

UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL  
INSTITUTO DE INFORMÁTICA  
PROGRAMA DE PÓS-GRADUAÇÃO EM MICROELETRÔNICA

SAMUEL NASCIMENTO PAGLIARINI

**VEasy: a Tool Suite Towards the Functional  
Verification Challenges**

Thesis presented in partial fulfillment  
of the requirements for the degree of  
Master of Microelectronics

Prof. Dr. Fernanda Lima Kastensmidt  
Advisor

Porto Alegre, april 2011

## CIP – CATALOGING-IN-PUBLICATION

Pagliarini, Samuel Nascimento

VEasy: a Tool Suite Towards the Functional Verification Challenges / Samuel Nascimento Pagliarini. – Porto Alegre: PG-MICRO da UFRGS, 2011.

121 f.: il.

Thesis (Master) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Microeletrônica, Porto Alegre, BR–RS, 2011. Advisor: Fernanda Lima Kastensmidt.

1. Functional verification. 2. Simulation. 3. Coverage metrics. 4. Automation. I. Kastensmidt, Fernanda Lima. II. Título.

UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL

Reitor: Prof. Carlos Alexandre Netto

Pró-Reitor de Coordenação Acadêmica: Prof. Rui Vicente Oppermann

Pró-Reitora de Pós-Graduação: Prof. Aldo Bolten Lucion

Diretor do Instituto de Informática: Prof. Flávio Rech Wagner

Coordenador do PGMICRO: Prof. Ricardo Augusto da Luz Reis

Bibliotecária-chefe do Instituto de Informática: Beatriz Regina Bastos Haro

*“In the days of my youth  
I was told what it means to be a man  
Now I’ve reached that age  
I’ve tried to do all those things the best I can  
No matter how I try  
I find my way into the same old jam”*

— L. Z.



## AGRADECIMENTOS

Foram dias difíceis. Esta dissertação me tomou um tempo absurdo, mesmo tendo sido planejada do primeiro ao último item. Inicialmente o texto que eu considerava acabado tinha 90 páginas. Alguns dias depois 100. Após uma revisão foram mais 15 páginas. Cada linha parece ter sido escrita pelo menos 2, 3, 4 vezes... nunca satisfeito com o teor ou qualidade de uma frase. De certa forma, foi uma aventura. E que agora chega ao fim, finalmente.

Mas este texto não serve para que eu relate o processo, e sim para que eu torne públicos os meus agradecimentos. Inicialmente, acho que o agradecimento mais válido de todos é para minha família: meu pai, Deraldo, minha mãe, Mariângela e minha irmã, Sâmara. Talvez eles não entendam nada que aqui esteja escrito, mas pouco importa. Esta dissertação só existe através deles. Da mesma forma que a minha graduação só foi possível através deles. Portanto, obrigado a vocês.

Durante o processo de escrita e desenvolvimento desta tese ninguém se fez mais presente que minha namorada, Aline. Abusei de sua paciência. Ironizei a proximidade através da distância, sempre trabalhando. Me dediquei de uma maneira que muitos achariam desnecessária ao Mestrado. Espero que os frutos de tanto esforço um dia sejam colhidos... contigo.

Nada mais justo que também agradecer a minha orientadora, Fernanda. Ela teve ousadia em aceitar um aluno novo, em uma área diferente das suas tradicionais. Desconfio que tenha sido uma parceria proveitosa para os dois lados.

Muitos amigos também me deram forças ao longo dos últimos anos. Uns mais distantes, espalhados pelo globo, outros aqui bem próximos. Foram dias muito insanos, tudo após a graduação tem sido assim. Matando um leão por dia. Às vezes dois. Uma incerteza que irrita, amenizada pela presença dos amigos. Por fim, re-edito uma frase célebre, daquela que é uma paixão eterna: “academia do povo só tem uma”. Quem sabe, sabe. E eu, caro leitor... eu sei. Em vermelho e branco.



# CONTENTS

<b>LIST OF ABBREVIATIONS AND ACRONYMS</b> . . . . .	11
<b>LIST OF FIGURES</b> . . . . .	13
<b>LIST OF TABLES</b> . . . . .	17
<b>ABSTRACT</b> . . . . .	19
<b>RESUMO</b> . . . . .	21
<b>1 INTRODUCTION</b> . . . . .	23
<b>1.1 IC Implementation Flow</b> . . . . .	26
<b>1.2 Verification</b> . . . . .	27
<b>1.3 Functional Verification</b> . . . . .	29
<b>2 STATE OF THE ART AND CHALLENGES</b> . . . . .	35
<b>2.1 Coverage metrics</b> . . . . .	35
2.1.1 Block coverage . . . . .	35
2.1.2 Expression coverage . . . . .	37
2.1.3 Toggle coverage . . . . .	39
2.1.4 Functional coverage . . . . .	40
2.1.5 Other types of coverage metrics . . . . .	41
<b>2.2 Stimuli generation</b> . . . . .	43
<b>2.3 Checking</b> . . . . .	43
<b>2.4 Verification methodologies</b> . . . . .	44
2.4.1 <i>e</i> Reuse Methodology (eRM) . . . . .	46
2.4.2 Reference Verification Methodology (RVM) . . . . .	47
2.4.3 Verification Methodology Manual (VMM) . . . . .	47
2.4.4 Advanced Verification Methodology (AVM) . . . . .	47
2.4.5 Open Verification Methodology (OVM) . . . . .	48
2.4.6 Universal Verification Methodology (UVM) . . . . .	48
<b>2.5 Verification plan</b> . . . . .	49
2.5.1 Overview . . . . .	49
2.5.2 Feature list . . . . .	50
2.5.3 Test list . . . . .	50
2.5.4 Coverage goals . . . . .	51
2.5.5 Other sections . . . . .	51

<b>3</b>	<b>EVALUATING THE CHALLENGES</b>	53
3.1	Measuring simulation overhead caused by coverage	53
3.2	Measuring simulation overhead caused by data generation	56
<b>4</b>	<b>VEASY FLOW AND THE VERIFICATION PLAN</b>	59
<b>5</b>	<b>LINTING</b>	65
5.1	BASE	65
5.2	BCSI	66
5.3	DCSI	66
5.4	DIRE	67
5.5	HCSI	67
5.6	IASS	67
5.7	IDNF	67
5.8	IDNP	68
5.9	LPNA	68
5.10	MBAS	68
5.11	NBCO	69
5.12	NOIO	70
5.13	RCAS	70
5.14	TIME	71
5.15	TINR	71
5.16	VWSN	71
5.17	WPAS	72
5.18	Linting Interface	72
<b>6</b>	<b>SIMULATION</b>	75
6.1	Combinational logic	75
6.2	Regular sequential logic	78
6.3	Reset sequential logic	79
6.3.1	No reset	80
6.3.2	Time zero	80
6.3.3	Ranged	80
6.3.4	Probabilistic	81
6.4	Benchmarking	81
6.5	Scaling properties	83
6.6	Waveform output	84
6.7	Validating the simulator	86
<b>7</b>	<b>COVERAGE</b>	87
7.1	Code coverage	87
7.1.1	VEasy's block coverage algorithm	87
7.1.2	VEasy's expression coverage algorithm	88
7.1.3	VEasy's toggle coverage algorithm	89
7.1.4	Experimental Results and Analysis	89
7.1.5	Code coverage analysis using the GUI	91
7.2	Functional coverage	92
7.2.1	VEasy's functional coverage collection algorithm	92
7.2.2	Functional coverage analysis using the GUI	93



<b>8</b>	<b>METHODOLOGY</b>	95
8.1	Some statistics of VEasy's implementation	98
<b>9</b>	<b>CASE STUDY: THE PASSWORD LOCK DEVICE</b>	101
9.1	DUT description	101
9.2	Verification Plan	102
9.3	Building a testcase	102
9.4	Comparing the methodology with traditional SystemVerilog	105
<b>10</b>	<b>CONCLUSION</b>	109
	<b>REFERENCES</b>	113
	<b>ATTACHMENTS</b>	119
I	Verilog testbench used to validate the simulator	119
II	Web browser rendering of a verification plan file	120
III	Verilator testbench format	121



## LIST OF ABBREVIATIONS AND ACRONYMS

AOP	Aspect Oriented Programming
ASIC	Application Specific Integrated Circuit
AVM	Advanced Verification Methodology
BFM	Bus Functional Model
CDG	Coverage Directed (test) Generation
CTS	Clock Tree Synthesis
DFT	Design For Testability
DUT	Design Under Test
DUV	Design Under Verification
EDA	Electronic Design Automation
eRM	<i>e</i> Reuse Methodology
ESD	Electro Static Discharge
eVC	<i>e</i> Verification Component
FPGA	Field-Programmable Gate Array
FSM	Finite State Machine
FV	Functional Verification
GUI	Graphical User Interface
HDL	Hardware Description Language
HVL	Hardware Verification Language
IC	Integrated Circuit
IEEE	Institute of Electrical and Electronics Engineers
IP	Intellectual Property
MBIST	Memory Built-in Self Test
MCDC	Modified Condition Decision Coverage
OO	Object Oriented
OVM	Open Verification Methodology

PLI	Programming Language Interface
PSL	Property Specification Language
RTL	Register Transfer Level
RVM	Reference Verification Methodology
SOP	Sum of Products
STA	Static Timing Analysis
URM	Universal Reuse Methodology
UVM	Universal Verification Methodology
VCD	Value Change Dump
VMM	Verification Methodology Manual

## LIST OF FIGURES

Figure 1.1:	A decreasing trend in the first silicon success rate. . . . .	23
Figure 1.2:	A decreasing trend in the first silicon success rate, including data from 2007. . . . .	24
Figure 1.3:	Causes that led to silicon failure and the respective occurrence probabilities. . . . .	24
Figure 1.4:	Causes that led to silicon failure and the estimated cost of a respin in different technology nodes. . . . .	25
Figure 1.5:	Example of an ASIC implementation flow. . . . .	26
Figure 1.6:	Relative cost of finding an error in the various stages of the design cycle. . . . .	28
Figure 1.7:	Several verification methods, classified by type. . . . .	29
Figure 1.8:	The space of design behaviors. . . . .	30
Figure 1.9:	The productivity and the verification gaps. . . . .	31
Figure 1.10:	Breakdown of the effort during the development of three different ASICs. . . . .	32
Figure 2.1:	Example of Verilog code for block coverage. . . . .	36
Figure 2.2:	Example of Verilog code for expression coverage. . . . .	37
Figure 2.3:	Control flow graph of a Verilog module. . . . .	42
Figure 2.4:	All possible paths of the control flow graph from Fig 2.3. . . . .	42
Figure 2.5:	FSM coverage report for states and arcs. . . . .	42
Figure 2.6:	A generic verification environment capable of performing checking. . . . .	44
Figure 2.7:	Current verification practices, divided by programming language. . . . .	45
Figure 2.8:	Current verification practices, divided by vendor. . . . .	45
Figure 2.9:	A timeline containing the introduction year of each of the current verification methodologies. . . . .	46
Figure 2.10:	eVC environment instance. . . . .	47
Figure 2.11:	Example of a VMM verification environment architecture. . . . .	48
Figure 2.12:	Simplified block diagram of the verification plan. . . . .	50
Figure 3.1:	Simulation overhead due to toggle coverage in simulator A. . . . .	54
Figure 3.2:	Simulation overhead due to toggle coverage in simulator B. . . . .	55
Figure 3.3:	Simulation overhead due to data generation. . . . .	56
Figure 4.1:	VEasy assisted flow. . . . .	59
Figure 4.2:	VEasy simulation flow . . . . .	60
Figure 5.1:	Example of a Verilog code that violates the BASE rule. . . . .	66

Figure 5.2:	Example of a Verilog code that violates the BCSI rule. . . . .	66
Figure 5.3:	Example of a Verilog code that violates the DCSI rule. . . . .	66
Figure 5.4:	Example of a Verilog code that violates the DIRE rule. . . . .	67
Figure 5.5:	Example of a Verilog code that violates the HCSI rule. . . . .	67
Figure 5.6:	Example of a Verilog code that violates the IASS rule. . . . .	68
Figure 5.7:	Example of a Verilog code that violates the IDNF rule. . . . .	68
Figure 5.8:	Example of a Verilog code that violates the IDNP rule. . . . .	68
Figure 5.9:	Example of a Verilog code that violates the LPNA rule. . . . .	69
Figure 5.10:	Example of a Verilog code that violates the MBAS rule. . . . .	69
Figure 5.11:	Example of a Verilog code that violates the NBCO rule. . . . .	70
Figure 5.12:	Example of a Verilog code that violates the NOIO rule. . . . .	70
Figure 5.13:	Example of a Verilog code that violates the RCAS rule. . . . .	70
Figure 5.14:	Example of a Verilog code that violates the TIME rule. . . . .	71
Figure 5.15:	Example of a Verilog code that violates the TINR rule. . . . .	71
Figure 5.16:	Example of a Verilog code that violates the VWSN rule. . . . .	72
Figure 5.17:	Example of a Verilog code that violates the WPAS rule. . . . .	72
Figure 5.18:	Linting environment GUI reporting an error. . . . .	73
Figure 6.1:	Pseudo-C code of the combinational logic evaluation. . . . .	76
Figure 6.2:	Example of a Verilog code in which the combinational logic might create a mismatch. . . . .	76
Figure 6.3:	Pseudo-C code of the regular sequential logic evaluation. . . . .	78
Figure 6.4:	Example of a Verilog code in which the sequential logic might be misinterpreted. . . . .	78
Figure 6.5:	Reset methods available in VEasy's GUI. . . . .	80
Figure 6.6:	VEasy simulation results. . . . .	81
Figure 6.7:	VEasy simulation results compared with Verilator. . . . .	82
Figure 6.8:	Scaling trends for sequential logic simulation. . . . .	83
Figure 6.9:	Scaling trends for combinational logic simulation. . . . .	84
Figure 6.10:	Initial portion of a VCD file. . . . .	85
Figure 6.11:	Simulation waveform equivalent to VCD of Fig. 6.10. . . . .	85
Figure 6.12:	VCD compare environment GUI reporting errors. . . . .	86
Figure 7.1:	Block coverage collection algorithm. . . . .	88
Figure 7.2:	Expression coverage collection algorithm. . . . .	88
Figure 7.3:	Toggle coverage collection algorithm. . . . .	89
Figure 7.4:	Coverage metric selection GUI. . . . .	91
Figure 7.5:	Code coverage analysis GUI. . . . .	91
Figure 7.6:	Toggle coverage analysis GUI. . . . .	92
Figure 7.7:	Functional coverage collection algorithm. . . . .	93
Figure 7.8:	Functional coverage analysis GUI. . . . .	93
Figure 8.1:	Layering example. . . . .	96
Figure 8.2:	Rule editor GUI. . . . .	97
Figure 8.3:	Execution order of VEasy's sequences. . . . .	98
Figure 9.1:	DUT block diagram. . . . .	101
Figure 9.2:	Transmission of PS/2 packet. . . . .	101
Figure 9.3:	VEasy's GUI being used to build a testcase. . . . .	103

Figure 9.4:	start sequence described in VEasy's format. . . . .	103
Figure 9.5:	data0 sequence described in VEasy's format. . . . .	104
Figure 9.6:	pressV sequence described in VEasy's format. . . . .	104
Figure 9.7:	SystemVerilog code used to control the simulation time. . . . .	105
Figure 9.8:	SystemVerilog task as a BFM. . . . .	106
Figure II.1:	Portion of a verification plan file when rendered by a web browser. . .	120
Figure III.2:	Verilator testbench format. . . . .	121





## LIST OF TABLES

Table 2.1:	SOP scoring table for the expression $(c \ \&\& \ (d \ \parallel \ e))$ . . . . .	38
Table 2.2:	Subexpressions hierarchy of $(c \ \&\& \ (d \ \parallel \ e))$ . . . . .	38
Table 2.3:	Subexpressions hierarchy of $(c \ \&\& \ (d \ \parallel \ e))$ and toggles of interest. . . . .	39
Table 2.4:	Example of an overview table of a verification plan. . . . .	49
Table 2.5:	Example of a feature list from a verification plan. . . . .	50
Table 2.6:	Example of a test list from a verification plan. . . . .	51
Table 2.7:	Example of a list of coverage goals from a verification plan. . . . .	51
Table 3.1:	Some properties of the circuits being analyzed. . . . .	53
Table 3.2:	Simulation overhead measured using simulator A. . . . .	55
Table 3.3:	Simulation overhead measured using simulator B. . . . .	55
Table 3.4:	Evaluation of stimuli generation time using commercial simulator A. . . . .	57
Table 6.1:	Erroneous combinational logic evaluation. . . . .	77
Table 6.2:	Correct combinational logic evaluation. . . . .	77
Table 6.3:	Erroneous sequential logic evaluation. . . . .	79
Table 6.4:	Correct sequential logic evaluation. . . . .	79
Table 7.1:	Simulation overhead measured using VEasy. . . . .	90
Table 7.2:	Average simulation overhead measured using simulator A. . . . .	90
Table 7.3:	Average simulation overhead measured using simulator B. . . . .	90
Table 7.4:	Average simulation overhead measured using VEasy. . . . .	90
Table 8.1:	Some statistics of VEasy's development. . . . .	99
Table 9.1:	Feature list of the password lock device. . . . .	102
Table 9.2:	Statistics of SystemVerilog testbenches. . . . .	105
Table 9.3:	Statistics of VEasy testcases. . . . .	106



## **ABSTRACT**

This thesis describes a tool suite, VEasy, which was developed specifically for aiding the process of Functional Verification. VEasy contains four main modules that perform linting, simulation, coverage collection/analysis and testcase generation, which are considered key challenges of the process. Each of those modules is commented in details throughout the chapters. All the modules are integrated and built on top of a Graphical User Interface. This framework enables the testcase automation methodology which is based on layers, where one is capable of creating complex test scenarios using drag-and-drop operations. Whenever possible the usage of the modules is exemplified using simple Verilog designs. The capabilities of this tool and its performance were compared with some commercial and academic functional verification tools. Finally, some conclusions are drawn, showing that the overall simulation time is considerably smaller with respect to commercial and academic simulators. The results also show that the methodology is capable of enabling a great deal of testcase automation by using the layering scheme.

**Keywords:** Functional verification, simulation, coverage metrics, automation.



## **VEasy: um Conjunto de Ferramentas Direcionado aos Desafios da Verificação Funcional**

### **RESUMO**

Esta dissertação descreve um conjunto de ferramentas, VEasy, o qual foi desenvolvido especificamente para auxiliar no processo de Verificação Funcional. VEasy contém quatro módulos principais, os quais realizam tarefas-chave do processo de verificação como *linting*, simulação, coleta/análise de cobertura e a geração de *testcases*. Cada módulo é comentado em detalhe ao longo dos capítulos. Todos os módulos são integrados e construídos utilizando uma Interface Gráfica. Esta interface possibilita o uso de uma metodologia de criação de *testcases* estruturados em camadas, onde é possível criar casos de teste complexos através do uso de operações do tipo *drag-and-drop*. A forma de uso dos módulos é exemplificada utilizando projetos simples escritos em Verilog. As funcionalidades da ferramenta, assim como o seu desempenho, são comparadas com algumas ferramentas comerciais e acadêmicas. Assim, algumas conclusões são apresentadas, mostrando que o tempo de simulação é consideravelmente menor quando efetuada a comparação com as ferramentas comerciais e acadêmicas. Os resultados também mostram que a metodologia é capaz de permitir um alto nível de automação no processo de criação de *testcases* através do modelo baseado em camadas.

**Palavras-chave:** Verificação funcional, simulação, métricas de cobertura, automação.



## 1 INTRODUCTION

The move to deep-submicron feature sizes in the latest Integrated Circuits (IC) designs has caused a paradigm shift, moving the emphasis from design to verification (BERMAN, 2005). Designers must create ICs with an excess of 50 million equivalent gates and still meet cost and time-to-market constraints. This paradigm shift has brought verification to a top position at the Electronic Design Automation (EDA) industry topics of research and development.

In a context where design complexity is continuously increasing (CHANG, 2007), the probability that an Integrated Circuit (IC) is going to fail is also increasing. The failure may be attributed to many sources from different types. Yet, the trend is clear: circuits are failing more and more. A 2003 study by Collett International Research (Collett International Research Inc., 2003) (FITZPATRICK; SCHUTTEN, 2003) revealed for the first time that the first silicon success rate, i.e, the rate that a given IC will function in a satisfactory manner in the first silicon spin, is dropping. Research has shown that the first silicon success rate has fallen from about 50% to 35% in a four-year time frame. This trend is illustrated in Fig. 1.1.

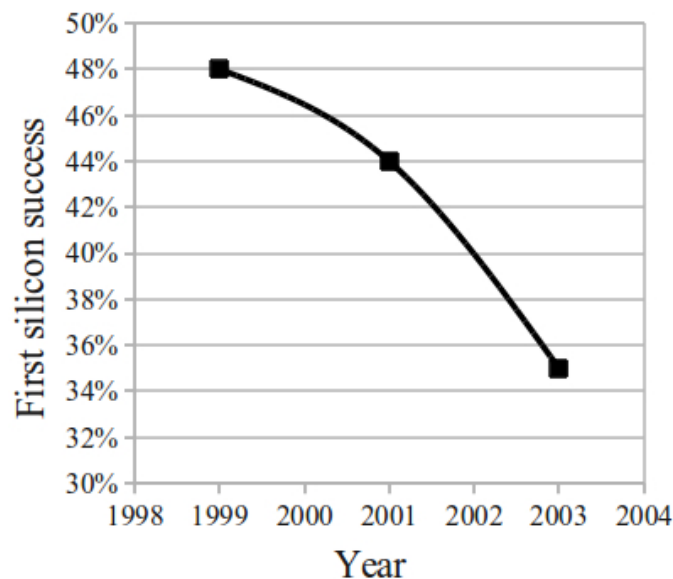


Figure 1.1: A decreasing trend in the first silicon success rate.

Later on a similar study was conducted by another company (Far West Research and Mentor Graphics, 2007). The trend that emerged was still the same: the first silicon success rate has fallen to 28%. The data from both studies are combined into Fig. 1.2.

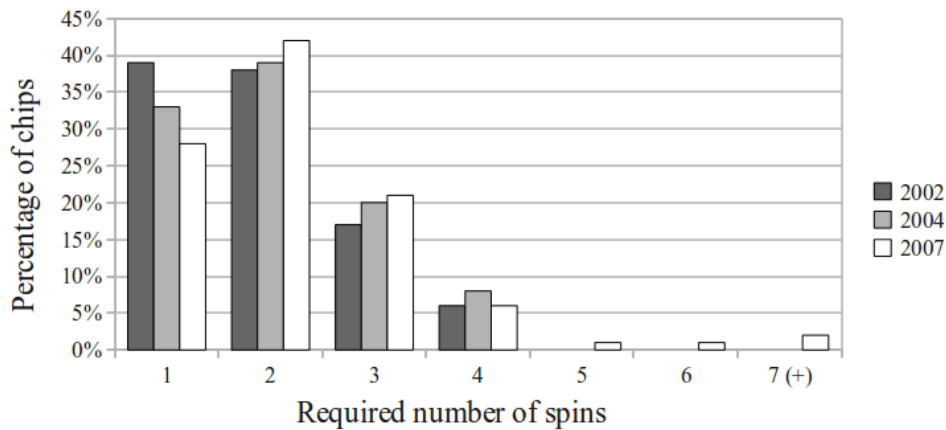


Figure 1.2: A decreasing trend in the first silicon success rate, including data from 2007.

The same studies also identified the sources of errors in chip design. Chips fail for many reasons ranging from physical effects like voltage drops (IR- drop), to mixed-signal issues, power issues, and logic/functional flaws. However, logic/functional flaws are the biggest cause of flawed silicon. Of all tapeouts that required a silicon respin, the Collett International Research study shows more than 65% contained logic or functional flaws, as shown in Fig. 1.3.

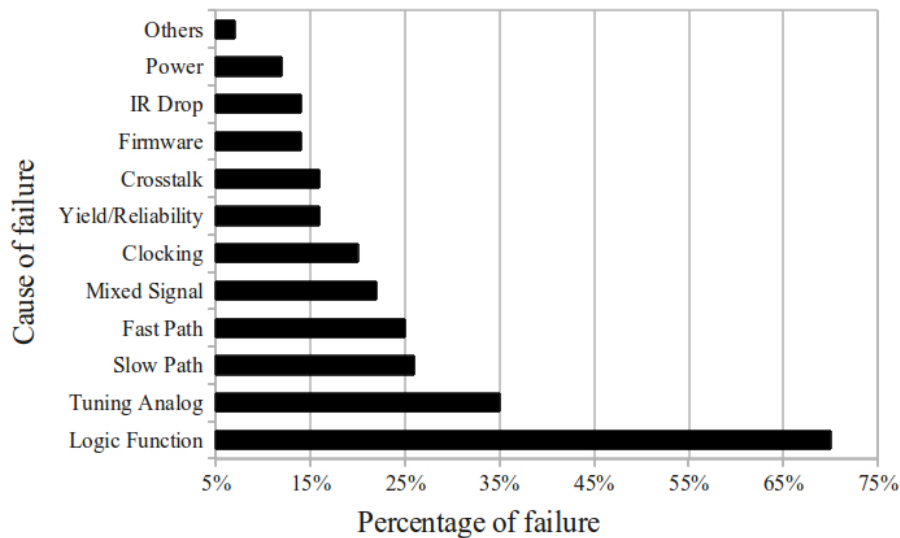


Figure 1.3: Causes that led to silicon failure and the respective occurrence probabilities.

The 2007 study from Farwest showed an even higher figure: about 80% of the respins were caused by some type of logic or functional error. Since the logic/functional errors were more common than the others, the same research tried to better identify those errors. For that purpose, the errors were classified as follows:

**Design errors** About 82% of designs with respins resulting from logic/functional flaws had design errors. This means that particular corner cases simply were not covered during the verification process, and bugs remained hidden in the design flow all the way through tapeout.



**Specification errors** About 47% of designs with respins resulting from logic/functional flaws had incorrect/incomplete specifications. Moreover, 32% of designs with respins resulting from logic/functional flaws had changes in specifications.

**Reused modules and imported IP** About 14% of all chips that failed had bugs in reused components or imported IP (Intellectual Property).

All presented data shows that a silicon respin is very common. Nevertheless, the costs involved in such respins might be extremely high, easily reaching six figure amounts for the earliest technologies (TRONCI, 2006) and nine figure amounts for predicted technologies (ITRS, 2005). The additional development time and costs of a respin must be considered as well. Thus, companies that are able to curb this trend have a huge advantage over their competitors, both in terms of the ensuing reduction in engineering costs and the business advantage of being to market sooner and with high-quality products. The key to time-to-market success, for many projects, is verification.

The illustration in Fig. 1.4 was obtained from a talk given by Janick Bergeron in a very recent conference (BERGERON, 2010). In such talk, the author highlights the fact that the sources of silicon failure have kept almost the same behavior throughout the technology shifts while the cost to keep such rates under control has risen exponentially. In other words, it means that verification has been doing its job but the costs of a mistake are rising with a fast-paced trend. Most of this apparent success achieved by verification comes from reuse. Although it is now possible to design chips with ten times more transistors than before, it is likely that such transistors execute the same functions that were already verified before. So, the overall picture still does not look good. This is an indication that our current verification practices are in need of changes.

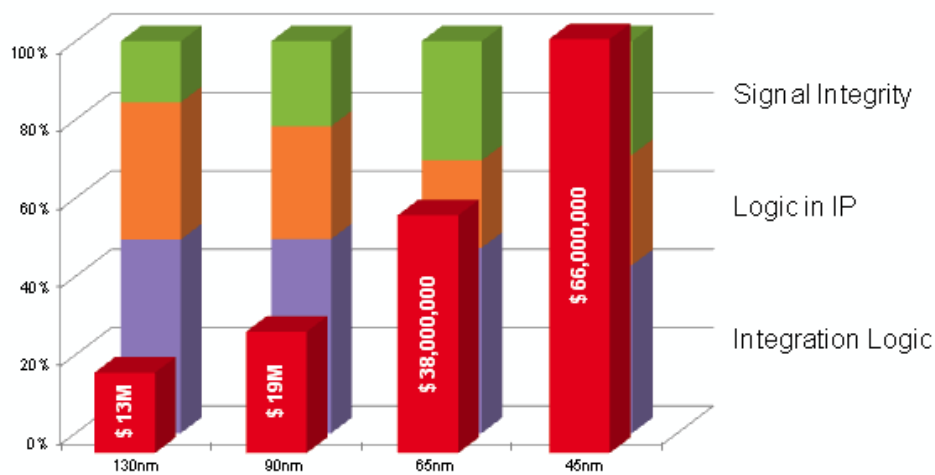


Figure 1.4: Causes that led to silicon failure and the estimated cost of a respin in different technology nodes.

