VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER SYSTEMS

NEW METHODS FOR INCREASING EFFICIENCY AND SPEED OF FUNCTIONAL VERIFICATION

DIZERTAČNÍ PRÁCE
PHD THESIS

AUTOR PRÁCE Ing. MARCELA ŠIMKOVÁ
AUTHOR

BRNO 2015

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER SYSTEMS

METODY AKCELERACE VERIFIKACE LOGICKÝCH OBVODŮ

NEW METHODS FOR INCREASING EFFICIENCY AND SPEED OF FUNCTIONAL VERIFICATION

DIZERTAČNÍ PRÁCE
PHD THESIS

AUTOR PRÁCE Ing. MARCELA ŠIMKOVÁ
AUTHOR

VEDOUCÍ PRÁCE Doc. Ing. ZDENĚK KOTÁSEK, CSc.
SUPERVISOR

BRNO 2015

Abstrakt

Při vývoji současných číslicových systémů, např. vestavěných systému a počítačového hardware, je nutné hledat postupy, jak zvýšit jejich spolehlivost. Jednou z možností je zvyšování efektivity a rychlosti verifikačních procesů, které se provádějí v raných fázích návrhu. V této dizertační práci se pozornost věnuje verifikačnímu přístupu s názvem funkční verifikace. Je identifikováno několik výzev a problému týkajících se efektivity a rychlosti funkční verifikace a ty jsou následně řešeny v cílech dizertační práce. První cíl se zaměřuje na redukci simulačního času v průběhu verifikace komplexních systémů. Důvodem je, že simulace inherentně paralelního hardwarového systému trvá velmi dlouho v porovnání s během v skutečném hardware. Je proto navrhnuta optimalizační technika, která umisťuje verifikovaný systém do FPGA akcelerátoru, zatím co část verifikačního prostředí stále běží v simulaci. Tímto přemístěním je možné výrazně zredukovat simulační režii. Druhý cíl se zabývá ručně připravovanými verifikačními prostředími, která představují výrazné omezení ve verifikační produktivitě. Tato režie však není nutná, protože většina verifikačních prostředí má velice podobnou strukturu, jelikož využívají komponenty standardních verifikačních metodik. Tyto komponenty se jen upravují s ohledem na verifikovaný systém. Proto druhá optimalizační technika analyzuje popis systému na vyšší úrovni abstrakce a automatizuje tvorbu verifikačních prostředí tím, že je automaticky generuje z tohoto vysoko-úrovňového popisu. Třetí cíl zkoumá, jak je možné docílit úplnost verifikace pomocí inteligentní automatizace. Úplnost verifikace se typicky měří pomocí různých metrik pokrytí a verifikace je ukončena, když je dosažena právě vysoká úroveň pokrytí. Proto je navržena třetí optimalizační technika, která řídí generování vstupů pro verifikovaný systém tak, aby tyto vstupy aktivovali současně co nejvíc bodů pokrytí a aby byla rychlost konvergence k maximálnímu pokrytí co nejvyšší. Jako hlavní optimalizační prostředek se používá genetický algoritmus, který je přizpůsoben pro funkční verifikaci a jeho parametry jsou vyladěny pro tuto doménu. Běží na pozadí verifikačního procesu, analyzuje dosažené pokrytí a na základě toho dynamicky upravuje omezující podmínky pro generátor vstupů. Tyto podmínky jsou reprezentovány pravděpodobnostmi, které určují výběr vhodných hodnot ze vstupní domény. Čtvrtý cíl diskutuje, zda je možné znovu použít vstupy z funkční verifikace pro účely regresního testování a optimalizovat je tak, aby byla rychlost testování co nejvyšší. Ve funkční verifikaci je totiž běžné, že vstupy jsou značně redundantní, jelikož jsou produkovány generátorem. Pro regresní testy ale tato redundance není potřebná a proto může být eliminována. Zároveň je ale nutné dbát na to, aby úroveň pokrytí dosáhnutá optimalizovanou sadou byla stejná, jako u té původní. Čtvrtá optimalizační technika toto reflektuje a opět používá genetický algoritmus jako optimalizační prostředek. Tentokrát ale není integrován do procesu verifikace, ale je použit až po její ukončení. Velmi rychle odstraňuje redundanci z původní sady vstupů a výsledná doba simulace je tak značně optimalizována.

Klíčová slova

Funční verifikace, verifikace založená na simulaci, Universal Verification Methodology, SystemVerilog, optimalizace, automatizace, genetický algoritmus, verifikace řízená pokrytím, metriky pokrytí.

Citace

Marcela Šimková: New Methods for Increasing Efficiency and Speed of Functional Verification [dizertační práce], Ústav počítačových systémů FIT VUT v Brně, Brno, CZ, 2015

Abstract

In the development of current hardware systems, e.g. embedded systems or computer hardware, new ways how to increase their reliability are highly investigated. One way how to tackle the issue of reliability is to increase the efficiency and the speed of verification processes that are performed in the early phases of the design cycle. In this Ph.D. thesis, the attention is focused on the verification approach called *functional verification*. Several challenges and problems connected with the efficiency and the speed of functional verification are identified and reflected in the goals of the Ph.D. thesis. The first goal focuses on the reduction of the simulation runtime when verifying complex hardware systems. The reason is that the simulation of inherently parallel hardware systems is very slow in comparison to the speed of real hardware. The optimization technique is proposed that moves the verified system into the FPGA acceleration board while the rest of the verification environment runs in simulation. By this single move, the simulation overhead can be significantly reduced. The second goal deals with manually written verification environments which represent a huge bottleneck in the verification productivity. However, it is not reasonable, because almost all verification environments have the same structure as they utilize libraries of basic components from the standard verification methodologies. They are only adjusted to the system that is verified. Therefore, the second optimization technique takes the high-level specification of the system and then automatically generates a comprehensive verification environment for this system. The third goal elaborates how the completeness of the verification process can be achieved using the intelligent automation. The completeness is measured by different coverage metrics and the verification is usually ended when a satisfying level of coverage is achieved. Therefore, the third optimization technique drives generation of input stimuli in order to activate multiple coverage points in the verified system and to enhance the overall coverage rate. As the main optimization tool the genetic algorithm is used, which is adopted for the functional verification purposes and its parameters are well-tuned for this domain. It is running in the background of the verification process, it analyses the coverage and it dynamically changes constraints of the stimuli generator. Constraints are represented by the probabilities using which particular values from the input domain are selected. The fourth goal discusses the re-usability of verification stimuli for regression testing and how these stimuli can be further optimized in order to speed-up the testing. It is quite common in verification that until a satisfying level of coverage is achieved, many redundant stimuli are evaluated as they are produced by pseudo-random generators. However, when creating optimal regression suites, redundancy is not needed anymore and can be removed. At the same time, it is important to retain the same level of coverage in order to check all the key properties of the system. The fourth optimization technique is also based on the genetic algorithm, but it is not integrated into the verification process but works offline after the verification is ended. It removes the redundancy from the original suite of stimuli very fast and effectively so the resulting verification runtime of the regression suite is significantly improved.

