   is integer range 1 to 5;
    signal    x, y : integer;
    signal    state, next_state : states := 1;
    subtype   atype    is integer range 0 to 2;
    type      fsm is array(states, bit, atype) of states;
    constant  fsms : fsm := (
              ((1, 1, 1), (2, 2, 2)),    --s1
              ((5, 4, 3), (5, 4, 3)),    --s2
              ((5, 4, 3), (5, 4, 3)),    --s3
              ((5, 4, 3), (5, 4, 3)),    --s4
              ((1, 1, 1), (1, 1, 1)));   --s
begin
OUTP: process (clock)
begin
    if clock'event and clock='1' then
        state <= next_state;
        case state is
            when 1 =>
                if start='1' then  x <= in1; y <= in2;
                                   pgcd <= 0; ready <= '0';
                end if;
            when 2 =>
                if      x>y then    x <= x-y;
                elsif   x<y then    y <= y-x;
                else                pgcd <= x; ready <= '1';
                end if;
            when 3 =>
                if      x>y then    x <= x-y;
                elsif   x<y then    y <= y-x;
                else                pgcd <= x; ready <= '1';
                end if;
            when 4 =>
                if      x>y then    x <= x-y;
                elsif   x<y then    y <= y-x;
                else                pgcd <= x; ready <= '1';
                end if;
            when 5 =>   ready <= '0';
        end case;
    end if;
end process;
SM: process (state, start, x, y)
    variable a    : atype;
begin
    next_state <= state;
    if      x>y then    a := 2;
    elsif   x<y then    a := 1;
    else                a := 0;
    end if;
    next_state <= fsms(state, start, a);
end process;
end;
```

B.2.2 Simulation time comparison

AL-FSM Modeling Style				Simulation time			
Style	**# of processes**	**# of tables**	**# of choice statements**	**Simulator 1**		**Simulator 2**	
				Sim:1E09ns	**Gain**	**Sim:1E09ns**	**Gain**
1.1	1 (clk+δ+λ)	0	1	2 093 840	1.05	**320 361**	**1.00**
1.2	1 (clk+δ+λ)	1 (state)	2	2 090 697	1.04	338 878	1.06
1.3	1 (clk+δ+λ)	2 (state, in)	1	**2 001 598**	**1.00**	351 605	1.10
1.4	2 (clk+λ, δ)	0	2	2 341 828	1.17	469 525	1.47
1.5	1 (clk+δ+λ)	0	1	2 108 883	1.05	321 353	1.00
1.6	1 (clk+δ+λ)	2 (state, in)	1	2 005 614	1.00	360 198	1.12
1.7	1 (clk+δ+λ)	1 (state+in)	1	2 003 531	1.00	363 843	1.14
1.8	1 (clk+δ+λ)	1 (state)	2	2 074 373	1.04	339 889	1.06
1.9	2 (clk+λ, δ)	1 (state)	2	2 384 840	1.19	521 390	1.63
1.10	2 (clk+λ, δ)	2 (state, in)	1	2 343 189	1.17	522 251	1.63
1.11	2 (clk+λ, δ)	1 (state)	2	2 342 388	1.17	518 676	1.62

Table B.1: *Comparison of AL-FSM modeling styles*

B.3 Register-transfer level model

B.3.1 Design description

The data path of the GCD circuit is depicted in figure B.2.

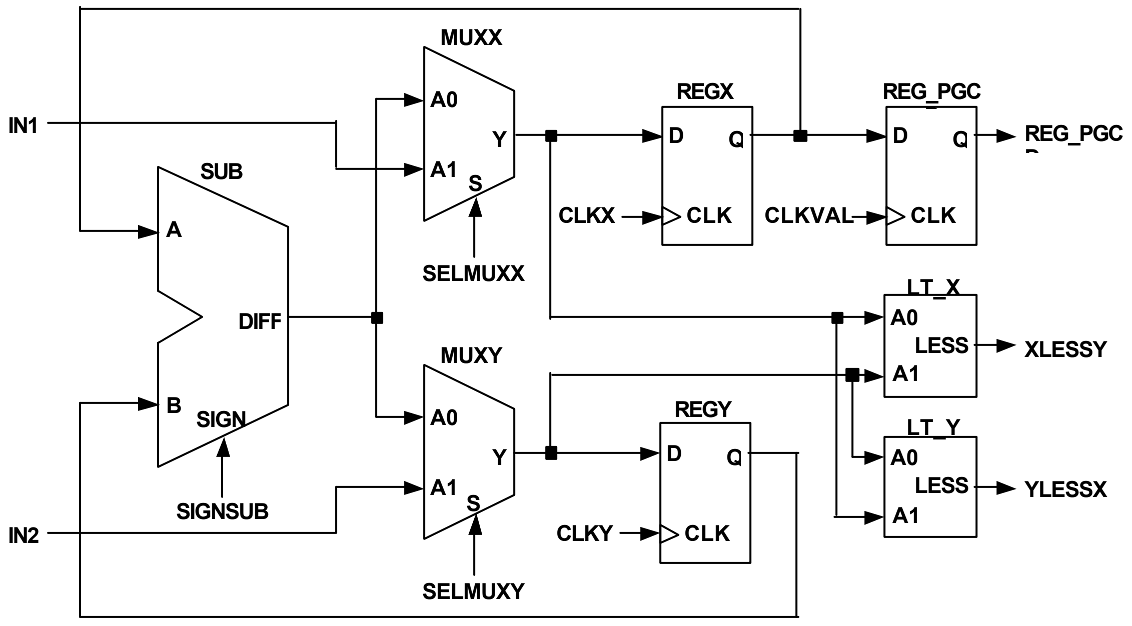


Figure B.2: *GCD data path*

The entire structural VHDL model of the RT-level GCD example is presented below. Then the description of the control part (entity *CONTROL*) in various modeling styles for both: Moor and Mealy machines is presented.

GCD model

```
library IEEE;
    use IEEE.std_logic_1164.all;
    use IEEE.std_logic_arith.all;
entity SUB is
port (  A, B    : in    std_logic_vector(0 to 7);
        Sign    : in    std_logic;
        DIFF    : out   std_logic_vector(0 to 7));
end SUB;
architecture DES_SUB of SUB is
begin
    DIFF <= (A-B) when Sign='0' else (B-A);
end DES_SUB;
---------------------------------------------------------------------
-- LT
---------------------------------------------------------------------
library IEEE;
    use IEEE.std_logic_1164.all;
    use IEEE.std_logic_arith.all;
entity LT is
port (  A0, A1  : in std_logic_vector(0 to 7);
        Less    : out std_logic);
end LT;
architecture DES_LT of LT is
begin
    Less <= '1'  when (to_integer(A0) < to_integer(A1))
                 else '0';
end DES_LT;
---------------------------------------------------------------------
-- OPERATIVE
---------------------------------------------------------------------
library IEEE;
    use IEEE.std_logic_1164.all;
entity OPERATIVE is
port (  CLK         : in    std_logic;
        SELMUXX     : in    std_logic;
        SELMUXY     : in    std_logic;
        WRITEX      : in    std_logic;
        WRITEY      : in    std_logic;
        WRITEVAL    : in    std_logic;
        SIGNSUB     : in    std_logic;
        IN1, IN2    : in    std_logic_vector(0 to 7);
        XLESSY      : out   std_logic;
        YLESSX      : out   std_logic;
        VAL_PGCD    : out   std_logic_vector(0 to 7));
end OPERATIVE;
architecture DES_OPERATIVE of OPERATIVE is
    component SUB
    port (  A, B    : in    std_logic_vector(0 to 7);
            Sign    : in    std_logic;
            DIFF    : out   std_logic_vector(0 to 7));
    end component;
    component MUX
    port (  A0, A1  : in    std_logic_vector(0 to 7);
            S       : in    std_logic;
            Z       : out   std_logic_vector(0 to 7));
    end component;
    component REG
    port (  CLK     : in    std_logic;
            D       : in    std_logic_vector(0 to 7);
            Q       : out   std_logic_vector(0 to 7));
    end component;
    component LT
    port (  A0, A1  : in    std_logic_vector(0 to 7);
            Less    : out   std_logic);
    end component;
    signal  A, B, SUB_DIFF       : std_logic_vector(0 to 7);
    signal  ZMUXX, ZMUXY         : std_logic_vector(0 to 7);
    signal  CLKX, CLKY, CLKVAL : std_logic;
begin
    SUB1: SUB  port map (A, B, SIGNSUB, SUB_DIFF);
    MUXX: MUX
            port map (SUB_DIFF, IN1, SELMUXX, ZMUXX);
    MUXY: MUX
            port map (SUB_DIFF, IN2, SELMUXY, ZMUXY);
    REGX: REG  port map (CLKX, ZMUXX, A);
    REGY: REG  port map (CLKY, ZMUXY, B);
    REG_PGCD: REG port map (CLKVAL, A, VAL_PGCD);
    LT_XLESSY: LT   port map (A, B, XLESSY);
    LT_YLESSX: LT   port map (B, A, YLESSX);
    CLKX        <= WRITEX AND CLK;
    CLKY        <= WRITEY AND CLK;
    CLKVAL <= WRITEVAL AND CLK;
end DES_OPERATIVE;
-------------------------------------------------------------------
-- CONTROL
-------------------------------------------------------------------
library IEEE;
    use IEEE.std_logic_1164.all;
entity CONTROL is
    port (  CLK         : in    std_logic;
            START       : in    std_logic;
            XLESSY      : in    std_logic;
            YLESSX      : in    std_logic;
            SELMUXX     : out   std_logic;
            SELMUXY     : out   std_logic;
            WRITEX      : out   std_logic;
            WRITEY      : out   std_logic;
            WRITEVAL    : out   std_logic;
            SIGNSUB     : out   std_logic;
            READY       : out   std_logic);
end CONTROL;
architecture DES_CONTROL of CONTROL is
    type    states is (s1, s2, s3, s4, s5);
begin
SM: process (CLK)
    variable state : states := s1;
begin
if clk'event and clk='1' then
    case state is
    when s1 =>
        if START='1' then
            state := s2;
            SELMUXX <= '1'; SELMUXY <= '1';
            WRITEX <= '1';    WRITEY <= '1';
            WRITEVAL <= '0'; READY <= '0';
        end if;
    when s2 =>
        if YLESSX = '1' then
            state := s3;
            SIGNSUB <= '0'; SELMUXX <= '0';
            WRITEX <= '1'; WRITEY <= '0';
            WRITEVAL <= '0';
        elsif XLESSY = '1' then
            state := s4;
            SIGNSUB <= '1'; SELMUXY <= '0';
            WRITEX <= '0'; WRITEY <= '1';
            WRITEVAL <= '0';
        else
            state := s5;
            WRITEX <= '0'; WRITEY <= '0';
            WRITEVAL <= '1'; READY <= '1';
        end if;
    when s3 =>
        if YLESSX = '1' then
            SIGNSUB <= '0'; SELMUXX <= '0';
            WRITEX <= '1'; WRITEY <= '0';
            WRITEVAL <= '0';
        elsif XLESSY = '1' then
            state := s4;
            SIGNSUB <= '1'; SELMUXY <= '0';
            WRITEX <= '0'; WRITEY <= '1';
            WRITEVAL <= '0';
        else
            state := s5;
            WRITEX <= '0'; WRITEY <= '0';
            WRITEVAL <= '1'; READY <= '1';
        end if;
    when s4 =>
        if YLESSX = '1' then
            state := s3;
            SIGNSUB <= '0'; SELMUXX <= '0';
            WRITEX <= '1'; WRITEY <= '0';
            WRITEVAL <= '0';
        elsif XLESSY = '1' then
            SIGNSUB <= '1'; SELMUXY <= '0';
            WRITEX <= '0'; WRITEY <= '1';
            WRITEVAL <= '0';
        else
            state := s5;
            WRITEX <= '0'; WRITEY <= '0';
            WRITEVAL <= '1'; READY <= '1';
        end if;
    when s5 => state := s1;
    end case;
end if;
end process;
end DES_CONTROL;
-------------------------------------------------------------------
-- PGCD
-------------------------------------------------------------------
library IEEE;
    use IEEE.std_logic_1164.all;
entity PGCD is
port (  CLK, START : in     std_logic;
        IN1, IN2    : in     std_logic_vector(0 to 7);
        VAL_PGCD    : out    std_logic_vector(0 to 7);
        READY       : out    std_logic);
end PGCD;
architecture DES_PGCD of PGCD is
    component OPERATIVE
    port (  CLK         : in    std_logic;
            SELMUXX     : in    std_logic;
            SELMUXY     : in    std_logic;
            WRITEX      : in    std_logic;
            WRITEY      : in    std_logic;
            WRITEVAL    : in    std_logic;
            SIGNSUB     : in    std_logic;
            IN1, IN2    : in    std_logic_vector(0 to 7);
            XLESSY      : out   std_logic;
            YLESSX      : out std_logic;
            VAL_PGCD    : out std_logic_vector(0to7));
    end component;
    component CONTROL
    port (  CLK         : in    std_logic;
            START       : in    std_logic;
            XLESSY      : in    std_logic;
            YLESSX      : in    std_logic;
            SELMUXX     : out   std_logic;
            SELMUXY     : out   std_logic;
            WRITEX      : out   std_logic;
            WRITEY      : out   std_logic;
            WRITEVAL    : out   std_logic;
            SIGNSUB     : out   std_logic;
            READY       : out   std_logic);
    end component;
    signal  S_CLK                       : std_logic;
    signal  S_SELMUXX, S_SELMUXY        : std_logic;
    signal  S_WRITEX, S_WRITEY          : std_logic;
    signal  S_WRITEVAL                  : std_logic;
    signal  S_SIGNSUB                   : std_logic;
    signal  S_XLESSY, S_YLESSX          : std_logic;
begin
    OP: OPERATIVE
    port map (  CLK         => S_CLK,
                SELMUXX     => S_SELMUXX,
                SELMUXY     => S_SELMUXY,
                WRITEX      => S_WRITEX,
                WRITEY      => S_WRITEY,
                WRITEVAL    => S_WRITEVAL,
                SIGNSUB     => S_SIGNSUB,
                IN1         => IN1,
                IN2         => IN2,
                XLESSY      => S_XLESSY,
                YLESSX      => S_YLESSX,
                VAL_PGCD    => VAL_PGCD);
    CT: CONTROL
    port map (  CLK         => CLK,
                START       => START,
          XLESSY      => S_XLESSY,
          YLESSX      => S_YLESSX,
          SELMUXX     => S_SELMUXX,
          SELMUXY     => S_SELMUXY,
          WRITEX      => S_WRITEX,
          WRITEY      => S_WRITEY,
          WRITEVAL    => S_WRITEVAL,
          SIGNSUB     => S_SIGNSUB,
          READY       => READY);
    S_CLK <= not(CLK);
end DES_PGCD;
```

B.3.2 Moore machine

B.3.2.1 Description of the modeling styles

The modeling styles 2.1-2.11 are described in section 2.3.4.3. Below the description of the *GCD* example in each of those styles is provided.

Style 2.1

```
architecture DES_CONTROL of CONTROL is
    type    states is (s1, s2, s3, s4, s5);
begin
SM: process (CLK)
    variable state : states := s1;
begin
if clk'event and clk='1' then
    case state is
    when s1 =>
        if START='1' then
            state := s2;
            SELMUXX <= '1'; SELMUXY <= '1';
            WRITEX <= '1'; WRITEY <= '1';
            WRITEVAL <= '0'; READY <= '0'; end if;
    when s2 =>
        if YLESSX = '1' then
            state := s3;
            SIGNSUB <= '0'; SELMUXX <= '0';
            WRITEX <= '1'; WRITEY <= '0';
            WRITEVAL <= '0';
        elsif XLESSY = '1' then
            state := s4;
            SIGNSUB <= '1'; SELMUXY <= '0';
            WRITEX <= '0'; WRITEY <= '1';
            WRITEVAL <= '0';
        else
            state := s5;
            WRITEX <= '0'; WRITEY <= '0';
            WRITEVAL <= '1'; READY <= '1'; end if;
    when s3 =>
        if YLESSX = '1' then
            SIGNSUB <= '0'; SELMUXX <= '0';
            WRITEX <= '1'; WRITEY <= '0';
            WRITEVAL <= '0';
        elsif XLESSY = '1' then
            state := s4;
            SIGNSUB <= '1'; SELMUXY <= '0';
            WRITEX <= '0'; WRITEY <= '1';
            WRITEVAL <= '0';
        else
            state := s5;
            WRITEX <= '0'; WRITEY <= '0';
            WRITEVAL <= '1'; READY <= '1'; end if;
    when s4 =>
        if YLESSX = '1' then
            state := s3;
            SIGNSUB <= '0'; SELMUXX  <= '0';
            WRITEX <= '1'; WRITEY <= '0';
            WRITEVAL <= '0';
        elsif XLESSY = '1' then
            SIGNSUB <= '1'; SELMUXY <= '0';
            WRITEX <= '0'; WRITEY <= '1';
            WRITEVAL <= '0';
        else
            state := s5;
            WRITEX <= '0'; WRITEY <= '0';
            WRITEVAL <= '1'; READY <= '1'; end if;
    when s5 =>  state := s1;
    end case;
end if;
end process;
end DES_CONTROL;
```

Style 2.2