Regarding such context, this master thesis aims at developing a new tool suite for achieving several goals related to verification. These goals are both academia and industry related. The former type of goal is deeply explored by the easiness to create simulation scenarios using a Graphical User Interface (GUI). This interface is suitable for teaching purposes, either at undergrad or graduation levels. A deep discussion regarding the use of the developed tool for teaching purposes is given in (PAGLIARINI; KASTENSMIDT, 2011a).

The industry related type of goal is explored by introducing the simulation performance that was achieved by the tool built-in simulator. But first, before introducing such topics, and in order to better understand the need for verification and development of new verification tools, let us start by revisiting the traditional IC implementation flow. The subtleties of this flow will determine where the verification efforts should be focused.

## 1.1 IC Implementation Flow

The process of developing an IC involves several steps that must be executed towards reaching the final circuit. Depending on the technology being used and also depending on the type of circuit (digital, analog or mixed-signal), this flow may have completely different steps. Also, if considering a re-programmable device such as a Field-Programmable Gate Array (FPGA), slight changes occur on the design flow. In the context of this thesis, the flow of Fig. 1.5 will be considered, which is a generic flow for an digital Application Specific Integrated Circuit (ASIC). Such flow is based on standard cell libraries, which is a widely used methodology for developing digital circuits (WESTE; HARRIS, 2010). This type of circuit is more prone to benefit from Functional Verification (FV) since it usually has a great complexity and a high design and production cost. In other words, the cost of fixing a logical error on this type of circuit is typically higher.

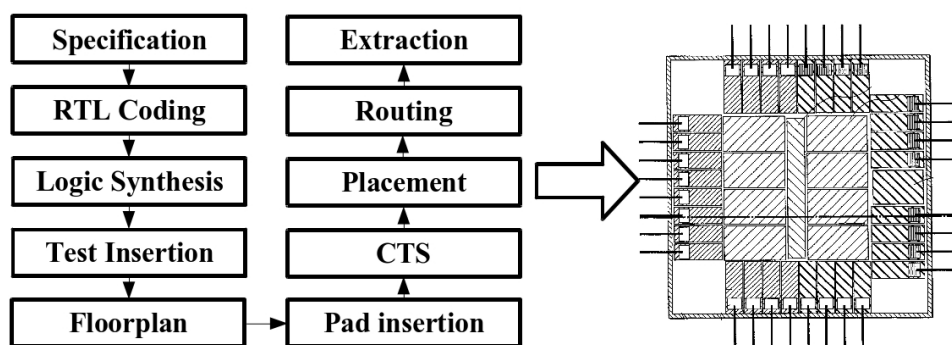


Figure 1.5: Example of an ASIC implementation flow.

Each of the tasks showed in Fig. 1.5 is some form of transformation that the current design representation goes through. For instance, the project specification, typically written in some natural language, is transformed into Register Transfer Level (RTL) code. This code is usually some form of synthesizable Hardware Description Language (HDL), like Verilog (IEEE, 1995) or VHDL (IEEE, 1987). Similarly, the other tasks perform the following transformations:

**Logic Synthesis** The RTL description is transformed into a gate-level netlist. This task is performed by tools referred as synthesizers, like RTL Compiler (Cadence Design Systems, Inc., 2010) and Design Compiler (Synopsys, Inc., 2010). Typically some form of Static Timing Analysis (STA) is used for calculating the expected timing of the circuit. Such data are then used to perform optimizations on the circuit. The final result of the process is, if possible at all, a circuit that respects the specified timing constraints.

**Test insertion** The design is instrumented with Design For Testability (DFT) structures, like scan-chains and Memory Built-in Self Test (MBIST). These are very important for determining if the chip works properly after it has been manufactured.

**Floorplan** In this step the circuit is organized and structured in a layout form for the first time. The power grid that will feed the circuit must be created and some routing or placement restrictions may be applied. The overall dimensions and aspect ratio of the chip must be defined as well.

**Pad insertion** The pads of the chip must be chosen and inserted in the layout. The pads are the communication channel the circuit has with the external environment. Considerations regarding Electro Static Discharge (ESD) must be done at this step.

**CTS** The Clock Tree Synthesis (CTS) is performed in this step. It is important for the clock signal of a chip to reach all the sequential elements at about the same time, therefore special optimizations are performed for the clock buffering and routing.

**Placement** The standard cells are placed in this step. The overall idea is that cells that must be connected must be kept close to each other. Elements other than the proximity must be considered, such as routing congestion. After the placement is done no cell overlap should occur.

**Routing** This step is responsible for creating the physical connections between all cells. The pads and the power rings must be connected as well.

**Extraction** This step is responsible for extracting the resistivity and capacitance of the final layout. The extracted data is used to perform proper tuning of the previous steps as well as to perform electric simulation for sign-off purposes.

After all these transformations are done the chip is considered ready. However, such transformations are prone to errors and/or misinterpretations. Thus, some form of checking the results from these transformations must be considered. That is the role of verification: avoiding errors in the design flow. Some verification tasks are very intensive and require several resources, mostly due to the simulation nature of the verification method. This intensive behavior translates directly into cost, either due to human resources or delays in deploying a design with a tight time-to-market. However, these costs are counterbalanced because the number of errors caught late in the design cycle is diminished. Such idea is illustrated by the chart in Fig. 1.6, which contains the relative cost of finding an error in the various stages of the design cycle (GIZOPOULOS, 2006).

Obviously, different types and flavors of verification are applied in different steps of the process. Some of these types are explored in the next section.

## 1.2 Verification

The previous section focused on what is referred as the IC implementation flow. Typically, this flow is executed in parallel with another flow, the verification one. In some cases, when a company has the resources to afford it, two completely separated teams are assigned to work on each flow (BERMAN, 2005). This introduces some redundancy in the process since different views of the same project will be constructed, which actually might lead to a more efficient error detection. Ideally, the only communication channel available between these teams is the actual specification of the project.

The transformations mentioned in the previous section are not error free. Thus, every step must be properly executed and the results must be checked as well. The first transformation that the design goes through is also the most critical one, at least from a

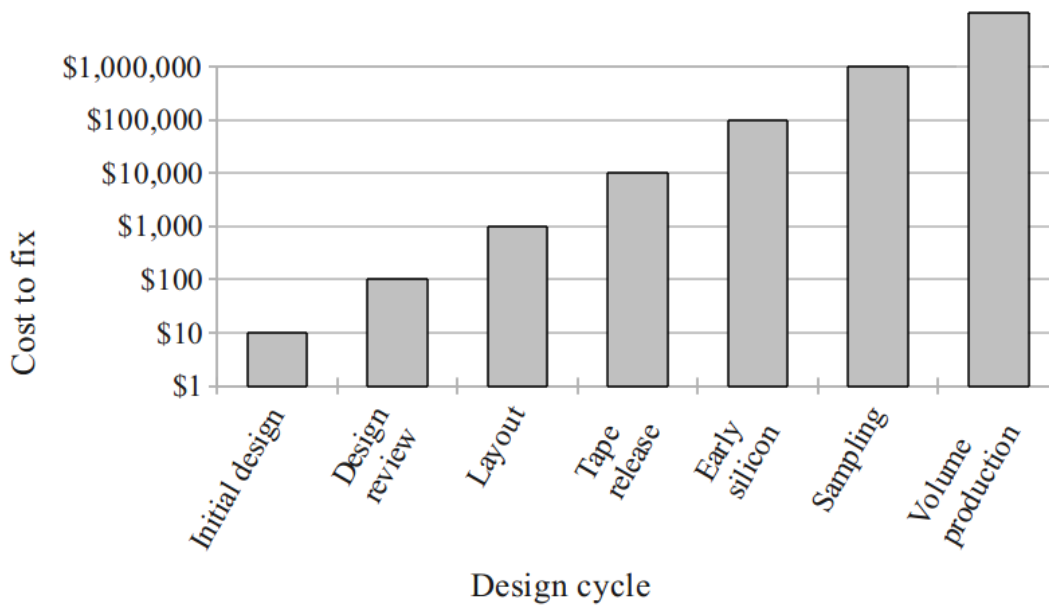


Figure 1.6: Relative cost of finding an error in the various stages of the design cycle.

verification point of view. The process of translating a written document to a hardware description is specially susceptible to misinterpretations. This transformation in particular is explored throughout the next section. The other transformations are more mechanic and automated, which makes them less sensitive to misinterpretation errors. Obviously, regular errors due to erroneous execution of the task are still possible. To exemplify some of the possible verification tasks that are performed in a modern design flow, the image in Fig. 1.7 shows several verification methods (MOLINA; CADENAS, 2007).

The image classifies verification methods into three major types: FV, equivalence checking and code coverage. A brief explanation on each type of method is given below. The FV type, specially the dynamic flavor, is explained in details in the next section.

**Functional Verification** The most widespread method of FV is dynamic in nature. The reason it is called dynamic is because input patterns/stimuli are generated and applied over a number of clock cycles to the design, and the corresponding result is collected and compared against a reference/golden model for conformance with the specification. The static functional verification, also referred as formal verification, performs the same comparison but using some sort of mathematical proof instead of simulation.

**Equivalence checking** Given two representations of the same design, equivalence checking is capable of reasoning if they are equivalent or not. This type of comparison is widely used after logic synthesis, i.e., to compare the gate-level netlist against the RTL representation.

**Code Coverage** Code coverage indicates lines of code that were visited in a simulation. Although code coverage is reasonably easy to collect, it is an indirect metric of the overall verification progress since there is no direct mapping between lines of code and a given design functionality. Code coverage and other types of coverage are explored in details in Section 2.1.

The terms used in the literature might mislead the readers since some methods are

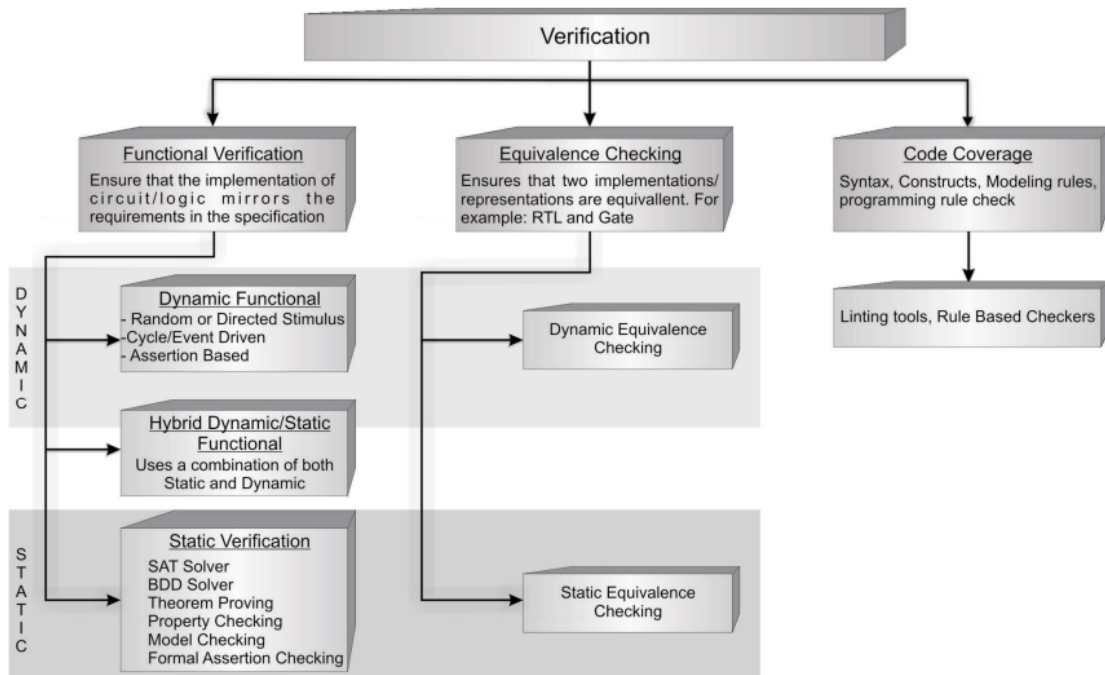


Figure 1.7: Several verification methods, classified by type.

shared by different techniques (e.g. assertions). This text uses the following convention: a technique is a collection of methods used in conjunction. A method is just an approach to prove a certain statement or property regarding a design. Thus, the illustration in Fig. 1.7 shows only methods, classified by type. Therefore a few words must also be said about techniques.

In general, the verification of a design has been accomplished following two principal techniques, simply referred as formal and functional verification (BERGERON, 2003). Regarding FV, one usually is referring to a set of methods that includes all of the following: random or directed stimulus, assertions, dynamic simulation and coverage. Regarding formal verification, one usually is referring to a set of methods that includes one of the following: theorem proving, property checking, formal assertion checking, etc.

To be perfectly clear, FV is mainly simulation based and it is the most common technique being used by the industry. The reason for the large adoption of FV is that, although new methodologies that benefit from formal and semi-formal methods have been proposed (COHEN et al., 2010) and sometimes adopted by the industry (ABADIR et al., 2003), these formal methods are still limited. Being so, the developed tool reported later in this text is aimed at FV.

Also, on the text that follows, the words Functional Verification (or the acronym FV) will be used referring exclusively to the Functional Verification technique, which is dynamic, i.e., the one that uses simulation. Formal methods will not be covered by this thesis. Equivalence checking will not be covered either.

### 1.3 Functional Verification

The primary goal of FV is to establish confidence that the design intent was captured correctly and preserved by the implementation (VSI Alliance, 2004) (BERGERON, 2003). The illustration of Fig. 1.8 shows how the design intent, the design specification

and the design's RTL code are related with each other to compose the space of design behaviors (PIZIALI, 2004).

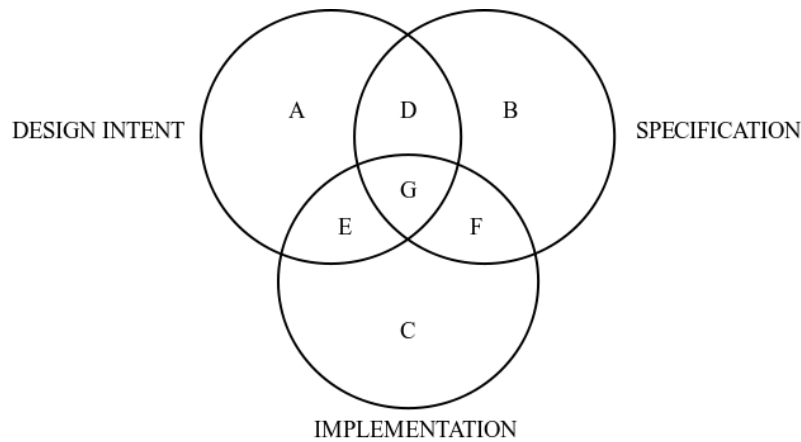


Figure 1.8: The space of design behaviors.

Each circle of Fig. 1.8 represents a set of behaviors. The design intent is a set of design requirements defined by the system architect, sometimes in conjunction with a customer. It is an abstraction of what the architect expects of a certain design overall functionality. The specification is a written document that tries to exhaustively enumerate those functionalities. The specification is the guide the engineers will follow to perform the coding. The implementation is the actual intent that was coded in the RTL code. The space outside all circles represents the unintended, unspecified and unimplemented behavior. The goal of verification is to match the three circles, bringing them into coincidence.

When performing verification, i.e., matching the circles from Fig. 1.8, some very distinct results appear since very often the circles do not match. Some regions from the image are of particular interest. Region G, for instance, actually represents the best possible scenario: a certain intent was defined, specified and implemented. FV's goal is to maximize this region. Another interesting region is D, which contains a desired intent that was specified but, for some reason, it was not implemented. On the other hand, a certain functionality might be specified and implemented although it was not intended. This is what the F region represents. Clearly this type of scenario led to a waste of resources and time.

It should be clear that verification is a necessary step in the development of today's complex digital designs, either functional or not. Hardware complexity continues to grow and that obviously impacts the verification complexity. The complexity growth is leading to a even more challenging verification. In fact, it has been shown that the verification complexity theoretically rises exponentially with hardware complexity (DEMPSTER; STUART; MOSES, 2001). Both increases in complexity have created gaps (BAILEY, 2004) (DENG, 2010). Those gaps come from comparing our capabilities at designing and verifying against the actual chip capacity, as shown by the illustration of Fig. 1.9.

The verification gap is a measurement of the difference between the ability to design (dashed line) and the ability to verify (thin dashed line), which is considerably lower than the overall productivity gap. Such gap comes from the difference between the verification ability and the actual chip capacity (black line). In other words, the industry current processes are capable of filling a chip with all kinds of complex logic. Yet, it is not capable of guaranteeing that the logic works properly.

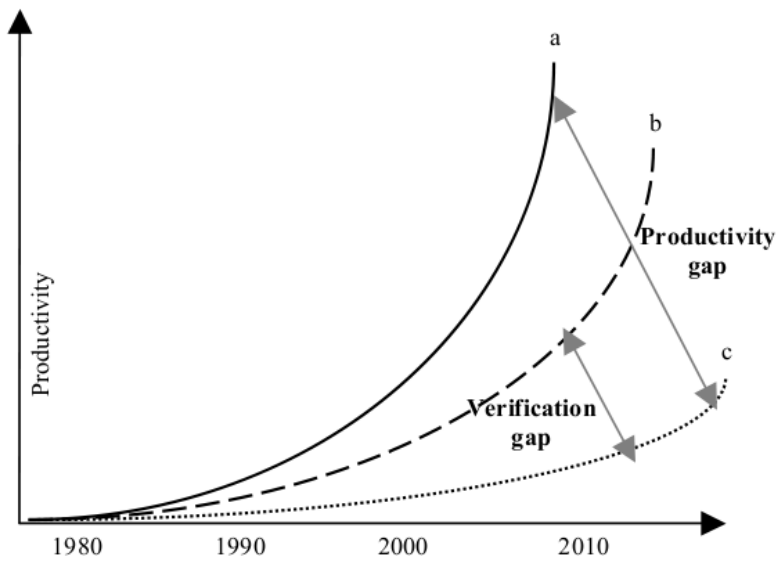


Figure 1.9: The productivity and the verification gaps.

Clearly the increase in verification complexity must be dealt properly and for that reason some new Hardware Verification Languages (HVLs) have been proposed in the last years, such as the *e* language (IEEE, 2006), the Property Specification Language (PSL) (IEEE, 2005) and the SystemVerilog language (IEEE, 2009). All those new languages were developed by the industry and later became standards of the Institute of Electrical and Electronics Engineers (IEEE). This confirms the fact that the ASIC industry already acknowledges that the verification process is extremely necessary and hard to accomplish.

It is also known that the verification itself occupies a fair share of the design cycle time. Although there is not a clear metric to evaluate such statement, i.e., to say that verification occupies a certain definitive percentage of the design cycle, some numbers have been estimated throughout the years. In 1998 there was a reference from the industry (STEFFORA, 1998) quoting that only a few years ago verification was responsible for 20-30% of the overall design effort. Yet, by that year, verification was already acknowledged to be responsible for even 70% of the design effort for certain designs. Another study (EVANS et al., 1998) from the same year tried to breakdown and measure all the tasks executed during the design process of three different ASICs. Such breakdown is shown in Fig. 1.10. By adding up all the tasks that are related to simulation and emulation one would come up with a 60-65% figure.

Almost identical numbers have been stated by other companies throughout the years. Some examples of such statements are found in reports from Mentor (2002), Virginia Tech (2004), Denali (2005), nVidia (2006), and Jasper (2008). All the sources pointing to these reports and a in-depth discussion on the matter was done by (MARTIN, 2008). These same reports also mentioned that, although the verification occupied a large share of the design cycle, there still was a large number of silicon chips that were flawed.

One of the possible explanations for the high number of flawed silicon spins is the lack of a concise metric for the verification progress. The completeness of a design verification is defined as follows: verification is complete when every possible input scenario has been applied to the Design Under Verification (DUV)<sup>1</sup>, and every possible output signal has

<sup>1</sup>In this thesis the terms DUV and Design Under Test (DUT) are used interchangeably to define the design that is currently being verified.

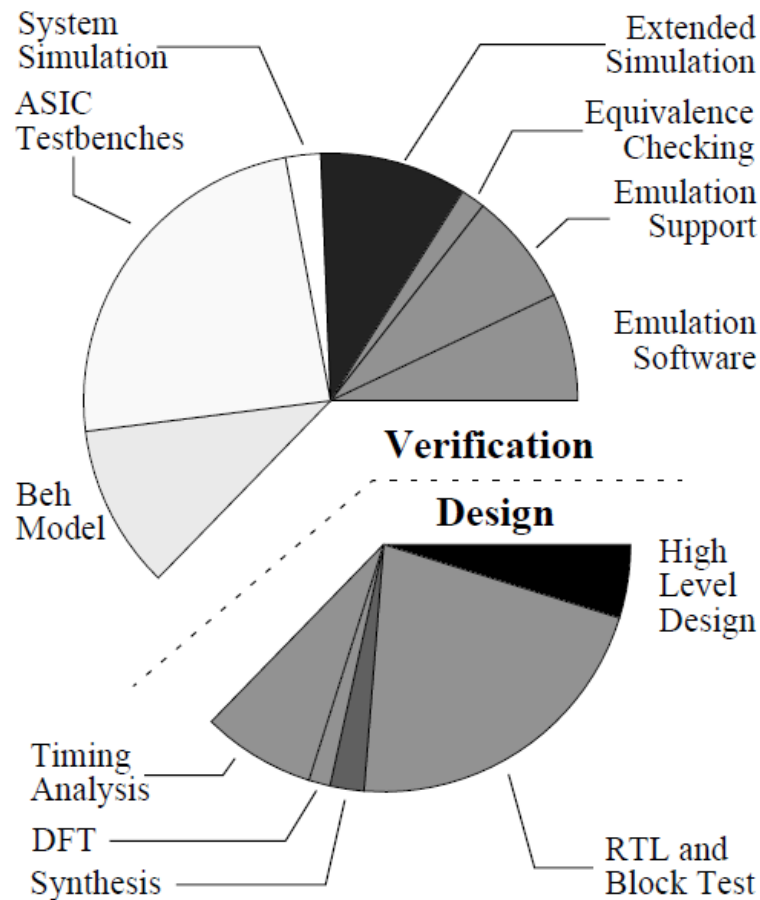


Figure 1.10: Breakdown of the effort during the development of three different ASICs.

been shown to have its intended value at every point in time (BORMANN et al., 2007). Measuring completeness is the actual problem that has driven verification engineers to adopt indirect metrics for measuring progress.

Those indirect metrics are referred as coverage metrics. The overall quality of the verification is obtained from coverage, either structural or functional coverage. Structural coverage is linked directly to the RTL code and therefore is also referred as code coverage. On the other hand, functional coverage is linked with the design functionalities. This is why functional coverage has become a key element in the verification process. However, the continuous increase in terms of the number of transistors per chip is resulting in a diminished validation effectiveness. Testcases being used in the simulations are more and more complex, causing simulation to get more expensive and to provide less coverage (YAN; JONES, 2010).

FV strives to cope with the complexity increase trend but some of the related challenges are overwhelming. One example of such challenge is the coverage collection and analysis. So far those challenges have been addressed with verification methodologies and EDA tools but there is a claim for more innovation and automation improvement. This is one of the goals that has driven the efforts of this thesis.

On the pages that follow, an EDA tool suite developed by the author will be described and compared. The name that such suite received is VEasy, which comes from the idea that verification should be easy. That is why VEasy's aim is to be a FV solution, including



a simulator and a Testcase<sup>2</sup>Automation interface. Therefore VEasy acts in both domains of improvement: the simulator as a simple EDA tool and the Testcase Automation solution as a verification methodology.

In the context of this thesis, verification is performed on the RTL representation of the design. In order to do that, FV often uses a combination of simple logic simulation and test cases (i.e. sequences of inputs) generated for asserting specific features of the design. Recently this approach has been enhanced by the use of constrained randomness. This element alone has aided the verification effort because it has broadened the search space of a test case while still keeping it controlled under a set of rules. This and other elements that are essential to FV will be discussed in the next chapters and sections.

The chapters that follow contain a description of the features found within the VEasy tool suite. Chapter 2, for example, describes some challenging tasks or concepts that are related to FV. Chapter 3 contains an evaluation of some of these tasks that are executed during the FV of a design, with emphasis on simulation and generation. The actual tool suite, VEasy, is presented in Chapter 4 together with the work flows and some general functionalities and features.

Later, on Chapter 5, the linting capabilities of the tool are demonstrated as well as a comprehensive list of all the available linting checks. Following this discussion, the simulation capabilities and algorithms of VEasy are described in details in Chapter 6. Chapter 7 deals with the coverage collection and analysis that is enabled by the tool, while Chapter 8 describes the methodologies that are used to create stimuli to the simulations, along with the GUI that enables some testcase automation capabilities. A case study is provided in Chapter 9 while the final chapter provides closure to the work with the final remarks and conclusions of this thesis.

---

<sup>2</sup>In this thesis the word testcase is used when referring to a simulation scenario created within our developed tool, while the word testbench is used when referring to a traditional simulation scenario written in some form of HDL/HVL.



## 2 STATE OF THE ART AND CHALLENGES

Before introducing the developed tool, VEasy, this chapter will discuss some of the challenges related to FV. Also, the current state of the art on each of the topics will be discussed.

### 2.1 Coverage metrics

As mentioned in the previous chapter, the measurement of verification progress is given by coverage metrics (GRINWALD et al., 1998). Although most of the metrics are simple and easy to understand, the way they relate to the actual hardware functionalities is not simple nor direct. Block, expression and toggle coverage are examples of coverage metrics with such behavior. Case studies of code coverage being used to measure the verification progress are very common in the literature and date back from more than 30 years ago. One example of such studies is presented by (REDWINE S.T., 1983), where code execution coverage was applied in the software test domain in the same way that block coverage is used in hardware verification. The same particular study also conducted function, input, output and function interaction coverage.

In this thesis coverage metrics are organized in two distinct groups: structural and functional. Structural coverage is linked directly to the RTL code and therefore is also referred as code coverage. This type of coverage is the most common metric being used by the industry today and collecting it is typically a completely automated process. This is not the case for functional coverage, which requires manual intervention to be collected. Therefore it has been a hard process for some companies to shift to a coverage model based on functional metrics.

On the other hand, functional coverage is linked with the design functionalities. Being so, the verification engineer or the architect responsible for defining the coverage model must be familiarized with the design. One of VEasy's goals is to bring this type of manual intervention to a minimum. The possibilities of defining functional coverage goals in VEasy are described in Subsection 7.2.2.

The subsections that follow contain a discussion regarding some of those coverage metrics.

#### 2.1.1 Block coverage

Block coverage is the most simple of the structural metrics. A block of code is defined by one or more statements that always execute together. The principle of this metric is to measure if each block of the RTL code was simulated at least once and, because of that, it has replaced line coverage metric (in which the execution of each line of code

is evaluated). Guaranteeing that each block was simulated eliminates the presence of simple errors due to unstimulated logic. Also, this type of metric is useful for detecting dead code<sup>1</sup> in the design, which might be contributing to extra logic in the final layout of the chip.

The block coverage metric is also used in the software test domain, as shown by (WONG et al., 1994), where the authors use the concept of block coverage for fault detection purposes. Even in such early studies it was already mentioned that block coverage, although an important metric, does not relate well with the actual functionality. One example is provided where two tests are examined: one that contains a decent block coverage (around 80%) and one that contains a high coverage (around 95%). Yet, the one with the high coverage is less efficient than the other when it comes to detecting faults. The authors of (BRIAND; PFAHL, 2000) also use block coverage as a metric in order to try to associate defect coverage with test coverage.

Typically, structures like *if/else* and *case* statements are considered as block starters for RTL code. For behavioral (algorithmic) codes, statements like *wait* and *sync* might be considered as well. The following code illustrated in Fig. 2.1 shows where blocks of code are defined, considering a module written in RTL Verilog.