Keywords

Functional verification, simulation-based verification, Universal Verification Methodology, SystemVerilog, optimization, automation, genetic algorithm, coverage-driven verification, coverage metrics.

Bibliographic Citation

Marcela Šimková: New Methods for Increasing Efficiency and Speed of Functional Verification, Ph.D. thesis, Department of Computer Systems FIT BUT, Brno, CZ, 2015

Prohlášení

Prohlašuji, že jsem tuto dizertační práci vypracovala samostatně pod vedením svého školitele Doc. Ing. Zdeňka Kotáska, CSc., a že jsem uvedla všechny literární prameny, ze kterých jsem v průběhu své práce čerpala.

. . . .
Marcela Šimková
14. září 2015

Poděkování

Děkuji všem, kteří mi byli oporou při sepisování této práce a byli mi cennými rádci v průběhu celého mého studia. Jmenovitě děkuji mému školiteli, panu Doc. Zdeňkovi Kotáskovi za vedení. Děkuji mému budoucímu manželovi Michalovi za lásku, podporu a povzbuzení. Také děkuji Ondřejovi Lengálovi, Michalovi Kajanovi a Liborovi Polčákovi za výběr tématu diplomové práce, který mně nasměroval i k tématu dizertační práce a za všechny jejich rady a čas, který mi věnovali. Děkuji mým kolegům z UPSY a z Codasipu, kteří se mnou spolupracovali.

Contents

Acronyms

ABV Assertion-Based Verification. 18

ALU Arithmetic-Logic Unit. 82

ASIC Application-Specific Integrated Circuit. 7

ASIPs Application Specific Instruction-set Processors. 44, 71

BDDs Binary Decision Diagrams. 46

CDV Coverage-Driven Verification. 18, 76

CTL Computation Tree Logic. 12

DPI Direct Programming Interface. 18

DRC Design Rule Checking. 13

DUV Design Under Verification. 7

EA Evolutionary Algorithms. 28

EC Evolutionary Computing. 28

FIFO First In First Out. 55

FPGA Field Programmable Gate Array. 7

FSM Finite State Machine. 45

GA Genetic Algorithm. 28, 37, 77

GPPs General Purpose Processors. 44

HDLs Hardware Description Languages. 11

HGEN Hash Generator. 55

HVLs Hardware Verification Languages. 11

IC Integrated Circuit. 12

IoT Internet of Things. 44

LLVM	Low Level Virtual Machine. 71
LTL	Linear Temporal Logic. 12
LVS	Layout Versus Schematic. 13
MPSoCs	Multi-Processor Systems-on-Chip. 44
NFL	No Free Lunch. 37
OOP	Object-Oriented Programming. 17
OVM	Open Verification Methodology. 8, 26
PSL	Property Specification Language. 18
RTL	Register-Transfer Level. 11
SoCs	Systems-on-Chip. 44
SVA	SystemVerilog Assertions. 18
UVM	Universal Verification Methodology. 8, 26
VCD	Value Change Dump. 53
VMM	Verification Methodology Manual. 25

Chapter 1

Introduction

1.1 Motivation

Computer systems play nowadays very important role in human everyday lives. They are everywhere and they help people in many ways: they assist them in their working lives, they help them in their households and they even entertain them. To cover these tasks perfectly, the most important property of these systems is their correctness with respect to their specification (i.e. ensuring that they do not behave in a faulty way). However, the complexity of modern computer systems is rising rapidly so achieving high degree of correctness is a difficult challenge. The discipline dealing with this issue is called *verification*.

According to The 2012 Wilson Research Group Functional Verification Study [37], which is a blind study supported by Mentor Graphics and conducted regularly by Wilson Research Group, verification became a very important part in the development of computer systems, as the ratio of the design and verification engineers is almost 1:1 and increased by 75 % since 2007 (see Figure 1.1).

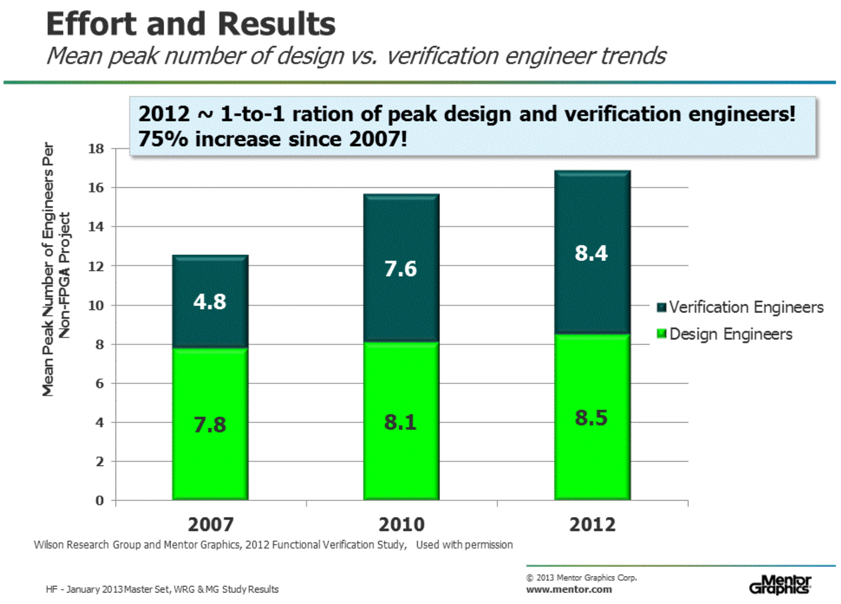


Figure 1.1: Source: The 2012 Wilson Research Group Functional Verification Study: The mean peak number of design vs. verification engineer trends.

For verification of computer hardware, a variety of options is available to engineers: (*i*) simulation and testing, (*ii*) functional verification, and (*iii*) formal analysis and verification.

Although simulation and testing might be seen as old-fashioned methods, they are still highly effective especially in the early phase of implementation and debugging of base system functions. The benefits of testing real hardware (either in the form of an Application-Specific Integrated Circuit (ASIC) or a configuration of a Field Programmable Gate Array (FPGA)) is the speed of testing (as it is performed in real time) and also the possibility to cover faults arising from the technology used for physical implementation of the logical circuit. Software simulation of hardware allows the developer to check that base system functions conform to system specification even before the circuit is physically assembled. This aspect plays more and more important role nowadays when the physical implementation of the systems into hardware is becoming critically expensive.

Functional verification can be seen as a more efficient approach to verification of systems before they are taped-out. This approach is based on simulation of the environment of the system, of the system itself being called a Design Under Verification (DUV), and uses *coverage-driven verification*, *constrained-random stimulus generation*, *assertion-based verification*, and other techniques to check the system correctness and maximize the efficiency of the overall verification process. The adaptation of functional verification techniques is illustrated in Figure 1.2.