```
architecture DES_CONTROL of CONTROL is
    subtype     states   is integer range 1 to 5;
    type        fsm is array (states, bit, bit, bit) of states;
    constant    fsms : fsm := (
                (((1, 1), (1, 1)), ((2, 2), (2, 2))),  --s1
                (((5, 4), (3, 3)), ((5, 4), (3, 3))),  --s2
                (((5, 4), (3, 3)), ((5, 4), (3, 3))),  --s3
                (((5, 4), (3, 3)), ((5, 4), (3, 3))),  --s4
                (((1, 1), (1, 1)), ((1, 1), (1, 1)))); --s5
    type        fsmy     is array (states, bit, bit, bit) of
                         std_logic_vector(0 to 6);
    constant fsmsy : fsmy := (
    ((("0110000", "0110000"), ("0110000", "0110000")),
    (("0111100", "0111100"), ("0111100", "0111100"))),       --s1
    ((("0000011", "1000100"), ("0001000", "0001000")),
    (("0000011", "1000100"), ("0001000", "0001000"))),       --s2
    ((("0000011", "1000100"), ("0001000", "0001000")),
    (("0000011", "1000100"), ("0001000", "0001000"))),       --s3
    ((("0000011", "1000100"), ("0001000", "0001000")),
    (("0000011", "1000100"), ("0001000", "0001000"))),       --s4
    ((("0000000", "0000000"), ("0000000", "0000000")),
    (("0000000", "0000000"), ("0000000", "0000000"))));      --s5
begin
SM: process (CLK)
    variable state : states := 1;
    variable res : std_logic_vector(0 to 6);
begin
    if clk'event and clk='1' then
        res := fsmsy(state, to_bit(START), to_bit(YLESSX),
                        to_bit(XLESSY));
        state := fsms(state, to_bit(START), to_bit(YLESSX),
                        to_bit(XLESSY));
        SIGNSUB     <= res(0);
        SELMUXX     <= res(1);
        SELMUXY     <= res(2);
        WRITEX      <= res(3);
        WRITEY      <= res(4);
        WRITEVAL    <= res(5);
        READY       <= res(6);
    end if;
end process;
end DES_CONTROL;
```

Style 2.3

```
architecture DES_CONTROL of CONTROL is
    subtype     states    is integer range 1 to 5;
    type        fsm is array (states, bit, bit, bit) of states;
    constant    fsms : fsm := (
                (((1, 1), (1, 1)), ((2, 2), (2, 2))),  --s1
                (((5, 4), (3, 3)), ((5, 4), (3, 3))),  --s2
                (((5, 4), (3, 3)), ((5, 4), (3, 3))),  --s3
                (((5, 4), (3, 3)), ((5, 4), (3, 3))),  --s4
                (((1, 1), (1, 1)), ((1, 1), (1, 1))));  --s5
    type        fsmy      is array (states, bit, bit, bit)
                          of std_logic_vector(0 to 6);
    constant    fsmsy : fsmy := (
    ((("0110000", "0110000"), ("0110000", "0110000")),
    (("0111100", "0111100"), ("0111100", "0111100"))),      --s1
    ((("0000011", "1000100"), ("0001000", "0001000")),
    (("0000011", "1000100"), ("0001000", "0001000"))),      --s2
    ((("0000011", "1000100"), ("0001000", "0001000")),
    (("0000011", "1000100"), ("0001000", "0001000"))),      --s3
    ((("0000011", "1000100"), ("0001000", "0001000")),
    (("0000011", "1000100"), ("0001000", "0001000"))),      --s4
    ((("0000000", "0000000"), ("0000000", "0000000")),
    (("0000000", "0000000"), ("0000000", "0000000"))));     --s5
begin
SM: process (CLK)
    variable state : states := 1;
begin
    if clk'event and clk='1' then
        out_vector  <= fsmsy(state, to_bit(START),
                           to_bit(YLESSX), to_bit(XLESSY));
        state       := fsms(state, to_bit(START),
                           to_bit(YLESSX), to_bit(XLESSY));
    end if;
end process;
end DES_CONTROL;
```

Style 2.4

```
architecture DES_CONTROL of CONTROL is
    subtype s is integer range 1 to 5;
    type    states  is record
            state   : s;
            outres  : std_logic_vector(0 to 6);
    end record;
    typ     fsmy    is array (s, bit, bit, bit) of states;
    constant fsmsy : fsmy := (
    ((((1, "0110000"), (1, "0110000")), ((1, "0110000"),
    (1, "0110000"))), (((2, "0111100"), (2, "0111100")),
    ((2, "0111100"), (2, "0111100")))),     --s1
    ((((5, "0000011"), (4, "1000100")), ((3, "0001000"),
    (3, "0001000"))), (((5, "0000011"), (4, "1000100")),
    ((3, "0001000"), (3, "0001000")))),     --s2
    ((((5, "0000011"), (4, "1000100")), ((3, "0001000"),
    (3, "0001000"))), (((5, "0000011"), (4, "1000100")),
    ((3, "0001000"), (3, "0001000")))),     --s3
    ((((5, "0000011"), (4, "1000100")), ((3, "0001000"),
    (3, "0001000"))), (((5, "0000011"), (4, "1000100")),
    ((3, "0001000"), (3, "0001000")))),     --s4
    ((((1, "0000000"), (1, "0000000")), ((1, "0000000"),
    (1, "0000000"))), (((1, "0000000"), (1, "0000000")),
    ((1, "0000000"), (1, "0000000")))));  --s5
begin
SM: process (CLK)
    variable st  : s := 1;
    variable res : states;
begin
    if clk'event and clk='1' then
        res  := fsmsy(st, to_bit(START), to_bit(YLESSX),
                    to_bit(XLESSY));
        st   := res.state;
        SIGNSUB    <= res.outres(0);
        SELMUXX    <= res.outres(1);
        SELMUXY    <= res.outres(2);
        WRITEX     <= res.outres(3);
        WRITEY     <= res.outres(4);
        WRITEVAL   <= res.outres(5);
        READY      <= res.outres(6);  end if;
end process; end DES_CONTROL;
```

Style 2.5

```
architecture DES_CONTROL of CONTROL is
    type    states is (s1, s2, s3, s4, s5);
    signal  state, next_state : states := s1;
begin
CLKD: process (CLK)
begin
    if clk'event and clk='1' then
        state <= next_state;
        case state is
        when s1 =>
            if START='1' then
                SELMUXX <= '1'; SELMUXY <= '1';
                WRITEX <= '1'; WRITEY <= '1';
                WRITEVAL <= '0'; READY <= '0'; end if;
        when s2 =>
            if YLESSX = '1' then
                SIGNSUB <= '0'; SELMUXX <= '0';
                WRITEX <= '1'; WRITEY <= '0';
                WRITEVAL <= '0';
            elsif XLESSY = '1' then
                SIGNSUB <= '1'; SELMUXY <= '0';
                WRITEX <= '0'; WRITEY <= '1';
                WRITEVAL <= '0';
            else
                WRITEX <= '0'; WRITEY <= '0';
                WRITEVAL <= '1'; READY <= '1'; end if;
        when s3 =>
            if YLESSX = '1' then
                SIGNSUB <= '0'; SELMUXX <= '0';
                WRITEX <= '1'; WRITEY <= '0';
                WRITEVAL <= '0';
            elsif XLESSY = '1' then
                SIGNSUB <= '1'; SELMUXY <= '0';
                WRITEX <= '0'; WRITEY <= '1';
                WRITEVAL <= '0';
            else
                WRITEX <= '0'; WRITEY <= '0';
                WRITEVAL <= '1'; READY <= '1'; end if;
        when s4 =>
            if YLESSX = '1' then
                SIGNSUB <= '0'; SELMUXX <= '0';
                WRITEX <= '1'; WRITEY <= '0';
                WRITEVAL <= '0';
            elsif XLESSY = '1' then
                SIGNSUB <= '1'; SELMUXY <= '0';
                WRITEX <= '0'; WRITEY <= '1';
                WRITEVAL <= '0';
            else
                WRITEX <= '0'; WRITEY <= '0';
                WRITEVAL <= '1'; READY <= '1'; end if;
        when s5 => null;
        end case;
    end if;
end process;
SM: process (state, START, XLESSY, YLESSX)
begin
    next_state <= state;
    case state is
    when s1 =>
        if    START='1'         then     next_state <= s2; end if;
    when s2 =>
        if    YLESSX = '1'      then     next_state <= s3;
        elsif XLESSY = '1'      then     next_state <= s4;
        else                             next_state <= s5; end if;
    when s3 =>
        if      YLESSX = '1'  then       null;
        elsif   XLESSY = '1'  then       next_state <= s4;
        else                             next_state <= s5; end if;
    when s4 =>
        if      YLESSX = '1'  then       next_state <= s3;
        elsif   XLESSY = '1'  then       null;
        else                             next_state <= s5; end if;
        when s5 =>  next_state <= s1;
        end case;
end process;
end DES_CONTROL;
```

Style 2.6

```
architecture DES_CONTROL of CONTROL is
        subtype     table     is std_logic_vector(15 downto 0);
        constant    t_max     : integer := 39;
        type        fsmt      is array (0 to t_max) of table;
        constant    fsm : fsmt := (
                    "0011110010001100", --s1 start = 1
                    "0011110010001101",
                    "0011110010001110",
                    "0011110010001111",
                    "0000000001001000", --s1 start = 0
                    "0000000001001001",
                    "0000000001001010",
                    "0000000001001011",
                    "1100000101010100", --s2 start = 1
                    "0010001100010101",
                    "0001000011010110",
                    "0001000011010111",
                    "1100000101010000", --s2 start = 0
                    "0010001100010001",
                    "0001000011010010",
                    "0001000011010011",
                    "1100000101011100", --s3 start = 1
                    "0010001100011101",
                    "0001000011011110",
                    "0001000011011111",
                    "1100000101011000", --s3 start = 0
                    "0010001100011001",
                    "0001000011011010",
                    "0001000011011011",
                    "1100000101100100", --s4 start = 1
                    "0010001100100101",
                    "0001000011100110",
                    "0001000011100111",
                    "1100000101100000", --s4 start = 0
                    "0010001100100001",
                    "0001000011100010",
                    "0001000011100011",
                    "1000000001101100", --s5 start = 1
                    "1000000001101101",
                    "1000000001101110",
                    "1000000001101111",
                    "1000000001101000", --s5 start = 0
                    "1000000001101001",
                    "1000000001101010",
                    "1000000001101011");
    signal  state     : std_logic_vector(2 downto 0) := "001";
begin
SM: process (CLK)
    variable    data      : table;
begin
    if clk'event and clk='1' then
        for i in 0 to t_max loop
            data := fsm(i);
            if (data(5 downto 0) = state&start&ylessx&xlessy)
                then exit; end if;
        end loop;
        state <= data(8 downto 6);
        SIGNSUB     <= data(9);
        SELMUXX     <= data(10);
        SELMUXY     <= data(11);
        WRITEX      <= data(12);
        WRITEY      <= data(13);
        WRITEVAL    <= data(14);
        READY       <= data(15);
    end if;
end process; end DES_CONTROL;
```

Style 2.7

```
architecture DES_CONTROL of CONTROL is
        type        states    is (s1, s2, s3, s4, s5);
        type        fsm is array (states, bit, bit, bit) of states;
        constant    fsms : fsm := (
                    (((s1, s1), (s1, s1)), ((s2, s2), (s2, s2))), --s1
                    (((s5, s4), (s3, s3)), ((s5, s4), (s3, s3))), --s2
                    (((s5, s4), (s3, s3)), ((s5, s4), (s3, s3))), --s3
                    (((s5, s4), (s3, s3)), ((s5, s4), (s3, s3))), --s4
                    (((s1, s1), (s1, s1)), ((s1, s1), (s1, s1))));--s5
        type        fsmy      is array (states, bit, bit, bit)
                              of std_logic_vector(0 to 6);
        constant    fsmsy     : fsmy := (
        ((("0110000", "0110000"), ("0110000", "0110000")),
        (("0111100", "0111100"), ("0111100", "0111100"))),          --s1
        ((("0000011", "1000100"), ("0001000", "0001000")),
        (("0000011", "1000100"), ("0001000", "0001000"))),          --s2
        ((("0000011", "1000100"), ("0001000", "0001000")),
        (("0000011", "1000100"), ("0001000", "0001000"))),          --s3
        ((("0000011", "1000100"), ("0001000", "0001000")),
        (("0000011", "1000100"), ("0001000", "0001000"))),          --s4
        ((("0000000", "0000000"), ("0000000", "0000000")),
        (("0000000", "0000000"), ("0000000", "0000000"))));         --s5
begin
SM: process (CLK)
    variable    state     : states := s1;
    variable    res       : std_logic_vector(0 to 6);
begin
    if clk'event and clk='1' then
        res  := fsmsy(state, to_bit(START), to_bit(YLESSX),
                    to_bit(XLESSY));
        state := fsms(state, to_bit(START), to_bit(YLESSX),
                    to_bit(XLESSY));
        SIGNSUB     <= res(0);
        SELMUXX     <= res(1);
        SELMUXY     <= res(2);
        WRITEX      <= res(3);
        WRITEY      <= res(4);
        WRITEVAL    <= res(5);
        READY       <= res(6);
    end if;
end process;
end DES_CONTROL;
```

Style 2.8

```
architecture DES_CONTROL of CONTROL is
        subtype s is integer range 1 to 5;
        type    states  is record
                state   : s;
                outres  : std_logic_vector(0 to 6);
        end record;
        type    fsmy    is array (s, bit, bit, bit) of states;
        constant    fsmsy : fsmy := (
                ((((1, "0110000"), (1, "0110000")), ((1, "0110000"),
                (1, "0110000"))), (((2, "0111100"), (2, "0111100")),
                ((2, "0111100"), (2, "0111100")))),     --s1
                ((((5, "0000011"), (4, "1000100")), ((3, "0001000"),
                (3, "0001000"))), (((5, "0000011"), (4, "1000100")),
                ((3, "0001000"), (3, "0001000")))),     --s2
                ((((5, "0000011"), (4, "1000100")), ((3, "0001000"),
                (3, "0001000"))), (((5, "0000011"), (4, "1000100")),
                ((3, "0001000"), (3, "0001000")))),     --s3
                ((((5, "0000011"), (4, "1000100")), ((3, "0001000"),
                (3, "0001000"))), (((5, "0000011"), (4, "1000100")),
                ((3, "0001000"), (3, "0001000")))),     --s4
                ((((1, "0000000"), (1, "0000000")), ((1, "0000000"),
                (1, "0000000"))), (((1, "0000000"), (1, "0000000")),
                ((1, "0000000"), (1, "0000000")))));    --s5
begin
SM: process (CLK)
    variable state : s := 1;
```

```
begin
    if clk'event and clk='1' then
        out_vector <= fsmsy(state, to_bit(START),
                to_bit(YLESSX), to_bit(XLESSY)).outres;
        state    := fsmsy(state, to_bit(START),
                to_bit(YLESSX), to_bit(XLESSY)).state;
    end if;
end process; end DES_CONTROL;
```

Style 2.9