---

```

1  module example(clk , a , b , c , d);
2
3  input clk ;
4  input a , b ;
5  output reg c , d ;
6
7  always @(posedge clk) begin
8      if (a == 1'b0) begin
9          c <= 1'b0 ;
10         d <= 1'b0 ;
11     end
12     else begin
13         case (b)
14             0: begin
15                 c <= 1'b0 ;
16                 d <= 1'b0 ;
17             end
18             1: c <= 1'b1 ;
19         endcase
20     end
21 end
22
23 endmodule

```

---

Figure 2.1: Example of Verilog code for block coverage.

The code listing shown in Fig. 2.1 has five blocks of code. These blocks start on the following lines:

**Line 8** Due to the *always* statement.

---

<sup>1</sup>Dead code is a portion of the code that, no matter which input or combination of inputs is considered, it will never be executed.

**Line 9** Due to the *if* statement.

**Line 13** Due to the *else* statement.

**Line 15** Due to the first clause of the *case* statement.

**Line 18** Due to the other clause of the *case* statement.

Notice that the *begin/end* pair is not necessary to start a new block of code (line 18 is an example of such situation). Also notice that the *always* statement (as in line 7) will always create a block, although this particular example has not a single assignment in such block.

### 2.1.2 Expression coverage

Expression coverage is a mechanism that factorizes logical expressions. Each factorized expression is then monitored during simulation runs. Thus, expressions are evaluated in greater detail (MICZO, 1986). Expression coverage complements block coverage because it tries to identify why a given block of code has been stimulated. The code listing from Fig. 2.2 illustrates such situation in lines 9 and 12. For instance, on line 9, a new block of code is started. The execution of this block of code may be triggered due to either of the inputs (*a* or *b*).

Expression coverage also subsumes block coverage, i.e., 100% expression coverage implies 100% block coverage. The opposite is not true. The principle of this metric is: given that the designer coded more than one situation in which a block might be executed, there was probably a different reason for each situation. One could think of the inputs *a* and *b* as two different configuration scenarios that were coded. So, it is necessary to confirm that both scenarios were simulated.

---

```

1  module example(a, b, c, d, e, f);
2
3  input a, b, c, d, e;
4  output reg f;
5
6  always @(*) begin
7      f = 1'b0;
8
9      if (a || b) begin
10         f = 1'b1;
11     end
12     if (c && (d || e)) begin
13         f = 1'b1;
14     end
15 end
16
17 endmodule

```

---

Figure 2.2: Example of Verilog code for expression coverage.

For expressions that are more complex, such as the one in line 12, more than one scoring method might be applied. Expressions can be scored by different methods, where

one trades the amount of data to analyze against the accuracy of the results. Examples of such scoring methods are SOP (Sum of Products), control and vector scoring, which are analyzed in the following subsections. A deeper discussion on different scoring methods for logical expressions is performed in an article by (AMMANN; OFFUTT; HUANG, 2003), where several techniques based in Modified Condition Decision Coverage (MCDC) (CHILENSKI; MILLER, 1994) are compared.

### 2.1.2.1 SOP scoring

The SOP scoring mode reduces expressions to a minimum set of expression inputs that make the expression both true and false. SOP scoring checks that each input has attained both 1 and 0 state at some time during the simulation. Therefore, the expression  $(c \ \&\& \ (d \ \parallel \ e))$  would be scored as shown in Tab. 2.1. The dash symbol used in the table represents a *do not care* situation.

Table 2.1: SOP scoring table for the expression  $(c \ \&\& \ (d \ \parallel \ e))$ .

c	d	e	result
1	-	1	1
1	1	-	1
-	0	0	0
0	-	-	0

### 2.1.2.2 Control scoring

Control scoring mode checks if each input has controlled the output value of the expression at some time during the simulation. If the input changes value, then the output also changes. In a way, control scoring improves verification accuracy by applying stronger requirements in order to call an expression input covered. Control scoring is sometimes referred to as sensitized condition coverage or focused condition coverage.

Also, control scoring breaks an expression into a hierarchy of subexpressions. For example, the expression  $(c \ \&\& \ (d \ \parallel \ e))$  would be represented as follows:

Table 2.2: Subexpressions hierarchy of  $(c \ \&\& \ (d \ \parallel \ e))$ .

Original expression	c	&&	(	d		e	)
First level hierarchy	<1>		<-----	2	----->		
Second level hierarchy				<3>		<4>	

The resulting subexpression analysis produced two levels of hierarchy and four different expressions, numbered from 1 to 4. The resulting coverage table for the expression  $(c \ \&\& \ (d \ \parallel \ e))$  using the control scoring method is illustrated in Tab. 2.3.

For instance, the and operation between <1> and <2> will be analyzed looking for three different combination of inputs. However, the combination of two zeros in both inputs is not considered due to the fact that, if you have two zero inputs and one of your inputs toggles, it would not cause a toggle in the result. Therefore, this toggle is not capable of controlling the expression. A similar analysis may be done for the or operation

Table 2.3: Subexpressions hierarchy of (c &amp;&amp; (d || e)) and toggles of interest.

<1>	&&	<2>	result
0		1	0
1		0	0
1		1	1
<3>		<4>	result
1		0	1
0		1	1
0		0	0

between <3> and <4>. Instead of two zeros, the combination of inputs that is not capable of toggling the result is two ones.

### 2.1.2.3 Vector scoring

Vector scoring mode is an extension of control scoring mode. With vector scoring, each bit of a multi-bit signal is scored and reported separately. Therefore, there is a great amount of data to analyze. Since each operand is multi-bit, each operand is initially scored using a reduction table (the logical or is used for reduction). The overall expression is scored using the same scheme from control scoring, i.e, using || and && operators. Considering the same expression scored in Tab. 2.3, but assuming that c is now 3 bits wide, an additional table would be used. In this table, each bit of c must have the opportunity to control the overall expression.

### 2.1.3 Toggle coverage

Toggle coverage measures if every bit of every signal has been properly stimulated. This measurement is done by observing if a given bit has toggled from zero to one and vice-versa. In the code of Fig. 2.2 all the inputs and outputs of the module are susceptible to toggle coverage. If the module had internal signals it would be necessary to cover them as well. Even for the example of Fig. 2.2, which is quite simple, there are 6 bits that must be observed at every clock cycle. For each bit both transitions must be considered, therefore there are 12 distinct situations that must be covered and, because of that nature, it is predictable that toggle coverage is very intensive to collect. In some cases it might be interesting to collect transitions that also include X and Z values. Naturally, this type of collection makes it even more challenging to perform toggle coverage.

Although VEasy does not handle circuits at the gate level there is an important feature of the toggle coverage metric that is related to those. At gate level the design is purely structural so the use of coverage analysis is limited since expressions and blocks no longer exist. Yet, the toggle coverage metric is still able to benefit the analysis at this level.

None of the coverage metrics present so far is considered 'enough', or some sort of full solution. It is very common to see verification efforts where all of these metrics were considered. This is mostly due to the fact that the metrics are capable of examining different behaviors or facts from a circuit description. For example, achieving 100% toggle coverage, in many cases, may not lead to 100% of the other metrics, as mentioned by (WANG; TAN, 1995).

Later, on Chapter 7, some considerations about coverage collection performance will be made, specially focusing on the performance of collecting toggle and block coverage metrics.

#### 2.1.4 Functional coverage

Before explaining what functional coverage is, it is important to understand why code coverage is not enough. As mentioned above, code coverage reflects how thorough the HDL code was exercised. A code coverage tool traces the code execution, usually by instrumenting or even modifying the HDL code. The set of features provided by code coverage tools usually includes line/block coverage, expression coverage, toggle coverage, etc. Some recent code coverage tools also include capabilities that automatically extract Finite State Machines (FSMs) from the design and ensure complete states and arc coverage. Nevertheless, code coverage is not enough. The reason is simple: most functional coverage requirements cannot be mapped into code structures.

For example, code coverage cannot indicate whether a particular simulation has visited all the legal combination of states in two orthogonal FSMs. Another example might be whether all the possible inputs have been tried while the DUT was in all the different internal states. Also, code coverage does not look at sequences of events, such as what else happened before, during, or after a given line of code was executed. In summary, whenever is necessary to correlate events or scenarios, code coverage fails. Yet, code coverage is a necessity since it is quite unacceptable to synthesize code that is either dead or unverified.

On the other hand, FV perceives the design from a user's (or system) point of view. Examples of questions that FV strives to answer are:

- Have you covered all of your typical scenarios?
- How about error cases? (sometimes these are as important as the correct scenarios)
- Which are the relevant corner cases?
- How to account for protocol scenarios and sequences?

The answers to all these questions comes from functional coverage, which also allows the measurement of relationships between scenarios (LACHISH et al., 2002). For instance, it might be interesting to cover that a situation was triggered from every state of a state machine. Another example: cover that a package was received when the buffer was empty and later when the buffer was full. Or even cover that a package of type A was followed by a package of type B. This characteristic elevates the discussion to specific transactions or bursts without overwhelming the verification engineer with bit vectors and/or signal names.

In a way, functional coverage might be seen as a black box testing enabler since it allows the verification to focus on the simulation scenarios from a higher level of abstraction. This level of abstraction enables natural translation from coverage results to test plan items, which will be later commented through Section 2.5. Also, functional coverage is usually divided into data-oriented and control-oriented. When performing data-oriented collection, mechanisms like covergroups of SystemVerilog are being used to sample and store values of registers or variables. On the other hand, when performing control-oriented



collection, mechanisms like assertions<sup>2</sup> are being analyzed to detect specific sequences of events that are of interest.

Also, it is important to realize that achieving a complete functional coverage is usually the primary goal of the FV. As a secondary goal then code coverage takes place. The reason is simple: there is no use to finding out if all blocks and expressions of a code have been exercised if that code does not implement all the requested functionalities. Another approach is to realize that most common, trivial or even general errors can be found when trying to reach full functional coverage. Complex errors that seem to have more of a random reason to appear, i.e., which the source is hard to identify, then might be investigated using code coverage metrics.

As functional and code coverage are complementary in nature, a tool or methodology that combines both approaches is extremely beneficial. This combined methodology will then provide a complete overview of the verification progress and a clearer correlation between the functional coverage definitions and the actual design implementation. VEasy, therefore, supports both types of coverage.

### 2.1.5 Other types of coverage metrics

This section enumerates some other types of coverage metrics that might be found in methodologies or simulators. Yet, the metrics presented so far in the latest section are more common than these.

#### 2.1.5.1 Line coverage

Line or statement coverage collects information to evaluate if each line (statement) of a given code was executed. Block coverage has replaced this type of metric in some simulators and methodologies in use nowadays.

#### 2.1.5.2 Path coverage

Path coverage analyzes the execution path that was stimulated in a given piece of code and it is achieved by handling the code as a decision diagram (or a control flow graph). Every possible decision must be taken at some point for the path coverage to be considered hole-free<sup>3</sup>. For example, the code of Fig. 2.2 contains four possible paths due to the two if statements.

One can see the four possible paths on Fig. 2.3. Each if statement is represented by a diamond shape and each assignment is represented by a rectangle. These possible paths are p1-p3, p1-p4, p2-p3 and p2-p4, as shown in Fig. 2.4.

#### 2.1.5.3 FSM related coverage

Somewhere between code coverage and functional coverage there are coverage metrics related to FSMs. Typical metrics are based on the analysis of arcs, states and transitions. State coverage reports which states were visited, transition coverage reports what transitions occurred and arc coverage reports why each transition occurred. The illustration that follows in Fig. 2.5 contains a FSM coverage report from a commercial tool.

---

<sup>2</sup>Assertions are checks usually embedded into the design. They are used to verify a designer's assumptions about how a design should operate. Assertions are very appropriate for verifying protocol scenarios at a design's interface. They are also used to express definitive properties about a system or circuit.

<sup>3</sup>A hole is a situation of interest that was not covered, e.g., a path or a block of code. Therefore a hole-free coverage metric is one that was fully covered.

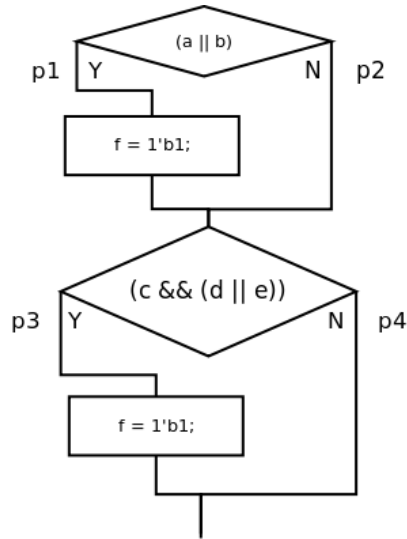


Figure 2.3: Control flow graph of a Verilog module.

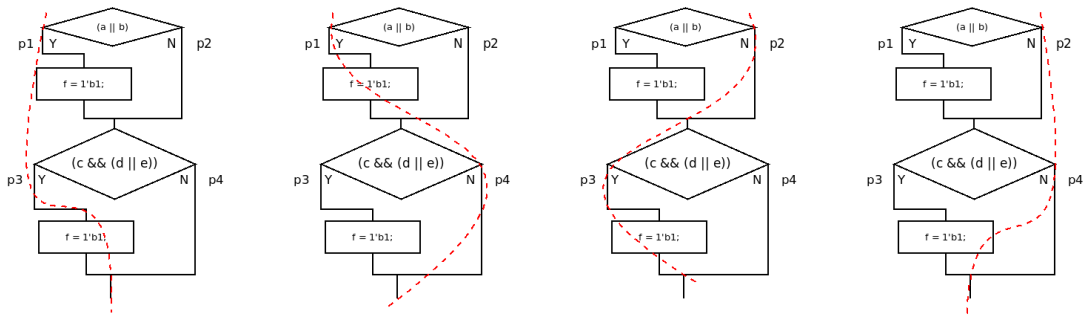


Figure 2.4: All possible paths of the control flow graph from Fig 2.3.

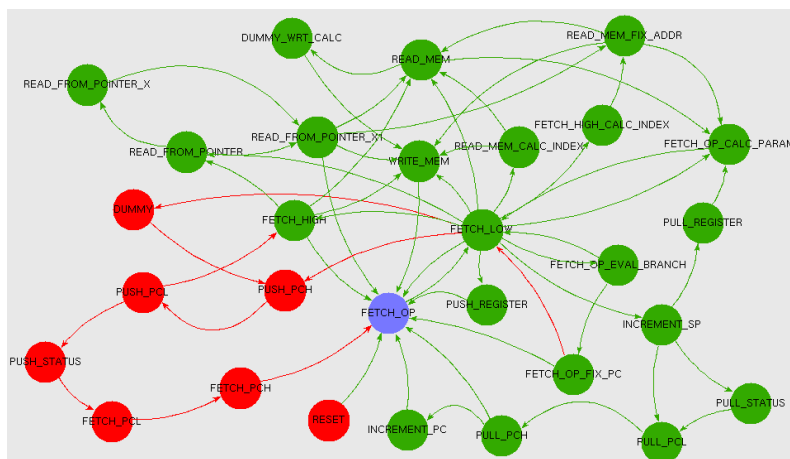


Figure 2.5: FSM coverage report for states and arcs.

The illustration does not consider transition coverage, instead it only shows arc and state coverage. Each state is represented by a circle and each arc is represented by an arrow. If an arc (or a state) has not been excited (visited) then it is displayed in red.

## 2.2 Stimuli generation

The current practice for verification of complex designs starts with a definition of a test list, comprised by a large set of events that the verification team would like to observe during the verification process (Section 2.5 contains a discussion on those). The list of tests might be implemented using random test generators that produce a large number of tests. Producing such tests in a suitable manner is an issue. Regarding design complexity, generating a set of testbenches that will properly stimulate the circuit and provide a meaningful answer (usually through a coverage report) is not simple. It requires deep design knowledge. Also, it requires experience obtained from the verification of previous projects. Otherwise engineers could be wasting time by writing two tests that achieve the same (or near the same) coverage goals.

Recently the approach for generating stimuli data has shifted to a constraint based approach such as the ones reported by (YUAN et al., 1999) and (KITCHEN; KUEHLMANN, 2007). This allows the simulation to reach more states with a test case that once was directed towards only a single feature of the design. This element alone has aided the verification effort because it has broadened the search space of a testcase while still keeping it controlled under a set of rules.

When a constraint based approach is used, usually it requires the use of a constraint solving engine. This adds another level of complexity to FV since two constraints, perhaps written by different engineers, could create a scenario that is not of desire. In other words, it may cause the simulation of a scenario that is not of interest for the design. There is also the issue of constraint contradiction, i.e., one constraint inhibits the other. Either way, simulation cycles are wasted if the applied constraints are not correct.

Another approach for generating stimuli data is referred as Coverage Directed (test) Generation (CDG). One example of such approach in the literature is presented by (FINE; ZIV, 2003), which is based on modeling the relationship between the coverage information and the directives to the test generator using Bayesian networks (a special type of directed graph).

## 2.3 Checking

When a designer completes the coding of a design unit, a single or a few modules implementing some elementary function, he or she verifies that it works as intended. This verification is casual and usually the waveform viewer is used to visually inspect the correctness of the response (BERGERON, 2006). Regarding the efforts of a verification engineer, instead of performing a casual waveform inspection, it is advisable to create some form of self-checking environment, like the one showed in Fig. 2.6.

The output of a module might be self-checked in several different ways. Usually the checking method is chosen based tightly on the stimuli generation method. For instance, if the input is created in a transaction fashion the output might be checked in the same manner. Another simpler example might use vectors to store simulation scenarios. Let us say that, for each set of inputs sent to the design, the correct response will also be known since the same set is submitted to the checker. So, a large vector is used to store both the inputs and the expected responses of the simulation.

In general, one is interested in checking both data and timing correctness from a design. One very common practice to check the flow of data in a design is to use a scoreboard technique with a reference model. A scoreboard is a temporary location for holding

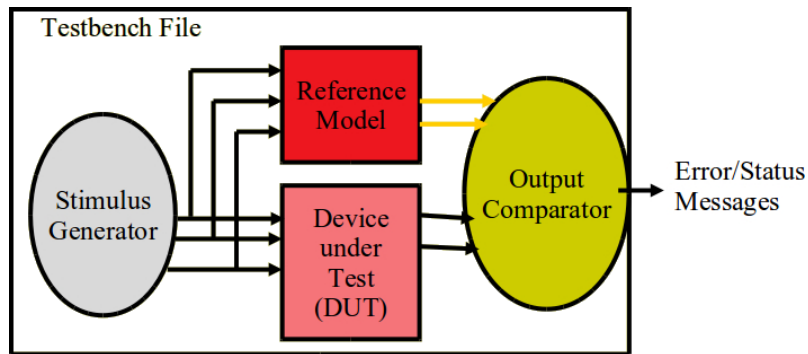


Figure 2.6: A generic verification environment capable of performing checking.

information that a checker will require (WILE; GOSS; ROESNER, 2005). The reference model dynamically produces the expected responses which are compared against the data already stored in the scoreboard. Once the simulation finished, if the scoreboard is not empty then something went wrong. For checking timing properties, it is more common to rely on some sort of assertion language, like the ones defined in the *e* language, SystemVerilog or PSL.

Checking is a complex subject since it is very design specific. There is no technique that is suitable for all types of designs. The verification engineer (or the verification environment architect) is responsible for choosing a method that fits well for some actual given design.

## 2.4 Verification methodologies

As explained by (IMAN; JOSHI, 2004), the focus on FV has consisted of a multi faceted approach where improvements in verification productivity were made possible through either:

- New verification methodologies, better suited for the verification of complex systems.
- Reusable verification components.
- HVLs to facilitate the new verification paradigms.

This section will discuss some of the verification methodologies that are in use as of today, as well as some of the associated HVLs of such methodologies. In the last few years different verification methodologies have been used by verification teams. The need for a verification methodology is clear nowadays: conforming to a set of standards allows the reuse of certain aspects of verification, thus diminishing the overall effort. The next subsections will detail some of the recent verification methodologies.

Before introducing each methodology and discussing its features, it is also important to assess which methodologies are currently most used. A few market researches have addressed this subject, such as (GRIES, 2009) and (Gary Smith EDA - GSEDA, 2009). The results of such researches is summarizes in Fig. 2.7 and Fig. 2.8, respectively.

The first research that was conducted was organized by a verification expert. He tried to answer the question of which verification solution is more widely used. Results show that a fair amount of the respondents (the horizontal axis of the image) still do not use

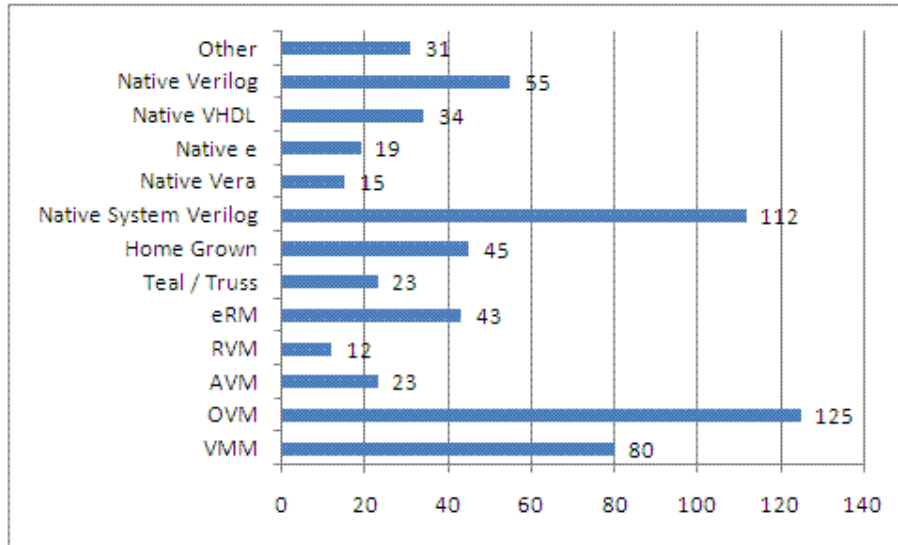


Figure 2.7: Current verification practices, divided by programming language.

any type of methodology, therefore performing verification using the native forms of the HVLs or homegrown solutions. These respondents were placed in the upper portion of the image. The use of languages that are not truly HVLs, like Verilog and VHDL, can also be seen in Fig. 2.7.

Regarding the ones that were placed in the bottom of the image, these are the respondents that actually rely on some form of methodology. A remarkable position is occupied by the Open Verification Methodology (OVM).

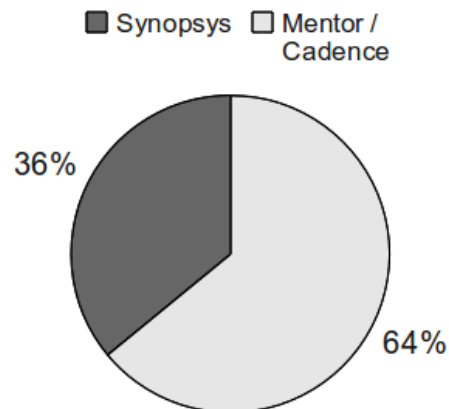


Figure 2.8: Current verification practices, divided by vendor.

The second research, organized by a provider of market intelligence and advisory services, has a different profile since it tried to identify which share of the market is occupied by which vendor. The image clearly states that Cadence and Mentor, together, have secured 64% of the market share. The reason for such is that both companies have joined strengths in developing and improving OVM. In other words, there is a great correlation between both researches. Also, the current trend seems to have SystemVerilog as the leading HVL since it is widely used in its native form and also through OVM and Verification Methodology Manual (VMM).

It is important to realize that, sometimes, a verification methodology is (almost) vendor specific. This means that only a few simulators are capable of handling such methodology or certain aspects of a methodology. For instance, each major vendor today has a simulator in its solutions portfolio. Attached to these simulators usually there is also at least one recommended methodology. This is the case of the OVM and the *e* Reuse Methodology (eRM), recommended and supported by Cadence. This is also the case of VMM, which is recommended and supported by Synopsys. A paradigm break is currently being adopted by the industry with the introduction of the Universal Verification Methodology (UVM). The evolution of all these methodologies is illustrated in Fig. 2.9.

2000	2001	2002	2003	2004	2005	2006	2007	2008	2009	2010	2011	...
Verity		Cadence										
vAdvisor		eRM					URM					
							urm +	OVM				
						Mentor	avm		ovm +			
						AVM				UVM		
			Synopsys						vmm			
			RVM		VMM							

Figure 2.9: A timeline containing the introduction year of each of the current verification methodologies.

Regarding Fig. 2.9, one can understand the evolution of the methodologies and the consortiums established between vendors. A few words regarding the methodologies illustrated in the image will be given in the next subsections. The exception is the vAdvisor, which was a collection of best practices for the *e* verification language developed in 2000 by Verity Design (now part of Cadence) and Universal Reuse Methodology (URM), which was a natural evolution of eRM. The methodologies will be addressed using a chronological order.

#### 2.4.1 *e* Reuse Methodology (eRM)

The basis of the eRM methodology is the reuse. Being so, the concept of an *e* Verification Component (eVC) was created, which is an abstraction that encapsulates the elements of a verification environment. This methodology comes with a comprehensive list of new concepts introduced by the *e* language, its syntax and semantics. The most remarkable feature of *e* is its Aspect Oriented Programming (AOP) semantic, which includes the regular Object Oriented (OO) semantic plus a greater level of customization. At the same time that AOP makes the *e* language unique, it also makes it more complex than a regular OO language.

In addition, the methodology allows the construction of architectural views and requirements of verification environments (i.e. randomly generated environments, coverage driven verification environments, etc.). The methodology also address standard ways to create the verification blocks in the architectural views (i.e. generators, checkers, monitors, coverage definitions, etc.). In other words, eRM defines a guideline and/or step-by-step instructions for building compliant eVCs, such as the one illustrated in Fig. 2.10 (IMAN; JOSHI, 2004).

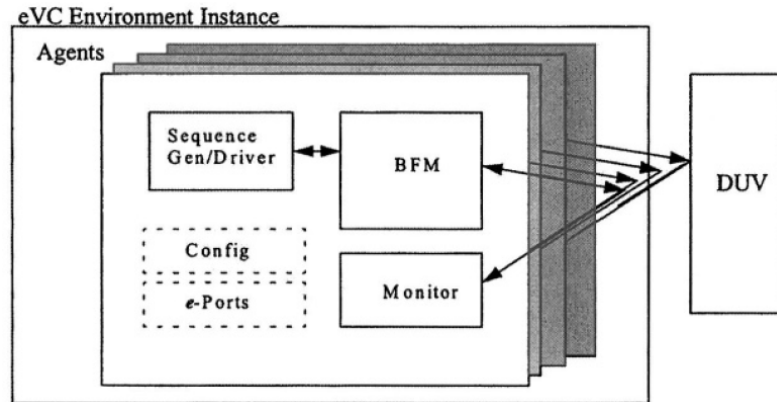


Figure 2.10: eVC environment instance.

#### 2.4.2 Reference Verification Methodology (RVM)

The Reference Verification Methodology (RVM) is a complete set of metrics and methods for performing FV. It was published by Synopsys in 2003 and it uses the Vera (HAQUE; MICHELSON; KHAN, 2001) language to create testbenches. This language was later used as the basis for creating SystemVerilog since it already had many of the desired features of SystemVerilog, like assertions and constrained randomness. The organization of a verification environment in RVM is similar to the one in VMM, as illustrated by Fig. 2.11.

#### 2.4.3 Verification Methodology Manual (VMM)

VMM was jointly authored by ARM and Synopsys back in 2005. It already uses SystemVerilog as the base programming language. The premise of the methodology is that it would finally enable users to take full advantage of all possible features of SystemVerilog in a concise way. So, engineers could benefit from assertions, reusability, testbench automation, coverage, formal analysis, and other advanced verification technologies, all within SystemVerilog.

Reuse is also a great concern of VMM. In order to have a common verification environment that facilitates reuse and extension to take full advantage of automation, a layered testbench architecture is used. This approach makes it easier to share common components between projects (ANDERSON et al., 2006). The VMM for SystemVerilog testbench architecture comprises five layers around the DUT, as shown in Fig. 2.11.

#### 2.4.4 Advanced Verification Methodology (AVM)

This was the first effort of Mentor to enter the FV methodology market. The Advanced Verification Methodology (AVM) was claimed by Mentor as the first open, non-proprietary methodology that supports system-through-RT level verification. The open claim comes from the way that it was provided, through the AVM Cookbook (GLASSER et al., 2008). Such book was released free of charges and it includes examples of code that could be cut and pasted into customer environments to build testbenches. Source code for base class libraries, utilities and implementation examples were also available for free.