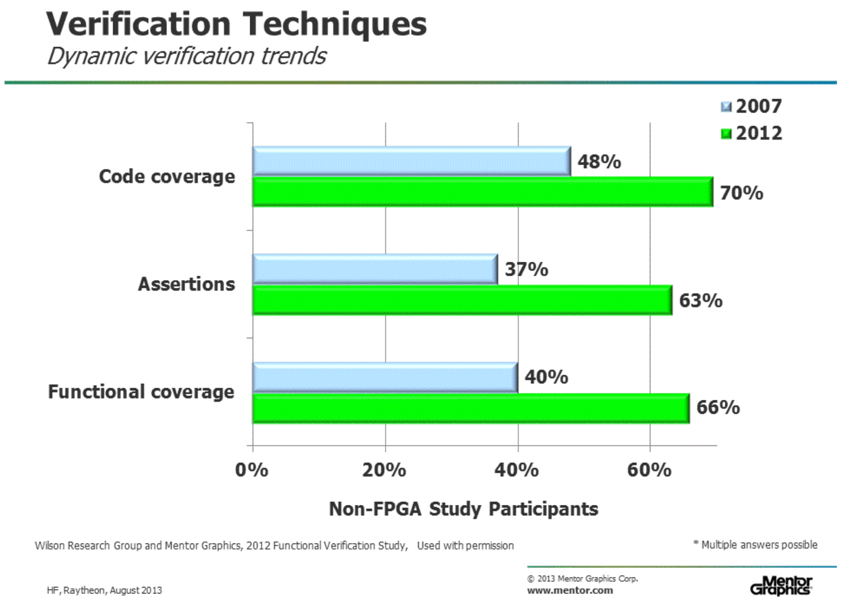


Figure 1.2: Source: The 2012 Wilson Research Group Functional Verification Study: Dynamic Verification Trends.

In order to achieve *completeness* (i.e. a certainty that the system does not violate its specification) of the process of verification, some formal techniques and tools can be used. They are adequate for detecting errors in highly concurrent and complex designs where traditional ways mentioned before are not sufficient. The adaptation of formal verification techniques is depicted in Figure 1.3.

As can be seen, a lot of different verification approaches with their own advantages and disadvantages exist. However, it is worth deciding carefully which of them is suitable for a specific scenario. Therefore, one of the goals of this thesis is to analyze and compare existing techniques for verification of hardware circuits and provide a simple guideline for verification engineers.

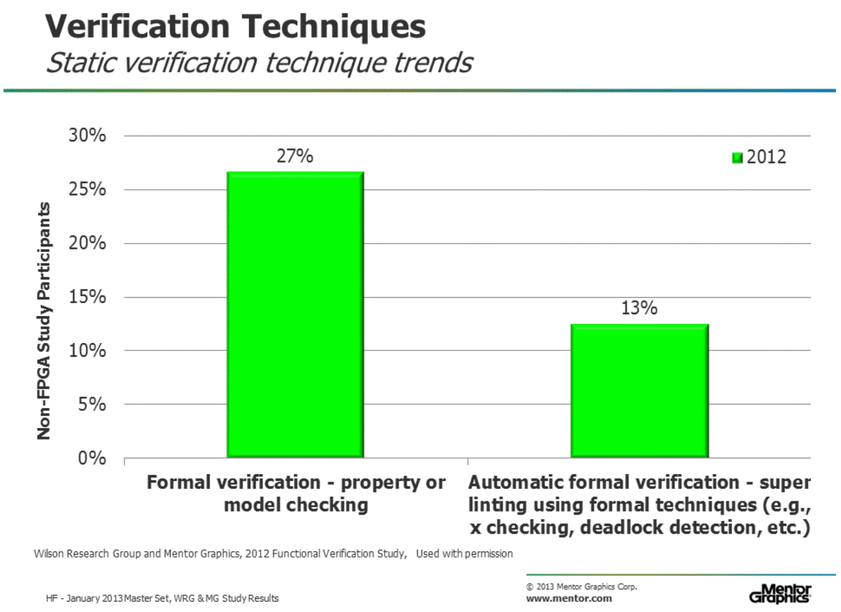


Figure 1.3: Source: The 2012 Wilson Research Group Functional Verification Study: Static Verification Techniques Trends.

1.2 Problem Statement

Today's highly competitive market of consumer electronics is very sensitive to the time it takes to introduce a new product (the so-called *time to market*). Figure 1.4 illustrates how many participants of the Wilson Research Group Functional Verification Study can handle the original time schedule of their projects. Unfortunately, it can be seen that for several years the trend is stable, around 67 % of projects are behind the schedule.

This has driven the demand for fast, efficient and cost-effective methods of verification of hardware systems. They must tackle several challenges:

- defining the appropriate metrics to measure the progress in verification,
- restricting the time needed to discover a next error,
- restricting the time to isolate and resolve the error,
- creating sufficient tests to verify the whole design and manage the verification process.

In Figure 1.5, the overview of errors that are most commonly discovered by verification is illustrated. It can be seen that logic and functional errors take the biggest portion of them, but the good news is that they can be effectively handled by pre-silicon verification approaches.

1.3 Thesis Contribution

We decided to focus our research on functional verification as it is extensively used in industry nowadays. It utilizes sophisticated programming languages for hardware verification, such as SystemVerilog [1], and standardized verification methodologies (e.g. Open Verification Methodology (OVM) [31], Universal Verification Methodology (UVM) [69]).

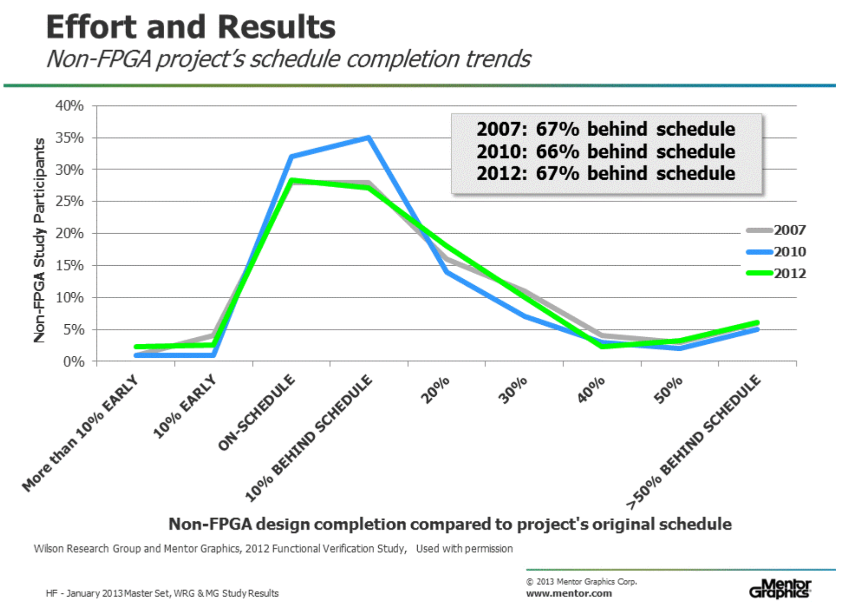


Figure 1.4: Source: The 2012 Wilson Research Group Functional Verification Study: Project's Schedule Completion Trends.