```
architecture DES_CONTROL of CONTROL is
    type    s is (s1, s2, s3, s4, s5);
    type    states  is record
            state   : s;
            outres  : std_logic_vector(0 to 6);
    end record;
    type    fsmy    is array (s, bit, bit, bit) of states;
constant    fsmsy : fsmy := (
        ((((s1, "0110000"), (s1, "0110000")), ((s1, "0110000"),
        (s1, "0110000"))), (((s2, "0111100"), (s2, "0111100")),
        ((s2, "0111100"), (s2, "0111100")))), --s1
        ((((s5, "0000011"), (s4, "1000100")), ((s3, "0001000"),
        (s3, "0001000"))), (((s5, "0000011"), (s4, "1000100")),
        ((s3, "0001000"), (s3, "0001000")))), --s2
        ((((s5, "0000011"), (s4, "1000100")), ((s3, "0001000"),
        (s3, "0001000"))), (((s5, "0000011"), (s4, "1000100")),
        ((s3, "0001000"), (s3, "0001000")))), --s3
        ((((s5, "0000011"), (s4, "1000100")), ((s3, "0001000"),
        (s3, "0001000"))), (((s5, "0000011"), (s4, "1000100")),
        ((s3, "0001000"), (s3, "0001000")))), --s4
        ((((s1, "0000000"), (s1, "0000000")), ((s1, "0000000"),
        (s1, "0000000"))), (((s1, "0000000"), (s1, "0000000")),
        ((s1, "0000000"), (s1, "0000000"))))); --s5
begin
SM: process (CLK)
    variable state : s := s1;
begin
    if clk'event and clk='1' then
        out_vector <= fsmsy(state, to_bit(START),
            to_bit(YLESSX), to_bit(XLESSY)).outres;
        state := fsmsy(state, to_bit(START), to_bit(YLESSX),
            to_bit(XLESSY)).state;
    end if;
end process;
end DES_CONTROL;
```

Style 2.10

```
architecture DES_CONTROL of CONTROL is
    subtype     states  is integer range 1 to 5;
    type        fsm     is array (states, bit, bit, bit) of states;
    constant    fsms : fsm := (
                (((1, 1), (1, 1)), ((2, 2), (2, 2))),  --s1
                (((5, 4), (3, 3)), ((5, 4), (3, 3))),  --s2
                (((5, 4), (3, 3)), ((5, 4), (3, 3))),  --s3
                (((5, 4), (3, 3)), ((5, 4), (3, 3))),  --s4
                (((1, 1), (1, 1)), ((1, 1), (1, 1)))); --s5
    type        fsmy    is array (states, bit, bit, bit)
                        of std_logic_vector(0 to 6);
    constant    fsmsy : fsmy := (
    ((("0110000", "0110000"), ("0110000", "0110000")),
    (("0111100", "0111100"), ("0111100", "0111100"))),      --s1
    ((("0000011", "1000100"), ("0001000", "0001000")),
    (("0000011", "1000100"), ("0001000", "0001000"))),      --s2
    ((("0000011", "1000100"), ("0001000", "0001000")),
    (("0000011", "1000100"), ("0001000", "0001000"))),      --s3
    ((("0000011", "1000100"), ("0001000", "0001000")),
    (("0000011", "1000100"), ("0001000", "0001000"))),      --s4
    ((("0000000", "0000000"), ("0000000", "0000000")),
    (("0000000", "0000000"), ("0000000", "0000000"))));     --s5
    signal  state, next_state : states := 1;
begin
CLKD: process (CLK)
    variable    res  : std_logic_vector(0 to 6);
begin
    if clk'event and clk='1' then
        state      <= next_state;
        res        := fsmsy(state, to_bit(START),
                      to_bit(YLESSX), to_bit(XLESSY));
        SIGNSUB    <= res(0);
        SELMUXX    <= res(1);
        SELMUXY    <= res(2);
        WRITEX     <= res(3);
        WRITEY     <= res(4);
        WRITEVAL   <= res(5);
        READY      <= res(6);
    end if;
end process;
SM: process (state, START, XLESSY, YLESSX)
begin
    next_state <= fsms(state, to_bit(START),
                  to_bit(YLESSX), to_bit(XLESSY));
end process;
end DES_CONTROL;
```

Style 2.11

```
architecture DES_CONTROL of CONTROL is
    subtype s is integer range 1 to 5;
    type        states  is record
                state   : s;
                outres  : std_logic_vector(0 to 6);
    end record;
    type        fsmy    is array (s, bit, bit, bit) of states;
    constant    fsmsy : fsmy := (
            1, "0110000"), (1, "0110000")), ((1, "0110000"),
            "0110000"))), (((2, "0111100"), (2, "0111100")),
            "0111100"), (2, "0111100")))),  --s1
            ((((5, "0000011"), (4, "1000100")), ((3, "0001000"),
            "0001000"))), (((5, "0000011"), (4, "1000100")),
            ((3, "0001000"), (3, "0001000")))),  --s2
            (5, "0000011"), (4, "1000100")), ((3, "0001000"),
            (3, "0001000"))), (((5, "0000011"), (4, "1000100")),
            ((3, "0001000"), (3, "0001000")))),  --s3
            ((((5, "0000011"), (4, "1000100")), ((3, "0001000"),
            (3, "0001000"))), (((5, "0000011"), (4, "1000100")),
            ((3, "0001000"), (3, "0001000")))),  --s4
            ((((1, "0000000"), (1, "0000000")), ((1, "0000000"),
            (1, "0000000"))), (((1, "0000000"), (1, "0000000")),
            ((1, "0000000"), (1, "0000000"))))); --s5
    signal      st, next_st : s := 1;
begin
CLKD: process (CLK)
    variable    res : states;
begin
    if clk'event and clk='1' then
        st   <= next_st;
        res  := fsmsy(st, to_bit(START), to_bit(YLESSX),
                to_bit(XLESSY));
        SIGNSUB    <= res.outres(0);
        SELMUXX    <= res.outres(1);
        SELMUXY    <= res.outres(2);
        WRITEX     <= res.outres(3);
        WRITEY     <= res.outres(4);
        WRITEVAL   <= res.outres(5);
        READY      <= res.outres(6);
    end if;
end process;
SM: process (st, START, XLESSY, YLESSX)
begin
    next_st <= fsmsy(st, to_bit(START), to_bit(YLESSX),
               to_bit(XLESSY)).state;
end process; end DES_CONTROL;
```

B.3.2.2 Simulation time comparison

RTL-FSM Modeling Style: Moore machine				Simulation time			
Style	# of processes	# of tables	# of choice statements	Simulator 1		Simulator 2	
				Sim:1E09ns	Gain	Sim:1E09ns	Gain
2.1	1 (clk+δ+λ)	0	1	11 304 575	1.00	1 692 934	1.00
2.2	1 (clk+δ+λ)	2 (state, out)	0	11 595 944	1.03	1 907 553	1.13
2.3	1 (clk+δ+λ)	2 (state, out)	0	13 285 113	1.18	1 873 113	1.11
2.4	1 (clk+δ+λ)	1 (state+out)	0	11 609 574	1.03	1 962 121	1.16
2.5	2 (clk+λ, δ)	0	2	11 468 651	1.01	1 740 753	1.03
2.6	1 (clk+δ+λ)	1	1 (in loop)	14 721 125	1.30	10 231 842	6.04
2.7	1 (clk+δ+λ)	2 (state, out)	0	11 590 356	1.03	1 918 078	1.13
2.8	1 (clk+δ+λ)	1 (state+out)	0	13 349 716	1.18	1 888 035	1.12
2.9	1 (clk+δ+λ)	1 (state+out)	0	13 322 337	1.18	1 894 584	1.12
2.10	2 (clk+λ, δ)	2 (state, out)	0	11 637 875	1.03	2 105 478	1.24
2.11	2 (clk+λ, δ)	1 (state+out)	0	11 686 454	1.03	2 158 274	1.27

Table B.2: *Comparison of RTL-FSM modeling styles for Moore machines*

B.3.3 Mealy machine

B.3.3.1 Description of the modeling styles

The *GCD* Mealy machine is modeled using the styles 3.1-3.9, which are described in section 2.3.4.4.

Style 3.1

```
architecture DES_CONTROL of CONTROL is
    type    states is (s1, s2, s3, s4, s5);
    signal  state : states := s1;
begin
SM: process (CLK)
begin
    if clk'event and clk='1' then
        case state is
        when s1 =>
            if   START='1'   then      state <= s2; end if;
        when s2 =>
            if      YLESSX = '1'then    state <= s3;
            elsif   XLESSY = '1'then    state <= s4;
            else                        state <= s5; end if;
        when s3 =>
            if      YLESSX = '1'then    null;
            elsif   XLESSY = '1'then    state <= s4;
            else                        state <= s5; end if;
        when s4 =>
            if      YLESSX = '1'then    state <= s3;
            elsif   XLESSY = '1'then    null;
            else                        state <= s5; end if;
        when s5 =>  state <= s1;
        end case;
    end if;
end process;
OUT1: process (state, START, XLESSY, YLESSX)
begin
    case state is
    when s1 =>
        if START='1' then
            SELMUXX <= '1'; SELMUXY <= '1';
            WRITEX <= '1'; WRITEY <= '1';
            WRITEVAL <= '0'; READY <= '0'; end if;
    when s2 =>
        if YLESSX = '1' then
            SIGNSUB <= '0'; SELMUXX  <= '0';
            WRITEX <= '1'; WRITEY <= '0';
            WRITEVAL <= '0';
        elsif XLESSY = '1' then
            SIGNSUB <= '1'; SELMUXY <= '0';
            WRITEX <= '0'; WRITEY <= '1';
            WRITEVAL <= '0';
        else
            WRITEX <= '0'; WRITEY <= '0';
            WRITEVAL <= '1'; READY <= '1'; end if;
    when s3 =>
        if YLESSX = '1' then
            SIGNSUB <= '0'; SELMUXX <= '0';
            WRITEX <= '1'; WRITEY <= '0';
            WRITEVAL <= '0';
        elsif XLESSY = '1' then
            SIGNSUB <= '1'; SELMUXY <= '0';
            WRITEX <= '0'; WRITEY <= '1';
            WRITEVAL <= '0';
        else
            WRITEX <= '0'; WRITEY <= '0';
            WRITEVAL <= '1'; READY <= '1'; end if;
    when s4 =>
        if YLESSX = '1' then
            SIGNSUB <= '0'; SELMUXX <= '0';
            WRITEX <= '1'; WRITEY <= '0';
            WRITEVAL <= '0';
        elsif XLESSY = '1' then
            SIGNSUB <= '1'; SELMUXY <= '0';
            WRITEX <= '0'; WRITEY <= '1';
```

```
                WRITEVAL <= '0';
            else
                WRITEX <= '0'; WRITEY <= '0';
                WRITEVAL <= '1'; READY <= '1'; end if;
        when s5 =>  null;
        end case;
    end process;
    end DES_CONTROL;
```

Style 3.2

```
architecture DES_CONTROL of CONTROL is
    type    states is (s1, s2, s3, s4, s5);
    signal  state, next_state : states := s1;
begin
SM: process (state, START, XLESSY, YLESSX)
begin
    next_state <= state;
    case state is
    when s1 =>
        if START='1' then
            next_state <= s2;
            SELMUXX <= '1'; SELMUXY <= '1';
            WRITEX <= '1'; WRITEY <= '1';
            WRITEVAL <= '0'; READY <= '0'; end if;
    when s2 =>
        if YLESSX = '1' then
            next_state <= s3;
            SIGNSUB <= '0'; SELMUXX <= '0';
            WRITEX <= '1'; WRITEY <= '0';
            WRITEVAL <= '0';
        elsif XLESSY = '1' then
            next_state <= s4;
            SIGNSUB <= '1'; SELMUXY <= '0';
            WRITEX <= '0'; WRITEY <= '1';
            WRITEVAL <= '0';
        else
            next_state <= s5;
            WRITEX <= '0'; WRITEY <= '0';
            WRITEVAL <= '1'; READY <= '1'; end if;
    when s3 =>
        if YLESSX = '1' then
            SIGNSUB <= '0'; SELMUXX <= '0';
            WRITEX <= '1'; WRITEY <= '0';
            WRITEVAL <= '0';
        elsif XLESSY = '1' then
            next_state <= s4;
            SIGNSUB <= '1'; SELMUXY <= '0';
            WRITEX <= '0'; WRITEY <= '1';
            WRITEVAL <= '0';
        else
            next_state <= s5;
            WRITEX <= '0'; WRITEY <= '0';
            WRITEVAL <= '1'; READY <= '1'; end if;
    when s4 =>
        if YLESSX = '1' then
            next_state <= s3;
            SIGNSUB <= '0'; SELMUXX <= '0';
            WRITEX <= '1'; WRITEY <= '0';
            WRITEVAL <= '0';
        elsif XLESSY = '1' then
            SIGNSUB <= '1'; SELMUXY <= '0';
            WRITEX <= '0'; WRITEY <= '1';
            WRITEVAL <= '0';
        else
            next_state <= s5;
            WRITEX <= '0'; WRITEY <= '0';
            WRITEVAL <= '1'; READY <= '1'; end if;
    when s5 =>  next_state <= s1;
    end case;
end process;
CLKD: process (CLK)
begin
    if clk'event and clk='1' then state <= next_state; end if;
end process; end DES_CONTROL;
```

Style 3.3

```
architecture DES_CONTROL of CONTROL is
    subtype   states  is integer range 1 to 5;
    type      fsm     is array (states, bit, bit, bit) of states;
    constant  fsms : fsm := (
              (((1, 1), (1, 1)), ((2, 2), (2, 2))),  --s1
              (((5, 4), (3, 3)), ((5, 4), (3, 3))),  --s2
              (((5, 4), (3, 3)), ((5, 4), (3, 3))),  --s3
              (((5, 4), (3, 3)), ((5, 4), (3, 3))),  --s4
              (((1, 1), (1, 1)), ((1, 1), (1, 1)))); --s5
    type      fsmy    is array (states, bit, bit, bit)
                      of std_logic_vector(0 to 6);
    constant  fsmsy : fsmy := (
    ((("0110000", "0110000"), ("0110000", "0110000")),
    (("0111100", "0111100"), ("0111100", "0111100"))),      --s1
    ((("0000011", "1000100"), ("0001000", "0001000")),
    (("0000011", "1000100"), ("0001000", "0001000"))),      --s2
    ((("0000011", "1000100"), ("0001000", "0001000")),
    (("0000011", "1000100"), ("0001000", "0001000"))),      --s3
    ((("0000011", "1000100"), ("0001000", "0001000")),
    (("0000011", "1000100"), ("0001000", "0001000"))),      --s4
    ((("0000000", "0000000"), ("0000000", "0000000")),
    (("0000000", "0000000"), ("0000000", "0000000"))));     --s5
    signal    state, next_state : states := 1;
begin
SM: process (state, START, XLESSY, YLESSX)
    variable res : std_logic_vector(0 to 6);
begin
    res  :=  fsmsy(state, to_bit(START), to_bit(YLESSX),
                to_bit(XLESSY));
    next_state  <=  fsms(state, to_bit(START),
                        to_bit(YLESSX), to_bit(XLESSY));
    SIGNSUB     <= res(0);
    SELMUXX     <= res(1);
    SELMUXY     <= res(2);
    WRITEX      <= res(3);
    WRITEY      <= res(4);
    WRITEVAL    <= res(5);
    READY       <= res(6);
end process;
CLKD: process (CLK)
begin
    if clk'event and clk='1' then state <= next_state; end if;
end process;
end DES_CONTROL;
```

Style 3.4

```
architecture DES_CONTROL of CONTROL is
    type    states is (s1, s2, s3, s4, s5);
    signal  state, next_state : states := s1;
begin
CLKD: process (CLK)
begin
    if clk'event and clk='1' then
        state <= next_state; end if;
end process;
ST: process (state, START, XLESSY, YLESSX)
begin
    next_state <= state;
    case state is
    when s1 =>
        if START='1' then next_state <= s2; end if;
    when s2 =>
        if      YLESSX = '1'then    next_state <= s3;
        elsif   XLESSY = '1'then    next_state <= s4;
        else                        next_state <= s5; end if;
    when s3 =>
        if      YLESSX = '1'then    null;
        elsif   XLESSY = '1'then    next_state <= s4;
        else                        next_state <= s5; end if;
    when s4 =>
```