The greatest distinction of this methodology is that it was aimed at two different languages: SystemC and SystemVerilog. The AVM also features an OO coding style to reduce the amount of testbench code and a modular architecture to enable reuse.

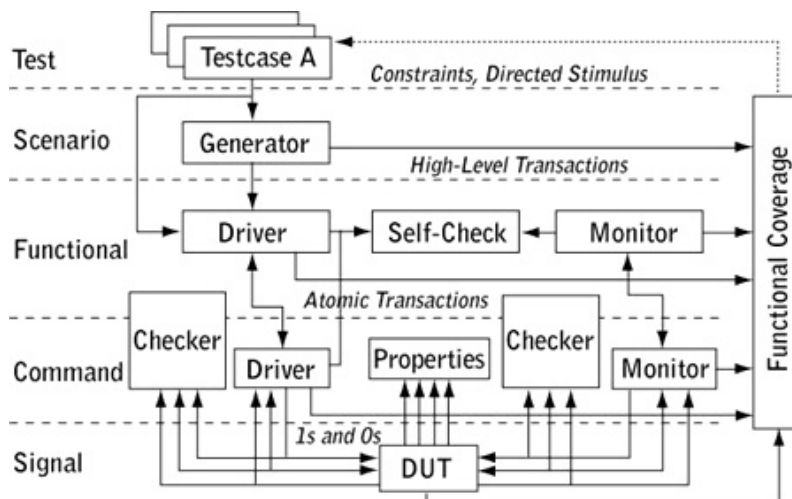


Figure 2.11: Example of a VMM verification environment architecture.

### 2.4.5 Open Verification Methodology (OVM)

Cadence and Mentor developed the OVM (CADENCE; MENTOR, 2007), an open-source SystemVerilog based class library and methodology. As expected, it defines a framework for reusable verification IP and testbenches. In a certain way, it combines the best practices of Cadence’s eRM (and later URM) with the best practices of Mentor’s AVM. It also has a Cookbook (GLASSER, 2009), in the same fashion that the AVM did.

Although the methodologies of each of the three major vendors had success within their own customer base, each ran only on its own simulator. There was no attempt at cross-vendor support or any form of standardization. OVM was the first key step in this direction in early 2008, when Cadence and Mentor delivered the first SystemVerilog version of OVM. Like its AVM and URM predecessors, the OVM was provided as open source. The actual improvement is that it was the first methodology to be tested and guaranteed to work on multiple simulators.

### 2.4.6 Universal Verification Methodology (UVM)

Although the OVM enjoyed wide adoption, it was not directly endorsed by any organization. This would only change in 2010 with the introduction of the UVM. The first serious industry attempt to standardize a verification methodology began in 2008, when the Verification IP Technical Sub-Committee was formed within Accellera. It was already clear that the recently launched OVM would be very successful. But, given its longer history, there were many VMM users as well. Thus, the initial focus for the standardization was to figure out how VMM-based environments could function in an OVM verification environment (and vice versa). Cadence, Mentor and Synopsys worked with representatives from numerous companies to define and validate a standard interoperability solution that could link the two methodologies.

The results of such standardization are still not clear, since the UVM 1.0 standard by Accellera was only released a few days before this thesis was finished.



## 2.5 Verification plan

Most verification efforts are initially described through a verification plan, where each test that will be implemented is described in details. The verification plan is to the FV what the specification is to the coding process. Also, most tasks related to verification are required to be properly checked, in a sense that management and planning play an important role in the verification process. So, a document like a verification plan is very important to manage the process.

Verification plan's development is often a laborious process held at the start of a project. Developing a comprehensive verification plan is difficult and is one thing, while keeping it in sync with the requirements as they evolve is quite another. The first task requires a lot of creative thinking while the latter requires a precise management. Attempts to manually keep the requirements, the plan, and the execution environment in sync are time consuming and, most of the time unreliable, to say the least. Analysis of execution results is also a intensive task due to the volume of data generated in the process. Globally distributed teams executing the same plan also add a new dimension of complexity.

Historically, teams have used standard Office tools (word processor, spreadsheet editor, etc.) for managing verification projects. However, this is useful for small projects with a single and centralized team with only a few members. It is laborious and time consuming to keep such spreadsheets up-to-date while changes occur in requirements, plans and execution. Some more elaborated solutions rely on Wiki pages (EBERSBACH; GLASER; HEIGL, 2005) (or a library of pages), since the Wiki solution is already aimed at collaborative work. Finally, there are also solutions based on executable verification plans (Cadence Design Systems, Inc., 2005) (International Business Machines Corporation - IBM, 2009), which are responsible for centralizing the verification efforts of a project. To achieve this kind of solution an automated tool is required.

One possible generic format for a verification plan is given on the next subsections, where a verification plan is assumed to be divided into four parts: an overview of the plan, a feature list, a test list and the coverage goals. Later, on Chapter 4, the format of the verification plan used by VEasy is described.

### 2.5.1 Overview

This portion of the plan contains an overview of the whole process. Usually it contains a table with the general configurations and/or characteristics of that plan. One example of such table is given in Tab. 2.4.

Table 2.4: Example of an overview table of a verification plan.

<b>DUT</b>	project21
<b>Verification method</b>	Functional (by simulation)
<b>Design files</b>	file1.v file2.v top.v
<b>Verification methodology</b>	OVM
<b>HVL</b>	SystemVerilog
<b>Simulator</b>	SimX
<b>Extras</b>	Assertions coded in PSL

Also, this section usually includes block diagrams from both the design and the ver-

ification environment. Such block diagrams are used to explicitly define the hierarchy and relations between the modules (from the design) and units (from the verification environment). One example of a possible and simplified hierarchy is shown in Fig. 2.12 (BERGERON et al., 2005). In the image it is possible to see a list of testcases interacting with the environment while the environment interacts with the DUT. The actual units here are not shown since this type of hierarchy decision is typically dependent of the verification methodology being used, as it was highlighted in Section 2.4.

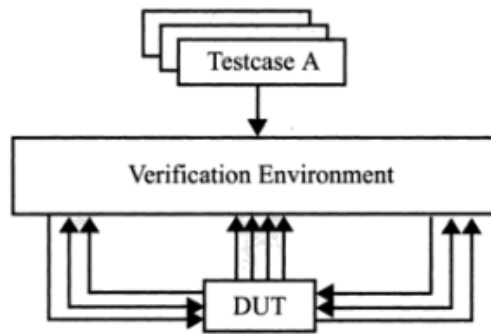


Figure 2.12: Simplified block diagram of the verification plan.

### 2.5.2 Feature list

This is actually the first step in the verification of a design. This section of a verification plan contains an exhaustive list of all the expected functionalities of a design, which were extracted from the design specification. In case of a multi-module design one table for each module might be created (in a bottom-up type of verification, for example). In (HAQUE; MICHELSON; KHAN, 2001), the authors propose a methodical approach for extracting significant and relevant features of a design by first analyzing the interfaces of the design, then the functions and finally the corner cases.

One example of such feature list is shown in Tab. 2.5.

Table 2.5: Example of a feature list from a verification plan.

Feature ID	Description
F1	The DUT must perform operation A.
F2	The DUT must accept on-the-fly configuration changes from operation mode M1 to M2 and vice-versa.
F3	The DUT must accept packets from protocol versions 1 and 2.

It is possible to notice that the features in this type of list are strictly functionally related. In other words, this list of features must not contain features that express power or timing constraints, e.g. “the design must work at 100Mhz”. This type of constraint must be addressed in the actual design specification.

### 2.5.3 Test list

This section of a verification plan contains an exhaustive list of all the tests that will be written in order to check the list of features. Each test is then later transformed into a testbench typically using some HVL. Therefore a test is associated with at least one

feature. One example of such list is given below in Tab. 2.6, which contains tests that match the features of Tab. 2.5.

Table 2.6: Example of a test list from a verification plan.

Test ID	Related feat.	Description
T1	F1, F2	The DUT configuration is set to mode M1 and the operation A is performed.
T2	F2	A packet that changes the operating mode must be send every [1000:10000] cycles.
T3	F3	Random data must be sent using only packets of version 1.
T4	F3	Random data must be sent using only packets of version 2.
T5	F3	Random data must be sent using packets of versions 1 and 2.

#### 2.5.4 Coverage goals

This section of a verification plan contains all the coverage metrics and their goals. A simple table like the one in Tab. 2.7 is used to define the coverage goals and the current values. The first two columns define the type of coverage while the others define the goal for each metric and the current value measured in the simulation runs of a certain day, respectively. Finally, such tables might also contain the next hypothetical actions defined in order to increase the coverage.

Table 2.7: Example of a list of coverage goals from a verification plan.

Coverage type	Coverage metric	Goal	Current value	Action
Structural	Block	100%	99% as Jan. 1.	Evaluate possible dead code on module A.
Structural	Expression	-	95% as Jan. 5.	Use more seeds in nightly runs. Schedule simulation grid for next week.
Structural	Toggle	100%	95% as Jan. 9.	Randomize data on memory bus.
Functional	Data	100%	98% as Jan. 2.	Create specific test to augment cross coverage of configuration modes and packet types.
Functional	Control	100%	50% as Jan. 1.	Debug failing assertions on all modules. Evaluate assertions reports, find holes.

#### 2.5.5 Other sections

Other sections are also seen in verification plans and are generally used to store management information. One data commonly managed is the number of simulator licenses being used and a schedule to avoid lack of licenses. Also, it is common to manage the

human resources required to perform the DUT verification in the verification plan. This task is achieved with a list of names and responsibilities.

### 3 EVALUATING THE CHALLENGES

Two of the goals of VEasy is to allow simpler and faster ways to perform FV. Before implementing the tool some preliminary experiments were conducted in order to put figures in the actual challenges that verifying a design imposes. Initially, the coverage collection time and the data stimuli generation time were measured extensively using different simulators and designs. The next section deals with the coverage collection measurements while Section 3.2 will deal with the generation evaluation.

#### 3.1 Measuring simulation overhead caused by coverage

Before explaining how the developed tool suite collects and analyzes coverage, it is important to understand and measure the impact that coverage has on simulation. For this task a set of circuits was chosen along with a set of commercial simulators. The circuits were chosen based on the different logic constructions they contain. Four circuits were chosen: *dffnrst* is a D type flip-flop with reset, *fsm* is a Finite State Machine (FSM) with 8 states and each state performs an 8-bit wide operation, *adder* is a 16 bit adder with a enable signal while *t6507lp* (PAGLIARINI; ZARDO, 2009) is an 8-bit microprocessor with 100 opcodes and 10 addressing modes.

A deeper analysis of the properties found on those circuits is performed in Tab. 3.1, where the number of blocks, expressions and toggle bins is shown for each circuit. Toggle bins are defined as the total number of transitions of interest. Since we are only interested in one-to-zero and zero-to-one transitions, dividing the number of toggle bins by two gives the total number of signals of a given circuit. For example, the *dffnrst* circuit has 8 toggle bins to cover 4 signals, the *d* input, the *q* output, plus *clock* and *reset* signals.

Table 3.1: Some properties of the circuits being analyzed.

	<b>dffnrst</b>	<b>fsm</b>	<b>adder</b>	<b>t6507lp</b>
# of blocks	3	6	15	294
# of expressions	1	2	10	96
# of toggle bins	8	70	74	604

Two commercial simulators from major vendors were evaluated using those circuits. One of the simulators is widely used in the Field Programmable Gate Array (FPGA) domain while the other is widely used in the ASIC domain. Neither of the simulators allow the execution and/or publishing of benchmark simulations so these simulators will be referred hereinafter to as simulator A and simulator B, respectively.

For each circuit a Verilog testbench was developed using some simple constraints: all the data signals are kept completely random except for *reset*, which is triggered only once and during the first simulation cycle. All testbenches were configured to run up to 10 million clock cycles.

In order to perform a fair comparison, all simulations were performed using the same computer configuration (64bit OS, 6GB of memory and a quad-core processor operating at 2.66Ghz). Also, no waveform output was requested from the simulators. No *\$display()* or *printf()* operations were performed in the testbenches. File writing operations were kept at a minimum, just the necessary to analyze the coverage data post simulation. Also, coverage was configured to be only collected for the DUT's code since both simulators try by default to cover the testbenches as well.

The simulation overhead for enabling toggle coverage in simulator A is shown in Fig. 3.1. The simulation overhead for enabling toggle coverage in simulator B is shown in Fig. 3.2.

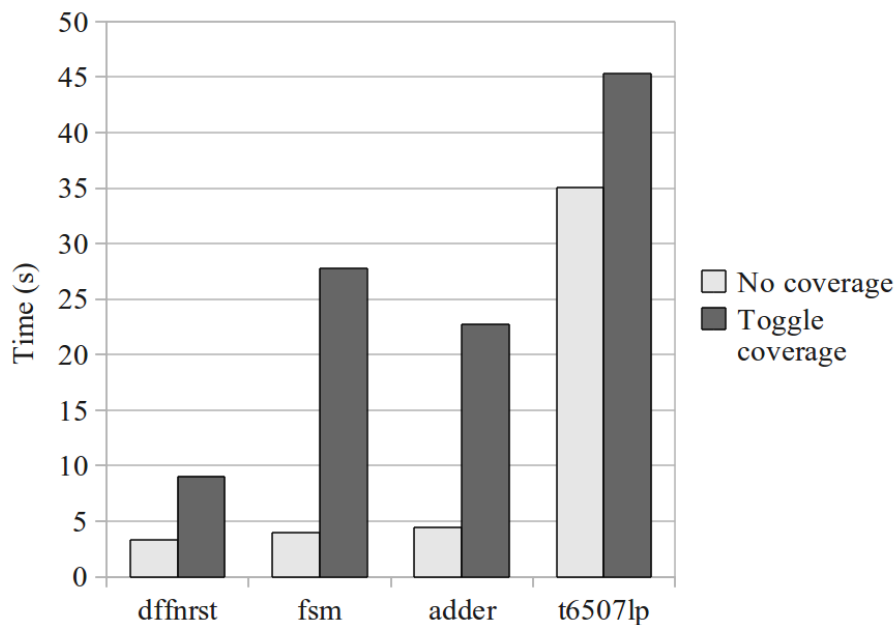


Figure 3.1: Simulation overhead due to toggle coverage in simulator A.

The overheads measured in both simulators may be considered high. Regarding simulator A, one may notice that the *fsm* circuit has the highest proportional overhead, while regarding simulator B, one may notice that simulating the *t6507lp* circuit with toggle coverage more than doubles the simulation time. That is why some companies choose to perform most of the coverage tasks only near the project's end. Although this choice may increase the number of simulation cycles performed during the development of the project, verification engineers will receive feedback of the quality of the testbenches late. Clearly this scenario might lead to inefficient use of engineering resources. It also worth mentioning that the same testbench is simulated more than once when considering regression testing, which increases the relevance of coverage overheads.

Figures 3.1 and 3.2 show only the overhead created by toggle coverage. Although the other coverage metrics also represent significant overheads, toggle coverage is the most severe one, as shown by Tab. 3.2, which summarizes the overheads created by all the three metrics plus the overhead of applying the three metrics combined, i.e., at the same time,

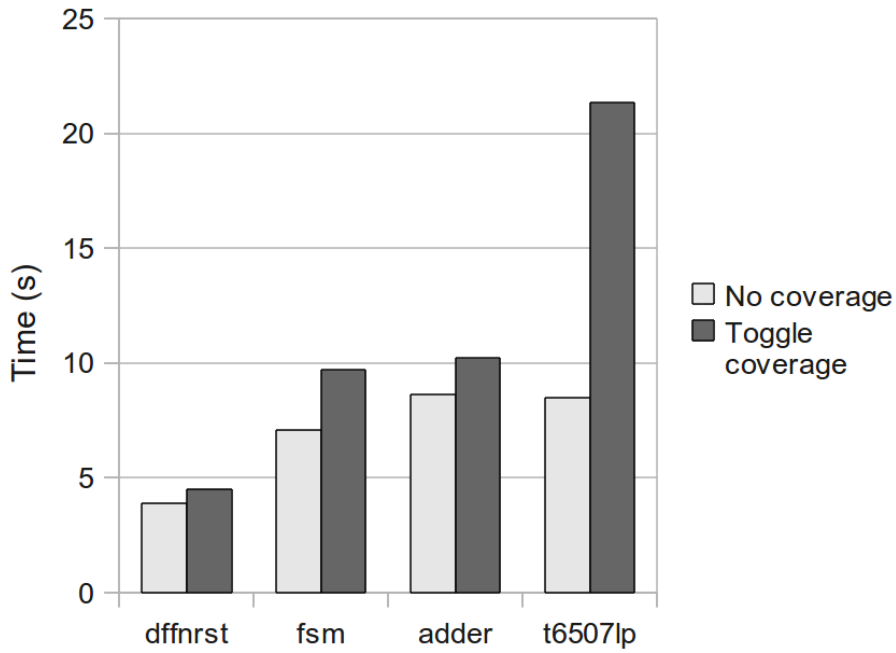


Figure 3.2: Simulation overhead due to toggle coverage in simulator B.

when considering only simulator A. Table 3.3 contains the same data regarding simulator B.

Table 3.2: Simulation overhead measured using simulator A.

Circuit	Block	Expression	Toggle	All combined
dffnrst	163.11%	160.67%	175.91%	175.91%
fsm	532.24%	539.55%	598.99%	600.25%
adder	370.54%	371.21%	407.14%	407.37%
t6507lp	12.43%	0.28%	29.04%	29.35%

Table 3.3: Simulation overhead measured using simulator B.

Circuit	Block	Expression	Toggle	All combined
dffnrst	2.56%	1.28%	15.51%	38.46%
fsm	1.41%	1.41%	36.48%	60.56%
adder	1.16%	1.16%	18.60%	39.53%
t6507lp	5.88%	7.06%	151.00%	208.24%

As seen on both tables, the time it takes to simulate a circuit with all coverage metrics combined is directly related to the toggle coverage simulation time. This is specially true for simulator A. It is also possible to notice that simulating the largest of the circuits (*t6507lp*) has created the smallest of the overheads in simulator A. Actually, the simulation time of this circuit is already considerable without coverage, as shown in Fig. 3.1. Therefore the measured overhead is not so severe. Yet, this is a particularity of simulator A since the results shown for simulator B and later from our own simulator reveal

otherwise. This type of scenario has influenced our circuit selection, in which we have purposely chosen circuits that are purely sequential (a flip-flop), purely combinational (an adder) and also mixed circuits of different sizes (a simple fsm and a processor).

It is also important to mention that both commercial simulators are event-driven. VEasy, which is a cycle-accurate simulator, will be compared against these simulators later. Since the simulators have different internal mechanisms, we chose not to compare the actual simulation time but instead we are comparing only the overheads. That is the reason why the values reported so far are percent wise. Otherwise, the values measured using VEasy would be smaller by at least one order of magnitude.

### 3.2 Measuring simulation overhead caused by data generation

From a user point of view it is clear that creating the data sequences that will excite (all) the functionalities of a design is a complex task. Now, assuming those sequences are already done, it is interesting to measure the overhead that they create in the overall simulation. For this purpose the same testbenches of the last section were evaluated plus a new circuit was considered, which is referred as *seq*. This circuit is sort of a sequence detector circuit.

In order to perform the experiment one huge difference was introduced: the actual DUT was disconnected from each of the testbenches, i.e., the signals are still being generated but they are not connected to any module. Then the time it takes to simulate the testbenches alone was measured. The comparison against the original simulation time of the testbenches is presented in Fig. 3.3.

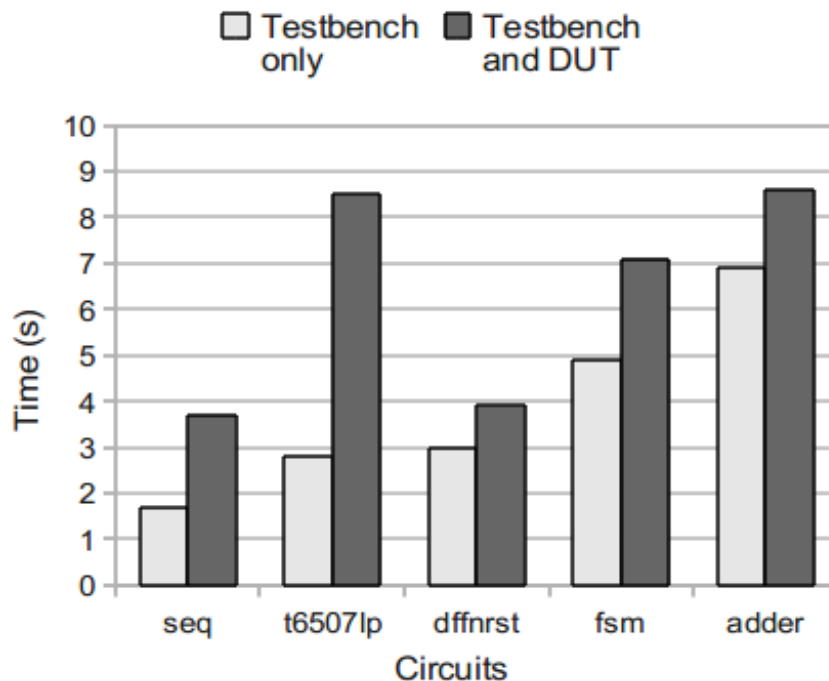


Figure 3.3: Simulation overhead due to data generation.

The results from Fig. 3.3 were gathered using commercial simulator A. One might notice that the data generation is responsible for a large portion of the actual simulation time of a verification environment. For circuits like *dffnrst* and *adder* one might notice that the generation of stimuli data (*testbench only*) corresponds to a great portion of the



simulation time (*testbench and DUT*). The actual percentage values were calculated by dividing the testbench only time by the regular simulation time. These values are showed in Tab. 3.4.

Table 3.4: Evaluation of stimuli generation time using commercial simulator A.

<b>Circuit</b>	<b>seq</b>	<b>t6507lp</b>	<b>dffnrst</b>	<b>fsm</b>	<b>adder</b>
<b>Simulation of TB only (s)</b>	1.7	2.8	3	4.9	6.9
<b>Simulation of TB and DUT (s)</b>	3.7	8.5	3.9	7.1	8.6
<b>Generation time with respect to simulation of TB and DUT</b>	45.95%	32.94%	76.92%	69.01%	80.23%

One might see that the data generation might be responsible for more than 80% of the simulation time. This is clearly a substantial value which makes generation one of the main sources of simulation overhead. Some of the results presented in this chapter plus a detailed discussion of the sources of simulation overhead are given in (HAACKE; PAGLIARINI; KASTENSMIDT, 2011).

At this point two of the main sources of simulation overhead have been defined: generation and coverage collection. There are other sources, remarkably the evaluation of assertions and the rule solving procedures when constrained randomness is applied. Yet these two are deeply related with the design while the ones showed in this chapter have more general applicability. Thus, the results that will be later showed will address the coverage and generation overheads. But, before starting the discussion of such results, VEasy will be presented in the next chapter.



## 4 VEASY FLOW AND THE VERIFICATION PLAN

As a starting point to describe VEasy it is important to mention the two work-flows of the tool and the way they interact with each other. The tool suite has two very distinct work-flows:

- Assisted flow.
- Simulation flow.

The assisted flow is supposed to be used at the beginning of a project since it will analyze the project prior to simulation. Only the assisted flow of the tool performs linting (BENING; FOSTER, 2001), which starts when the Verilog (IEEE, 1995) description of the DUT is parsed and analyzed. Chapter 5 deals with the linting capabilities of VEasy and describes several possible violations in details. The assisted flow receives this name because the linting debug interface assists the user in removing violations from the source code.

Once the description complies with the linting rules the simulation flow is enabled. This way linting guarantees that the input is written using strictly RTL constructions (i.e. it is synthesizable), therefore no behavioral-only constructions were used. Being so, the internal VEasy simulator is capable of simulating the design. These checks are important because they provide an easy way to analyze and debug the code being developed by the user even before synthesis is performed.

From that same input that contains the design description, the interfaces (i.e., input and output signals) and the special signals (i.e., clock and reset) are automatically extracted. This information is then used to build a template of a verification plan, as shown in Fig. 4.1.

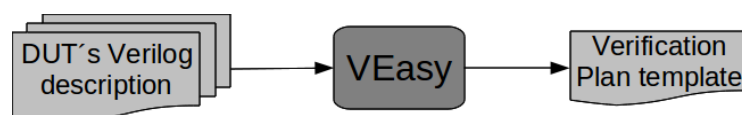


Figure 4.1: VEasy assisted flow.

The template generated in the assisted flow later becomes the input of the simulation flow, which is illustrated in Fig. 4.2. Initially, a verification plan file is loaded. This file contains all the information that is required to generate and simulate the test scenarios that the user creates through the GUI. VEasy then is ready to create a simulation snapshot, i.e. an executable file, that combines the circuit description and the test generation functionality. At this point the use of a golden model is optional.

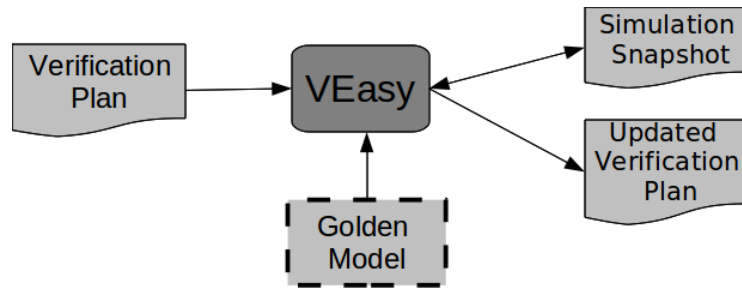


Figure 4.2: VEasy simulation flow

All the code generated by VEasy is ANSI-C (ANSI, 1989) compliant, which allows it to be used in a several combinations of platforms and compilers. For the experiments performed later in this thesis the GCC compiler (STALLMAN, 2001) was used, along with some optimization options (JONES, 2005).

After the simulation is complete the tool automatically stores the coverage results, saves them into the same verification plan and provides this info for the user analysis. If suitable a new simulation round may be started in a attempt to reach coverage holes.

The verification plan file used by VEasy is actually a complete view of the verification effort. It includes the traditional lists of features and associated test cases but it also contains simulation and coverage data. This approach makes it a unified database of the current verification progress. The following is an example of a full verification plan for a simple design, a D type flip flop (coverage data has been omitted).

```

= RTL =
== General ==
* rtl_path /home/samuel/VEasy/work
* rtl_file dffnrst.v
* rtl_md5sum 0xa36e351cc8e4fdea83d9824fc99b1bf5
== Special signals ==
* rtl_clock clk
* rtl_reset rst
= FUV =
== Input Interface ==
* d 1
== Output Interface ==
* q 1
== Sequence ==
* seq_name one
* seq_level 0
* seq_list
* seq_members
* d 1 PHY
* keep == 1
== Sequence ==
* seq_name random
* seq_level 0
* seq_list
* seq_members
* d 1 PHY
== Sequence ==
* seq_name zero
* seq_level 0
* seq_list
  
```

```

* seq_members
* d 1 PHY
  * keep == 0
== Sequence ==
* seq_name one_zero
* seq_level 1
* seq_list one zero one zero one zero
* seq_members
== Sequence ==
* seq_name one_zero_then_random
* seq_level 2
* seq_list one_zero random random random random random random
random random random random random random random random
random
* seq_members
= COV =
* cov_branch true
* cov_exp true
* cov_toggle true
* cov_input false
* cov_output false
= RST =
* rst_type 1
= SIM =
* sim_cycles 10000000
* sim_start start
* sim_vcd false
= VPLAN =
== Feature ==
* feat_id F1
* feat_text The design must put D into Q at every
clock cycle.
== Test ==
* test_id T1
* test_text Sends a string of alternating ones and zeros.
* test_for F1

```

Notice that the verification plan file is written using a Wiki-like syntax. This makes it easier to be parsed since there are several distinguishable marks (tokens) in the text. And, at the same time, the Wiki-like format aids the visualization of the file since it might be rendered by a web browser. In the attachments there is an example of how a portion of the above described file would look like when rendered.

A simple analysis of the verification plan file format reveals that it is organized in sections. These sections are marked by the pairs of equal signs, one at the beginning and one at the end of a line (e.g. = RTL =). The following is an explanation on the contents of each section of the file also followed by an explanation about each of the possible configuration variables that are embedded into the file.