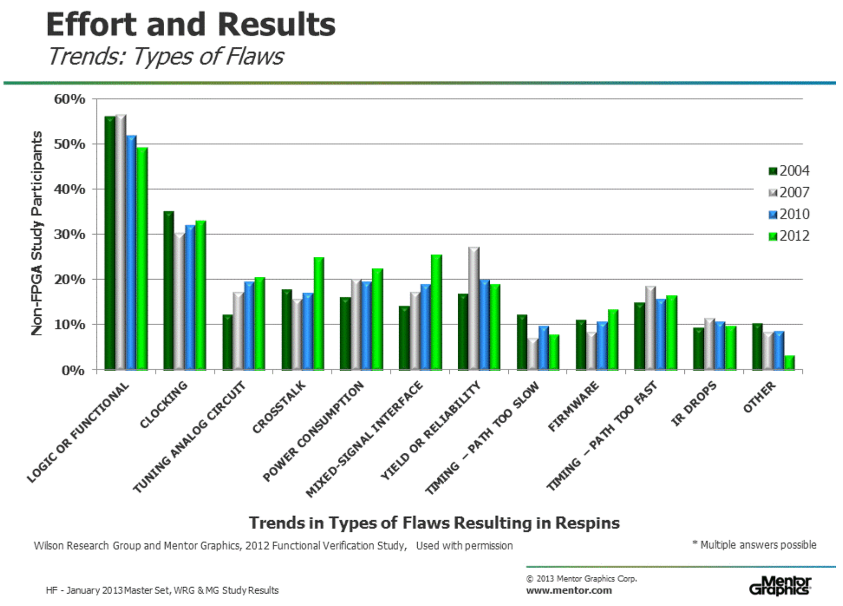


Figure 1.5: Source: The 2012 Wilson Research Group Functional Verification Study: Types of flaws.

In the thesis, every of the challenges mentioned in Section 1.2 is taken into account. It is believed that the main contributions of this thesis are as follows.

- Various coverage metrics are discussed, how they capture design specifications and functionalities and how they allow to measure the progress in verification. It is outlined, how the functional coverage points (monitors) must be defined as it is quite tricky and a non-trivial problem.

- The bottleneck concerning generation of suitable stimuli for the DUV that can adequately activate all coverage points and achieve high coverage rate is also targeted in the thesis. An optimization technique is proposed that works in the background of the verification process and automatically without the human intervention drives generation of stimuli so the uncovered properties of the system are checked.
- The time bottleneck is eliminated by accelerating functional verification runs in the FPGA accelerator and by automated generation of verification environments with respect to the UVM methodology.
- The verification process must be successfully managed also in the later phases of the development of hardware systems, when the functionality is slightly modified or some optimizations to the design are made. Therefore, an optimization technique is proposed that helps to create small but coverage-effective regression suites from the stimuli already used in functional verification.

1.4 Thesis Organization

The text of the thesis is divided into thirteen chapters. Chapter 2 introduces the history of techniques and tools for hardware design and verification. The history overview is followed by theoretical principles of widely used verification approaches: simulation and testing, functional verification and formal verification. Chapter 3 describes the functional verification in more details as well as the SystemVerilog language, its features and steps to build effective and reusable verification environments. Chapter 4 describes evolutionary computing, especially the genetic algorithm, its parameters and applications in practice. The reason for mentioning evolutionary computing and algorithms is that the genetic algorithm is used as the main optimization tool in the techniques proposed in this thesis. Chapter 5 discusses the functional verification state-of-the-art and the available related work in the field of optimization of functional verification. Many literature and research papers are examined and evaluated. In Chapter 6, the main goals of this Ph.D. thesis are outlined. With respect to these goals, four chapters follow, each of them is devoted to one optimization technique that is uniquely proposed in this thesis and targets one of the goals. A detailed description of each technique is included as well as the experimental work. The first optimization technique in Chapter 7 targets acceleration of functional verification using the FPGA technology. The optimization technique presented in Chapter 8 automates generation of the UVM-based verification environments. In Chapter 9, automation and optimization of coverage-driven verification is described. The optimization technique used in this chapter utilizes the genetic algorithm as the main optimization tool. Also in the optimization technique proposed in Chapter 10 the genetic algorithm is used, but this time for finding optimal regression suites that are built from stimuli used in functional verification. Chapter 11 summarizes the work and outlines its possible further enhancements. Finally, Chapters 12 and 13 contain additional information related to the thesis, mainly the list of author's publications and research projects.

Chapter 2

Verification Approaches

The discipline dealing with checking whether a hardware system satisfies a given correctness specification is called *verification*. The main purpose of verification is to find as many design errors as possible before the system is deployed or even produced. This chapter presents the history of various design and verification techniques in the last four decades and also verification approaches that are currently mostly used at different stages of the hardware development. Furthermore, the Register-Transfer Level (RTL) verification approaches are presented in more details because they are used to discover most of the functional and logical errors in the pre-silicon debugging. Moreover, a simple comparison of these approaches is included, so the verification engineers can get an opinion which of them is suitable for specific verification issues and scenarios.

2.1 Brief History of Design and Verification Methods

Over the years, as technology progressed and the complexity of hardware designs increased, new verification methods were needed and developed. During the last 40 years many advanced and effective techniques and tools for verification appeared (the history overview is taken from [19]).

In the 70's, hardware was designed by drawing schematics on the paper, later electronically, but verification was mainly performed by detailed review of the schematics. During the 80's, simulation tools became popular, but they were predominantly proprietary. Testbenches were hard to build with non-standardized tools, and verification relied on manual assessment of simulation results. In 1987, VHDL became a standard and Hardware Description Languages (HDLs) were generally accepted as means of design and verification. *Static verification tools* also started to emerge around this time in order to help in static analysis determining whether designs conform to design rules. As the density of designs increased, the use of more intelligent testbenches became a common practice. Self-checking testbenches with direct testing were then followed by testbenches that used *constrained-random stimulus generation* and finally *transaction-based* testbenches. The implementation of advanced testbenches along with the introduction of better tools (coverage checkers, faster RTL simulators) increased rapidly the popularity of simulation. However, as designs became even more complex, HDLs of that time were not sufficient for effective verification, since only very restricted functions and data types could be defined.

To overcome these deficiencies, Hardware Verification Languages (HVLs) were introduced. These HVLs were tailored for simulation and integration with HDLs. HVLs introduced the concept of *functional coverage* and helped in tracking the progress of verification. Evolution of verification methodologies and languages led towards standardization of the SystemVerilog language (its current standard being IEEE 1800-2009 [1]).

Assertion-based verification is a concept originally introduced in the scope of formal (static) verification. This approach uses a simple language based on a temporal logic (such as Linear Temporal Logic (LTL) or Computation Tree Logic (CTL)) to describe specifications of parts of a system in the form of a set of temporal formula. In the case of formal verification tools, the state space of the system is then exhaustively searched to check whether all specified temporal formula are valid in all accessible states of the verified system. In case a state that invalidates some formula is found, a *counter-example* (also called a *witness*) is provided in the form of a waveform of a sequence of input and internal signals that leads to the failing state. In case of dynamic verification tools, in each clock cycle of the simulation all formulas are checked for validity and in the case of a failure the simulation is stopped with an error message.