```
        if      YLESSX = '1'then    next_state <= s3;
        elsif   XLESSY = '1'then    null;
        else                        next_state <= s5; end if;
    when s5 =>  next_state <= s1;
    end case;
end process;
OUT1: process (state, START, XLESSY, YLESSX)
begin
    case state is
    when s1 =>
        if START='1' then
            SELMUXX <= '1'; SELMUXY <= '1';
            WRITEX <= '1'; WRITEY <= '1';
            WRITEVAL <= '0'; READY <= '0'; end if;
    when s2 =>
        if YLESSX = '1' then
            SIGNSUB <= '0'; SELMUXX <= '0';
            WRITEX <= '1'; WRITEY <= '0';
            WRITEVAL <= '0';
        elsif XLESSY = '1' then
            SIGNSUB <= '1'; SELMUXY <= '0';
            WRITEX <= '0'; WRITEY <= '1';
            WRITEVAL <= '0';
        else
            WRITEX <= '0'; WRITEY <= '0';
            WRITEVAL <= '1'; READY <= '1';end if;
    when s3 =>
        if YLESSX = '1' then
            SIGNSUB <= '0'; SELMUXX <= '0';
            WRITEX <= '1'; WRITEY <= '0';
            WRITEVAL <= '0';
        elsif XLESSY = '1' then
            SIGNSUB <= '1'; SELMUXY <= '0';
            WRITEX <= '0'; WRITEY <= '1';
            WRITEVAL <= '0';
        else
            WRITEX <= '0'; WRITEY <= '0';
            WRITEVAL <= '1'; READY <= '1'; end if;
    when s4 =>
        if YLESSX = '1' then
            SIGNSUB <= '0'; SELMUXX <= '0';
            WRITEX <= '1'; WRITEY <= '0';
            WRITEVAL <= '0';
        elsif XLESSY = '1' then
            SIGNSUB <= '1'; SELMUXY <= '0';
            WRITEX <= '0'; WRITEY <= '1';
            WRITEVAL <= '0';
        else
            WRITEX <= '0'; WRITEY <= '0';
            WRITEVAL <= '1'; READY <= '1'; end if;
    when s5 =>  null;
    end case;
end process;
end DES_CONTROL;
```

Style 3.5

```
architecture DES_CONTROL of CONTROL is
    subtype s is integer range 1 to 5;
    type    states  is record
            state   : s;
            outres  : std_logic_vector(0 to 6);
    end record;
    type    fsmy    is array (s, bit, bit, bit) of states;
    constant   fsmsy : fsmy := (
            ((((1, "0110000"), (1, "0110000")), ((1, "0110000"),
            (1, "0110000"))), (((2, "0111100"), (2, "0111100")),
            ((2, "0111100"), (2, "0111100")))),    --s1
            ((((5, "0000011"), (4, "1000100")), ((3, "0001000"),
            (3, "0001000"))), (((5, "0000011"), (4, "1000100")),
            ((3, "0001000"), (3, "0001000")))),    --s2
            ((((5, "0000011"), (4, "1000100")), ((3, "0001000"),
            (3, "0001000"))), (((5, "0000011"), (4, "1000100")),
            ((3, "0001000"), (3, "0001000")))),    --s3
            ((((5, "0000011"), (4, "1000100")), ((3, "0001000"),
            (3, "0001000"))), (((5, "0000011"), (4, "1000100")),
            ((3, "0001000"), (3, "0001000")))),    --s4
            ((((1, "0000000"), (1, "0000000")), ((1, "0000000"),
            (1, "0000000"))), (((1, "0000000"), (1, "0000000")),
            ((1, "0000000"), (1, "0000000")))));   --s5
    signal  st, next_st : s := 1;
begin
CLKD: process (CLK)
begin
    if clk'event and clk='1' then st <= next_st; end if;
end process;
STM: process (st, START, XLESSY, YLESSX)
    variable    res  : states;
begin
    res  :=  fsmsy(st, to_bit(START), to_bit(YLESSX),
             to_bit(XLESSY));
    next_st <= res.state;
end process;
OUT1: process (st, START, XLESSY, YLESSX)
    variable res : states;
begin
    res  :=  fsmsy(st, to_bit(START), to_bit(YLESSX),
             to_bit(XLESSY));
    SIGNSUB     <= res.outres(0);
    SELMUXX     <= res.outres(1);
    SELMUXY     <= res.outres(2);
    WRITEX      <= res.outres(3);
    WRITEY      <= res.outres(4);
    WRITEVAL    <= res.outres(5);
    READY       <= res.outres(6);
end process;
end DES_CONTROL;
```

Style 3.6

```
architecture DES_CONTROL of CONTROL is
    subtype     states  is integer range 1 to 5;
    type        fsm is array (states, bit, bit, bit) of states;
    constant    fsms : fsm := (
                (((1, 1), (1, 1)), ((2, 2), (2, 2))),  --s1
                (((5, 4), (3, 3)), ((5, 4), (3, 3))),  --s2
                (((5, 4), (3, 3)), ((5, 4), (3, 3))),  --s3
                (((5, 4), (3, 3)), ((5, 4), (3, 3))),  --s4
                (((1, 1), (1, 1)), ((1, 1), (1, 1))));  --s5
    type        fsmy    is array (states, bit, bit, bit)
                        of std_logic_vector(0 to 6);
    constant    fsmsy : fsmy := (
    ((("0110000", "0110000"), ("0110000", "0110000")),
    (("0111100", "0111100"), ("0111100", "0111100"))),      --s1
    ((("0000011", "1000100"), ("0001000", "0001000")),
    (("0000011", "1000100"), ("0001000", "0001000"))),      --s2
    ((("0000011", "1000100"), ("0001000", "0001000")),
    (("0000011", "1000100"), ("0001000", "0001000"))),      --s3
    ((("0000011", "1000100"), ("0001000", "0001000")),
    (("0000011", "1000100"), ("0001000", "0001000"))),      --s4
    ((("0000000", "0000000"), ("0000000", "0000000")),
    (("0000000", "0000000"), ("0000000", "0000000"))));     --s5
    signal   state : states := 1;
begin
SM: process (CLK)
begin
    if clk'event and clk='1' then
        state <= fsms (state, to_bit(START), to_bit(YLESSX),
                       to_bit(XLESSY)); end if;
end process;
OUT1: process (state, START, XLESSY, YLESSX)
    variable res : std_logic_vector(0 to 6);
begin
    res  := fsmsy(state, to_bit(START), to_bit(YLESSX),
            to_bit(XLESSY));
    SIGNSUB     <= res(0);
    SELMUXX     <= res(1);
    SELMUXY     <= res(2);
    WRITEX      <= res(3);
    WRITEY      <= res(4);
    WRITEVAL    <= res(5);
    READY       <= res(6);
end process;
end DES_CONTROL;
```

Style 3.7

```
architecture DES_CONTROL of CONTROL is
    subtype s is integer range 1 to 5;
    type    states  is record
            state   : s;
            outres  : std_logic_vector(0 to 6);
    end record;
    type    fsmy    is array (s, bit, bit, bit) of states;
    constant   fsmsy : fsmy := (
            ((((1, "0110000"), (1, "0110000")), ((1, "0110000"),
            (1, "0110000"))), (((2, "0111100"), (2, "0111100")),
            ((2, "0111100"), (2, "0111100")))),     --s1
            ((((5, "0000011"), (4, "1000100")), ((3, "0001000"),
            (3, "0001000"))), (((5, "0000011"), (4, "1000100")),
            ((3, "0001000"), (3, "0001000")))),     --s2
            ((((5, "0000011"), (4, "1000100")), ((3, "0001000"),
            (3, "0001000"))), (((5, "0000011"), (4, "1000100")),
            ((3, "0001000"), (3, "0001000")))),     --s3
            ((((5, "0000011"), (4, "1000100")), ((3, "0001000"),
            (3, "0001000"))), (((5, "0000011"), (4, "1000100")),
            ((3, "0001000"), (3, "0001000")))),     --s4
            ((((1, "0000000"), (1, "0000000")), ((1, "0000000"),
            (1, "0000000"))), (((1, "0000000"), (1, "0000000")),
            1, "0000000"), (1, "0000000")))));      --s5
    signal  st   : s := 1;
```

```
begin
SM: process (CLK)
    variable res : states;
begin
    if clk'event and clk='1' then
        res  :=   fsmsy(st, to_bit(START), to_bit(YLESSX),
                  to_bit(XLESSY));
        st   <= res.state;
    end if;
end process;
OUT1: process (st, START, XLESSY, YLESSX)
    variable res : states;
begin
    res  :=   fsmsy(st, to_bit(START), to_bit(YLESSX),
              to_bit(XLESSY));
    SIGNSUB    <= res.outres(0);
    SELMUXX    <= res.outres(1);
    SELMUXY    <= res.outres(2);
    WRITEX     <= res.outres(3);
    WRITEY     <= res.outres(4);
    WRITEVAL   <= res.outres(5);
    READY      <= res.outres(6);
end process;
end DES_CONTROL;
```

Style 3.8

```
architecture DES_CONTROL of CONTROL is
    subtype s is integer range 1 to 5;
    type    states  is record
            state   : s;
            outres  : std_logic_vector(0 to 6);
    end record;
    type    fsmy    is array (s, bit, bit, bit) of states;
    constant     fsmsy : fsmy := (
        ((   ((1, "0110000"), (1, "0110000")),
        ((1, "0110000"), (1, "0110000"))),
        (((2, "0111100"), (2, "0111100")),
        ((2, "0111100"), (2, "0111100")))),     --s1
        ((((5, "0000011"), (4, "1000100")),
        ((3, "0001000"), (3, "0001000"))),
        (((5, "0000011"), (4, "1000100")),
        ((3, "0001000"), (3, "0001000")))),     --s2
        ((((5, "0000011"), (4, "1000100")),
        ((3, "0001000"), (3, "0001000"))),
        (((5, "0000011"), (4, "1000100")),
        ((3, "0001000"), (3, "0001000")))),     --s3
        ((((5, "0000011"), (4, "1000100")),
        ((3, "0001000"), (3, "0001000"))),
        (((5, "0000011"), (4, "1000100")),
        ((3, "0001000"), (3, "0001000")))),     --s4
```

```
        ((((1, "0000000"), (1, "0000000")),
        ((1, "0000000"), (1, "0000000"))),
        (((1, "0000000"), (1, "0000000")),
        ((1, "0000000"), (1, "0000000")))));   --s5
    signal   st, next_st : s := 1;
begin
SM: process (st, START, XLESSY, YLESSX)
    variable res : states;
begin
    res  :=   fsmsy(st, to_bit(START), to_bit(YLESSX),
              to_bit(XLESSY));
    next_st <= res.state;
    SIGNSUB    <= res.outres(0);
    SELMUXX    <= res.outres(1);
    SELMUXY    <= res.outres(2);
    WRITEX     <= res.outres(3);
    WRITEY     <= res.outres(4);
    WRITEVAL   <= res.outres(5);
    READY      <= res.outres(6);
end process;
CLKD: process (CLK)
begin
    if clk'event and clk='1' then    st <= next_st; end if;
end process; end DES_CONTROL;
```

Style 3.9

```
architecture DES_CONTROL of CONTROL is
    subtype    states   is integer range 1 to 5;
    type       fsm      is array (states, bit, bit, bit) of states;
    constant   fsms : fsm := (
               (((1, 1), (1, 1)), ((2, 2), (2, 2))),  --s1
               (((5, 4), (3, 3)), ((5, 4), (3, 3))),  --s2
               (((5, 4), (3, 3)), ((5, 4), (3, 3))),  --s3
               (((5, 4), (3, 3)), ((5, 4), (3, 3))),  --s4
               (((1, 1), (1, 1)), ((1, 1), (1, 1)))); --s5
    type       fsmy     is array (states, bit, bit, bit)
                        of std_logic_vector(0 to 6);
    constant   fsmsy : fsmy := (
    ((("0110000", "0110000"), ("0110000", "0110000")),
    (("0111100", "0111100"), ("0111100", "0111100"))),      --s1
    ((("0000011", "1000100"), ("0001000", "0001000")),
    (("0000011", "1000100"), ("0001000", "0001000"))),      --s2
    ((("0000011", "1000100"), ("0001000", "0001000")),
    (("0000011", "1000100"), ("0001000", "0001000"))),      --s3
    ((("0000011", "1000100"), ("0001000", "0001000")),
    (("0000011", "1000100"), ("0001000", "0001000"))),      --s4
    ((("0000000", "0000000"), ("0000000", "0000000")),
    (("0000000", "0000000"), ("0000000", "0000000"))));     --s5
    signal   state, next_state : states := 1;
```