**RTL** This section deals with the RTL related configurations of the verification plan, such as the special signals and the path to the actual design files. The meaning of each configuration variables of this section is:

**rtl\_path** Sets the path of the folder where the design files are to be found.

**rtl\_file** Sets the name of the design files, including extension.

**rtl\_md5sum** Stores a md5sum, which is a 128-bit MD5 hash (RIVEST, 1992). It is used by VEasy to identify changes in a source file. If any of such changes is detected then the current coverage percentages might be invalid.

**rtl\_clock** Sets the name of the master clock signal in the design.

**rtl\_reset** Sets the name of the reset signal in the design. It might be left empty for designs without reset.

**FUV** This section deals with the Functional Verification related configurations, such as the design interfaces and the list of sequences. The input interface of the design is described using the token `== Input Interface ==` while the output interface uses the token `== Output Interface ==`. Each sequence definition starts with a `== Sequence ==` token. The configuration variables of this section are detailed in the following text. The actual layered methodology is later addressed in Chapter 8. At this point it is important to realize that each sequence has a name, a level and some yet to be explained content.

**seq\_name** Sets the name of the current sequence being defined.

**seq\_level** Sets the level in which the current sequence is placed in the hierarchy.

**seq\_list** List of sequences that are used to compose the current sequence. It may contain any number of sequences in it as well as zero sequences.

**seq\_members** List of members that are part of the current sequence. It may contain any number of members in it. Sequences from *layer0*, i.e., sequences which level is 0, always have all the physical inputs of the design as members.

**COV** This section deals with the coverage related configurations, such as turning on or off certain metrics. The configuration variables of this section are detailed as follows:

**cov\_block** Enables block coverage in the simulation.

**cov\_exp** Enables expression coverage in the simulation.

**cov\_toggle** Enables toggle coverage in the simulation.

**cov\_input** Enables input functional coverage in the simulation.

**cov\_output** Enables output functional coverage in the simulation.

**RST** This section deals with the reset methodology related configurations, such as when and how often reset must be performed. The configuration variables of this section are detailed as follows. The different methodologies available for performing reset are explained in Section 6.3.

**rst\_type** Enables one of the reset methodologies available. It may contain extra operands to define the reset probability or timing. Check section 6.3 for more details.

**SIM** This section deals with the simulation related configurations, such as the number of cycles and the output format. The configuration variables of this section are detailed as follows:

**sim\_cycles** Defines the total number of clock cycles that will be performed in the next simulation run.

**sim\_start** Defines which sequence will be used as a starting point for the next simulation run.

**sim\_vcd** Enables the simulation to be exported to Value Change Dump (VCD) format. Check section 6.6 for more details.

**VPLAN** This section deals with the lists of features and tests of the verification plan, as mentioned in Subsections 2.5.2 and 2.5.3. The configuration variables of this section are detailed as follows:

**feat\_id** Sets the id of the current design's feature being described.

**feat\_text** Detailed textual description of the feature.

**test\_id** Sets the id of the current test being described.

**test\_text** Detailed textual description of the test.

**test\_for** List of the features that are possible to excite using this test. Must contain at least one feature otherwise the test is meaningless.

The next sections will detail different features of the tool. The order in which the features are presented is the same as they are arranged in VEasy's flow. Some additional and general information regarding the tool might be found in (PAGLIARINI; KASTENS-MIDT, 2011b). Let us start by describing the linting feature.





## 5 LINTING

In computer programming lint was the name originally given to a particular program that flagged some suspicious and/or non-portable constructs (likely to carry bugs in it). Originally it was made for linting source code written in the C language. The term has evolved and now is generically applied to any tool that flags suspicious or erroneous usage in any computer language. In other words, what lint-like tools perform is a static analysis of source code.

Linting is a must have step in a flow, as recognized by experts (BAILEY, 2005). From a verification point of view, linting might be interpreted as an additional verification effort to find potential design problems such as mismatches in vector sizes, for example. It might also be seen as a more generic approach to formal verification checks in the entire design flow (YAROM; PATIL, 2007).

As previously mentioned, linting guarantees that the input is written using strictly RTL constructions, i.e., it is synthesizable. Therefore no behavioral-only constructions were used. These checks are important because they provide an easy way to analyze and debug the code being developed even before synthesis is performed. It is also worth mentioning that some companies have tight standards for code writing and linting might also apply in that situation, i.e., it might be used to check if a piece of code complies with the company code writing rules and/or best practices.

Most of the linting rules that VEasy checks for are listed in the next sections, which are named after the actual rule identification in the tool. Each section also contains a brief description of the violation and an example of a violating code. The examples are written in Verilog.

### 5.1 BASE

The BASE linting rule detects any illegal blocking assignment at a sequential always block. Regarding Verilog, there are two assign operators: `<=` and `=`. These are referred to as the non-blocking and blocking assignment operators. They are used to infer sequential and combinational logic, respectively. So, using the blocking assignment within a block that is supposed to be sequential is considered an error. The following piece of code contains one example of such violation in line 6, where the reg *a* is being assigned erroneously.

---



---

```

1 module example( clk , in );
2 input clk , in ;
3 reg a ;
4
5 always @(posedge clk) begin
6     a = in ;
7 end
8 endmodule

```

---



---

Figure 5.1: Example of a Verilog code that violates the BASE rule.

## 5.2 BCSI

The BCSI linting rule detects any binary constant sized incorrectly. The following piece of code contains one example of such violation in line 5. One can see that the *a* signal is only one bit wide but the assignment in line 5 is trying to store two bytes in it. Although the value could be truncated and stored it is considered a bad practice to perform assignments using containers or constants of different sizes.

---



---

```

1 module example( clk , in );
2 input clk , in ;
3 wire a ;
4
5 assign a = 2'b00 ;
6 endmodule

```

---



---

Figure 5.2: Example of a Verilog code that violates the BCSI rule.

## 5.3 DCSI

The DCSI linting rule detects any decimal constant sized incorrectly. The following piece of code contains one example of such violation in line 5. One can see that now the *a* signal is two bits wide but the assignment in line 5 is trying to store the decimal value 10 in it, which would require 4 bits.

---



---

```

1 module example( clk , in );
2 input clk , in ;
3 wire [1:0] a ;
4
5 assign a = 2'd10 ;
6 endmodule

```

---



---

Figure 5.3: Example of a Verilog code that violates the DCSI rule.

## 5.4 DIRE

The DIRE linting rule detects any directives and alerts the user that it will be ignored. Directives are used for different purposes in Verilog, from controlling the simulation scale/step pair to defining the generation of blocks conditionally (e.g. ``ifdef` directive). Verilog directives are recognizable by the ``` character. The following piece of code contains examples of such violations in lines 1 and 2<sup>1</sup>.

---



---

```

1  `timescale 1ns / 10ps
2  `include moduleb.v
3
4  module example( clk , in );
5  ...
6  endmodule

```

---



---

Figure 5.4: Example of a Verilog code that violates the DIRE rule.

## 5.5 HCSI

The HCSI linting rule detects any hexadecimal constant sized incorrectly in the same manner that BCSI and DCSI do. The following piece of code contains one example of such violation in line 5, where the register *a* is being assigned the value *b* (11 in decimal). Such value clearly does not fit into a 2 bit wide register.

---



---

```

1  module example( clk , in );
2  input clk , in ;
3  wire [1:0] a ;
4
5  assign a = 2'hb ;
6  endmodule

```

---



---

Figure 5.5: Example of a Verilog code that violates the HCSI rule.

## 5.6 IASS

The IASS linting rule detects any assignments to inputs, which by default are not controllable by the inner module. The following piece of code described in Fig. 5.6 contains one example of such violation in line 5.

## 5.7 IDNF

The IDNF linting rule detects any references made to identifiers (any reg, wire, local-param or param) that are not found within the current module. The following piece of code described in Fig. 5.7 contains one example of such violation where a typo has been made and instead of using the identifier *a* the user has used the identifier *aa*.

<sup>1</sup>It is considered a bad practice to define the design's hierarchy by including files. There are exceptions when including a file is acceptable, e.g. a global constant definition file.

---



---

```

1 module example( clk , in );
2 input clk , in ;
3 wire [1:0] a ;
4
5 assign in = 1'b0 ;
6 endmodule

```

---



---

Figure 5.6: Example of a Verilog code that violates the IASS rule.

---



---

```

1 module example( clk , in );
2 input clk , in ;
3 wire [1:0] a ;
4
5 assign aa = 1'b0 ;
6 endmodule

```

---



---

Figure 5.7: Example of a Verilog code that violates the IDNF rule.

## 5.8 IDNP

The IDNP linting rule detects any references made to identifiers that are not found within the current module port list. The following piece of code contains one example of such violation where a input named *in* has been declared (line 2) but it is not found in the module's list of ports, which only contains the *clk* signal.

---



---

```

1 module example( clk );
2 input clk , in ;
3
4 endmodule

```

---



---

Figure 5.8: Example of a Verilog code that violates the IDNP rule.

## 5.9 LPNA

The LPNA linting rule detects any parameter (or localparameter) that has been named in any style other than using all letters in uppercase. Although this is not exactly an error it is considered a bad practice to create code that way, i.e., regs and wires should be named using lowercase letters while parameters should be named using uppercase letters. VEasy is able to detect and warn the user if such behavior is found. The following piece of code described in Fig. 5.9 contains one example of such violation in line 4, where one can see that a localparameter was declared with the name *start*. Another violation is present in line 6, where a register was declared with the name *TEMP*.

## 5.10 MBAS

The MBAS linting rule detects if any variable (register or wire) has been assigned in multiple blocks. Since this type of behavior might lead to simulation mismatches be-

---

```

1  module example( clk , in );
2  input clk , in ;
3
4  localparameter start = 0;
5
6  reg TEMP;
7
8  endmodule

```

---

Figure 5.9: Example of a Verilog code that violates the LPNA rule.

tween the Verilog description and the actual synthesized hardware, it is not allowed. For example, in the following piece of code of Fig. 5.10, it is not possible to determine if the register *ff* is used to build combinational (as suggested by line 6) or sequential logic (as suggested by line 10).

The example contains one particular situation where the actual logic behavior cannot be defined. Yet, if the two always blocks of the code were of the same type the error still would be reported. In this case the error would be reported because race conditions might occur, although the logic behavior is known.

---

```

1  module example( clk , in );
2  input clk , in ;
3  reg ff ;
4
5  always @(*) begin
6      ff = 0;
7  end
8
9  always @(posedge clk) begin
10     ff <= 1;
11 end
12
13 endmodule

```

---

Figure 5.10: Example of a Verilog code that violates the MBAS rule.

## 5.11 NBCO

The NBCO linting rule detects any illegal non-blocking assignment at a combinational always block. Using the non-blocking assignment within a block that is supposed to be combinational is considered an error since the non-blocking operator might be interpreted as a scheduler for later assignment. Yet, combinational logic is not allowed to schedule assignments. For example, the following piece of code of Fig. 5.11 contains a violation of such rule in line 6.

---



---

```

1 module example ( in );
2 input in ;
3 reg a ;
4
5 always @( * ) begin
6     a <= in ;
7 end
8 endmodule

```

---



---

Figure 5.11: Example of a Verilog code that violates the NBCO rule.

## 5.12 NOIO

The NOIO linting rule detects any module that has no inputs/outputs. A violation of such rule is exemplified by the piece of code illustrated in Fig. 5.12. It is worth mentioning that traditional Verilog testbenches usually have no IOs, i.e. they are self-contained. Yet, VEasy is not able to accept such testbenches since they are usually described in a behavioral way. Therefore what this linting rule actually reports is DUT's modules that contain no I/O.

---



---

```

1 module example ( ) ;
2 reg a ;
3
4 always @( * ) begin
5     a <= 0 ;
6 end
7 endmodule

```

---



---

Figure 5.12: Example of a Verilog code that violates the NOIO rule.

## 5.13 RCAS

The RCAS linting rule detects any illegal continuous assignment to a reg variable. In Verilog continuous assignments are only allowed for the wire data type. A violation of such rule is exemplified by the piece of code illustrated in Fig. 5.13 (line 6).

---



---

```

1 module example ( in );
2 input in ;
3 reg a ;
4 wire b ;
5
6 assign a = in ; // invalid
7 assign b = in ; // valid
8 endmodule

```

---



---

Figure 5.13: Example of a Verilog code that violates the RCAS rule.

## 5.14 TIME

The TIME linting rule detects any code construction that deals with timing, e.g. *wait* statements. This type of statement is not allowed since VEasy cannot honor such statements (due to its cycle-accurate behavior). A violation of such rule is exemplified by the piece of code illustrated in Fig. 5.14 (lines 7 and 11).

```

1  module example( clk , in );
2  input clk , in ;
3
4  reg a ;
5  wire b ;
6
7  assign b = #10 a ;
8
9  always @(posedge clk) begin
10     a <= in ;
11     wait ( b == a ) ;
12 end
13 endmodule

```

Figure 5.14: Example of a Verilog code that violates the TIME rule.

## 5.15 TINR

The TINR linting rule detects any template that is not recognizable, i.e., a piece of code from which neither sequential nor combinational behavior might be inferred. A violation of such rule is exemplified by the piece of code illustrated in Fig. 5.15 (line 6). Later when the simulation engine of VEasy is presented it will be shown how VEasy detects sequential and combinational logic by actually detecting templates. Describing logic in any other way will most certainly result in errors.

```

1  module example( clk , in );
2  input clk , in ;
3
4  reg a ;
5
6  always @(posedge clk , in) begin
7     a <= in ;
8 end
9 endmodule

```

Figure 5.15: Example of a Verilog code that violates the TINR rule.

## 5.16 VWSN

The VWSN linting rule detects any variable declared with the same name as other variable, regardless of the data types. A violation of such rule is exemplified by the piece

of code illustrated in Fig. 5.16 (lines 4 and 5). This type of violation is usually detected by compilers but since VEasy's compiling step is actually performed together with linting this rule had to be created.

---



---

```

1  module example( clk , in );
2  input clk , in ;
3
4  reg a ;
5  wire a ;
6
7  endmodule

```

---



---

Figure 5.16: Example of a Verilog code that violates the VWSN rule.

## 5.17 WPAS

The WPAS linting rule detects any illegal procedural assignment to a wire. A violation of such rule is exemplified by the piece of code illustrated in Fig. 5.16 (line 7).

---



---

```

1  module example( clk , in );
2  input clk , in ;
3
4  wire a ;
5
6  always @(posedge clk) begin
7      a <= in ;
8  end
9
10 endmodule

```

---



---

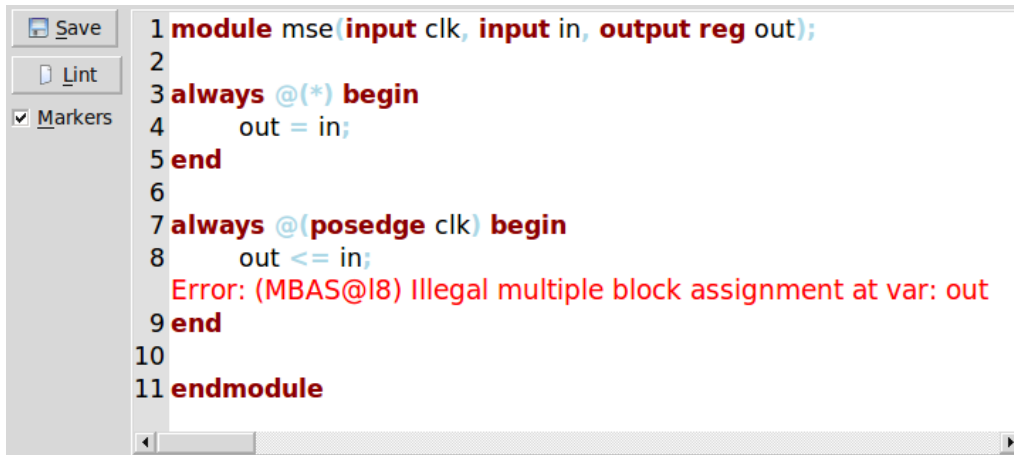
Figure 5.17: Example of a Verilog code that violates the WPAS rule.

## 5.18 Linting Interface

As seen in Section 5.10, one example of a linting rule is the multiple block assignment violation, which ensures that a reg or wire is only assigned in a single fashion through out the code (typically an always block). The illustration of Fig 5.18 contains one example of such rule being analyzed in the linting environment GUI, which also serves as a code editor with syntax highlighting capabilities. Using the environment enables the user to perform changes in the source code, save it and lint again until it is free of errors.

On the remainder of this thesis other screenshots will be presented. What they have in common is that the entire GUI of VEasy was built using Nokia's Qt framework (Nokia Corporation, 2008). This particular framework is considered cross-platform since the same source code is properly compiled in either Windows, Linux, Mac and others. This feature is achieved mostly because the user interfaces are stored in an intermediate format that is platform independent. Moreover, the sections of VEasy that deal directly





The screenshot shows a GUI window with a menu bar containing 'Save', 'Lint', and 'Markers' (checked). The main area displays Verilog code with line numbers 1 through 11. The code is as follows:

```
1 module mse(input clk, input in, output reg out);  
2  
3 always @(*) begin  
4     out = in;  
5 end  
6  
7 always @(posedge clk) begin  
8     out <= in;  
9 end  
10  
11 endmodule
```

An error message is displayed in red text on line 8: "Error: (MBAS@18) Illegal multiple block assignment at var: out".

Figure 5.18: Linting environment GUI reporting an error.

with source code editing or highlighting have these capabilities enabled by the QScintilla library (Riverbank Computing Limited, 2010).



## 6 SIMULATION

Since the nature of FV is to rely on simulation, somehow VEasy had to be able to simulate designs. Instead of relying on a third-party simulator it was decided that VEasy would have its own simulation engine. In order to improve the number of cycles simulated per second, VEasy integrates the test case generation (i.e. the generation of inputs that build a certain test case) with the circuit description. Also, VEasy integrates the coverage collection within the same simulation snapshot. The ability to integrate these three components into one engine has enabled a fast circuit simulation, even when coverage is considered, as it will be shown by the following sections.

But first, let us understand how the simulation engine works. At first, the portion of the simulation snapshot that contains the circuit description is obtained from extracting three profiles from the DUT's code:

- The combinational logic
- The regular sequential logic
- The reset sequential logic

In summary, what VEasy does is profile matching and translation to C code. The profile matching is performed together with the linting procedure. Since most constructions of the Verilog language are similar to the ones in C language translating is quite easy, so the actual update logic (i.e., the assignments) is very similar. One exception is the bit handling and manipulation that Verilog has and C does not. This issue is resolved by using masks and logical operators (& and |) to set or clear specific bits.

Now, each profile is handled in a different manner by VEasy. The following sections will detail each of these profiles.

### 6.1 Combinational logic

The combinational logic simulation is performed using a signature based method. A pseudo-C code of the method is given below in Fig. 6.1 and an explanation of the algorithm follows.

First, a signature is declared in line 1, which is an array sized according to the number of signals being updated by the combinational logic. The combinational signals may be either primary outputs, internal wires or regs. Since a signal is not allowed to be updated in two separate always blocks, it is always known if a given signal has a sequential or a combinational profile. Later another signature array is declared but this time it is a local signature (line 4). What follows is the execution of the update logic in line 7, i.e., the

---



---

```

1  int signature [signals];
2
3  void comb() {
4      int local_signature [signals];
5
6      do_it_again :
7      execute_update_logic ();
8
9      foreach (signal v)
10         local_signature [v] = v;
11
12     if (signature == local_signature)
13         return ;
14     else {
15         signature = local_signature ;
16         goto do_it_again ;
17     }
18 }

```

---



---

Figure 6.1: Pseudo-C code of the combinational logic evaluation.

circuit is evaluated once. After the evaluation is done each signal is copied into the local signature array as shown in lines 9 and 10<sup>1</sup>.

At this time both signatures are compared. If they match then the evaluation is done and the combinational logic is considered to be frozen since not a single signal changed from one evaluation to another. If the signatures do not match then the signature array is updated with the current local signature and the evaluation starts again.

One might wonder why such signature based method is being used. The reason for that lies in the actual combinational logic behavior, where once a input has changed the output must automatically follow that change. Yet, when writing a hardware description such behavior can not be described directly. Instead we write one line of code after the other and perform the evaluation in a certain order. That order might lead to a mismatch in the actual result of a computation unless a method like the one described above is used. In order to exemplify such situation a Verilog code is shown in Fig. 6.2.

---



---

```

1  module example (in);
2  input in;
3  reg a, b;
4
5  always @(*) begin
6      b = a;
7      a = in;
8  end
9  endmodule

```

---



---

Figure 6.2: Example of a Verilog code in which the combinational logic might create a mismatch.

<sup>1</sup>The actual indexing has been simplified by the foreach statement.

It is fairly easy to comprehend that at the end of the evaluation both signals  $a$  and  $b$  should contain the value of the input  $in$ . Yet, analyzing the code from Fig. 6.2, one could reach two different results. First let us perform a simple evaluation of the assignments in the order they are written, as shown in Tab. 6.1<sup>2</sup>.

Table 6.1: Erroneous combinational logic evaluation.

Time	Signal		Code being evaluated
t = 0	a = U	b = U	-
t = 1	a = U	b = U	b = a;
t = 2	a = 1	b = U	a = in;

As shown by Tab. 6.1, a simple evaluation of the statements using the order in which they were written is not satisfactory. The resulting value of  $b$  is erroneous. In order to perform a correct evaluation, the assignments could be evaluated once again until the logic is halted. This is the idea of the algorithm presented in Fig. 6.1. Table 6.2<sup>3</sup> contains the same analysis but this time performing a signature-based evaluation.

Table 6.2: Correct combinational logic evaluation.

Time	Signal		Code being evaluated	local_sig	sig	Match?
t = 0	a = U	b = U	-	-	-	
t = 1	a = U	b = U	b = a;	-	-	
t = 2	a = 1	b = U	a = in;	-	-	
t = 3	a = 1	b = U	compare(local_sig, sig)	1U	-	No
t = 4	a = 1	b = 1	b = a;	1U	1U	
t = 5	a = 1	b = 1	a = in;	1U	1U	
t = 6	a = 1	b = 1	compare(local_sig, sig)	11	1U	No
t = 7	a = 1	b = 1	b = a;	11	11	
t = 8	a = 1	b = 1	a = in;	11	11	
t = 9	a = 1	b = 1	compare(local_sig, sig)	11	11	Yes

One might see that only at the 10th evaluation cycle the result is finally correct. In order to diminish the number of evaluation cycles, the assignments could be organized in another order. For the example illustrated in Fig. 6.1, it would be easy to create another evaluation order: evaluating  $a = in$  first would reduce the number of evaluation cycles. There are algorithms for determining such ordering, such as the ones described by (WANG; MAURER, 1990). This ordering is also referred as levelization in the literature. Yet it was not addressed in this thesis.

<sup>2</sup>The data in the following tables considers that initially both signals  $a$  and  $b$  have a undefined value U. It is also considered that the input  $in$  is 1. The first column contains a timing analogy but it does not actually represent time. Those timings could be interpreted as delta cycles, internal to the simulation engine.

<sup>3</sup>Both signatures are 2 bits wide because the circuit contains two combinational signals being updated.

## 6.2 Regular sequential logic

Simulating the regular sequential logic also requires some code manipulation. The behavior that the regular sequential logic simulation must provide is the concurrency of assignments, as if a clocking signal were reaching all the signals at the same time. For that purpose a method that uses local copies of the signals was developed. A pseudo-C code of the method is given below in Fig. 6.3

```

1  void seq() {
2      foreach (signal v)
3          local_signalv = v;
4
5      execute_update_logic();
6
7      foreach (signal v)
8          v = local_signalv;
9
10     return;
11 }

```

Figure 6.3: Pseudo-C code of the regular sequential logic evaluation.

First, all the values of the signals are copied into the *local\_signal* copy (lines 2 and 3). Later the update logic is executed (line 5). The method only works because the execution of the update logic is done using only the local signal copies instead of using the actual signal values. Finally, after the update logic is executed the actual signals are updated (lines 7 and 8). Therefore the order in which the signals are assigned is no longer important.

In order to justify the need for the simulation method, a Verilog code is given in Fig. 6.4. The following code contains the description of two flip-flops, *ffa* and *ffb*. Since no reset logic is described, the initial values of both flip-flops is unknown.

```

1  module example(clk);
2  input clk;
3
4  reg ffa , ffb;
5
6  always @(posedge clk) begin
7      ffa <= 1;
8      ffb <= ffa;
9  end
10
11 endmodule

```

Figure 6.4: Example of a Verilog code in which the sequential logic might be misinterpreted.

The analysis of the code in Fig. 6.4 could lead to two different results. Let us consider that the circuit simulation has been started and that the clock signal is toggling. At the

first clock edge the flip-flop *ffa* will be assigned with the value 1. This assignment is trivial. The actual problem lies within the second assignment. One could perform a direct evaluation of line 8 and copy the value of *ffa* into *ffb*, as shown in Tab. 6.3<sup>4</sup>.

Table 6.3: Erroneous sequential logic evaluation.

Time	Signal		Code being evaluated
t = 0	<i>ffa</i> = U	<i>ffb</i> = U	-
t = 1	<i>ffa</i> = 1	<i>ffb</i> = U	<i>ffa</i> <= 1;
t = 2	<i>ffa</i> = 1	<i>ffb</i> = 1	<i>ffb</i> <= <i>ffa</i> ;

Yet, the evaluation performed in Tab. 6.3 is erroneous. Since the logic is sequential there must be observed a scheduling effect on all assignments. Therefore, it takes two clock cycles for the *ffb* to copy the value of the input. On the other hand, the evaluation performed in Tab. 6.4 uses the proposed method that creates local copies of the sequential signals. One can see that at the end of the evaluation the flip-flop *ffb* still contains a undefined value. Once again, the evaluation only contains one clock edge. If another cycle were simulated the value of the flip-flop *ffb* would then be the same as the input.

Table 6.4: Correct sequential logic evaluation.

Time	Signal		Local copies		Code being evaluated
t = 0	<i>ffa</i> = U	<i>ffb</i> = U	local_ffa = U	local_ffb = U	-
t = 1	<i>ffa</i> = U	<i>ffb</i> = U	local_ffa = 1	local_ffb = U	<i>ffa</i> <= 1;
t = 2	<i>ffa</i> = U	<i>ffb</i> = U	local_ffa = 1	local_ffb = U	<i>ffb</i> <= <i>ffa</i> ;
t = 3	<i>ffa</i> = 1	<i>ffb</i> = U	local_ffa = 1	local_ffb = U	updating

### 6.3 Reset sequential logic

Handling the reset sequential logic simulation is trivial since it is similar to the other profiles described so far. Yet, mostly for practical purposes, VEasy separates the reset logic from the regular sequential logic. This allows VEasy to handle the reset more clearly, without worrying about simulating the rest of the logic at the same time.

It is usual to describe hardware using either asynchronous or synchronous reset. VEasy is able to detect and simulate both reset types. If the circuit contains an asynchronous reset then it is treated very similarly to the combinational logic. On the other hand, if the circuit has a synchronous reset then it is treated very similarly to the regular sequential logic.

Also, each of the profiles described so far is built into a different C function. The simulation process will repeatedly call these functions until the desired number of simulation cycles is reached. Separating the reset behavior from the regular sequential logic behavior might save some simulation time since there is no need to always evaluate a reset condition at every cycle, e.g, checking if the reset signal of a circuit rose. The reset evaluation

<sup>4</sup>The table contains the evaluation of only one clock edge.

can be detached from the rest of the circuit depending on the reset method chosen by the user.

As mentioned, the definition of when to reset comes from outside the actual simulation, from a reset method chosen by the user. The next sections will detail the four possible reset profiles available within VEasy. The illustration of Fig. 6.5 shows the reset method being chosen using VEasy's GUI.



Figure 6.5: Reset methods available in VEasy's GUI.

### 6.3.1 No reset

In this reset method no reset is created by the simulation environment. This method is suitable for circuits that do not require a reset. If the circuit indeed has a reset than such logic is completely ignored and will never be evaluated. It is only kept in the simulation snapshot for coverage reasons.

### 6.3.2 Time zero

In this reset method only one reset cycle is created by the simulation environment. The chosen cycle is the first, therefore the name time zero. The reset signal is never actually evaluated since the simulation snapshot is slightly modified. The main simulation loop is modified by removing one cycle from it, the first one. The reset handling function is then explicitly called. All the other cycles in the main loop are simplified, which saves some simulation time.