2.2 Current Verification Approaches

With the growing complexity of hardware systems, verification became a bottleneck in the development cycle and therefore, it was and still is typically divided into several stages. Every representation of the system at varying levels of abstraction (register-transfer level, gate-level, hardware-level) should be verified independently as is depicted in Figure 2.1 and described in the following text.

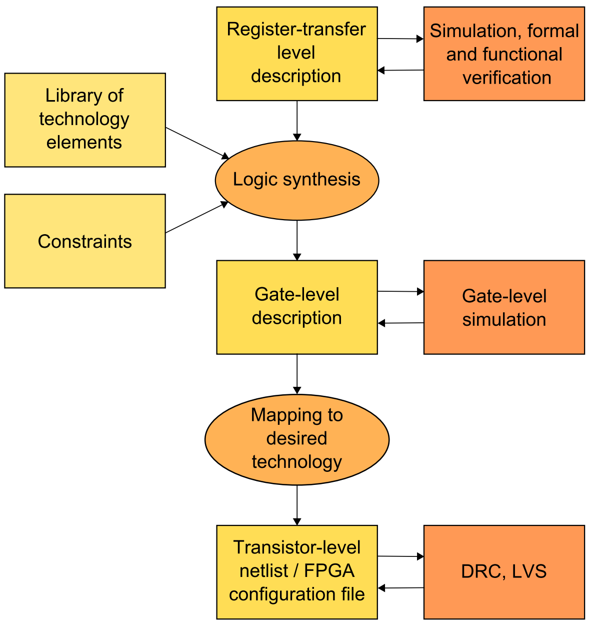

Register-transfer level description
Simulation, formal and functional verification
Library of technology elements
Logic synthesis
Constraints
Gate-level description
Gate-level simulation
Mapping to desired technology
Transistor-level netlist / FPGA configuration file
DRC, LVS

Figure 2.1: Verification stages.

- RTL is a design abstraction level which models a synchronous digital circuit in terms of the flow of digital signals (data) between hardware registers and the logical operations performed on those signals. RTL represents a high-level description of a circuit and is implemented in HDLs like VHDL, Verilog, SystemC, or SystemVerilog. At this level, several verification approaches running in RTL simulators can be applied, with *logic simulation*, *static analysis*, *model checking* and *functional verification* being the most popular.

- *Gate-level netlist* is a description of a circuit which is derived from the RTL representation in the process of logic synthesis. During the logic synthesis, a library of logic elements of a given technology (Integrated Circuit (IC), ASIC, FPGA) and user constraints for timing, power or area are considered. Among techniques that are used for verification of gate-level netlist belongs the *gate-level simulation*.

- *Hardware-level description* can be represented by the transistor-level netlist for IC/ASIC or by the bitstream configuration file for FPGAs. These descriptions are created by specific tools while utilizing concrete physical elements of a target technology. In the case of FPGAs, logic elements are mapped and placed onto physical FPGA elements and interconnections are built in the process of routing. Similar steps are performed in case of ICs/ASICs and as a result, specific physical layout is created. For the verification purposes, Design Rule Checking (DRC) or Layout Versus Schematic (LVS) techniques can be utilized.

Except of the design faults, many defects can arise during the production phase. Therefore, a few techniques have been developed in order to allow the application of manufacturing tests. They are labeled as *Design for Test* and in most cases they insert some specific logic into a tested circuit in order to explore and test all important parts of the circuit properly.

2.3 RTL Verification

Verification approaches working at the behavioral level or register-transfer level of abstraction typically use the HDL description of a circuit for the construction of the simulation model (in the case of the simulation-based approaches), the mathematical model (in the case of the formal approaches) or they simply work only with the source code (in the case of the static analysis). They are described thoroughly in the following subsections.

2.3.1 Logic Simulation and Testing

Logic simulation and testing are often called *bug hunting* methods as their main purpose is to find as many errors as possible in a hardware design during the early (pre-silicon) stages of the development cycle.

A designer typically prepares a simple testbench with a sample of input stimuli and manually checks responses of the system to these stimuli in RTL simulation. This scenario is demonstrated in Figure 2.2. In such a way, a designer can check basic system functions properly. However, the overall behavior can be explored only in a limited number of situations (because of the non-complex testbench). Therefore, the logic simulation provides only a *partial* guarantee of the system correctness.

One of the main strengths of simulations is that software simulators working at the RTL level provide a perfect debugging environment with many functions. All signals of the simulated design are readily accessible. Once an error has been reproduced in this environment, the process of debugging can be performed extremely efficiently. For this reason, it is convenient to have the capability of executing the same tests in order to reproduce the failing scenarios.

The disadvantage of simulation is its low performance which directly depends on the complexity of the simulated design. The reason is that processes are running one after another in the simulator, while in a real hardware, the are running concurrently. Practically, simulation is so computationally intensive that it can take days or even weeks to simulate a large design. For this reason, detailed software simulation can be often performed only on small portions of a complex design and cannot be interconnected to real hardware environment [59].

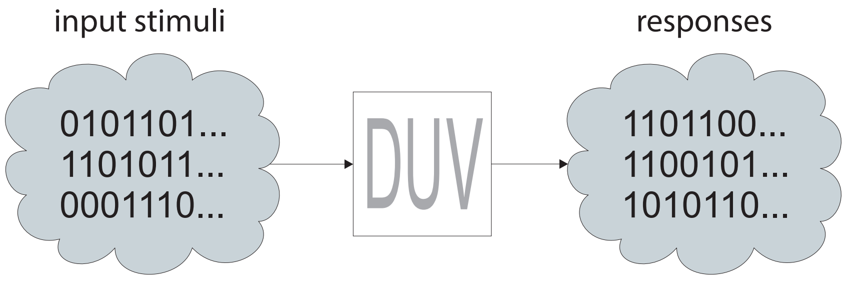


Figure 2.2: A simple testbench architecture.

2.3.2 Static Analysis and Formal Verification

Static Analysis is an approach with a high level of automation. Static analysis tries to avoid execution of the system being examined by analyzing its source code instead. It also gathers some approximate information about the system from its source code. Various forms of static analysis exist, like type analysis, bug pattern searching, equational data-flow analysis, constrained-based analysis, and abstract interpretation. Static analysis is not exclusively intended for checking correctness of systems only, it is used also for optimization, code generation, etc. The main advantages are that it can handle very large systems and does not need a model of the environment of a system. However, it can produce many false alarms and various analysis are specialized just for certain specific tasks.

Formal verification is an approach based on an exhaustive exploration of the state space of a system, hence it is potentially able to formally *prove* the correctness of the given system according to its specification. Formal methods are mainly based on formal mathematical roots. As the task of formal verification is in general undecidable, formal verification methods may not guarantee termination. A typical reason is the so called *state space explosion* problem for the real-world systems. Some methods deal with the issue of termination using abstraction through upper-approximation which may in turn introduce false alarms. Alternatively, a method is allowed to stop with a *don't know* answer or it may become not fully automated (some human intervention is required). If the formal verification run of a system fails, it may be still useful as it can find some errors in the meantime. Another disadvantage is the need to provide formal specifications of the behavior of the system, which makes this approach often hard to use.