```
begin
CLKD: process (CLK)
    begin
        if clk'event and clk='1' then state <= next_state; end if;
    end process;
ST: process (state, START, XLESSY, YLESSX)
    begin
        next_state <=      fsms(state, to_bit(START),
                           to_bit(YLESSX), to_bit(XLESSY));
    end process;
OUT1: process (state, START, XLESSY, YLESSX)
    variable res : std_logic_vector(0 to 6);
    begin
        res  :=   fsmsy(state, to_bit(START), to_bit(YLESSX),
                  to_bit(XLESSY));
        SIGNSUB     <= res(0);
        SELMUXX     <= res(1);
        SELMUXY     <= res(2);
        WRITEX      <= res(3);
        WRITEY      <= res(4);
        WRITEVAL    <= res(5);
        READY       <= res(6);
    end process; end DES_CONTROL;
```

B.3.3.2 Simulation time comparison

RTL-FSM Modeling Style: Mealy machine				Simulation time			
Style	# of processes	# of tables	# of choice statements	Simulator 1		Simulator 2	
				Sim:1E09ns	Gain	Sim:1E09ns	Gain
3.1	2 (clk+δ, λ)	0	2	5 389 289	1.12	822 923	1.12
3.2	2 (δ+λ, clk)	0	1	5 396 710	1.12	868 059	1.18
3.3	2 (δ+λ, clk)	2 (state, out)	0	4 808 054	1.00	740 745	1.01
3.4	3 (clk, δ, λ)	0	2	5 368 790	1.12	835 832	1.14
3.5	3 (clk, δ, λ)	1 (state+out)	0	4 809 486	1.00	735 698	1.00
3.6	2 (clk+δ, λ)	2 (state, out)	0	4 848 091	1.01	789 795	1.07
3.7	2 (clk+δ, λ)	1 (state+out)	0	4 916 069	1.02	888 948	1.21
3.8	2 (δ+λ, clk)	1 (state+out)	0	4 877 944	1.01	756 709	1.03
3.9	3 (clk, δ, λ)	2 (state, out)	0	4 807 112	1.00	756 959	1.03

Table B.3: *Comparison of RTL-FSM modeling styles for Mealy machines*

B.4 Test examples

The finite state machine modeling styles have been tested also on other examples described in detail in [99, 100]. Those examples have been implemented in the test model generation tool (section 2.5). The finite state machine generated by the tool has 10000 states Here, the results of the simulation performance evaluation are presented.

Finite state machine modeling				
Model description	Simulation			
	Simulator 1		Simulator 2	
	Time [s]	Ratio	Time [s]	Ratio
FSM Moore type				
1 process (both: next state & output)	**2389.3**	**1.00**	**195.3**	**1.00**
2 processes (next state & output)	4879.7	2.04	358.4	1.84
3 processes (synch & asynch out)	7886.7	3.30	839.6	4.30
1 process / 2 tables (next state & output)	2447.5	1.02	220.1	1.13
1 process / 2 tables (next state & output vector)	2807.9	1.18	216.1	1.11
table look-up	3111.4	1.30	1180.4	6.04
FSM Mealy type				
2 processes (clock/state & output)	4910.3	1.82	368.6	1.82
2 processes (next state/output & clock/state)	3023.4	1.12	226.5	1.12
3 processes (clock/state & next state & output)	5654.9	2.10	412.9	2.04
2 processes (next state/output & clk/state) / 2 tables	**2693.6**	**1.00**	203.9	1.01
3 processes (clk/state & nxt state & output) / 1 table	2694.4	1.00	**202.5**	**1.00**

Table B.4: *Finite state machine modeling comparison: test example*

Annex C

Application of optimization, transformation and abstraction methods to an industrial example

C.1 Design characteristics

As an industrial case a telecommunication system VHDL model has been used. This model has been provided by the Italtel company [8].

The device is called ILC16 (16 HDLC Channels Italtel Link Controller) and is used in the Italtel telephone exchange Linea UT. Linea UT is the Italtel product line of digital switching systems for N-ISDN and B-ISDN/ATM wired network applications and for multimedia and wireless networks.

Within the complex UT architecture the Command Module (CM) is devoted to manage basic information for routing, taxing and diagnosis, collecting data from user cards controllers and supplying routing data to the switching matrix. CM is a fault tolerant processing unit based on macro synchronous master/slave architecture. The whole CM exchanges data with the environments using a single proprietary protocol, PROSSIM (PROtocollo Seriale Sincrono di InterModulo). This serial protocol is based on the more general HDLC.

The ILC16 device is a custom ASIC implementation of 16 Channel HDLC Communication Controller with a mixed HW/SW architecture. Combination of more ILC16s can manage a large number of PROSSIM links.

The particular implementation of the ILC16 device that is considered here is a VLSI device devoted to support the two standard Italtel serial communication protocols, synchronous (Prossim) and asynchronous (Prosa). It can interface up to 16 serial links and it is designed to implement the HDLC protocol on 16 synchronous channels. The HDLC protocol is one of the most common OSI layer 2 protocols on which many other common layer 2 protocols are based.

The management of the HDLC protocol on the 16 available channels is executed by the internal RISC microcontroller. The device includes also eight programmable asynchronous links based on UART cells directly mapped into the bus interface slave space and can support many asynchronous start/stop protocols. The proposed model doesn't cover these last features. Each link allows managing data in transmitting (tx) and receiving (rx) modes at the same time. During the rx mode, the data are inserted into the data storage memory for a future analysis.

Table C.1 presents the characteristics of the device modules used in the experiments.

Design characteristics						
Design	Inputs	Outputs	Signals	Variables	Processes	Lines of code
fiforx	8	3	27	2	7	1113
rxbit	4	3	21	1	6	1341
bus_interface_rx	15	10	28	2	6	1720

Table C.1: *Design characteristics*

C.2 Simulation tools and environment

The simulation tools used for experiments are:

- VB VHDL SysSim 98.0 from VeriBest, Inc (indicated as Simulator 1) [161]
- ModelSim PE/Plus 4.7b from Model Technology, Inc.
  (indicated as Simulator 2) [97]

All the experiments have been carried out on the Pentium II 200MHz computer with 128MB RAM, running WindowsNT 4.0 operating system.

C.3 Results

The model transformation rules have been sequentially applied to the model. The tables below present summaries of the results obtained for the particular modules of the device. The first column is the type of the transformation applied, the second column provides the some details about the transformation, while the following columns indicate the simulation time in [s] of the model, and the gain in comparison to the initial model for two simulation tools.

In the tables the following abbreviations are used:

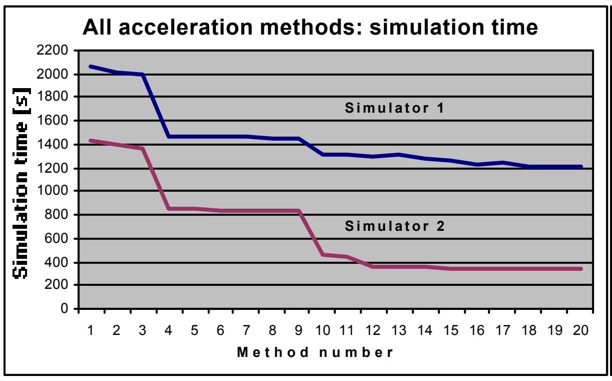
IM – initial model	DA – data-type abstraction
TR – transformation rule	BA – behavioral abstraction
OR – optimization rule	OA – object abstraction

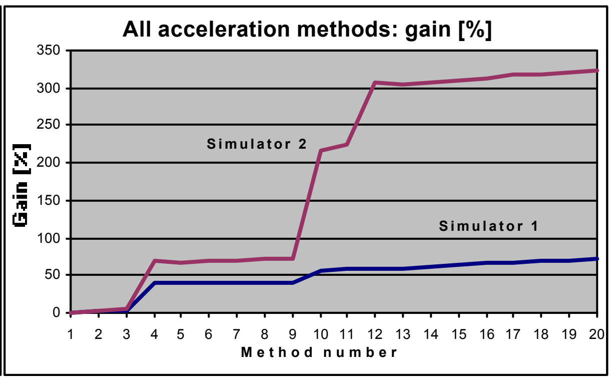
C.3.1 Model 1: fiforx

C.3.1.1 All methods

Model: fiforx			**Simulator 1**		**Simulated time: 1E09 ns**		
All Methods				**Simulation Time**		**Gain**	
No	**Method**	**Description**	**Compilation**	**Simulation**	**Total**	**Gain %**	**Ratio %**
1	IM	Initial model	24.095	2 036.027	2 060.122	0.00	100.0
2	TR1	Process merge 1: 2 processes merged	12.808	2 000.066	2 012.874	2.35	97.7
3	TR2	Process merge 2: 2 processes merged	13.039	1 974.459	1 987.498	3.65	96.5
4	OA	7 signals replaced: PROSSIMO, PROSSIMO_ME, PNUMBYTE, PW, PR, P_DATO, P_TAG	12.648	1 460.079	1 472.727	39.88	71.5
5	OR1	Delete package reference: package FRAME	12.218	1 454.682	1 466.900	40.44	71.2
6	OR2	Constant instantiation: INDET, OVRUN	12.287	1 453.550	1 465.837	40.54	71.2
7	OR3	Function inline expansion: INC	12.047	1 447.492	1 459.539	41.15	70.8
8	TR3	Transformation if->case: 6 transformations	12.007	1 446.020	1 458.027	41.30	70.8
9	OR4	Delete package reference: package PFIFO Constant instantiation	12.208	1 445.709	1 457.917	41.31	70.8
10	DA1	Abstraction: std_logic_vector(7 to 0) to integer 2 ports: DAT_IN, DATA_OUT 1 signal: REG_DATO 1 shared variable: P_DATO	12.318	1 299.799	1 312.117	57.01	63.7
11	DA2	Abstraction: std_logic_vector(7 downto 0) to integer 2 ports: TAG_IN, TAG_OUT 1 signal: REG_TAG 1 shared variable: P_TAG	13.009	1 294.711	1 307.720	57.54	63.5
12	DA3	Abstraction: std_logic to bit 7 ports: CLK_RX, EN_HDLC_H, ICLK, RD_H, RES_L, WR_H, VALID_H	13.669	1 288.903	1 302.572	58.16	63.2
13	DA4	Abstraction: std_logic to bit 2 signals: HDLC_RES_L, HDLC_EV_H	12.588	1 295.192	1 307.780	57.53	63.5
14	BA1	IOI: EN_HDLC_H=1	12.719	1 264.797	1 277.516	61.26	62.0
15	BA2	IOI: RD_H=1	12.559	1 241.842	1 254.401	64.23	60.9
16	BA3	IOI: RES_L=1	12.829	1 220.285	1 233.114	67.07	59.9
17	BA4	IOI: TAG_IN=0	12.859	1 227.061	1 239.920	66.15	60.2
18	BA5	IOI: WR_H=1	12.849	1 201.304	1 214.153	69.68	58.9
19	BA6	OO: TAG_OUT abstracted	13.680	1 199.403	1 213.083	69.83	58.9
20	BA7	OO: VALID_H abstracted	**12.318**	**1 191.041**	**1 203.359**	**71.20**	**58.4**

Model: fiforx			Simulator 2		Simulated time: 1E09 ns		
All Methods				Simulation Time		Gain	
No	Method	Description	Compilation	Simulation	Total	Gain %	Ratio %
1	IM	Initial model	0.671	1 431.509	1 432.180	0.00	100.0
2	TR1	Process merge 1: 2 processes merged	0.671	1 400.644	1 401.315	2.20	97.8
3	TR2	Process merge 2: 2 processes merged	0.661	1 367.336	1 367.997	4.69	95.5
4	OA	7 signals replaced: PROSSIMO, PROSSIMO_ME, PNUMBYTE, PW, PR, P_DATO, P_TAG	0.661	849.151	849.812	68.53	59.3
5	OR1	Delete package reference: package FRAME	0.601	853.467	854.068	67.69	59.6
6	OR2	Constant instantiation: INDET, OVRUN	0.611	839.888	840.499	70.40	58.7
7	OR3	Function inline expansion: INC	0.611	838.886	839.497	70.60	58.6
8	TR3	Transformation if->case: 6 transformations	0.610	831.606	832.216	72.09	58.1
9	OR4	Delete package reference: package PFIFO Constant instantiation	0.631	831.376	832.007	72.14	58.1
10	DA1	Abstraction: std_logic_vector(7 to 0) to integer 2 ports: DAT_IN, DATA_OUT 1 signal: REG_DATO 1 shared variable: P_DATO	0.641	452.611	453.252	215.98	31.6
11	DA2	Abstraction: std_logic_vector(7 downto 0) to integer 2 ports: TAG_IN, TAG_OUT 1 signal: REG_TAG 1 shared variable: P_TAG	0.631	442.186	442.817	223.42	30.9