### 6.3.3 Ranged

In this reset method several reset cycles may be created by the simulation environment. The user is responsible for defining the probability of reset occurrences by using two parameters: a maximum and a minimum number of regular cycles that will be simulated between each reset cycle. For example, let us assume that the minimum value is 100 and the maximum value is 200 and also that a reset cycle occurred at time  $t = 0$ . It is then expected that a new reset cycle will happen somewhere in between 100 and 200 cycles after that. Exactly when the reset will happen is not known, since it is randomly chosen by VEasy. It is only guaranteed that the minimum and maximum range will be honored.

This method also allows some simulation time savings. The main simulation loop is once again modified. Actually it is split in two portions: one that lasts the minimum value and never evaluates the reset signal. The other portion lasts the difference from the maximum value to the minimum value and always evaluates the reset signal.



### 6.3.4 Probabilistic

In this reset method several reset cycles may be created by the simulation environment. The user is responsible for defining the probability of reset occurrences by using a single parameter. This parameter defines the reset probability directly. For example, let us assume that the probability was set with the value 0.01. This means that there is a 1% chance that each cycle will be a reset cycle. There is no guarantee about the distribution of these cycles in time. So, although it is not likely to happen, it is possible that two reset cycles will happen one after the other.

This method allows no simulation time savings since the reset signal must be evaluated every cycle.

## 6.4 Benchmarking

The same set of simple circuits described in Section 3.1 was used for the purpose of comparing the performance of the developed simulator. Fig. 6.6 shows those results. All simulations of Fig. 6.6 were done using 10 millions of clock cycles. The reset signal was asserted only during the first simulation cycle. The other signals were generated every cycle with a random value. The commercial simulators A and B are the same ones used in the previous chapters. Icarus Verilog (WILLIAMS, 1999) is a Verilog simulator distributed as free software. The scale on the Y axis of Fig. 6.6 is logarithmic.

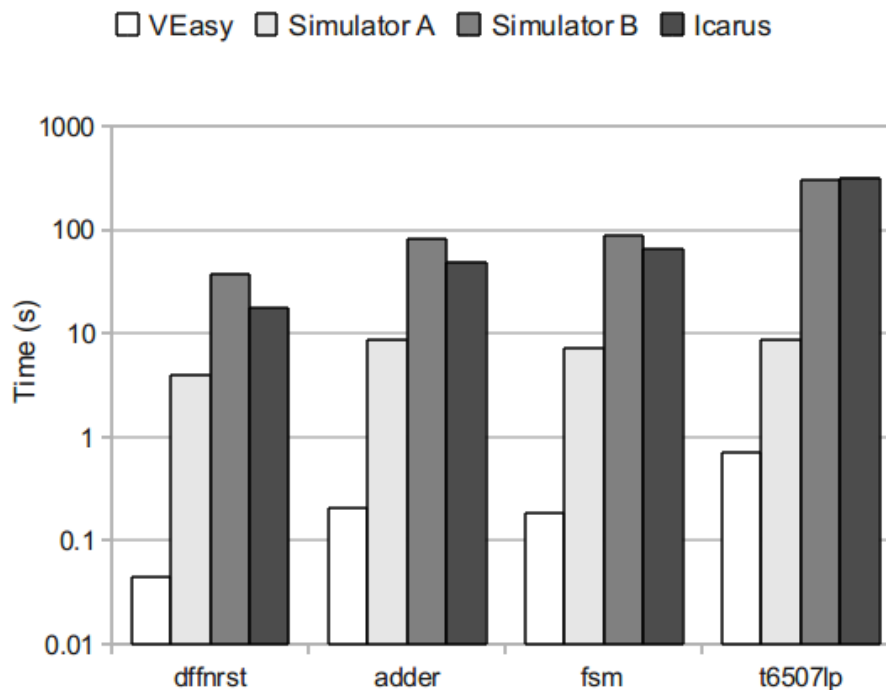


Figure 6.6: VEasy simulation results.

One might notice that, when compared against the simulation times of simulator A, VEasy performs, on average, the same simulation but within less than 5% of the time simulator A requires. This speed-up allows for a faster verification. This is one of the cornerstones of VEasy: by using a cycle-accurate simulator the simulation performance is increased by one order of magnitude. Yet, this approach is not used by most commercial simulators.

One explanation lies in the fact that companies try to create a single simulation engine that is capable of handling their entire flow, i.e., these simulators are used as sign-off simulators. The simulators must handle mixed signal simulations as well as gate-level simulations. This is clearly not the goal of VEasy. VEasy’s goal is to handle RTL code only, which is the actual format used for verification purposes.

The cycle accurate behavior of VEasy is determinative for the measured speed-ups. But it is also worth mentioning that VEasy benefits directly by the compiler optimizations of GCC. Without the optimizations there still is a significant speed-up, but the figure is not as good as 5%. Also, the reset methods of the previous sections also contribute to the speed-up. VEasy also benefits from using a third-party random number generator, referred as SFMT (SAITO; MATSUMOTO, 2008), which implements a modified Mersenne Twister algorithm.

On a secondary batch of benchmarking runs, VEasy was compared with another cycle accurate simulator, referred as Verilator (SNYDER; GALBI; WASSON, 1994). This simulator has been already been compared with several others and, for most designs, it has shown simulation times that beat the other simulators by orders of magnitude. The same set of circuits was submitted to both simulators plus two new circuits were considered: *synth\_comb\_1k* and *synth\_seq\_1k*. These circuits are synthetic and they evaluate exactly 1000 combinational or sequential assignments per cycle, respectively. The simulation results are summarized in Fig. 6.7.

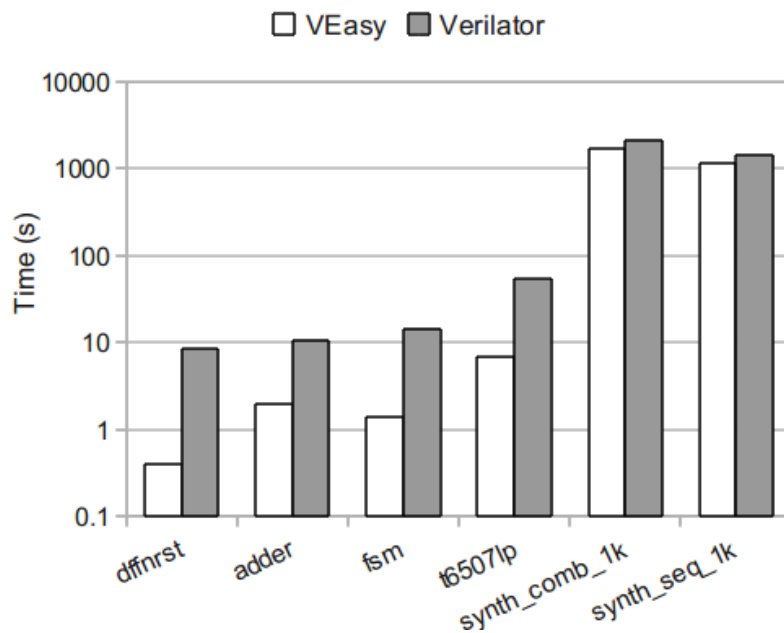


Figure 6.7: VEasy simulation results compared with Verilator.

In Fig. 6.7, one might notice that, since the two simulators have simulation times of the same magnitude, it is possible to compare them using more simulation cycles. The chosen number of cycles was 100 million cycles, ten times more than before. Even in such scenarios VEasy still simulates all of the evaluated circuits faster than Verilator.

One detail about the benchmarking must be taken into account: both simulators require the use of an external C/C++ compiler. Being so, in order to conduct a fair comparison both simulators were configured to use the same compiler (GCC) and the same

optimization directives (the most important one being the  $-O3^5$  one).

The second benchmark batch required the writing of testbenches, the same way that the first benchmark did. Yet, instead of writing testbenches in some form of HDL/HVL language, it was necessary to write testbenches in C/C++ for Verilator. One example of such testbench format is given in the attachments.

One might also notice that the results presented in Fig. 6.7 point out that VEasy simulates all of the circuits faster than Verilator. But, when considering the two synthetic circuits, VEasy and Verilator seen to have a very similar simulation time. This is due to the scale in the vertical axis of Fig. 6.7. In order to further examine such behavior the next section contains an analysis of the scaling trends of both simulators.

## 6.5 Scaling properties

In order to evaluate the scaling properties of both cycle-based simulators, a set of synthetic circuits was developed. The use of synthetic circuits was necessary because they contain no branch decisions, so the number of actual concurrent assignments in a given cycle is always known. A total of 16 synthetic circuits were evaluated. These circuits contain 1, 2, 5, 10, 50, 100, 500 or 1000 concurrent assignments, either combinational or sequential ones. The simulation time of both simulators was measured for each circuit and the results are presented in Fig. 6.8 and Fig. 6.9. The horizontal axis of both images is represented in a logarithmic scale. Each circuit was simulated 10 million cycles.

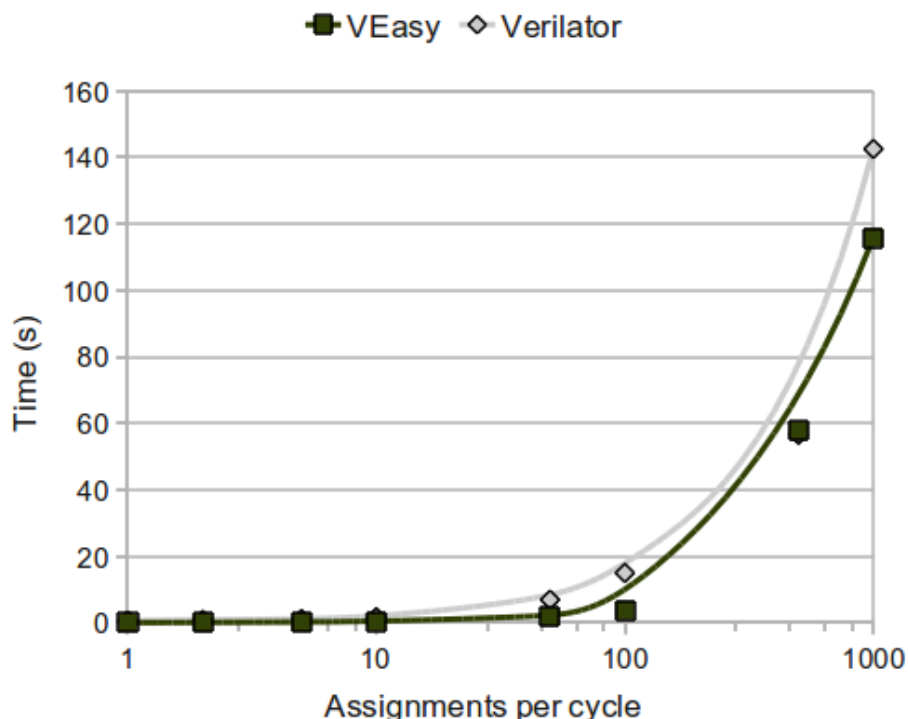


Figure 6.8: Scaling trends for sequential logic simulation.

Regarding Fig. 6.8 and Fig. 6.9, the gray line represents the behavior of Verilator while the black line represents the behavior of VEasy. One might notice that the gray line

<sup>5</sup>This switches the optimization level 3 on. More information on such optimizations is given in (JONES, 2005).

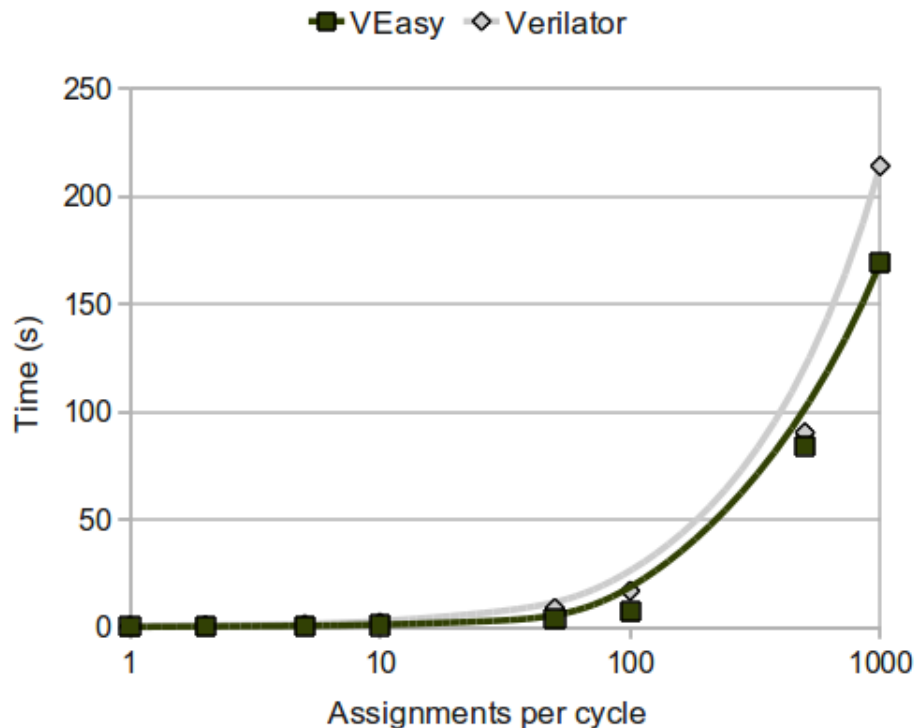


Figure 6.9: Scaling trends for combinational logic simulation.

is growing more rapidly than the black one in both images. This indicates that VEasy is able to handle combinational and sequential assignments more efficiently than Verilator. The discussion regarding VEasy's scaling trends is explored in detail in (PAGLIARINI; HAACKE; KASTENSMIDT, 2011a).

## 6.6 Waveform output

Although simulation using waveforms is not an effective way of verifying a design, it is important to have such possibility when an error has been identified. Thus, the waveform output is allowed in VEasy mostly for debugging purposes. The format chosen to be used within VEasy is the VCD format (IEEE, 1995). The VCD file is an ASCII-based format for dumpfiles generated by several EDA logic simulation tools.

The illustration of Fig. 6.10 contains one example of a VCD file generated by VEasy. Initially all VCD files have a header with some general information. This header was suppressed from the image since it only contained the date and time in which the file was created. The actual data begins with the *\$scope* token found on line 1. After the scope is defined what follows is a declaration of all wires and regs of the Verilog file (lines 2-5). The variable *q* is the only one of the reg type since this is a flip-flop. All other variables are wires since they are either inputs or outputs.

Each of the variables declared within the scope gets an alias. This alias is unique and will be used to reference a given variable later in the file. For example, the *clk* wire received the double quote (") alias. Once all variables are declared the current scope is finished with the *\$upscope* token. The actual simulation data begins after the *\$enddefinitions* token.

Now, there are two types of statements allowed in the simulation data portion of the file. Either a variable is being assigned a value or the simulation time is changing. The

---

```

1 $scope module main $end
2 $var wire 1 " clk $end
3 $var wire 1 $ d $end
4 $var reg 1 % q $end
5 $var wire 1 # rst $end
6 $upscope $end
7 $enddefinitions $end
8
9 0%
10 1#
11 1"
12 #1
13 0"
14 #2
15 0#
16 0$
17 0%
18 1"
19 #3
20 0"
21 #4

```

---

Figure 6.10: Initial portion of a VCD file.

first type of statement is seen in lines 9, 10 and 11, for example. These lines are assigning values to the signals *q* (alias %), *rst* (alias #) and *clk* (alias "). The second type of statement always contains a hash symbol in the first character of a line<sup>6</sup>, as seen in lines 12, 14, 19 and 21. At this point one is capable of identifying the clock signal toggling. A negative clock edge happens whenever the simulation time is an odd integer.

The same VCD file described in Fig. 6.10 is showed in the waveform of Fig. 6.11. This waveform was visualized using the GTKWave free software (BYBELL, 1998).

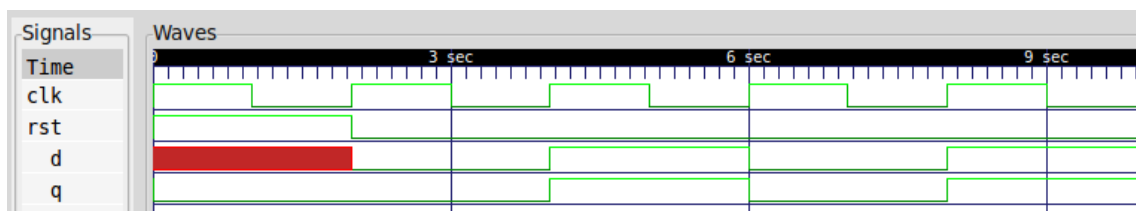


Figure 6.11: Simulation waveform equivalent to VCD of Fig. 6.10.

The next section deals with the simulator validation process, which had the VCD files as the main input.

<sup>6</sup>These hashes do not have the same meaning as in the Verilog language in which #4 is read as “wait 4 time units”. Instead it is read as “advance to time t= 4”.

## 6.7 Validating the simulator

When the simulator was capable of simulating a circuit for the first time it became necessary to check if the simulation results were correct. Initially this comparison was made manually either by comparing the waveforms of VEasy against the ones of another simulator, or by checking a given result in the simulation console. Either way, these approaches were not feasible for analyzing data from large simulations. Thus, it was decided that it was necessary to create a secondary tool, capable of comparing the simulation output of VEasy. This is also a reason why the VCD format was chosen: all simulators that were evaluated in the experiments reported so far are capable of generating VCD files.

The actual work then was to create a tool that is able to parse VCD files, compare them and output warnings whenever differences were found. Later this tool was also included in the VEasy GUI. The illustration of Fig. 6.12 contains a screenshot of the interface that is available to the user. In the image it is possible to see that the user initially provides the location of two VCD files. Once the compare button is hit the tool starts the actual comparison.

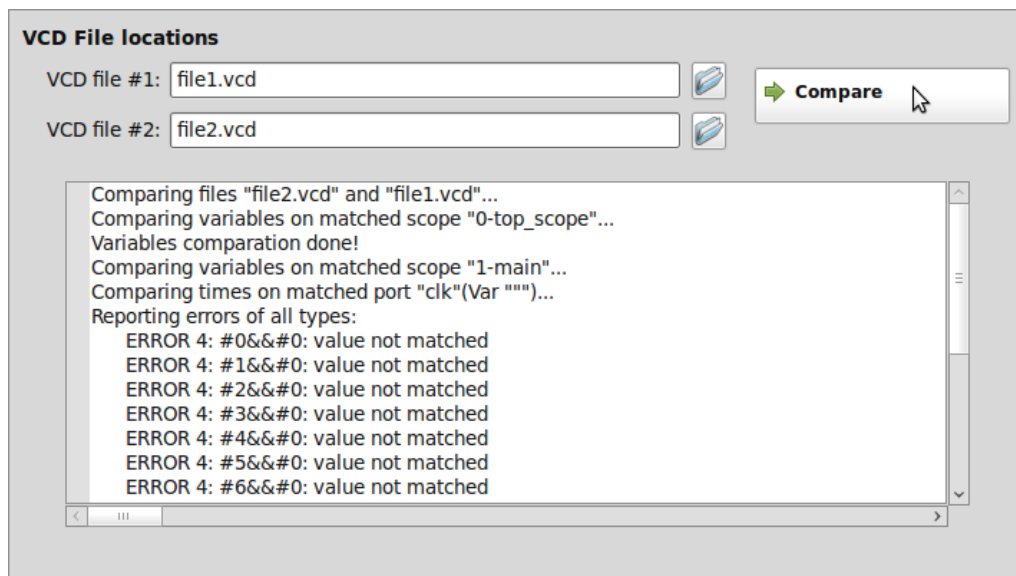


Figure 6.12: VCD compare environment GUI reporting errors.

The VCD files being compared in Fig. 6.12 were purposely modified so several errors are being reported. In the first VCD the clock started the simulation with a zero value and then started to toggle. In the second VCD it started with a one value. Therefore there is a mismatch in all simulation cycles, as reported in the image.

The process of debugging the simulator demanded a large amount of time. First, it was necessary to create testbenches in Verilog to be used with the commercial simulators. Then it was necessary to mimic the behavior of those testbenches using VEasy's sequences. Finally, when an error was found it was necessary to debug it and fix it. Some errors detected were not actual errors, since the testbenches were different. Yet, several bugs were found in the development process of the software. Some statistics of these are given in Chapter 10. At the time this thesis was written, none of the aforementioned circuits had a simulation mismatch when compared against both commercial simulators A and B. An example of one Verilog testbench that was used to validate the simulator is given in the attachments of this thesis.

## 7 COVERAGE

Coverage is a main concept within VEasy. Specific algorithms were created for collecting and analyzing coverage data. In all the metrics that will be presented, the respective code that is responsible for collection will be inserted in the same simulation snapshot of the actual circuit description. This alternative is faster than performing inter-process communication or some form of Programming Language Interface (PLI) (SUTHERLAND, 1998) operation supported by Verilog.

First, let us start by explaining the code coverage related features of VEasy and then let us proceed with the functional coverage ones.

### 7.1 Code coverage

Code coverage in VEasy is possible through three different metrics: block, expression and toggle. Each of these metrics will be addressed in the next subsections.

#### 7.1.1 VEasy's block coverage algorithm

The C code from Fig. 7.1 shows how VEasy collects block coverage information. The idea of this algorithm is to perform the coverage necessary operations only once and then disable that operation by replacing it with another. This goal is achieved by using two different functions. Such technique is known as a jump or branch table.

First, a new *Handler* type is defined as a function pointer in line 1 of Fig. 7.1. Following, lines 2 and 3 create two function prototypes: *cover()* and *do\_nothing()*. Next, an array of the *Handler* type is declared and sized according to the number of blocks of the circuit (the code being considered in this example has 4 blocks). The contents of the array are initially set to point only to the *cover()* function. On line 8 the *cover()* function is defined with a single parameter, the identification of the block being covered (*block\_id*). The coverage storage is performed and then the *jump\_table* is updated to reference the *do\_nothing()* function. In other words, the coverage storage for that given block id will not be executed again. Finally, on line 15, the *do\_nothing()* function is defined as being empty.

It is necessary to call the *cover()* function at some point in the simulation snapshot. For that purpose the simulation snapshot generated by VEasy is instrumented with calls to the *jump\_table*. The actual code that performs the storage is not relevant, although it is sufficient to say that it contains more than one memory access or even an file I/O operation. Disabling the execution of this operation creates an initial small overhead that is justified by the savings it enables later when a large simulation is performed.

---



---

```

1  typedef void (*Handler)( int );
2  void cover(int block_id);
3  void do_nothing (int block_id);
4
5  Handler jump_table[4] = {cover ,
6    cover , cover , cover };
7
8  void cover (int block_id)
9  {
10     /* expensive coverage storage */
11     { ... }
12     jump_table[block_id] = do_nothing;
13 }
14
15 void do_nothing (int block_id)
16 {
17     /* empty */
18 }

```

---



---

Figure 7.1: Block coverage collection algorithm.

### 7.1.2 VEasy's expression coverage algorithm

The expression coverage collection in VEasy is handled by multi dimensional arrays. No efforts have been made to optimize this type of collection mechanism since it is already simplified. Considering the expression  $(c \ \&\& \ (d \ || \ e))$ , first presented in Fig. 2.2, it would be stored in an array of 3 dimensions (since the expression has 3 distinct inputs). In order to collect it, it would be evaluated as follows in Fig 7.2, at every clock cycle.

---



---

```

1  int exp_cov_0[2][2][2] = {0};
2
3  /* main simulation loop */
4  while (VEasy_i < MAX_TICKS) {
5      VEasy_exec_next_sequence();
6      dut_comb();
7      dut_seq();
8
9      exp_cov_0[c][d][e] = 1;
10 }

```

---



---

Figure 7.2: Expression coverage collection algorithm.

In the first line of code in Fig. 7.2, the array that stores the collection is declared. Each expression has a unique array, identified by a number appended to the string *exp\_cov\_*. Lines 5, 6 and 7 perform the circuit simulation using the sequence methodology and the simulation methods explained in Chapter 6. After that, the expression is evaluated member by member. The members of the expression are used as the array indexes in line 9.

Although a very simple example is given, this type of collection algorithm is capable



of handling larger expressions as well. If a given member of a given expression is more than one bit wide then the array size has to be properly modified. Let us say that the input  $c$  is 3 bits wide. Then the array would be sized as `exp_cov_0[8][2][2]`. The maximum size and or addressing of an array is typically limited by compilers. If such limitation is detected then VEasy is able to split large expressions into smaller ones.

### 7.1.3 VEasy's toggle coverage algorithm

The C code from Fig. 7.3 shows how VEasy collects toggle information at each cycle. The collection is executed by the `doToggleCoverage()` function, which is called with three parameters: the old value of a given signal from the last simulation cycle (*oldvalue*), the new value of the same signal from the current simulation cycle (*newvalue*) and finally a pointer to the array that holds the toggle status of the signal (*tog[]*).

---

```

1  void doToggleCoverage(int oldvalue , int newvalue ,
2  int tog [2])
3  {
4      int mask;
5
6      mask = oldvalue ^ newvalue;
7
8      tog [0] |= ((~ oldvalue) & mask);
9      tog [1] |= (oldvalue & mask);
10 }
```

---

Figure 7.3: Toggle coverage collection algorithm.

The first operation that the algorithm performs is to find out if any bit of the signal being evaluated has toggled. Therefore, an integer variable is declared on line 4 and updated on line 6. The exclusive or of line 6 creates sort of a mask pattern: if any bit is set on the mask it means that the given bit has toggled. However, it does not tell if the bit has toggle from one-to-zero or vice-versa.

Lines 8 and 9 perform a logical and between the mask and the *oldvalue*. For the purpose of detecting a zero-to-one toggle, the *oldvalue* is inverted on line 8. For the purpose of detecting an one-to-zero toggle, the *oldvalue* is kept the same on line 9. Now, after performing the logical and, the result is stored at the *tog* pointer after performing a logical or. The first position of the array pointed by the *tog* pointer stores the zero-to-one toggles while the second one stores the one-to-zero toggles. The or operation guarantees that results from past simulation cycles are not overwritten.

Analyzing the algorithm from Fig. 7.3, one notices that there is no bit shifting. In other words, the algorithm works at the word level. Each Verilog signal is directly mapped to an integer, which is analyzed in its entirety. Also, there are no branch conditions on the algorithm. These two properties are very positive for the algorithm execution time.

The collection algorithms described so far are compared in the next section.

### 7.1.4 Experimental Results and Analysis

The same circuits that were evaluated in Section 3.1 were also evaluated using VEasy, considering the three coverage metrics in separate and also the three combined. Results are shown in Tab. 7.1. Again, the toggle coverage creates the most severe simulation

overhead. The overhead seen on the *dffnrst* circuit is the highest one since the simulation time was increased by 35%. Yet, this value is still lower than the 598% overhead seen in simulator A and the 36% seen in simulator B.

Table 7.1: Simulation overhead measured using VEasy.

	<b>dffnrst</b>	<b>adder</b>	<b>fsm</b>	<b>t6507lp</b>
Block	3.31%	1.43%	0.07%	2.48%
Expression	4.96%	0.71%	9.16%	13.86%
Toggle	35.54%	25.00%	23.86%	17.82%
All	42.98%	31.67%	44.82%	39.60%

The average overheads of each of the three simulators is shown in the following tables. Simulator A average values are showed in Tab. 7.2, simulator B in Tab. 7.3 and finally VEasy in Tab. 7.4.

Table 7.2: Average simulation overhead measured using simulator A.

<b>Block</b>	<b>Expression</b>	<b>Toggle</b>	<b>All</b>
269.58%	267.93%	302.77%	303.22%

Table 7.3: Average simulation overhead measured using simulator B.

<b>Block</b>	<b>Expression</b>	<b>Toggle</b>	<b>All</b>
2.75%	2.73%	55.40%	86.70%

Table 7.4: Average simulation overhead measured using VEasy.

<b>Block</b>	<b>Expression</b>	<b>Toggle</b>	<b>All</b>
1.82%	7.17%	25.55%	39.77%

It was surprising to see the overheads measured in simulator A, which are extremely high, with an average overhead of 303%. Simulator B also has a considerable average overhead of 86%. VEasy, on the other hand, has an average overhead of 39%, which is less than half of the overhead introduced by simulator B.