There are several distinct verification techniques used in the formal field. The most important and the best known are described here:

- *Model checking* is an algorithmic approach of checking whether a given system satisfies a given property through systematic exploration of the state space of the system. This principle is illustrated in Figure 2.3 where the green circle represents the initial state, yellow circles the progress in the state space exploration and the red flashes stand for errors hidden in the system. System properties are specified using temporal logics like CTL, CTL*, LTL, PLTL or by using *assertions*. Model checking is regarded as a successful method frequently used to uncover well-hidden errors.

  The ultimate goal of this technique is to answer the question: "*Does the model A satisfy the* *property Q?*". A positive reply guarantees that the underlying property holds for all behaviors of the model (at the level of the mathematical proof). A negative reply is usually accompanied by a counter-example that represents a particular run of the system which leads to a violation of the desired property.

The main obstacle encountered by the model checking algorithms is the state space explosion problem. A system easily handled by a simulator, a compiler or an interpreter may be unable to be verified in a reasonable time and by using reasonable resources by a model checker. In practice, model checking requires a good expertise from the user, especially to deal with formulation of properties.

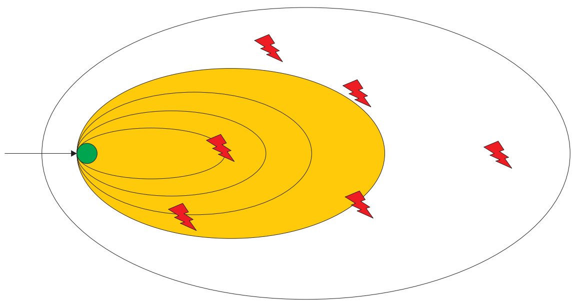


Figure 2.3: Exploration of the state space by model checking.

- *Theorem proving* represents a deductive verification method. It is similar to the classical mathematical way of proving theorems starting with axioms and inferring further facts using rules of the correct inference. This approach is very general but semi-automated as it requires a significant manual intervention of users.

- *Equivalence checking* proves that two representations of a circuit exhibit exactly the same behavior (they are logically equivalent). This approach uses mainly the RTL description of a circuit as a golden reference model. The behavior of the same circuit but described at different level of abstraction (typically the gate-level netlist), is then compared to this reference model. The initial netlist usually undergoes a number of modifications, optimizations or replacements of logic elements during synthesis. Throughout this process, the original functionality and the behavior described by the original code must be maintained.

To conclude, no ideal formal technique that would provide fully automated proofs for all types of systems exists. The choice of the best suited approach strongly depends on the concrete verification problem. Currently in practice, mainly critical system parts are formally verified, e.g. pipeline behavior in processors or dedicated controllers.

2.3.3 Functional Verification

Functional verification is based on simulation (therefore, it also cannot prove the correctness of a system) but uses more sophisticated testbenches with additional features.

First of all, it generates a suite of constrained-random input stimuli and compares the behavior of DUV with the behavior specified by a provided reference model. The reference model is prepared according to the specification in HVL, C/C++ or other languages that are supported. The progress in verification is measured using coverage metrics which express different measurable properties of DUV (functions, source code, instructions, etc.). To facilitate the process of verification and to formally express the intended behavior of a system, *assertions* may be used. All these functional verification techniques will be described in detail in Chapter 3.

Unlike model-checking, functional verification explores the state space of a system in different manner, more to the depth (as can be seen in Figure 2.4). In this way, also states of the system that were not covered by model-checking (because of the state-space explosion problem) can be reached.

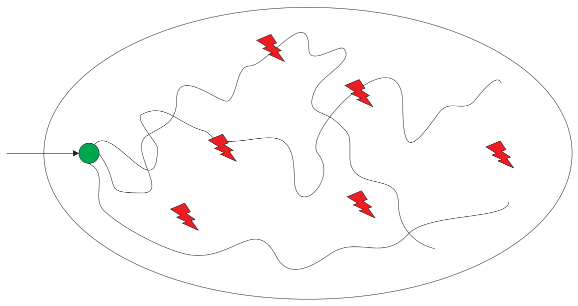


Figure 2.4: Exploration of the state space in functional verification.

Further, this approach is based more on the following idea: the longer the verification run is, the higher is the chance that there are no errors left in the state space (or, at least, the most critical and reachable errors have been found and removed). In Chapter 3, functional verification is elaborated in more details.

Chapter 3

Functional Verification in SystemVerilog

In the rest of this Ph.D. thesis, the attention is paid to functional verification as it is one of the most dominant verification approaches used in industry [90, 77, 7] for the pre-silicon checking of hardware systems. The focus is taken on identifying problems that need to be solved, especially those connected with the speed, the automation and the completeness of functional verification. Verification environments for functional verification are implemented in the SystemVerilog language usually with respect to some well-known verification methodology. Afterwards, they are running in some RTL simulator, e.g. , QuestaSim from Mentor Graphics, Riviera-PRO from Aldec, VCS from Synopsys or Incisive Enterprise Simulator from Cadence.

SystemVerilog is a complex programming language for hardware design and especially for functional verification. While created as the next generation of the Verilog language, it has adopted features from many other programming languages with great impact on its simulation and verification capabilities. SystemVerilog provides a basis for building techniques that increase the efficiency of verification processes. The description of some of these techniques follows:

- **Object-Oriented Programming (OOP).** This approach allows easier design of large systems with support of common design patterns or reusable components. Verification environments are more modular and thus easier to develop and debug. The mechanisms of encapsulation, inheritance and polymorphism support the reuse of verification components, which leads to an increase in productivity. Several orders of re-usability are possible. The first order is when the same verification environment is used across multiple test cases on the same project. The second order occurs when some verification components are used several times in the same project environment. The third order is when some verification components are used across different verification environments for different systems. The reuse was significantly improved when verification methodologies that are introduced in Section 3.2 were incorporated into functional verification. They provide libraries of freely available and reusable components that are just modified for particular use, mostly in terms of interfaces and reference models.

- **Constrained-random stimuli generation.** For checking full functionality of a larger design it becomes more difficult to create a complete set of stimuli. A suitable solution is to create test cases automatically using constrained-random stimuli generation to target corner cases and stress conditions. Test scenarios are restricted to be valid using constraints. Constraints define the correct form of the generated data and can be also used to guide verification tests to interesting DUV states. Due to randomly generated stimuli, it is necessary to use coverage mechanisms to explore the DUV state space evenly.

- **Assertion-Based Verification (ABV).** This is a technique used to formally express the intended design behaviour, internal synchronization, and expected operations, using assertions (i.e. properties that must hold at all times). Assertions can be expressed at many levels of the device including internal and external interfaces (to detect critical protocol violations), clock-domain crossings and state machines. Assertions create monitors at critical points of the design without having to create separate testbenches where these points would be externally visible. ABV also guides the verification task and accelerates the verification because it provides feedback at the internal level of the device as it is possible to locate the cause of a problem faster than from the output of a simulation. Two examples of assertion languages are Property Specification Language (PSL) and SystemVerilog Assertions (SVA).