12	DA3	Abstraction: std_logic to bit 7 ports: CLK_RX, EN_HDLC_H, ICLK, RD_H, RES_L, WR_H, VALID_H	0.641	350.518	351.159	307.84	24.5
13	DA4	Abstraction: std_logic to bit 2 signals: HDLC_RES_L, HDLC_EV_H	0.621	353.512	354.133	304.42	24.7
14	BA1	IOI: EN_HDLC_H=1	0.631	349.894	350.525	308.58	24.5
15	BA2	IOI: RD_H=1	0.631	348.459	349.090	310.26	24.4
16	BA3	IOI: RES_L=1	0.631	346.916	347.547	312.08	24.3
17	BA4	IOI: TAG_IN=0	0.631	343.027	343.658	316.75	24.0
18	BA5	IOI: WR_H=1	0.621	341.300	341.921	318.86	23.9
19	BA6	OO: TAG_OUT abstracted	0.631	340.758	341.389	319.52	23.8
20	BA7	OO: VALID_H abstracted	**0.621**	**338.296**	**338.917**	**322.58**	**23.7**

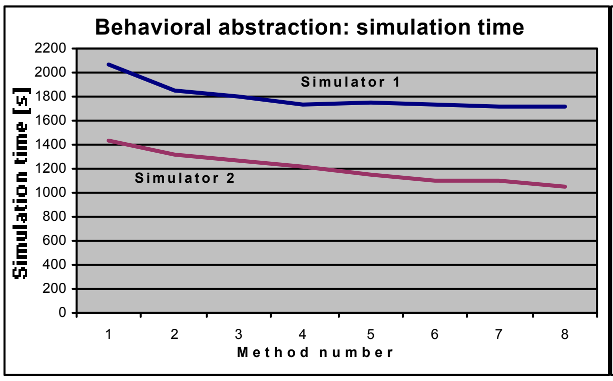


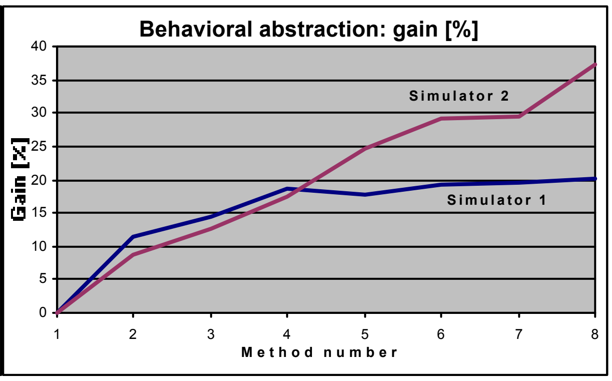


C.3.1.2 Behavioral abstraction

Model: fiforx			Simulator 1		Simulated time: 1E09 ns		
Behavioral Abstraction				Simulation Time		Gain	
No	Method	Description	Compilation	Simulation	Total	Gain %	Ratio %
1	IM	Initial model	24.095	2 036.027	2 060.122	0.00	100.0
2	BA1	IOI: EN_HDLC_H=1	13.249	1 835.471	1 848.720	11.44	89.7
3	BA2	IOI: RD_H=1	12.839	1 786.751	1 799.590	14.48	87.4
4	BA3	IOI: RES_L=1	12.849	1 721.386	1 734.235	18.79	84.2
5	BA4	IOI: TAG_IN=0	13.379	1 734.886	1 748.265	17.84	84.9
6	BA5	IOI: WR_H=1	12.718	1 713.345	1 726.063	19.35	83.8
7	BA6	OO: TAG_OUT abstracted	13.059	1 709.032	1 722.091	19.63	83.6
8	BA7	OO: VALID_H abstracted	**12.518**	**1 702.630**	**1 715.148**	**20.11**	**83.3**

Model: fiforx			Simulator 2		Simulated time: 1E09 ns		
Behavioral Abstraction				Simulation Time		Gain	
No	Method	Description	Compilation	Simulation	Total	Gain %	Ratio %
1	IM	Initial model	0.671	1 431.509	1 432.180	0.00	100.0
2	BA1	IOI: EN_HDLC_H=1	0.681	1 318.097	1 318.778	8.60	92.1
3	BA2	IOI: RD_H=1	0.671	1 272.317	1 272.988	12.51	88.9
4	BA3	IOI: RES_L=1	0.661	1 217.765	1 218.426	17.54	85.1
5	BA4	IOI: TAG_IN=0	0.661	1 148.479	1 149.140	24.63	80.2
6	BA5	IOI: WR_H=1	0.651	1 107.449	1 108.100	29.25	77.4
7	BA6	OO: TAG_OUT abstracted	0.661	1 105.121	1 105.782	29.52	77.2
8	BA7	OO: VALID_H abstracted	**0.651**	**1 043.431**	**1 044.082**	**37.17**	**72.9**

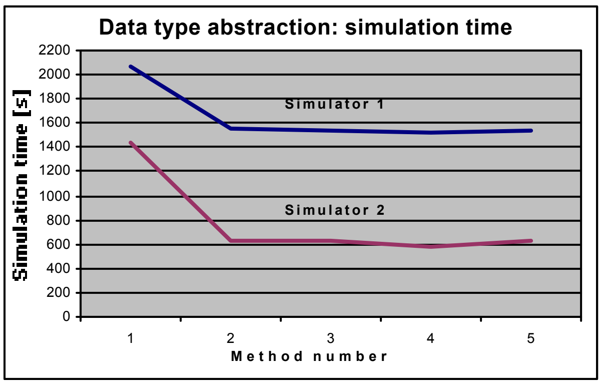


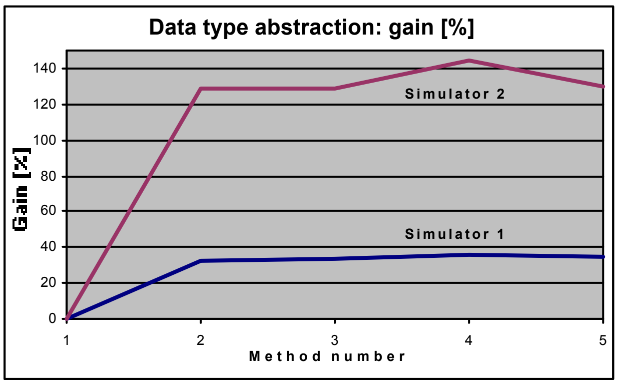


C.3.1.3 Data-type abstraction

Model: fiforx			**Simulator 1**		**Simulated time: 1E09 ns**		
Data-type Abstraction				**Simulation Time**		**Gain**	
No	**Method**	**Description**	**Compilation**	**Simulation**	**Total**	**Gain %**	**Ratio %**
1	IM	Initial model	24.095	2 036.027	2 060.122	0.00	100.0
2	DA1	Abstraction: std_logic_vector(7 to 0) to integer 2 ports: DAT_IN, DATA_OUT 1 signal: REG_DATO 1 shared variable: P_DATO	12.872	1 540.565	1 553.437	32.62	75.4
3	DA2	Abstraction: std_logic_vector(7 downto 0) to integer 2 ports: TAG_IN, TAG_OUT 1 signal: REG_TAG 1 shared variable: P_TAG	12.654	1 524.236	1 536.890	34.04	74.6
4	DA3	Abstraction: std_logic to bit 7 ports: CLK_RX, EN_HDLC_H, ICLK, RD_H, RES_L, WR_H, VALID_H	12.244	1 501.954	1 514.198	36.05	73.5
5	DA4	Abstraction: std_logic to bit 2 signals: HDLC_RES_L, HDLC_EV_H	**12.532**	**1 521.230**	**1 533.762**	**34.32**	**74.5**

Model: fiforx			**Simulator 2**		**Simulated time: 1E09 ns**		
Data-type Abstraction				**Simulation Time**		**Gain**	
No	**Method**	**Description**	**Compilation**	**Simulation**	**Total**	**Gain %**	**Ratio %**
1	IM	Initial model	0.671	1 431.509	1 432.180	0.00	100.0
2	DA1	Abstraction: std_logic_vector(7 to 0) to integer 2 ports: DAT_IN, DATA_OUT 1 signal: REG_DATO 1 shared variable: P_DATO	0.621	625.630	626.251	128.69	43.7
3	DA2	Abstraction: std_logic_vector(7 downto 0) to integer 2 ports: TAG_IN, TAG_OUT 1 signal: REG_TAG 1 shared variable: P_TAG	0.671	624.107	624.778	129.23	43.6
4	DA3	Abstraction: std_logic to bit 7 ports: CLK_RX, EN_HDLC_H, ICLK, RD_H, RES_L, WR_H, VALID_H	0.671	585.291	585.962	144.42	40.9
5	DA4	Abstraction: std_logic to bit 2 signals: HDLC_RES_L, HDLC_EV_H	**0.671**	**622.425**	**623.096**	**129.85**	**43.5**

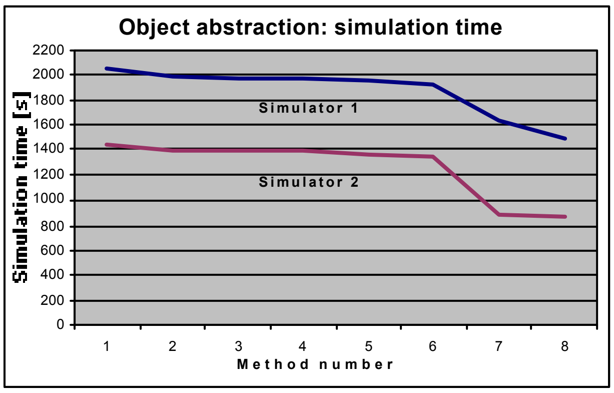


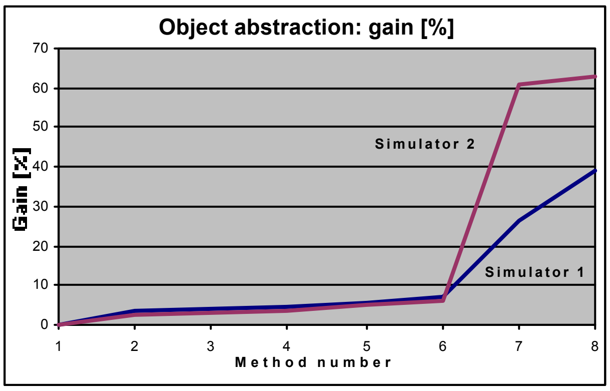


C.3.1.4 Object abstraction

Model: fiforx			Simulator 1		Simulated time: 1E09 ns		
Object Abstraction				Simulation Time		Gain	
No	Method	Description	Compilation	Simulation	Total	Gain %	Ratio %
1	IM	Initial model	24.095	2 036.027	2 060.122	0.00	100.0
2	OA1	PROSSIMO: 16 statements changed	12.848	1 975.360	1 988.208	3.62	96.5
3	OA2	PROSSIMO_ME: 4 statements changed	12.437	1 968.210	1 980.647	4.01	96.1
4	OA3	PNUMBYTE: 15 statements changed	12.418	1 955.251	1 967.669	4.70	95.5
5	OA4	PW: 12 statements changed	12.709	1 936.635	1 949.344	5.68	94.6
6	OA5	PR: 11 statements changed	12.698	1 907.293	1 919.991	7.30	93.2
7	OA6	P_DATO: 15 statements changed	12.668	1 616.684	1 629.352	26.44	79.1
8	OA7	P_TAG: 15 statements changed	**13.860**	**1 468.461**	**1 482.321**	**38.98**	**72.0**

Model: fiforx			Simulator 2		Simulated time: 1E09 ns		
Object Abstraction				Simulation Time		Gain	
No	Method	Description	Compilation	Simulation	Total	Gain %	Ratio %
1	IM	Initial model	0.671	1 431.509	1 432.180	0.00	100.0
2	OA1	PROSSIMO: 16 statements changed	0.671	1 394.285	1 394.956	2.67	97.4
3	OA2	PROSSIMO_ME: 4 statements changed	0.661	1 388.727	1 389.388	3.08	97.0
4	OA3	PNUMBYTE: 15 statements changed	0.681	1 384.351	1 385.032	3.40	96.7
5	OA4	PW: 12 statements changed	0.681	1 360.737	1 361.418	5.20	95.1
6	OA5	PR: 11 statements changed	0.671	1 349.801	1 350.472	6.05	94.3
7	OA6	P_DATO: 15 statements changed	0.671	889.549	890.220	60.88	62.2
8	OA7	P_TAG: 15 statements changed	**0.671**	**878.042**	**878.713**	**62.99**	**61.4**



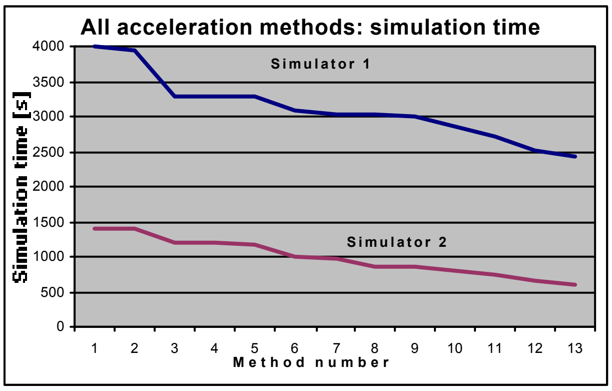


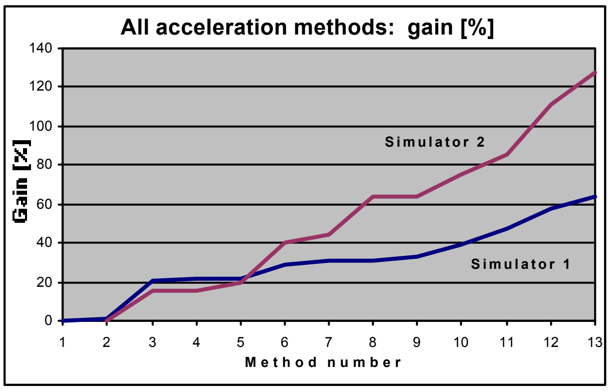
C.3.2 Model 2: rxbit

C.3.2.1 All methods

Model: rxbit			Simulator 1		Simulated time: 1E09 ns		
All Methods				Simulation Time		Gain	
No	Method	Description	Compilation	Simulation	Total	Gain %	Ratio %
1	IM	Initial model	28.271	3 958.392	3 986.663	0.00	100.0
2	TR1	Transformantion: if -> case	29.323	3 909.041	3 938.364	1.23	98.8
3	OA	7 signals replaced: PDATO, PI, PUNI, PNUMUNI, PONE, PROSSIMOCONT, PROSSIMO	32.016	3 257.834	3 289.850	21.18	82.5
4	OR1	Constant instantiation: FLAG, NA, ABRT	32.898	3 252.637	3 285.535	21.34	82.4
5	OR2	Function transfer from package to architecture: MUX_FOR_RXBIT	35.821	3 242.513	3 278.334	21.61	82.2
6	OR3	Function inline expansion: MUX_FOR_RXBIT 25 transformations	24.145	3 056.385	3 080.530	29.41	77.3
7	DA1	Abstraction: std_logic_vector (7 downto 0) to integer 2 ports: BIT_IN, DAT_OUT 2 signals: REG_DATO, ONE 2 shared variables: PDATO, PONE	9.544	3 032.060	3 041.604	31.07	76.3
8	DA2	Abstraction: std_logic to bit 4 ports: CLK_RX, EN_HDLC_H, RES_L, VALID_H	9.684	3 021.228	3 030.912	31.53	76.0
9	DA3	Abstraction: std_logic to bit 1 signal: UNI 1 shared variable: PUNI	15.913	2 983.077	2 998.990	32.93	75.2
10	BA1	IOI: EN_HDLC_H=1	9.844	2 842.404	2 852.248	39.77	71.5
11	BA2	IOI: RES_L=1	9.654	2 696.826	2 706.480	47.30	67.9
12	BA3	OO: TAG_OUT abstracted	9.333	2 511.440	2 520.773	58.15	63.2
13	BA4	OO: VALID_H abstracted	**9.404**	**2 418.490**	**2 427.894**	**64.20**	**60.9**

Model: rxbit			Simulator 2		Simulated time: 1E09 ns		
All Methods				Simulation Time		Gain	
No	Method	Description	Compilation	Simulation	Total	Gain %	Ratio %
1	IM	Initial model	0.641	1 396.508	1 397.149	0.00	100.0
2	TR1	Transformantion: if -> case	0.661	1 396.358	1 397.019	0.01	100.0
3	OA	7 signals replaced: PDATO, PI, PUNI, PNUMUNI, PONE, PROSSIMOCONT, PROSSIMO	0.641	1 205.654	1 206.295	15.82	86.3
4	OR1	Constant instantiation: FLAG, NA, ABRT	0.641	1 205.393	1 206.034	15.85	86.3
5	OR2	Function transfer from package to architecture: MUX_FOR_RXBIT	0.621	1 167.388	1 168.009	19.62	83.6
6	OR3	Function inline expansion: MUX_FOR_RXBIT 25 transformations	0.601	991.480	992.081	40.83	71.0
7	DA1	Abstraction: std_logic_vector (7 downto 0) to integer 2 ports: BIT_IN, DAT_OUT 2 signals: REG_DATO, ONE 2 shared variables: PDATO, PONE	0.591	967.900	968.491	44.26	69.3
8	DA2	Abstraction: std_logic to bit 4 ports: CLK_RX, EN_HDLC_H, RES_L, VALID_H	0.591	848.476	849.067	64.55	60.8
9	DA3	Abstraction: std_logic to bit 1 signal: UNI 1 shared variable: PUNI	0.601	849.989	850.590	64.26	60.9
10	BA1	IOI: EN_HDLC_H=1	0.581	798.535	799.116	74.84	57.2
11	BA2	IOI: RES_L=1	0.591	753.392	753.983	85.30	54.0
12	BA3	OO: TAG_OUT abstracted	0.590	662.862	663.452	110.59	47.5
13	BA4	OO: VALID_H abstracted	**0.581**	**612.486**	**613.067**	**127.89**	**43.9**

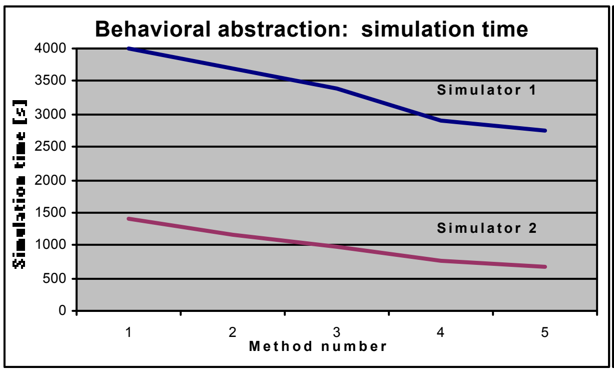


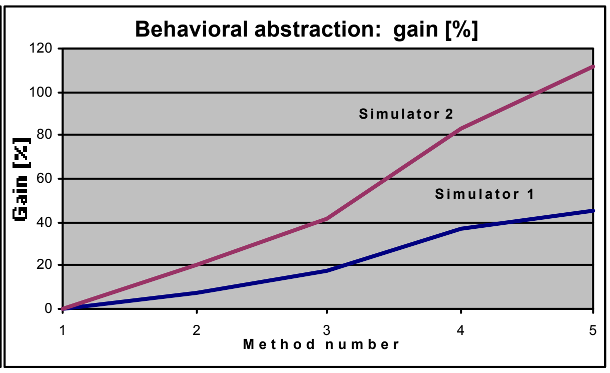


C.3.2.2 Behavioral abstraction

Model: rxbit			Simulator 1		Simulated time: 1E09 ns		
Behavioral Abstraction				Simulation Time		Gain	
No	Method	Description	Compilation	Simulation	Total	Gain %	Ratio %
1	IM	Initial model	28.271	3 958.392	3 986.663	0.00	100.0
2	BA1	IOI: EN_HDLC_H=1	9.243	3 691.215	3 700.458	7.73	92.8
3	BA2	IOI: RES_L=1	9.333	3 387.406	3 396.739	17.37	85.2
4	BA3	OO: TAG_OUT abstracted	9.874	2 904.267	2 914.141	36.80	73.1
5	BA4	OO: VALID_H abstracted	**9.073**	**2 742.856**	**2 751.929**	**44.87**	**69.0**

Model: rxbit			Simulator 2		Simulated time: 1E09 ns		
Behavioral Abstraction				Simulation Time		Gain	
No	Method	Description	Compilation	Simulation	Total	Gain %	Ratio %
1	IM	Initial model	0.641	1 396.358	1 396.999	0.00	100.0
2	BA1	IOI: EN_HDLC_H=1	0.631	1 160.338	1 160.969	20.33	83.1
3	BA2	IOI: RES_L=1	0.621	985.882	986.503	41.61	70.6
4	BA3	OO: TAG_OUT abstracted	0.631	762.124	762.755	83.15	54.6
5	BA4	OO: VALID_H abstracted	**0.620**	**659.721**	**660.341**	**111.56**	**47.3**

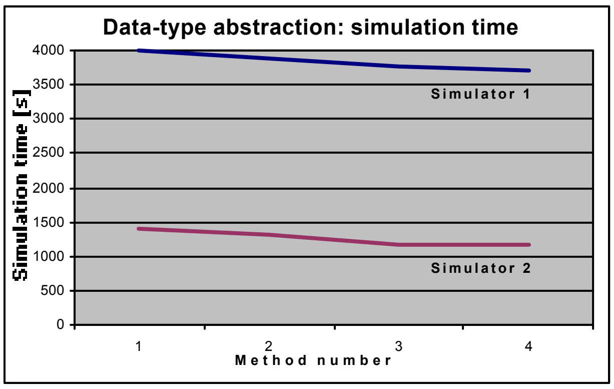


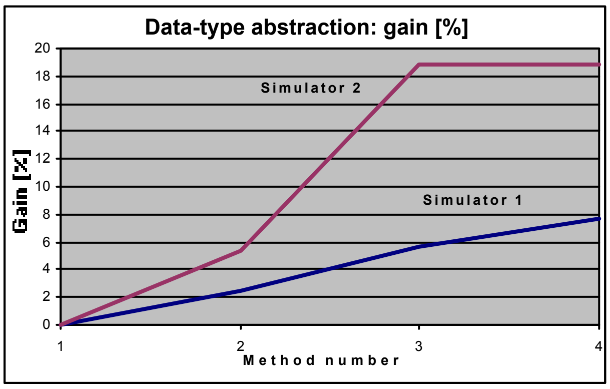


C.3.2.3 Data-type abstraction

Model: rxbit			Simulator 1		Simulated time: 1E09 ns		
Data-type Abstraction				Simulation Time		Gain	