Most of the results presented in this chapter are also given in (PAGLIARINI; HAACKE; KASTENSMIDT, 2011b), which has a special focus on the toggle and block algorithms.

### 7.1.5 Code coverage analysis using the GUI

When using VEasy the coverage collection is fully automated. The user is only required to choose which metrics of interested should be collected. The possible choices are exemplified in Fig. 7.4. After choosing the metrics and performing the simulation, the tool allows for coverage analysis, as shown in Fig. 7.5. The user then might check for coverage *holes* and create new scenarios that will most likely hit that feature or code area.

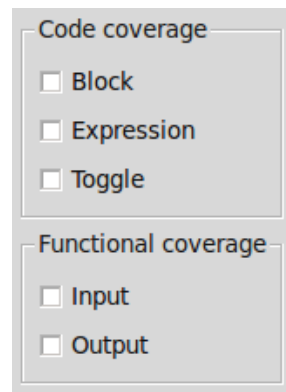


Figure 7.4: Coverage metric selection GUI.

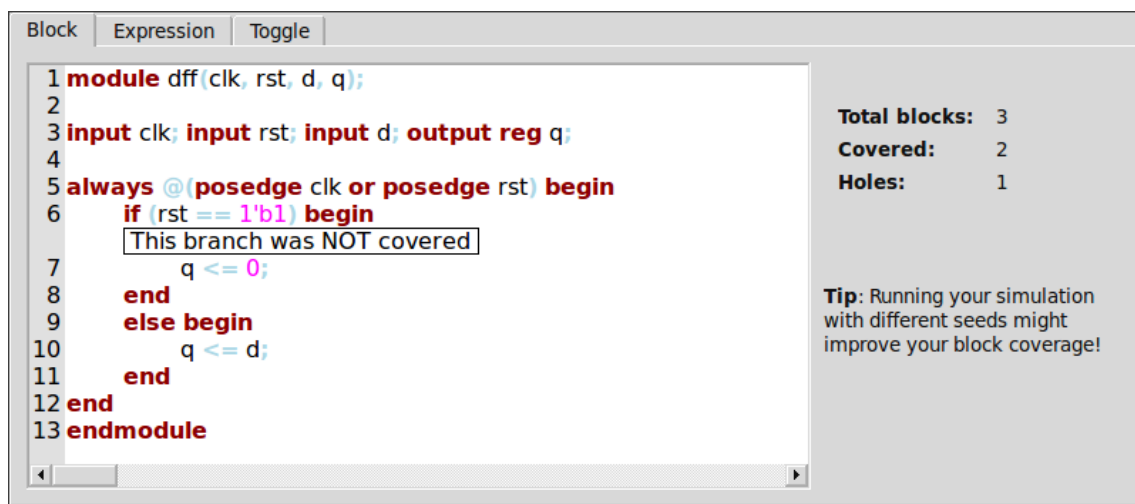


Figure 7.5: Code coverage analysis GUI.

The illustration of Fig. 7.5 shows the *hole* analysis environment of block coverage. Each block that has not been simulated is highlighted. In the example of Fig. 7.5, the code being analyzed is a single D type flip-flop. The simulation of such flip-flop was performed without asserting the reset signal, therefore a portion of the code was left unexcited (line 7).

The illustration of Fig. 7.6 shows the *hole* analysis environment of toggle coverage. Each signal that has not been fully simulated is marked with a red N letter. In the example of Fig. 7.6, the code being analyzed is an adder but since it was simulated only a few cycles not all of the signals had a chance to toggle.

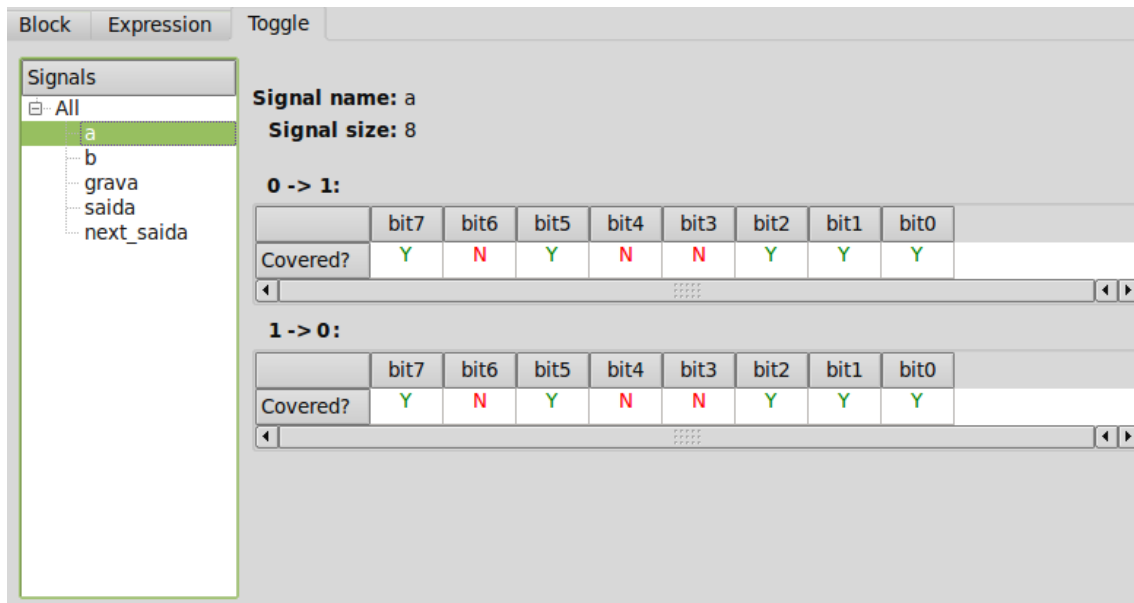


Figure 7.6: Toggle coverage analysis GUI.

## 7.2 Functional coverage

Functional coverage in VEasy is possible through covering the actual inputs and outputs of the DUT or by covering specific members of the layered methodology. The methodology itself is only explained in Chapter 8 yet it is sufficient to know that there are several data members and that using the GUI they might be covered.

### 7.2.1 VEasy's functional coverage collection algorithm

The functional coverage collection in VEasy is quite simple. For each given member there is an associated size. The size is used to determine the total number of buckets<sup>1</sup> for that member, e.g., a member that is 8 bits wide will have 256 buckets. Yet, for most designs there is no interest in covering all possible values of a member. It is more usual to cover a certain range or some pre-defined values of interest. VEasy also allows such restrictions using either the GUI or the verification plan format. The illustration of Fig. 7.7 contains one example of how functional coverage is collected. The design being considered is the same flip-flop from Fig. 7.5.

First, on line 1, an array of integers is declared. The size of the array is defined by the size  $S$  of the member ( $2^S$ ). This particular example is only one bit wide, therefore the number 2 was used. Later, on line 3, the simulation main loop starts. At every clock cycle the simulation delegates the control to the methodology, which decides the next sequence and generates all the stimuli data. The simulation proceeds by executing the evaluation of the combinational and sequential logic (lines 6 and 7). Finally, on line 9, the bucket associated with the  $q$  output is incremented by one. The actual value of  $q$  is used as the array index.

The example described above is quite simple since no restrictions have been created for the buckets. If some restriction is applied (as in Fig. 7.8) then a few changes take

<sup>1</sup>A bucket is any possible scenario that is a candidate for functional coverage. In code coverage we are usually interested in covering if a given block or expression was covered. Yet, for functional coverage sometimes it is interesting to see how many times a scenario actually occurred. Every time that scenario happens it is said that the bucket got a *hit*.

```

1  int q_output_cov [2];
2
3  /* main simulation loop */
4  while (VEasy_i < MAX_TICKS) {
5      VEasy_exec_next_sequence ();
6      dff_comb ();
7      dff_seq ();
8      /* output coverage section */
9      q_output_cov [q]++;
10 }

```

Figure 7.7: Functional coverage collection algorithm.

place: the size of the array is modified according to a range or to a list of possible values. Also, the array is no longer indexed in the same way. Instead a secondary function is used to convert the values into indexes. Thus, the skeleton of the simulation remains almost the same showed in Fig. 7.7.

The next section shows how it is possible to create such restrictions using the GUI.

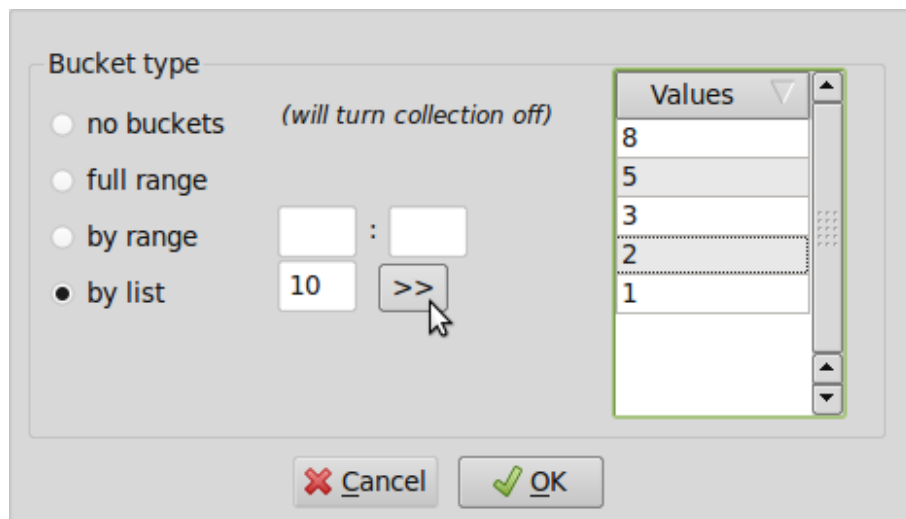


Figure 7.8: Functional coverage analysis GUI.

### 7.2.2 Functional coverage analysis using the GUI

Figure 7.8 shows the GUI available for creating restrictions in the buckets. First, the user is required to choose the type of bucket that will be used. The default behavior is to use no buckets at all, which will cause the collection to be turned off. It is also possible to choose between a full range, a restricted range or a list. If a restricted range is chosen then two values must be provided, one for the lower range boundary and another for the upper boundary. This image in particular is showing a user creating a bucket restriction by means of a list. Such list already has a few different values and upon a user click it will have the value 10 as well.



## 8 METHODOLOGY

The main feature of the tool is the testcase automation capability. Using a GUI the user is able to drag-and-drop sequence items to build larger sequences. One example of such operation is shown in Section 9.3. The methodology that allows the automation is entirely based on layers. A layer is just a container used to build sequences progressively. For instance, a *layer1* sequence contains items from *layer0* while a *layerN* sequence contains items from *layerN-1* and below.

Each sequence has three mandatory fields: the sequence's name, the sequence's level in the hierarchy and the member list. The verification plan file described in Chapter 4 contained all of these fields, which were referred as *seq\_name*, *seq\_level* and *seq\_members*. Each member of the member list is entitled to a list of rules (i.e. constraints). These constraints are identified by the token *keep* in the verification plan file.

Sequences from any layer other than *layer0* also have a list of sequences (referred as *seq\_list*). This creates the possibility of combining more and more sequences from different layers to allow the construction of a sequence library. This library holds sophisticated testcases that are of particular interest for the verification of a design. There are only a few rules and/or guidelines that must be observed:

1. Sequences from *layer0* are the only ones capable of interfacing with the design (i.e. they are the only ones that may contain physical members).
2. Sequences from *layer0* will always generate values for all the inputs of the design, whether they are constrained or not.
3. If a member is not under any constraint then it is assigned a random value within the member's range of possible values.
4. Sequences from *layer0* are the only ones that can make the simulation time advance.
5. The only communication channel between two layers is through logical members (i.e., members that are not physical). All data exchange relies on this approach.
6. All members should be uniquely identified to allow any sequence to use them unambiguously. The exception is when two sequences actually are trying to use the same member to perform communication. In general, the use of prefixes is recommended.
7. Each sequence must have at least one sequence item in its list in order to be executable, except for *layer0* sequences which should not have a list at all.

As mentioned it is also possible to add logical members into sequences of all layers, even if a given sequence is of *layer0*. Fig. 8.1 contains an example of a possible layering which contains physical and logical members.

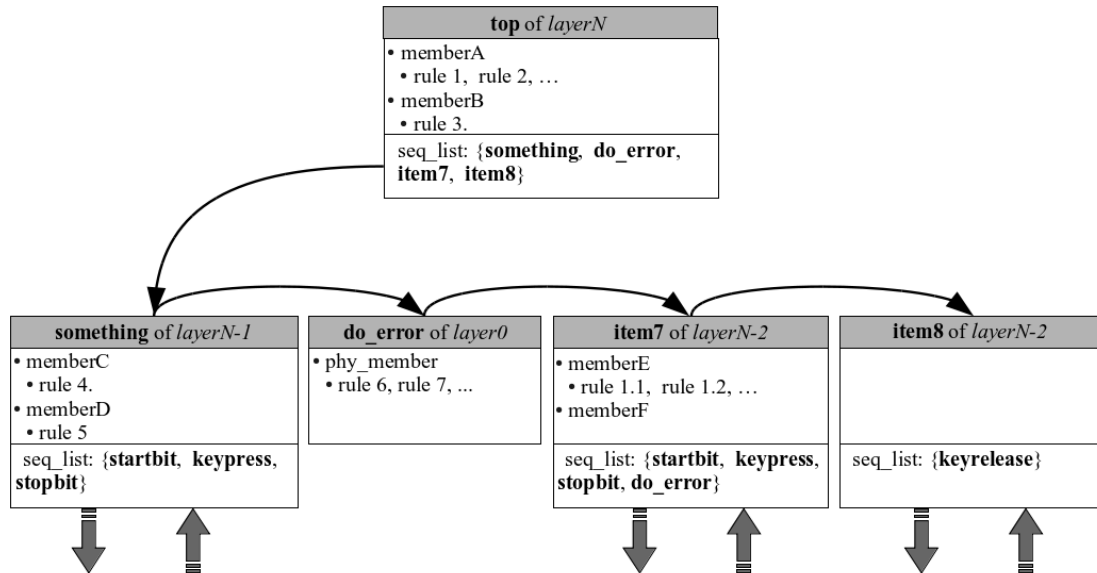


Figure 8.1: Layering example.

The example shows a sequence called **top** of a *layerN*. This sequence has two logical members (memberA and memberB) and each member is constrained with its own set of rules. This same sequence has four items in its list of sequence items (*seq\_list*). One of these items is referred as **do\_error**, which is a sequence from *layer0*. This sequence in particular has no list of items since it is in the bottom of the hierarchy. Fig. 8.1 also shows a sequence that has no members at all, referred as **item8**. The arrows in the image represent the simulation flow. The striped gray arrows are used to point out that there is a hierarchy below each of those sequences. These hierarchies will be fully simulated before the flow continues through the regular black arrows.

As mentioned, each member of a layer, either physical or logical, might be constrained using rules. Currently the tool supports seven rule types:

- Keep value less than (<)
- Keep value less or equal than (<=)
- Keep value greater than (>)
- Keep value greater or equal than (>=)
- Keep value equal to (==)
- Keep value ranged between ([a:b])
- Keep value in a list of possible values ([a,b,c])

All these rules might be created using the GUI. By double clicking in a certain member a modal dialog windows pops up. This is the rule editor, in which the user is able to



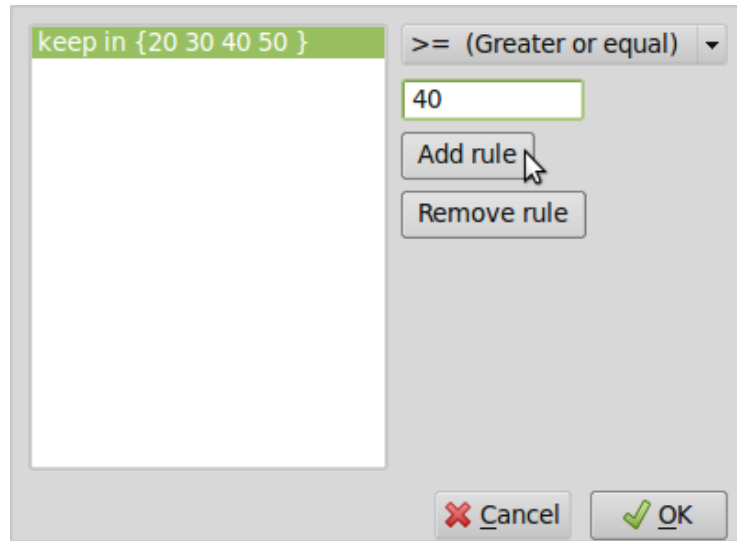


Figure 8.2: Rule editor GUI.

create, edit and remove rules of a certain member. The illustration of Fig. 8.2 shows the rule editor interface available to the user.

The image shows a scenario where there is already a constraint applied to a given member. Such constraint limits the possible values that the member might assume to only four: 20, 30, 40 and 50. Now, if the user clicks the ‘Add rule’ button, a new constraint will be created. Combining the two constraints one will realize that now the member is only allowed to assume two values: 40 and 50. Using this type of rule system, the user is able to create a testcase which is more (or less) focused. For example, in the initial stages of the verification one could create only a single set of rules that allow the simulation of the design respecting its interface protocols. Later one might increase the number of rules in order to simulate a certain situation more often. If no constraints were used it would be necessary to (completely) rewrite testbenches in order to excite specific features.

Another feature added to the methodology is the possibility to create expressions. This is the approach to be used when the user needs the value of a member with some modification applied. For instance, it is very common for a certain sequence to use a combination or only a portion of the data from another sequence that is higher placed in the hierarchy. For example, in a *layerN* the user might create a complete stream of data, while in *layer0* it would send chunks of that stream.

In VEasy this is achieved by using expressions. Such expressions are constantly reading the value of the members they are related. If a change occurs in those members then the expression will perform some operation. Expressions may be interpreted as a virtual member of a sequence.

A pseudo-C code describing the sequence execution order performed by the methodology is shown below in Fig. 8.3. The hierarchy used in the code matches the one in Fig. 8.1 and it also contains one example of expression usage.

In the code from Fig. 8.3 there are two sequences, one from *layerN* and one from *layer0*. For each sequence there is a function associated, as shown in lines 1 and 18. On the first sequence there are logical members, which are generated on lines 4 and 5, before the expression is updated on line 8. These members are generated obeying the rules associated with each of them. Later, each item in the *seq\_list* of that sequence is simulated, as shown in lines 11, 12 and 13. The sequence from *layer0*, **do\_error**, has a

---



---

```

1  void seq_top ()
2  {
3      // first members are generated using the rules
4      memberA = gen_memberA(rule 1, rule 2, ...);
5      memberB = gen_memberB(rule 3);
6
7      // then the expressions update
8      expl = memberA + memberB;
9
10     // and the seq_list is processed
11     seq_something ();
12     seq_do_error ();
13     ...
14     // until there are no more sequences
15     return ;
16 }
17
18 void seq_do_error ()
19 {
20     phy_member = gen_phy_member(rule 6, rule 7);
21
22     // simulate the DUT
23     dut_comb ();
24     dut_seq ();
25
26     // increase the simulation time and return
27     time++;
28     return ;
29 }

```

---



---

Figure 8.3: Execution order of VEasy's sequences.

single physical member (*phy\_member*) which is also generated on line 20.

The two functions called on lines 23 and 24 are the ones related to the design's logic. The first function executes all the combinational statements of the design while the second one executes all the sequential ones, using the algorithms previously discussed. Finally, the simulation time is advanced in line 27.

## 8.1 Some statistics of VEasy's implementation

Developing the interface that allows the methodology of this chapter plus the simulation and linting engines required a lot of work. The methodology actually required planning as well as coding since it should interface properly with the simulator. Fortunately the actual work was recorded and assessed throughout the months. In order to provide a better insight of the amount of time spent developing VEasy, Tab. 8.1 contains some statistics of the effort.

During the entire development of the project a bug tracking system was used. As of

Table 8.1: Some statistics of VEasy's development.

<b>Bugs tracked</b>	120
<b>C++ files and headers</b>	80
<b>Lines of C++ code</b>	18000+
<b>Qt interfaces (UIs)</b>	15+
<b>Builds</b>	4300+
<b>Lines of Verilog</b>	24000+

March 2011 around 120 different bugs were registered and later fixed<sup>1</sup>. The number of C++ files and headers also was measured, reaching around 80 files. These files contain more than 18 thousand significant lines of code<sup>2</sup>. These do not include any automatically generated code, either by VEasy or by Qt.

Regarding Qt, more than 15 user interfaces (UIs) were developed. In other words, there at least 15 different screens in the application. The actual number is higher since some of the interfaces are dynamically generated on-the-fly which difficult the assessment. Another interesting figure comes from the number of builds the tool had. A build is a successful compilation of the entire source code, performed only when a change in the source files was detected. In other words, this accounts for the total number of times the project makefiles were built.

Finally, it is important to say that a large number of circuits were written for evaluating the simulation and linting engines. The total number of lines of Verilog code written for this project was also accounted, reaching a total value of more than 24 thousand lines. This is another indication that the simulation engine is well defined and provides accurate simulation data. Regarding all these statistics it is clear that the amount of work required to get here was definitely not a small one.

Bearing in mind the actual amount of work required to develop VEasy, let us proceed with a study case design in the next chapter. Such chapter presents a case study verification of a password lock device. There, several sequences will be created and discussed in details.

<sup>1</sup>The bug tracking system is publicly available at <http://trac6.assembla.com/veasy>.

<sup>2</sup>The metric used for counting the number of significant lines of code was the same one used to measure the size of a Linux distribution in (WHEELER, 2002), available at <http://cloc.sourceforge.net>.



## 9 CASE STUDY: THE PASSWORD LOCK DEVICE

The methodology described in the previous chapter is used here to build a testcase for a real design in the sections that follows.

### 9.1 DUT description

Our case study DUT is a module that acts as a password lock device. The password is encoded within the module and for the purpose of this thesis was defined as “veasy”. Fig. 9.1 contains the simplified block diagram of the DUT.

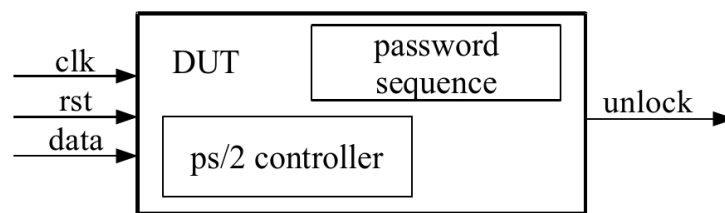


Figure 9.1: DUT block diagram.

The module has two submodules. First, there is the PS/2 controller, which is responsible for detecting the keys that the user is pressing. The password sequence submodule is responsible for recognizing the sequence of keys and check if it matches the encoded password. Since a PS/2 interface is used, all the data entry is serial. The PS/2 protocol is based on the RS 232-C serial protocol (Electronic Industries Association, 1969), which specifies that each key should be transmitted using 11 bits. The first bit is the *start bit* and it is always zero. The next eight bits are the encoded data that represents a key being pressed or released on the keyboard. The next bit is a parity bit while the last bit is always one. The last bit is referred as the *stop bit*. Fig. 9.2 shows an example of a transmission of a PS/2 packet.

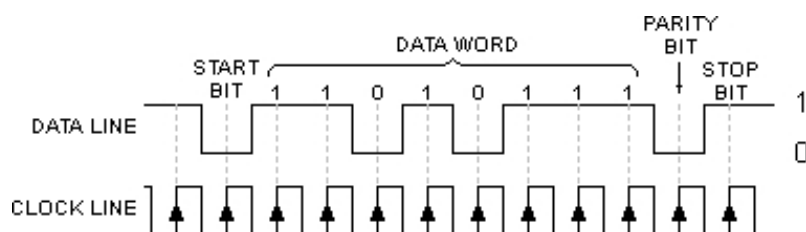


Figure 9.2: Transmission of PS/2 packet.

## 9.2 Verification Plan

In order to verify the DUT a plan was written. The plan contains three features. Table 9.1 shows the features that were considered during the verification of the module. For each feature a different testcase was later associated.

Table 9.1: Feature list of the password lock device.

Feature ID	Description
F1	The module must accept all valid inputs, which are the keys from A-Z and 0-9.
F2	The module must assert the unlock output when the input matches the password.
F3	The module must not assert the unlock output if the password is invalid.

## 9.3 Building a testcase

In order to perform the verification the user is required to send stimulus to the DUT. In other words, it means the input protocol of the DUT must be used. The use of sequences eases this matter since sequences have an ordering mechanism implied. It also lowers the amount of code that must be written since the sequences are reused whenever possible. The sequence creation can be performed using the drag-and-drop features of the GUI or by actual coding using VEasy's verification plan format, which was described in Chapter 4.

For each entry in the feature list of Tab. 9.1, there must be at least one testcase associated. The following testcase was built for the feature F2 and it was built using four different layers: the mandatory *layer0* and three more. The process of creating a testcase always starts by defining at least one *layer0* sequence that contains all of the physical inputs of the DUT. The process continues by building sequences from higher layers. For the creation of a *layerN* sequence it is only required that the user adds at least one *layerN-1* sequence item.

This particular testcase has 11 sequences from *layer0* as shown by Fig. 9.3. These sequences are referred as **start**, **data0** through **data7**, **parity** and **stop**. One can see an obvious match between the sequence's name and the desired behavior of that sequence. Fig. 9.3 also shows a sequence creation scenario where the user is dragging a sequence (**press\_and\_releaseY**) from the left panel list and dropping it into the list of sequence items from a *layer3* sequence (**correct\_password**). In order to actually send a correct password to the DUT several sequences had to be created. Let us go through that process.

First, each of the sequences from *layer0* were created. Since these are from *layer0* they must contain all the inputs of the module. Those inputs are referred as the physical members. In the proposed case study circuit there is only one functional input<sup>1</sup> (data).

The following piece of code in Fig. 9.4 shows how VEasy stores the sequences into the verification plan file. This code is automatically generated by VEasy based on the

<sup>1</sup>A functional input is any input other than clock and reset.



Figure 9.3: VEasy's GUI being used to build a testcase.

data that the user placed in the GUI. This code in particular was generated for the **start** sequence.

```

=====
1 == Sequence ==
2 * seq_name start
3 * seq_level 0
4 * seq_list
5 * seq_members
6 * data 1 PHY
7 * keep == 0
=====

```

Figure 9.4: start sequence described in VEasy's format.

The first line of Fig. 9.4 creates a new sequence (**start**). Lines 2 and 3 are used to identify the sequence with a name and a level in the hierarchy, respectively. Line 4 defines the list of sequence items but, since this is a sequence from *layer0*, this field must be left empty. Finally, line 6 defines the actual member, *data*, which is physical and is one bit wide, while line 7 defines a single rule for that member (making it always generate with a zero value).

The code for the **stop** sequence is obviously similar to the one presented so far, only the sequence name and the constraint are modified. However, the **parity** and **data0** through **data7** sequences require the use of an expression. Thus, the **data0** sequence is described in Fig. 9.5.

Again, the first line creates a sequence while lines 2 and 3 are used to identify it. The difference is that this sequence uses an expression. On line 6 the expression is declared with the name *aux* and a size of only 1 bit. Following, in line 7 the expression is defined with the BIT macro. This macro takes a desired bit from a source byte (in this case it

---



---

```

1 == Sequence ==
2 * seq_name data0
3 * seq_level 0
4 * seq_list
5 * seq_members
6 * aux 1 EXP
7 * aux = BIT(0, key);
8 * data 1 PHY
9 * keep == aux

```

---



---

Figure 9.5: data0 sequence described in VEasy's format.

is taking only the first bit of a member referred as *key*). Finally, on line 8 the physical member data is declared while in line 9 it is constrained. In other words, data is tied to the first bit of a member referred as *key*. Yet, the *key* member has not been declared so far. Such member belongs to another sequence from a higher layer. One example of such sequence is the **pressV** sequence, which is a sequence from *layer1*. This sequence is described in the illustration of Fig. 9.6.

---



---

```

1 == Sequence ==
2 * seq_name pressV
3 * seq_level 1
4 * seq_list {start, data0, data1, data2, data3, data4,
5 data5, data6, data7, parity, stop}
6 * seq_members
7 * key 8 LOG
8 * keep key == 42

```

---



---

Figure 9.6: pressV sequence described in VEasy's format.