- **Cooperation with other programming languages.** Direct Programming Interface (DPI) allows SystemVerilog code to call functions in other programming languages as if they were native SystemVerilog functions. Data can be passed between the two domains through function arguments and results. Inter-operable environments and components may be used to reduce the effort required to verify a complete product in case some parts of the product are already prepared in other programming languages.

- **Coverage-Driven Verification (CDV).** *Coverage* is an important part in functional verification [60]. Let us define a terminology connected with the coverage at first.

  Coverage metric is one measurable attribute of a circuit, e.g. the number of executed lines of code or the number of checked arithmetical operations. In general, it is possible to specify different coverage metrics in functional verification which are connected either with the source code or with the intended functionality.

  Coverage space represents an n-dimensional region defined by n coverage metrics.

  Coverage model is an abstract representation of a circuit composed of selected $i \leq n$ coverage metrics and their relationship. It forms an i-dimensional subspace of the n-dimensional coverage space.

  A 3-dimensional coverage space is illustrated in Figure 3.1. The dimensions of this coverage space are built by 3 different coverage metrics. A 2-dimensional subspace forms one coverage model.

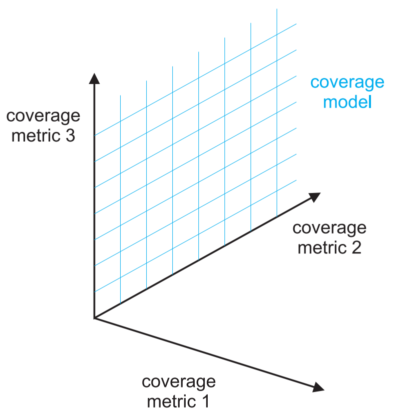


Figure 3.1: The 3-dimensional coverage space.

Coverage models create a basis for measuring the quality and completeness of functional verification. In other words, coverage models allows to measure how well input stimuli cover the most of the possible behaviors (measurable properties) of DUV. It became also a part of various industrial standards like DO-178B, DO-178C, and ISO 26262.

Achieving *coverage closure* means provoking the occurrence of each (or some threshold) of the measurable properties [60]. RTL simulators offer coverage analysis and produce statistics about which coverage items were covered during the verification runs. If there are holes (unexplored areas) in the coverage analysis, the verification effort is directed to the preparation of suitable test scenarios which will be able to cover these holes. That is the reason why this approach is called coverage-driven verification. One option is to manually change the constraints of the pseudo-random generator, the second option is to prepare direct tests.

The list of supported coverage metrics follows, some of them are generated automatically in a simulator, other must be written by hand.

- *Functional coverage* is specified manually; it measures how well input stimuli cover the functional specification of DUV. It focuses mostly on the semantics, e.g.: Did the test cover all possible commands or did the simulation trigger a buffer overflow? For more precise definition and examples, see Chapter 4 in [60] or Chapter 18 in IEEE SystemVerilog standard [1].
- *Structural coverage* is generated automatically by a simulation tool, so no extra HVL code needs to be written because the code coverage tool included in many simulators instruments the design automatically by analyzing the source code and adding hidden code to gather statistics. In general, structural coverage measures how well input stimuli cover the implementation (the source code) of DUV. For more precise definition and examples, see Chapter 5 in [60] or Chapter 29 in IEEE SystemVerilog standard [1]. Typical structural coverage metrics are defined below.
    * *Toggle coverage* measures how many times signals and latches in the HDL model of a hardware system changed their logic values during the simulation. The absence of signal change activity indicates that any input stimulus did not target a specific corner area of the verified system. It considers mainly the block-level structure of the design not its functional significance.
    * *Code coverage* takes the syntactical structure of an HDL specification and measures which HDL lines were executed by a verification run. The limitation of code coverage is the missing semantic insight; there is no knowledge about the correctness of the system in the case of coverage holes. Code coverage metric can be concretized to *statement coverage*, *branch coverage*, *condition coverage* and *expression coverage*.
    * *Finite State Machine coverage* – associates the state-machine structures in HDL designs and measures visited states (*state coverage*) and transitions between these states (*transition coverage*).

3.1 Verification Process

Verification is a complex task, therefore, a lot of effort should go into specifying when a DUV can be considered as fully verified. Inspired by [8] we introduce the main steps of a verification process (Figure 3.2):

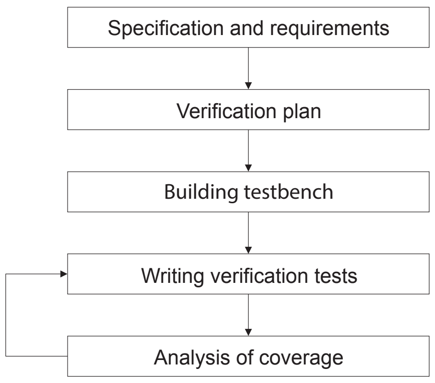


Figure 3.2: Steps of the verification process.

3.1.1 Specification and Requirements

In order to check a new implementation of DUV for its functional correctness, we need a reference description, either a text specification or a previous *golden model* reference implementation which represents the intention of the design. In many cases, the specification is given on a higher level of abstraction so it does not capture the detailed behaviour of the design.

The requirements for the appropriate verification process are built according to the specification of the functionality of the device. The requirements should be defined as soon as possible in the project life cycle, ideally while the architectural design is in progress. It is recommended that the requirements are identified by all members of the project team: hardware and software developers as well as verification engineers. A few examples of requirements are: definition of legal input and output patterns according to the protocols of used interfaces, description of incorrect behaviour of the device, identification of potential corner cases, supported configurations, errors from which the design can or cannot recover and how the system responses and behaves in such cases.

3.1.2 Verification Plan

A verification plan helps verification engineers to understand how verification should be done. It is closely tied to the specification and requirements and contains a description of which features need to be exercised and which techniques and tools should be used to achieve the specific goals so that the progress in verification could be easily measured.

Moreover, it should contain a precise definition of all the resources that will be needed and not only the computational resources but also human and financial resources.

- *The stimuli generation plan* chooses the character of input sequences:

  a) **Direct tests** — each test contains direct sequences of stimuli which are targeted at a very specific set of design elements. The DUV is stimulated with these sequences and the resulting log files and waveforms are then manually reviewed to make sure that the design behaves according to the specification. Direct tests typically find errors that are expected to be in the design and cover the state space very slowly. Creating complex stimuli set is a very time-consuming and challenging task because this approach requires a verification engineer to write many tests in order to achieve high coverage.

b) **Constrained-random stimuli generation tests** — as designs become more complex it is impossible to use only direct tests. A more efficient way to verify complex designs thoroughly is with constrained-random stimuli generation. The widest possible range of stimuli can help to find bugs which were not anticipated. Random tests explore the space much faster than direct tests, reduce the number of required tests, and increase productivity and quality of the verification process.