No	Methoc	Description	Compilation	Simulation	Total	Gain %	Ratio %
1	IM	Initial model	28.271	3 958.392	3 986.663	0.00	100.0
2	DA1	Abstraction: std_logic_vector (7 downto 0) to integer 2 ports: BIT_IN, DAT_OUT 2 signals: REG_DATO, ONE 2 shared variables: PDATO, PONE	10.375	3 880.300	3 890.675	2.47	97.6
3	DA2	Abstraction: std_logic to bit 4 ports: CLK_RX, EN_HDLC_H, RES_L, VALID_H	9.694	3 765.868	3 775.562	5.59	94.7
4	DA3	Abstraction: std_logic to bit 1 signal: UNI 1 shared variable: PUNI	**9.964**	**3 691.275**	**3 701.239**	**7.71**	**92.8**

Model: rxbit			Simulator 2		Simulated time: 1E09 ns		
Data-type Abstraction				Simulation Time		Gain	
No	Methoc	Description	Compilation	Simulation	Total	Gain %	Ratio %
1	IM	Initial model	0.641	1 396.358	1 396.999	0.00	100.0
2	DA1	Abstraction: std_logic_vector (7 downto 0) to integer 2 ports: BIT_IN, DAT_OUT 2 signals: REG_DATO, ONE 2 shared variables: PDATO, PONE	0.631	1 324.549	1 325.180	5.42	94.9
3	DA2	Abstraction: std_logic to bit 4 ports: CLK_RX, EN_HDLC_H, RES_L, VALID_H	0.641	1 174.560	1 175.201	18.87	84.1
4	DA3	Abstraction: std_logic to bit 1 signal: UNI 1 shared variable: PUNI	**0.631**	**1 174.890**	**1 175.521**	**18.84**	**84.1**

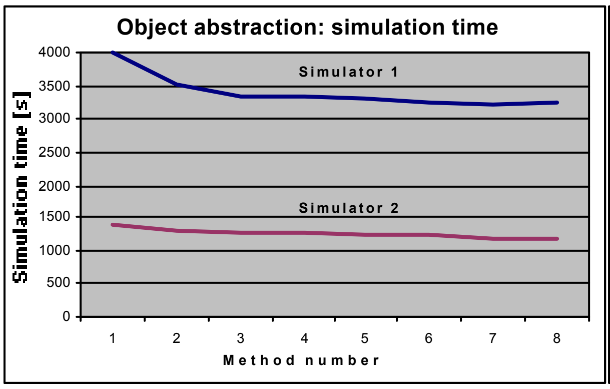


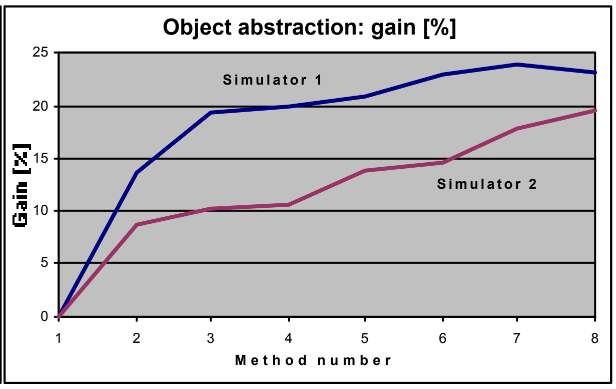


C.3.2.4 Object abstraction

Model: rxbit			Simulator 1		Simulated time: 1E09 ns		
Object Abstraction				Simulation Time		Gain	
No	Methoı	Description	Compilation	Simulation	Total	Gain %	Ratio %
1	IM	Initial model	28.271	3 958.392	3 986.663	0.00	100.0
2	OA1	PDATO: 27 statements changed	10.004	3 497.439	3 507.443	13.66	88.0
3	OA2	PI: 27 statements changed	9.243	3 331.701	3 340.944	19.33	83.8
4	OA3	PUNI: 7 statements changed	9.874	3 313.875	3 323.749	19.94	83.4
5	OA4	PNUMUNI: 7 statements changed	9.073	3 289.540	3 298.613	20.86	82.7
6	OA5	PONE: 27 statements changed	10.075	3 233.960	3 244.035	22.89	81.4
7	OA6	PROSSIMOCONT: 7 statements changed	8.863	3 210.787	3 219.650	23.82	80.8
8	OA7	PROSSIMO: 27 statements changed	**9.764**	**3 230.606**	**3 240.370**	**23.03**	**81.3**

Model: rxbit			Simulator 2		Simulated time: 1E09 ns		
Object Abstraction				Simulation Time		Gain	
No	Methoc	Description	Compilation	Simulation	Total	Gain %	Ratio %
1	IM	Initial model	0.641	1 396.358	1 396.999	0.00	100.0
2	OA1	PDATO: 27 statements changed	0.631	1 285.699	1 286.330	8.60	92.1
3	OA2	PI: 27 statements changed	0.621	1 267.652	1 268.273	10.15	90.8
4	OA3	PUNI: 7 statements changed	0.621	1 263.116	1 263.737	10.55	90.5
5	OA4	PNUMUNI: 7 statements changed	0.640	1 225.341	1 225.981	13.95	87.8
6	OA5	PONE: 27 statements changed	0.631	1 217.591	1 218.222	14.68	87.2
7	OA6	PROSSIMOCONT: 7 statements changed	0.621	1 184.804	1 185.425	17.85	84.9
8	OA7	PROSSIMO: 27 statements changed	**0.621**	**1 167.399**	**1 168.020**	**19.60**	**83.6**



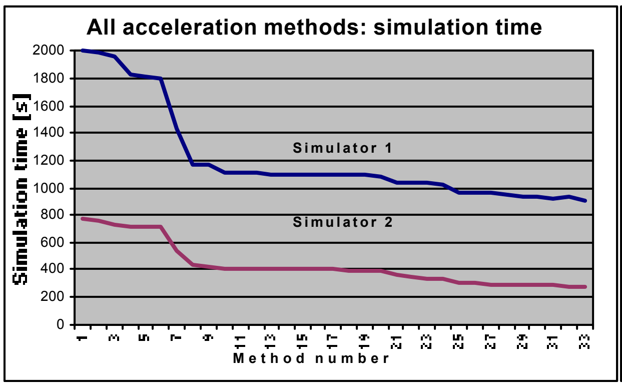


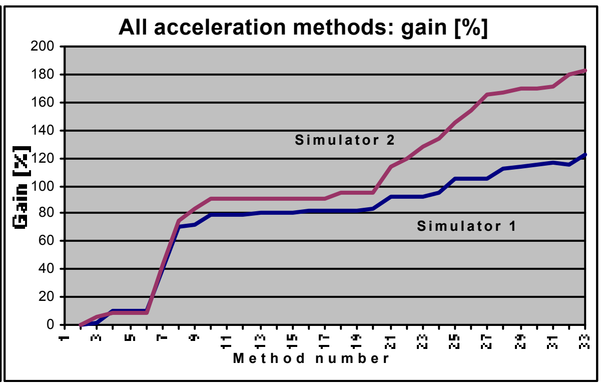
C.3.3 Model 3: bus_interface_rx

C.3.3.1 All methods

Model: bus_interface_rx			Simulator 1		Simulated time: 1E09 ns		
All Methods				Simulation Time		Gain	
No	Method	Description	Compilation	Simulation	Total	Gain %	Ratio %
1	IM	Initial model	16.974	1 976.362	1 993.336	0.00	100.0
2	TR1	Transformation: if -> case	15.933	1 975.210	1 991.143	0.11	99.9
3	TR2	Process merge: 2 processes merged	14.661	1 944.796	1 959.457	1.73	98.3
4	OA	3 signals abstracted: PBUFDATO, PNEWADR, PROSSIMO_GB	15.473	1 804.675	1 820.148	9.52	91.3
5	OR1	Delete package reference: 1 constant DIM_ADD_EXT moved	12.698	1 791.696	1 804.394	10.47	90.5
6	OR2	Constant instantiation: DIM_ADD_EXT:=23	12.438	1 789.564	1 802.002	10.62	90.4
7	DA1	Abstraction: std_logic_vector(23 downto 0) to integer 2 ports: ADR, POINTER 2 signals: NEWADR, PNEWADR	12.588	1 417.989	1 430.577	39.34	71.8
8	DA2	Abstraction: std_logic_vector (31 downto 0) to integer 2 ports: DATO_TO_SEND, DATO 2 signals: BUFDATO, PBUFDATO	12.478	1 161.871	1 174.349	69.74	58.9
9	DA3	Abstraction: std_logic_vector (1 downto 0) to integer 1 port: SIZ	12.138	1 155.261	1 167.399	70.75	58.6
10	DA4	Abstraction: std_logic_vector (1 downto 0) to integer 1 port: TT	11.566	1 108.674	1 120.240	77.94	56.2
11	DA5	Abstraction: std_logic to bit 1 port: BG_L	12.057	1 102.335	1 114.392	78.87	55.9
12	DA6	Abstraction: std_logic to bit 1 port: CORRECT_L	11.236	1 102.416	1 113.652	78.99	55.9
13	TR3	Removal of unused signal: EN_DMA_H	12.228	1 092.912	1 105.140	80.37	55.4
14	DA7	Abstraction: std_logic to bit 1 port: ICLK	11.977	1 092.130	1 104.107	80.54	55.4
15	DA8	Abstraction: std_logic to bit 1 port: INDOK_H	12.017	1 091.930	1 103.947	80.56	55.4
16	DA9	Abstraction: std_logic to bit 1 port: LAST_H	12.899	1 087.544	1 100.443	81.14	55.2
17	DA10	Abstraction: std_logic to bit 3 ports: REQ_H, RES_L, TA_L	11.607	1 087.203	1 098.810	81.41	55.1
18	DA11	Abstraction: std_logic to bit 1 port: BR_L	12.207	1 084.550	1 096.757	81.75	55.0
19	DA12	Abstraction: std_logic to bit 1 port: DATOK_H	11.687	1 086.482	1 098.169	81.51	55.1
20	DA13	Abstraction: std_logic to bit 1 port: RIND_H	12.128	1 081.816	1 093.944	82.22	54.9
21	BA1	IOI: BG_L=1	11.777	1 029.961	1 041.738	91.35	52.3
22	BA2	IOI: CORRECT_L=1	11.686	1 029.380	1 041.066	91.47	52.2
23	BA3	IOI: INDOK_H=1	12.348	1 027.337	1 039.685	91.72	52.2
24	BA4	IOI: LAST_H=1	11.587	1 011.695	1 023.282	94.80	51.3
25	BA5	IOI: RES_L=1	12.127	956.315	968.442	105.83	48.6
26	BA6	IOI: REQ_H=1	11.747	954.793	966.540	106.23	48.5
27	BA7	IOI: TA_L=1	12.378	954.763	967.141	106.11	48.5
28	BA8	OO: BB_L abstracted	11.697	925.911	937.608	112.60	47.0
29	BA9	OO: ADR abstracted	11.056	920.103	931.159	114.07	46.7
30	BA10	OO: SIZ and TT abstracted	12.308	914.725	927.033	115.02	46.5
31	BA11	OO: BR_L abstracted	11.757	907.345	919.102	116.88	46.1
32	BA12	OO: RIND_H abstracted	11.076	911.932	923.008	115.96	46.3
33	BA13	OO: DATOK_H, TS_L abstracted	**12.188**	**883.971**	**896.159**	**122.43**	**45.0**

Model: bus_interface_rx			**Simulator 2**		**Simulated time: 1E09 ns**		
All Methods				**Simulation Time**		**Gain**	
No	**Method**	**Description**	**Compilation**	**Simulation**	**Total**	**Gain %**	**Ratio %**
1	IM	Initial model	0.661	765.141	765.802	0.00	100.0
2	TR1	Transformation: if -> case	0.681	759.902	760.583	0.69	99.3
3	TR2	Process merge: 2 processes merged	0.671	727.656	728.327	5.15	95.1
4	OA	3 signals abstracted: PBUFDATO, PNEWADR, PROSSIMO_GB	0.701	707.438	708.139	8.14	92.5
5	OR1	Delete package reference: 1 constant DIM_ADD_EXT moved	0.631	706.035	706.666	8.37	92.3
6	OR2	Constant instantiation: DIM_ADD_EXT:=23	0.640	704.072	704.712	8.67	92.0
7	DA1	Abstraction: std_logic_vector(23 downto 0) to integer 2 ports: ADR, POINTER 2 signals: NEWADR, PNEWADR	0.621	537.082	537.703	42.42	70.2
8	DA2	Abstraction: std_logic_vector (31 downto 0) to integer 2 ports: DATO_TO_SEND, DATO 2 signals: BUFDATO, PBUFDATO	0.621	438.350	438.971	74.45	57.3
9	DA3	Abstraction: std_logic_vector (1 downto 0) to integer 1 port: SIZ	0.621	416.619	417.240	83.54	54.5
10	DA4	Abstraction: std_logic_vector (1 downto 0) to integer 1 port: TT	0.621	403.000	403.621	89.73	52.7
11	DA5	Abstraction: std_logic to bit 1 port: BG_L	0.631	402.208	402.839	90.10	52.6
12	DA6	Abstraction: std_logic to bit 1 port: CORRECT_L	0.631	401.888	402.519	90.25	52.6
13	TR3	Removal of unused signal: EN_DMA_H	0.621	401.748	402.369	90.32	52.5
14	DA7	Abstraction: std_logic to bit 1 port: ICLK	0.631	401.608	402.239	90.38	52.5
15	DA8	Abstraction: std_logic to bit 1 port: INDOK_H	0.631	401.667	402.298	90.36	52.5
16	DA9	Abstraction: std_logic to bit 1 port: LAST_H	0.631	401.647	402.278	90.37	52.5
17	DA10	Abstraction: std_logic to bit 3 ports: REQ_H, RES_L, TA_L	0.631	401.618	402.249	90.38	52.5
18	DA11	Abstraction: std_logic to bit 1 port: BR_L	0.621	392.614	393.235	94.74	51.3
19	DA12	Abstraction: std_logic to bit 1 port: DATOK_H	0.631	393.166	393.797	94.47	51.4
20	DA13	Abstraction: std_logic to bit 1 port: RIND_H	0.630	391.042	391.672	95.52	51.1
21	BA1	IOI: BG_L=1	0.621	355.622	356.243	114.97	46.5
22	BA2	IOI: CORRECT_L=1	0.611	347.589	348.200	119.93	45.5
23	BA3	IOI: INDOK_H=1	0.621	333.399	334.020	129.27	43.6
24	BA4	IOI: LAST_H=1	0.621	326.690	327.311	133.97	42.7
25	BA5	IOI: RES_L=1	0.621	310.566	311.187	146.09	40.6
26	BA6	IOI: REQ_H=1	0.611	300.783	301.394	154.09	39.4
27	BA7	IOI: TA_L=1	0.611	287.914	288.525	165.42	37.7
28	BA8	OO: BB_L abstracted	0.601	285.831	286.432	167.36	37.4
29	BA9	OO: ADR abstracted	0.611	283.257	283.868	169.77	37.1
30	BA10	OO: SIZ and TT abstracted	0.611	283.027	283.638	169.99	37.0
31	BA11	OO: BR_L abstracted	0.611	282.056	282.667	170.92	36.9
32	BA12	OO: RIND_H abstracted	0.601	273.042	273.643	179.85	35.7
33	BA13	OO: DATOK_H, TS_L abstracted	**0.591**	**269.648**	**270.239**	**183.38**	**35.3**

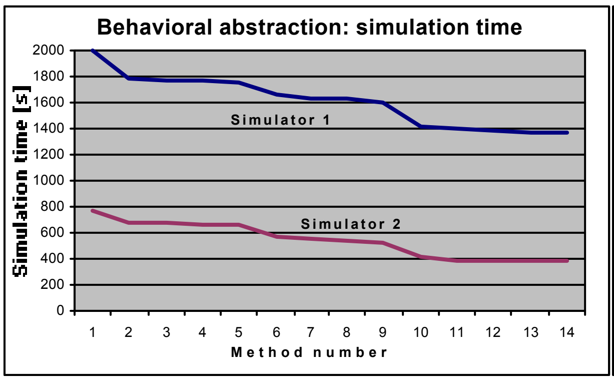


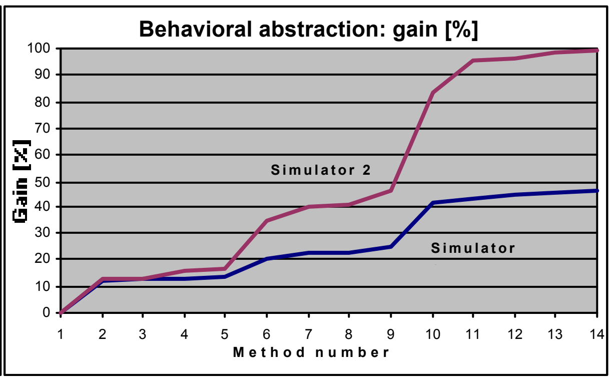


C.3.3.2 Behavioral abstraction

Model: bus_interface_rx			Simulator 1		Simulated time: 1E09 ns		
Behavioral Abstraction				Simulation Time		Gain	
No	Method	Description	Compilation	Simulation	Total	Gain %	Ratio %
1	IM	Initial model	16.974	1 976.362	1 993.336	0.00	100.0
2	BA1	IOI: BG_L=1	12.698	1 764.337	1 777.035	12.17	89.1
3	BA2	IOI: CORRECT_L=1	12.188	1 760.561	1 772.749	12.44	88.9
4	BA3	IOI: INDOK_H=1	12.278	1 757.637	1 769.915	12.62	88.8
5	BA4	IOI: LAST_H=1	12.298	1 742.786	1 755.084	13.57	88.0
6	BA5	IOI: RES_L=1	12.788	1 643.003	1 655.791	20.39	83.1
7	BA6	IOI: REQ_H=1	12.879	1 617.656	1 630.535	22.25	81.8
8	BA7	IOI: TA_L=1	11.988	1 611.177	1 623.165	22.81	81.4
9	BA8	OO: BB_L abstracted	13.329	1 583.377	1 596.706	24.84	80.1
10	BA9	OO: ADR abstracted	12.708	1 397.279	1 409.987	41.37	70.7
11	BA10	OO: SIZ and TT abstracted	12.999	1 382.228	1 395.227	42.87	70.0
12	BA11	OO: BR_L abstracted	13.129	1 369.019	1 382.148	44.22	69.3
13	BA12	OO: RIND_H abstracted	12.789	1 358.994	1 371.783	45.31	68.8