Since this sequence is no longer a *layer0* one, it has a non-empty *seq\_list*. This sequence also has a logical member, *key*, which is 8 bits wide and has a single constraint. This constraint keeps it equal to 42, which is the code for the letter V. All other sequences from *layer1* are built similarly, given that the value of the key is properly set in each sequence. In other words, the simulation environment is now capable of creating sequences that correspond to each key of the password.

Yet, the protocol also requires that a different code is sent every time a key is released. In order to do that, for each key, there must be a sequence that presses it and another one that releases it, e.g. **pressV** and **releaseV**. This behavior makes it necessary to create sequences on *layer2*. These sequences are built linking the press and release of the same given key. Finally, on *layer3*, all sequences are organized in an order that sends the correct password to the device, which can be seen in Fig. 9.3. So, to be perfectly clear, upon the execution of the **correct\_password** sequence, the correct password will be sent to the DUT by pressing and releasing the keys V-E-A-S-Y in the proper order.

The other test cases for the other features use similar constructions for the sequences. The difference is mostly in the constraining applied to the keys, which is no longer a constant value. On the first test case, the one from feature F1, the key is constrained using a list of possible values. The third test case, the one from feature F3, does the same but



guarantees that the correct password is never sent.

## 9.4 Comparing the methodology with traditional SystemVerilog

The same testcases that are related to the three features of Tab. 9.1 were coded into three SystemVerilog (IEEE, 2009) testbenches. These were built in the most traditional manner, i.e., without using sequences. However, comparing both approaches is not simple. Instead, some statistics and facts from both implementations are presented in this section.

Initially there must be said that in both solutions there is a need to be time accurate, i.e., to comply with the protocol. The main difference is that the SystemVerilog code needs to advance the simulation time manually. That behavior is seen throughout all the testbenches, in which there are several constructions like the following:

```

1 data = key [ 0 ];
2 @negedge ( clk );
3 data = key [ 1 ];
4 @negedge ( clk );
5 ...

```

Figure 9.7: SystemVerilog code used to control the simulation time.

This type of construction that sets a value and then waits for the DUT to execute is not necessary when the layered methodology of this thesis is used. Whenever a sequence from *layer0* is used, the simulation knows that it must execute the DUT code and advance the simulation time. This allows the user to focus on the behavior when constructing a testcase, leaving the timing control to the tool.

The number of significant lines of code was used to evaluate the SystemVerilog coding of the testbenches. This information is shown in Tab. 9.2. All three testcases built using the proposed methodology have not used a single line of code made by the user. Instead the user had to build sequences, choose a layer for each sequence and build the expressions to use data shared between layers. This information is summarized in Tab. 9.3.

Table 9.2: Statistics of SystemVerilog testbenches.

Feature ID	Lines of code
F1	118
F2	111
F3	139

As one can see in Tab. 9.3, the number of expressions used is the same for all testcases. This is due to the fact that all expressions are from *layer0* sequences, which are reused in all the testcases. Eight of these expressions are used in the **data0** through **data7** sequences (as showed in Fig. 9.5) while the last one is used in the **parity** sequence. As a matter of fact, several sequences were reused between the testcases, remarkably the ones used to press and release a given key.

Table 9.3: Statistics of VEasy testcases.

Feature ID	Number of layers	Number of sequences	Number of expressions
F1	3	14	9
F2	4	27	9
F3	4	15	9

The SystemVerilog testbenches also did the same, being able to benefit from reuse. The SystemVerilog solution made use of a Bus Functional Model (BFM), which usually is a task with parameters. In this case study it could be a task with a single parameter, the key to be sent. The illustration of Fig. 9.8 contains the task used in the analysis performed.

```

1  task press_key;
2  input [7:0] key;
3
4  begin
5      data = 0; // start bit
6      @(negedge clk);
7      data = key[0];
8      @(negedge clk);
9      data = key[1];
10     @(negedge clk);
11     data = key[2];
12     @(negedge clk);
13     data = key[3];
14     @(negedge clk);
15     data = key[4];
16     @(negedge clk);
17     data = key[5];
18     @(negedge clk);
19     data = key[6];
20     @(negedge clk);
21     data = key[7];
22     @(negedge clk);
23     data = ^key;
24     @(negedge clk);
25     data = 1; // stop bit
26 endtask

```

Figure 9.8: SystemVerilog task as a BFM.

Other constructions were used as well, like constraints to define the current key being pressed and a class to encapsulate such constraints. To send the keys in the proper order a randsequence construction was used.

Although the entire tool suite provides a GUI to the user, it must be said, once again, that every operation available through the GUI is also available through the verification

plan format. The GUI might be the method of choice of a inexperienced user or student while the verification plan format might be more suitable for advanced users.



## 10 CONCLUSION

This thesis tried to explore different issues within the context of FV. Yet, one could argue that a given concept might have been left behind. So, initially, before actually concluding, an overview of the different possibilities of future works will be given. First, it is important to state how important assertions are in a modern verification flow. This technique alone has brought a great improvement in the quality of the verification efforts, since it allows a great improvement in block-level verification quality (CHEN, 2003). At the moment no support for assertions of any type has been planned for VEasy. The amount of work required to support any assertion language would be tremendous and certainly would not allow this thesis to end in a manageable timing.

One feature that would be an excellent addition is the possibility to perform some type of CDG. The results in Chapter 7 might be used to choose a structural metric that is more suitable for such coverage-aware generation. There are some strong references to this approach in the literature, which would certainly be useful for comparing/assessing against.

There is a great number of possible small changes to the simulation engine, i.e., fine tuning. During the evolution of this thesis different approaches to simulation have been tried, although only one made to the final text. Some of the approaches that were not detailed had benefits for certain types of circuits or certain styles of logic description. Perhaps the simulation engine could be improved even more by applying some type of mixed algorithm, capable of deciding which simulation engine is better for each circuit or portion of a circuit. Also, as previously mentioned, simulation could benefit from some type of ordering mechanism for the evaluation of assignments. All these would be great contributions to the tool suite.

One feature (and actually a goal) that later was dropped due to time issues is the possibility to export the verification environment built in VEasy to an already established verification methodology, like the ones mentioned in Section 2.4. This is actually a work in progress that unfortunately was not finished until the final version of this text.

The constraint solver used in VEasy was not detailed in the text, only the actual type of rules were detailed. The reason is because the constraint solver is very simple. When there is a single rule or multiple rules of the same type applied to a member, that member will be assigned a random value using a single draw of a random number. When there is more than one type of rule applied to the same member, either physical or logical, it may require more than one draw. This issue is easily avoided if a true constraint solver is used, like the ones built with binary decision diagrams.

Unfortunately it is not possible to give a complete view of the user interface available in the tool. Only some screenshots of the most relevant operations are given in this thesis. The problem is that these images cannot provide the actual way the tool is used. Almost

every action allowed in the tool is automated, most of the times by drag-and-drop operations. For instance, to build a large sequence the user only is required to populate that sequence by dropping other sequences in it. There is also the possibility to select multiple sequences using the shift key, very similar to the applications used for file exploring available in window managers. All these features are very useful for easing the intrinsic learning curve of FV.

Now that the possible future works and some statistics of the development process have been discussed, let us proceed with the actual concluding remarks. First and most important, the subject of this thesis is a very neglected one. It is very hard to see a undergrad (or even graduation) course that contains classes on this subject. Yet, the issues related to verification (and FV) are gigantic, as discussed in Chapter 1. Also, as clearly stated by (BERMAN, 2005), the focus has shifted a lot from design to verification. Actually there are more job positions in the industry that are related to verification than to design. The courseworks of our universities must also shift to reflect such scenario. In this context, this thesis presented a collection of tools that is prone to help.

Regarding Chapter 2, a revision of the state of the art was given. Several verification methodologies were discussed. Yet, they were not compared against VEasy. Such comparison, if made, would be extremely questionable and perhaps even inappropriate. First, because VEasy is not a true commercial methodology. Such methodologies, event the first ones, already have a great amount of complexity included. VEasy's goal was, and still is, to avoid such complexities.

In other words, in VEasy one cannot distinguish what is a BFM or what is a sequencer, a synchronizer or a driver. In VEasy there is no strict layer separation. One then is able to focus in producing stimuli in a layered, reusable way. Although for a large project the need for a clear layer distinction is justifiable, sometimes for small projects it is not. Once again, if one already has a steep learning curve (regarding FV as a whole), one does not need to worry about a perfect tailored layer distinction. Yet, one has evolved from producing direct stimuli to layered stimuli. This is already a great achievement allowed by VEasy.

In the same chapter different concepts within FV have been discussed, like generation and coverage. The latter was deeply discussed since it reflects the quality of the verification (GRINWALD et al., 1998). VEasy included a few of the discussed metrics in its simulation engine. But, most important, it allows the automatic collection and the assisted analysis of those. These are great features that help the engineer/student to evaluate its efforts so far and where to focus next.

Some of the challenges reported in Chapter 2 were also explored in Chapter 3, in which the relevance of those challenges was detailed with some numeric figures. Those figures were given to the overheads of generating data and collecting coverage. First, using two commercial simulators, the coverage collection overhead was measured. Such overhead ranges from 1% to 600%. Although the former number seems acceptable, the latter one clearly is not. Later, in Chapter 7, the same numbers were gathered for VEasy, which has an average overhead 39.7%. This overhead is smaller than the overheads seem in both commercial simulators. This improvement may be used to enable the evaluation of coverage more often in the simulations performed. Thus, engineers would benefit since they could have an early access to coverage results.

A description of the developed tool only begins in Chapter 4. In this chapter the two flows of VEasy are explained as well as the verification plan format. The assisted flow or a similar feature is not commonly present in commercial solutions. There are some ex-

ceptions, but usually these are third-party provided solutions like integrated development environments that help to automate the testbench creation process.

Another feature of the tool, that is detailed in Chapter 4, is the verification plan format. The fact that the file is stored in a wiki-like syntax makes it already prone for collaborative environments. Plus, it is easily rendered by web browsers. This feature plus the fact that the format might be considered an executable verification plan, to the best of the author's knowledge, is unique to VEasy. These two features together certainly might be used to improve verification management.

In Chapter 5 several linting rules available in VEasy were detailed. This type of rule is common regarding the Verilog language, sometimes because of the natural behavior of it, which is loosely typed. Also, linting is very useful for students still learning the language, since it detects most of the common sources of error. For example, describing mixed synchronous and asynchronous logic.

The simulation in VEasy was addressed by developing a simulator from the ground up. This task required a great amount of work but the results reported in Chapter 6 made it worth. One can see that VEasy's simulation was compared against academic and commercial simulations, achieving a better performance than the other simulators. It is also worth mentioning that the algorithms developed are able to handle a great number of concurrent assignments per cycle. In other words, the simulation engine of VEasy is able to better handle the scaling effect of increasing the circuit complexity.

The most complex aspect of VEasy is the methodology. Thus, this particular feature of the tool was deeply refined until a suitable user interface was reached. Such interface is illustrated in Fig. 9.3. The possibility of building large sequences by creating some simple ones and later extending them using drag-and-drop operations is the key element of this feature. As the methodology is naturally layered, reuse is easily achieved. For example, the case study circuit described in Chapter 9 benefits from reuse, at least in a test-level.

As of today, FV by means of simulation is the de-facto choice of the industry when regarding verification. In that context, this thesis described a tool that enhances the traditional verification flow with efficient coverage collection, generation and simulation. Also, the tool could be used for training purposes in graduate or undergraduate courses, helping students to fight the learning curve of FV.





## REFERENCES

- ABADIR et al. Formal Verification Successes at Motorola. **Formal Methods in System Design**, [S.l.], n.22, p.117–123, 2003.
- AMMANN, P.; OFFUTT, J.; HUANG, H. Coverage criteria for logical expressions. In: SOFTWARE RELIABILITY ENGINEERING, 2003. ISSRE 2003. 14TH INTERNATIONAL SYMPOSIUM ON. **Proceedings...** [S.l.: s.n.], 2003. p.99 – 107.
- ANDERSON, T. et al. SystemVerilog reference verification methodology: rtl. **EE Times**, [S.l.], May. 2006.
- ANSI. **The C Programming Language Standard**. 1989. n.X3.159.
- BAILEY, B. A new vision for scalable verification. **EE Times**, [S.l.], n.18, Feb. 2004.
- BAILEY, B. **The Functional Verification of Digital Systems**. [S.l.]: IEC Publications, 2005.
- BENING, L.; FOSTER, H. **Principles of verifiable RTL design: a functional coding style supporting verification processes in verilog**. [S.l.]: Springer, 2001.
- BERGERON, J. **Writing Testbenches: functional verification of hdl models**. 2.ed. Boston: Kluwer Academic, 2003.
- BERGERON, J. **Writing Testbenches using SystemVerilog**. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2006.
- BERGERON, J. The Future of Functional Verification. In: SNUG ISRAEL, 2010. PROCEEDINGS, Herzliya, Israel. **Proceedings...** [S.l.: s.n.], 2010.
- BERGERON, J. et al. **Verification Methodology Manual for SystemVerilog**. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2005.
- BERMAN, V. IEEE P1647 and P1800: two approaches to standardization and language design. **Design & Test of Computers, IEEE**, [S.l.], v.22, n.3, p.283 – 285, May 2005.
- BORMANN, J. et al. Complete Formal Verification of TriCore2 and Other Processors. In: DESIGN AND VERIFICATION CONFERENCE. **Proceedings...** [S.l.: s.n.], 2007.
- BRIAND, L.; PFAHL, D. Using simulation for assessing the real impact of test-coverage on defect-coverage. **Reliability, IEEE Transactions on**, [S.l.], v.49, n.1, p.60 –70, Mar. 2000.

BYBELL, T. **GTKWave**. 1998. Available at <<http://gtkwave.sourceforge.net/>>. Accessed February 10, 2011.

Cadence Design Systems, Inc. **Incisive Management Datasheet**. 2005. Available at <[http://www.cadence.com/rl/Resources/datasheets/incis\\_verif\\_mgr\\_ds.pdf](http://www.cadence.com/rl/Resources/datasheets/incis_verif_mgr_ds.pdf)>. Accessed December 10, 2010.

Cadence Design Systems, Inc. **Encounter RTL Compiler Datasheet**. 2010. Available at <[http://www.cadence.com/rl/resources/datasheets/encounter\\_rtlcompiler.pdf](http://www.cadence.com/rl/resources/datasheets/encounter_rtlcompiler.pdf)>. Accessed October 21, 2010.

CADENCE; MENTOR. **Open Verification Methodology White Paper**. 2007.

CHANG, M. Foundry Future: challenges in the 21st century. In: SOLID-STATE CIRCUITS CONFERENCE, 2007. ISSCC 2007. DIGEST OF TECHNICAL PAPERS. IEEE INTERNATIONAL. **Proceedings...** [S.l.: s.n.], 2007. p.18 –23.

CHEN, K.-C. Assertion-based verification for SoC designs. In: ASIC, 2003. PROCEEDINGS. 5TH INTERNATIONAL CONFERENCE ON. **Proceedings...** [S.l.: s.n.], 2003. v.1, p.12 – 15 Vol.1.

CHILENSKI, J.; MILLER, S. Applicability of modified condition/decision coverage to software testing. **Software Engineering Journal**, [S.l.], v.9, n.5, p.193 –200, Sept. 1994.

COHEN, O. et al. Designers Work Less with Quality Formal Equivalence Checking. In: DESIGN AND VERIFICATION CONFERENCE. **Proceedings...** [S.l.: s.n.], 2010.

Collett International Research Inc. **IC/ASIC functional verification study**. [S.l.]: Collett International Research Inc., 2003.

DEMPSTER, D.; STUART, M.; MOSES, C. **Verification Methodology Manual**: techniques for verifying hdl designs. 2.ed. [S.l.]: Teamwork International, 2001.

DENG, Y. GPU Accelerated VLSI Design Verification. In: COMPUTER AND INFORMATION TECHNOLOGY (CIT), 2010 IEEE 10TH INTERNATIONAL CONFERENCE ON. **Proceedings...** [S.l.: s.n.], 2010. p.1213 –1218.

EBERSBACH, A.; GLASER, M.; HEIGL, R. **Wiki: web collaboration**. [S.l.]: Springer, 2005.

Electronic Industries Association. **EIA Standard RS-232-C Interface Between Data Terminal Equipment and Data Communication Equipment Employing Serial Data Interchange**. 1969. n.RS 232-C.

EVANS, A. et al. Functional verification of large ASICs. In: DESIGN AUTOMATION CONFERENCE, 35., New York, NY, USA. **Proceedings...** ACM, 1998. p.650–655. (DAC '98).

Far West Research and Mentor Graphics. **EDA Market Statistics Service Report**. [S.l.]: Far West Research, 2007.

FINE, S.; ZIV, A. Coverage directed test generation for functional verification using Bayesian networks. In: DESIGN AUTOMATION CONFERENCE, 2003. PROCEEDINGS. **Proceedings...** [S.l.: s.n.], 2003. p.286 – 291.

FITZPATRICK, T.; SCHUTTEN, R. **Design for verification**: blueprint for productivity and product quality. [S.l.]: Synopsys Inc., 2003.

Gary Smith EDA - GSEDA. **2009 Market Share Research**. 2009. Available at <<http://www.garysmitheda.com>>. Accessed January 16, 2011.

GIZOPOULOS, D. **Advances in Electronic Testing**: challenges and methodologies (frontiers in electronic testing). Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2006.

GLASSER, M. **Open Verification Methodology Cookbook**. [S.l.]: Springer, 2009. (SpringerLink Engineering).

GLASSER, M. et al. **Advanced Verification Methodology Cookbook**. [S.l.]: Mentor Graphics, 2008.

GRIES, H. **Verification Methodology Poll Results**. 2009. Available at <<http://theasicguy.com/2009/02/11/verification-methodology-poll-results/>>. Accessed January 16, 2011.

GRINWALD, R. et al. User Defined Coverage - A Tool Supported Methodology for Design Verification. In: DESIGN AUTOMATION CONFERENCE, 35., San Francisco, United States. **Proceedings...** [S.l.: s.n.], 1998. p.158–163.

HAACKE, P. A.; PAGLIARINI, S. N.; KASTENSMIDT, F. L. **Evaluating Stimuli Generation Using the VEasy Functional Verification Tool Suite**. SIM - South Symposium on Microelectronics, 2011. Submitted.

HAQUE, F.; MICHELSON, J.; KHAN, K. **The Art Of Verification with VERA**. [S.l.]: Verification Central, 2001.

IEEE. **Standard VHDL Language Reference Manual**. 1987. n.1076.

IEEE. **Standard for the Verilog Hardware Description Language**. 1995. n.1364.

IEEE. **Standard for the Property Specification Language (PSL)**. 2005. n.1850.

IEEE. **Standard for the Functional Verification Language e**. 2006. n.1647.

IEEE. **Standard for SystemVerilog - Unified Hardware Design, Specification, and Verification Language**. 2009. n.1800.

IMAN, S.; JOSHI, S. **The E hardware verification language**. [S.l.]: Kluwer Academic Publishers, 2004.

International Business Machines Corporation - IBM. **Enterprise verification management solution**. 2009. Available at <<http://www-01.ibm.com/software/rational/offerings/cadence/>>. Accessed December 10, 2010.

ITRS. **INTERNATIONAL TECHNOLOGY ROADMAP FOR SEMICONDUCTORS**. Available at <<http://www.itrs.net/links/2005itrs/design2005.pdf>>. Accessed October 21, 2010.

JONES, M. T. Optimization in GCC. **Linux J.**, Seattle, WA, USA, v.2005, p.11–, March 2005.

KITCHEN, N.; KUEHLMANN, A. Stimulus generation for constrained random simulation. In: ICCAD '07: PROCEEDINGS OF THE 2007 IEEE/ACM INTERNATIONAL CONFERENCE ON COMPUTER-AIDED DESIGN. **Proceedings...** [S.l.: s.n.], 2007. p.258 –265.

LACHISH, O. et al. Hole analysis for functional coverage data. In: DESIGN AUTOMATION CONFERENCE, 39., New York, NY, USA. **Proceedings...** ACM, 2002. p.807–812. (DAC '02).

MARTIN, G. **The Myths of EDA: the 70% rule.** 2008. Available at <<http://www.chipdesignmag.com/martins/>>. Accessed February 11, 2011.

MICZO, A. **Digital logic testing and simulation.** New York, NY, USA: Harper & Row Publishers, Inc., 1986.

MOLINA, A.; CADENAS, O. FUNCTIONAL VERIFICATION: approaches and challenges. **Latin American Applied Research**, [S.l.], n.37, p.65–69, 2007.

Nokia Corporation. **Qt Framework.** 2008. Available at <<http://qt.nokia.com/products/>>. Accessed January 10, 2011.

PAGLIARINI, S. N.; HAACKE, P. A.; KASTENSMIDT, F. L. **RTL Simulation Using the VEasy Functional Verification Tool Suite.** NEWCAS, 2011. Submitted.

PAGLIARINI, S. N.; HAACKE, P. A.; KASTENSMIDT, F. L. **Evaluating Coverage Collection Using the VEasy Functional Verification Tool Suite.** LATW, 2011. Accepted for publication.

PAGLIARINI, S. N.; KASTENSMIDT, F. L. **VEasy: a tool suite for teaching vlsi functional verification.** MSE - Microelectronic Systems Education, 2011. Submitted.

PAGLIARINI, S. N.; KASTENSMIDT, F. L. **VEasy: a functional verification tool suite.** SIM - South Symposium on Microelectronics, 2011. Submitted.

PAGLIARINI, S. N.; ZARDO, G. O. **t6507lp IP core.** 2009. Available at <<http://opencores.org/project,t6507lp>>. Accessed December 1, 2011.

PIZIALI, A. **Functional Verification Coverage Measurement and Analysis.** [S.l.]: Kluwer Academic, 2004.

REDWINE S.T., J. An Engineering Approach to Software Test Data Design. **Software Engineering, IEEE Transactions on**, [S.l.], v.SE-9, n.2, p.191 – 200, Mar. 1983.

Riverbank Computing Limited. **QScintilla - a Port to Qt of Scintilla.** 2010. Available at <<http://www.riverbankcomputing.co.uk/software/qscintilla/intro>>. Accessed January 10, 2011.

RIVEST, R. **The MD5 Message-Digest Algorithm.** United States: RFC Editor, 1992.

SAITO, M.; MATSUMOTO, M. SIMD-Oriented Fast Mersenne Twister: a 128-bit pseudorandom number generator. In: **Monte Carlo and Quasi-Monte Carlo Methods 2006**. [S.l.]: Springer, 2008. p.607–622.

SNYDER, W.; GALBI, D.; WASSON, P. **Verilator**. 1994. Available at <<http://www.veripool.org/wiki/verilator/>>. Accessed November 16, 2010.

STALLMAN, R. Using and Porting the GNU Compiler Collection. **M.I.T. Artificial Intelligence Laboratory**, [S.l.], 2001.

STEFFORA, A. DAI Enters Transaction-Based Verification Market. In: **Electronic News**. [S.l.: s.n.], 1998.

SUTHERLAND, S. Transitioning to the new PLI standard. In: VERILOG HDL CONFERENCE AND VHDL INTERNATIONAL USERS FORUM, 1998. IVC/VIUF. PROCEEDINGS., 1998 INTERNATIONAL. **Proceedings...** [S.l.: s.n.], 1998. p.20–21.

Synopsys, Inc. **Design Compiler 2010 Datasheet**. Available at <<http://www.synopsys.com/Tools/Implementation/RTLSynthesis/Documents/DesignCompiler2010-ds.pdf>>. Accessed October 21, 2010.

TRONCI, E. Special section on Recent Advances in Hardware Verification: introductory paper. **International Journal on Software Tools for Technology Transfer (STTT)**, [S.l.], v.8, p.355–358, 2006.

VSI Alliance. **Specification for VC/SoC Functional Verification**. [S.l.]: VSI Alliance, 2004.

WANG, T.-H.; TAN, C. G. Practical code coverage for Verilog. In: IEEE INTERNATIONAL VERILOG HDL CONFERENCE, 4., Washington, DC, USA. **Proceedings...** IEEE Computer Society, 1995. p.99–. (IVC '95).

WANG, Z.; MAURER, P. M. LECSIM: a leveled event driven compiled logic simulation. In: ACM/IEEE DESIGN AUTOMATION CONFERENCE, 27., New York, NY, USA. **Proceedings...** ACM, 1990. p.491–496. (DAC '90).

WESTE, N.; HARRIS, D. **CMOS VLSI Design: a circuits and systems perspective**. USA: Addison-Wesley Publishing Company, 2010.

WHEELER, D. A. **More Than a Gigabuck**: estimating gnu/linux's size. 2002. Available at <<http://www.dwheeler.com/sloc/redhat71-v1/redhat71sloc.html>>. Accessed February 22, 2011.

WILE, B.; GOSS, J.; ROESNER, W. **Comprehensive Functional Verification: the complete industry cycle (systems on silicon)**. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2005.

WILLIAMS, S. **Icarus Verilog**. 1999. Available at <<http://www.icarus.com/eda/verilog/>>. Accessed November 16, 2010.

WONG, W. et al. Effect of test set size and block coverage on the fault detection effectiveness. In: SOFTWARE RELIABILITY ENGINEERING, 1994. PROCEEDINGS., 5TH INTERNATIONAL SYMPOSIUM ON. **Proceedings...** [S.l.: s.n.], 1994. p.230–238.

YAN, C.; JONES, K. Efficient Simulation Based Verification by Reordering. In: DESIGN AND VERIFICATION CONFERENCE. **Proceedings...** [S.l.: s.n.], 2010.

YAROM, I.; PATIL, V. Smart-lint: improving the verification flow. In: HAIFA VERIFICATION CONFERENCE ON HARDWARE AND SOFTWARE, VERIFICATION AND TESTING, 2., Berlin, Heidelberg. **Proceedings...** Springer-Verlag, 2007. p.81–91. (HVC'06).

YUAN, J. et al. Modeling design constraints and biasing in simulation using BDDs. In: ICCAD '99: PROCEEDINGS OF THE 1999 IEEE/ACM INTERNATIONAL CONFERENCE ON COMPUTER-AIDED DESIGN, Piscataway, NJ, USA. **Proceedings...** IEEE Press, 1999. p.584–590.

## ATTACHMENTS

### I Verilog testbench used to validate the simulator

```
module t_fsm();

reg [7:0] a;
reg [7:0] b;
reg grava;
reg clk;
reg rst;
wire [7:0] saida;

fsm fsm(
.a(a),
.b(b),
.clk(clk),
.rst(rst),
.saida(saida)
);

always begin
#1 clk = !clk;
a = $random;
b = $random;
end

initial begin
clk = 0;
rst = 0;
#1;
rst = 1;
#20000000 $finish();
end

endmodule
```

## II Web browser rendering of a verification plan file

```

- rtl_md5sum 0xa36e351cc8e4fdea83d9824fc99b1bf5

```

**Special signals**

- rtl\_clock clk
- rtl\_reset rst

**FUV**

**Input Interface**

- d 1

**Output Interface**

- q 1

**Sequence**

- seq\_name one
- seq\_level 0
- seq\_list
- seq\_members
  - d 1 PHY
    - keep == 1

**Sequence**

- seq\_name random
- seq\_level 0
- seq\_list
- seq\_members
  - d 1 PHY

Figure II.1: Portion of a verification plan file when rendered by a web browser.



### III Verilator testbench format

---

```

1 #include <verilated.h> // Defines common routines
2 #include "Vfsm.h" // From Verilating "fsm.v"
3 Vfsm *fsm; // Instantiation of module
4 unsigned int main_time = 0; // Current simulation time
5 double sc_time_stamp () { // Called by $time in Verilog
6     return main_time;
7 }
8
9 int main(int argc, char** argv) {
10     Verilated::commandArgs(argc, argv);
11     // Remember args
12     fsm = new Vfsm; // Create instance
13     fsm->rst = 0; // assert reset
14
15     while (!Verilated::gotFinish()) {
16         if (main_time > 2) {
17             fsm->rst = 1; // Deassert reset
18         }
19
20         if (main_time == 20000000) {
21             fsm->final();
22             break;
23         }
24
25         if ((main_time % 2) == 0) {
26             fsm->clk = 0;
27         }
28         else {
29             fsm->clk = 1;
30             fsm->a = random();
31             fsm->b = random();
32         }
33
34         main_time++;
35         fsm->eval();
36     }
37 }

```

---

Figure III.2: Verilator testbench format.