14	BA13	OO: DATOK_H, TS_L abstracted	**12.318**	**1 350.772**	**1 363.090**	**46.24**	**68.4**

Model: bus_interface_rx			Simulator 2		Simulated time: 1E09 ns		
Behavioral Abstraction				Simulation Time		Gain	
No	Method	Description	Compilation	Simulation	Total	Gain %	Ratio %
1	IM	Initial model	0.661	765.141	765.802	0.00	100.0
2	BA1	IOI: BG_L=1	0.671	680.308	680.979	12.46	88.9
3	BA2	IOI: CORRECT_L=1	0.661	677.755	678.416	12.88	88.6
4	BA3	IOI: INDOK_H=1	0.651	659.168	659.819	16.06	86.2
5	BA4	IOI: LAST_H=1	0.651	655.243	655.894	16.76	85.6
6	BA5	IOI: RES_L=1	0.661	567.917	568.578	34.69	74.2
7	BA6	IOI: REQ_H=1	0.651	545.805	546.456	40.14	71.4
8	BA7	IOI: TA_L=1	0.661	544.112	544.773	40.57	71.1
9	BA8	OO: BB_L abstracted	0.651	523.312	523.963	46.16	68.4
10	BA9	OO: ADR abstracted	0.651	416.189	416.840	83.72	54.4
11	BA10	OO: SIZ and TT abstracted	0.641	390.422	391.063	95.83	51.1
12	BA11	OO: BR_L abstracted	0.641	390.302	390.943	95.89	51.1
13	BA12	OO: RIND_H abstracted	0.631	385.585	386.216	98.28	50.4
14	BA13	OO: DATOK_H, TS_L abstracted	**0.621**	**384.062**	**384.683**	**99.07**	**50.2**

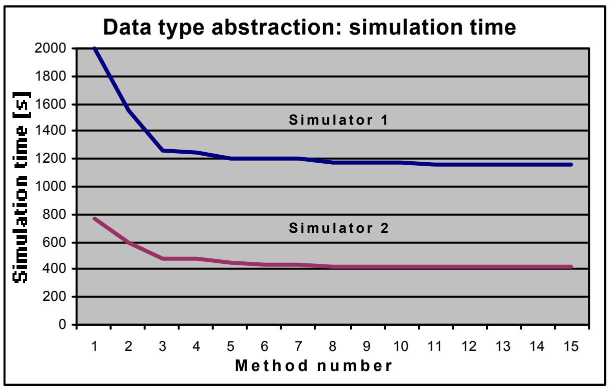


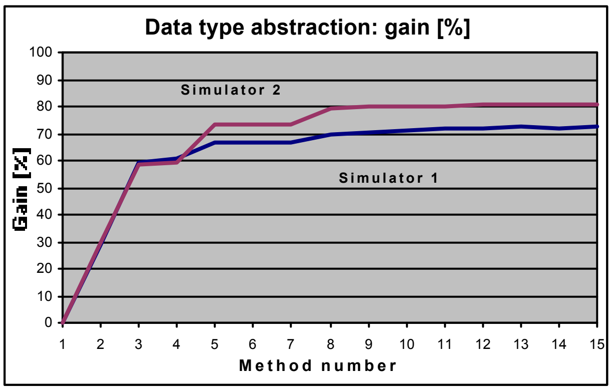


C.3.3.3 Data-type abstraction

Model: bus_interface_rx			Simulator 1		Simulated time: 1E09 ns		
Data-type Abstraction				Simulation Time		Gain	
No	Method	Description	Compilation	Simulation	Total	Gain %	Ratio %
1	IM	Initial model	16.974	1 976.362	1 993.336	0.00	100.0
2	DA1	Abstraction: std_logic_vector(23 downto 0) to integer 2 ports: ADR, POINTER 2 signals: NEWADR, PNEWADR	12.919	1 532.944	1 545.863	28.95	77.6
3	DA2	Abstraction: std_logic_vector (31 downto 0) to integer 2 ports: DATO_TO_SEND, DATO 2 signals: BUFDATO, PBUFDATO	12.007	1 242.267	1 254.274	58.92	62.9
4	DA3	Abstraction: std_logic_vector (1 downto 0) to integer 1 port: SIZ	12.658	1 228.447	1 241.105	60.61	62.3
5	DA4	Abstraction: std_logic_vector (1 downto 0) to integer 1 port: TT	11.837	1 186.477	1 198.314	66.35	60.1
6	DA5	Abstraction: std_logic to bit 1 port: BG_L	12.187	1 185.875	1 198.062	66.38	60.1
7	DA6	Abstraction: std_logic to bit 1 port: CORRECT_L	11.537	1 184.273	1 195.810	66.69	60.0
8	TR3	Removal of unused signal: EN_DMA_H	12.067	1 161.160	1 173.227	69.90	58.9
9	DA7	Abstraction: std_logic to bit 1 port: ICLK	12.227	1 159.978	1 172.205	70.05	58.8
10	DA8	Abstraction: std_logic to bit 1 port: INDOK_H	12.809	1 154.279	1 167.088	70.80	58.5
11	DA9	Abstraction: std_logic to bit 1 port: LAST_H	12.027	1 149.202	1 161.229	71.66	58.3
12	DA10	Abstraction: std_logic to bit 3 ports: REQ_H, RES_L, TA_L	12.838	1 148.381	1 161.219	71.66	58.3
13	DA11	Abstraction: std_logic to bit 1 port: BR_L	12.037	1 144.586	1 156.623	72.34	58.0
14	DA12	Abstraction: std_logic to bit 1 port: DATOK_H	12.538	1 145.798	1 158.336	72.09	58.1
15	DA13	Abstraction: std_logic to bit 1 port: RIND_H	**12.498**	**1 144.225**	**1 156.723**	**72.33**	**58.0**

Model: bus_interface_rx			Simulator 2		Simulated time: 1E09 ns		
Data-type Abstraction				Simulation Time		Gain	
No	Method	Description	Compilation	Simulation	Total	Gain %	Ratio %
1	IM	Initial model	0.661	765.141	765.802	0.00	100.0
2	DA1	Abstraction: std_logic_vector(23 downto 0) to integer 2 ports: ADR, POINTER 2 signals: NEWADR, PNEWADR	0.661	590.569	591.230	29.53	77.2
3	DA2	Abstraction: std_logic_vector (31 downto 0) to integer 2 ports: DATO_TO_SEND, DATO 2 signals: BUFDATO, PBUFDATO	0.661	481.953	482.614	58.68	63.0
4	DA3	Abstraction: std_logic_vector (1 downto 0) to integer 1 port: SIZ	0.661	480.061	480.722	59.30	62.8
5	DA4	Abstraction: std_logic_vector (1 downto 0) to integer 1 port: TT	0.661	441.765	442.426	73.09	57.8
6	DA5	Abstraction: std_logic to bit 1 port: BG_L	0.651	441.265	441.916	73.29	57.7
7	DA6	Abstraction: std_logic to bit 1 port: CORRECT_L	0.651	440.573	441.224	73.56	57.6
8	TR3	Removal of unused signal: EN_DMA_H	0.661	425.762	426.423	79.59	55.7
9	DA7	Abstraction: std_logic to bit 1 port: ICLK	0.661	425.342	426.003	79.76	55.6
10	DA8	Abstraction: std_logic to bit 1 port: INDOK_H	0.651	425.111	425.762	79.87	55.6
11	DA9	Abstraction: std_logic to bit 1 port: LAST_H	0.661	424.290	424.951	80.21	55.5
12	DA10	Abstraction: std_logic to bit 3 ports: REQ_H, RES_L, TA_L	0.661	423.349	424.010	80.61	55.4
13	DA11	Abstraction: std_logic to bit 1 port: BR_L	0.661	423.259	423.920	80.65	55.4
14	DA12	Abstraction: std_logic to bit 1 port: DATOK_H	0.661	423.288	423.949	80.64	55.4
15	DA13	Abstraction: std_logic to bit 1 port: RIND_H	**0.650**	**423.179**	**423.829**	**80.69**	**55.3**

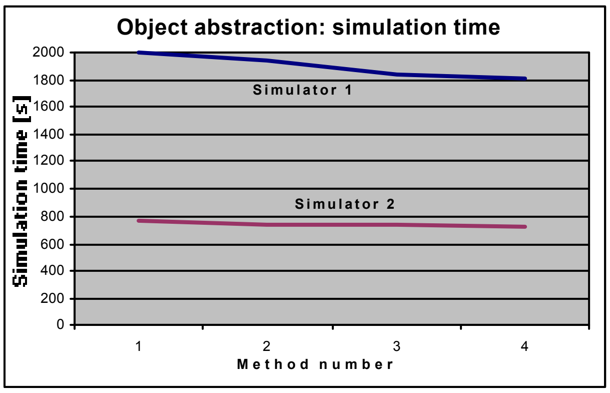


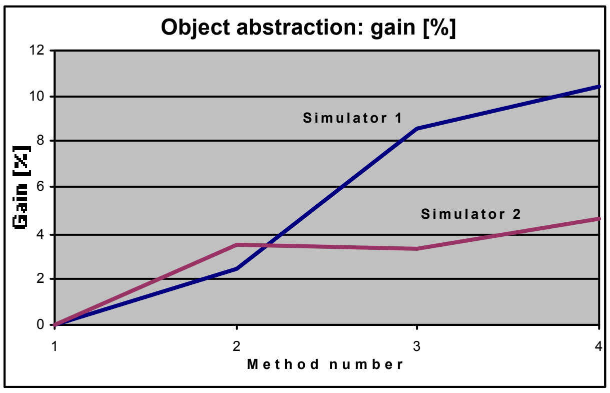


C.3.3.4 Object abstraction

Model: bus_interface_rx			Simulator 1		Simulated time: 1E09 ns		
Object Abstraction				Simulation Time		Gain	
No	Method	Description	Compilation	Simulation	Total	Gain %	Ratio %
1	IM	Initial model	16.974	1 976.362	1 993.336	0.00	100.0
2	OA1	PBUFDATO: 2 statements changed	13.179	1 931.828	1 945.007	2.48	97.6
3	OA2	PNEWADR: 20 statements changed	13.088	1 823.422	1 836.510	8.54	92.1
4	OA3	PROSSIMO_GB: 20 statements changed	**12.247**	**1 793.469**	**1 805.716**	**10.39**	**90.6**

Model: bus_interface_rx			Simulator 2		Simulated time: 1E09 ns		
Object Abstraction				Simulation Time		Gain	
No	Method	Description	Compilation	Simulation	Total	Gain %	Ratio %
1	IM	Initial model	0.661	765.141	765.802	0.00	100.0
2	OA1	PBUFDATO: 2 statements changed	0.671	738.731	739.402	3.57	96.6
3	OA2	PNEWADR: 20 statements changed	0.661	740.605	741.266	3.31	96.8
4	OA3	PROSSIMO_GB: 20 statements changed	**0.661**	**731.152**	**731.813**	**4.64**	**95.6**





RÉSUMÉ

La complexité des systèmes électroniques, due au progrès de la technologie microélectronique, nécessite une augmentation correspondante de la productivité des méthodes de conception et de vérification. Une faible performance de la simulation est un des obstacles majeurs à une conception rapide et peu coûteuse de produits de haute qualité. Dans cette thèse nous proposons des méthodes pour améliorer la performance d'une simulation dirigée par événements ou par horloge de modèles décrits en langages de description de matériel.

Nous présentons d'abord les méthodes automatisées d'optimisation et de transformation de modèles VHDL, pour l'accélérer la simulation dirigée par événements. Elles sont fondées sur une analyse précise de la performance en simulation de diverses constructions du langage VHDL, et permettent de convertir le modèle initial en un autre modèle plus efficace, tout en garantissant l'invariance de son comportement. D'autres techniques d'accélération utilisent l'abstraction du modèle : abstraction comportementale, de types de données ou d'objets et permettent de supprimer du modèle des détails inutiles dans le cas d'une simulation particulière. Des outils prototype compatibles avec les simulateurs existants sont développés.

Pour améliorer l'efficacité de la simulation dirigée par horloge, nous introduisons une représentation de la fonctionnalité du système par graphes de décision de haut niveau (DDs). Diverses formes de DDs – graphes vectoriels, compressés ou non et graphes orientés registres – sont définis pour optimiser une représentation du système sur plusieurs niveaux d'abstraction. De plus, de nouveaux algorithmes plus rapides d'évaluation des réseaux de DDs sont développés. Ils emploient, seuls ou en combinaison, les deux techniques de simulation : la technique dirigée par événements et l'évaluation rétrogradée. L'ensemble des prototypes fondé sur ces méthodes permet d'obtenir un gain de performances prometteur par rapport aux outils commerciaux.

ABSTRACT

The growing complexity of electronic systems stimulated by IC's technology progress demands a corresponding growth of the productivity of design and verification methods. The low performance of simulation is one of the obstacles preventing a delivery of high quality products in a short time and at a low cost. In this thesis we propose methods aimed at improving the simulation performance of event-driven and cycle-based simulation techniques of HDL models.

Automated optimization and transformation methods of VHDL models, developed to accelerate the event-driven simulation are presented first. These methods, based on the precise measure of simulation performance of VHDL language constructs, convert an initial VHDL model into another functionally equivalent VHDL model offering a better simulation performance. Other acceleration techniques, denoted as abstraction methods, focus on removing from a model all irrelevant details of its behavior or structure. We propose three such methods: behavioral abstraction, data-type abstraction and object abstraction. Prototype tools compatible with currently used simulators are developed to support automatic application of these methods.

For the purpose of improving of the cycle-based simulation efficiency a representation of a digital system by high-level decision diagrams (DDs) is introduced. Some forms of DDs: vector decision diagrams, compressed or not (VDDs and CVDDs) and register-oriented DDs are developed to optimize the representation of a system at different levels of abstraction. In addition, new simulation algorithms of a network of DDs are proposed to further accelerate the simulation execution. These algorithms implement separately or in combination two simulation techniques: the event-driven and back-tracing techniques. The prototype tools are build, based on the DDs simulator, which allow to efficiently simulate various types of decision diagrams with appropriate simulation algorithms.

DISCIPLINE Microélectronique

MOTS-CLES performance de simulation, modélisation des circuits intégrés, conception et simulation des circuits intégrés, méthodes de vérification, langages de description de matériel, HDL

ISBN 2-913329-52-7 Paperback
ISBN 2-913329-53-5 Electronic format