- *The coverage plan* specifies which coverage metrics will be used to track the progress in verification and what is the level of coverage that should be achieved in every such metric.
- *The checker plan* uses mechanisms for predicting the expected response and for comparing the observed response (typically from external outputs of DUV) against the expected one. These checks should be intelligent and independent of test cases. The following list introduces several means of predicting expected responses.

a) **Assertions** — these are used to verify the response of the device based on internal signals. Any detected discrepancy in the observed behaviour results in an error which is reported near to the origin of the functional defect. Assertions work well for verifying local signal relationships; they can detect errors in handshaking, state transitions and protocol rules. On the other hand, they are not well suited for detecting data transformations, computation and ordering errors. Despite the effectiveness of assertions, they are limited to the types of properties that can be expressed only using expressions in a clock temporal logic.

b) **Scoreboarding** — a scoreboard is used to dynamically predict the response of the device. Stimulus applied to the device is also passed to the transfer function which performs all transformations on the stimulus to produce the form of the final response. Modified stimulus is inserted into a data structure called transaction. The observed response from the DUV in the form of the transaction is forwarded to the comparison function which verifies whether it reflects the expected response or not (Figure 3.3).

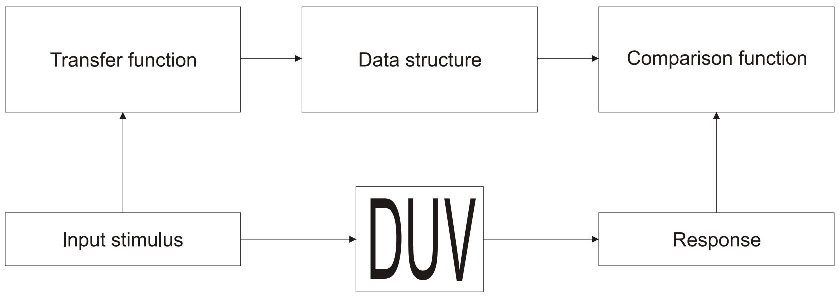


Figure 3.3: Scoreboarding.

The transfer function may be implemented using a reference (golden) model, even e.g. in the C, C++ language (and integrated into the testbench through the DPI). The data structure stores the expected response until it can be compared against the observed output. It means that DUV and the reference model can run asynchronously to each other. Scoreboarding works well for verifying the end-to-end response of a design and the integrity of the output data. It can efficiently detect errors in data computations, transformations and ordering. It can also easily identify missing or spurious data.

c) **Reference model** — like a scoreboard, it is used to dynamically predict the response of the device. The stimulus applied to the design is also passed to the reference model. The output of the reference model is compared against the observed response (Figure 3.4). The reference model mechanism has the same capabilities and challenges as the use of scoreboards. Unlike a scoreboard, the comparison function works directly from the output of the reference model so that it has to produce outputs in the same order as the design itself. However, there is no need to produce the output with the same latency or cycle accuracy. The comparison of the observed response is performed at the transaction-level, not at cycle-by-cycle level. The use of reference models depends heavily on their availability. If they are not available, scoreboard techniques are more efficient to be used. Reference models are often written in the C, C++ language and therefore easily exploitable by a SystemVerilog testbench using the mechanism of the DPI.

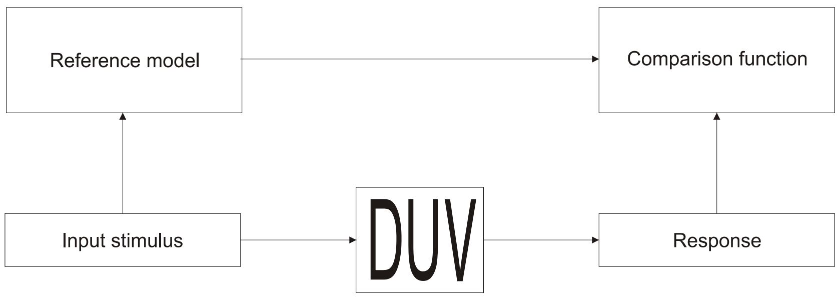


Figure 3.4: Reference model.

d) **Accuracy** — the simplest comparison function compares the observed output of the design with the predicted output on a cycle-by-cycle basis, so synchronously. This approach requires the response to be accurately predicted down to the cycle level.

e) **Offline checking** — used to predict responses of the design before or after a simulation of the design is done. In pre-simulation prediction, the offline checker produces a description of expected responses, which are dynamically verified against the observed responses during the simulation (Figure 3.5). Some utilities can perform post-simulation comparison (Figure 3.6). In both cases the response can be checked at the cycle-by-cycle or the transaction level with reordering. Off-line checkers work well for verifying the end-to-end response of a design and the integrity of the output data based on executable system-level specifications or mathematical models. Although offline checkers are usually used with a reference model, they can be used also with scoreboard techniques implemented as a separate offline program.

3.1.3 Building Testbench

Verification environments (also called testbenches) determine the correctness of the DUV. This is accomplished in general by:

1. generating stimuli,
2. applying stimuli to the DUV,
3. capturing the response,

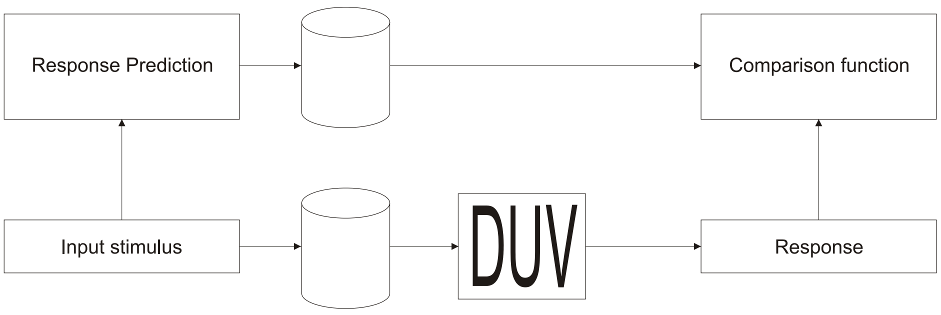


Figure 3.5: Pre-simulation offline checking.

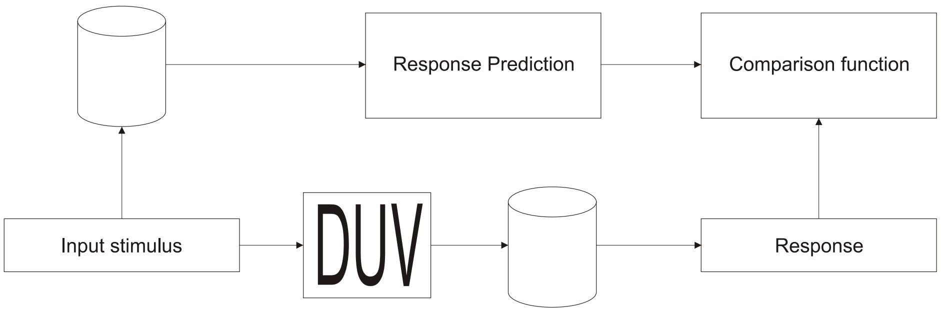


Figure 3.6: Post-simulation offline checking.

4. checking correctness of the response,
5. measuring progress against the overall verification goals.

The continuous growth in the complexity of hardware designs requires a modern, systematic and automated approach to creating verification environments. One of the big challenges in developing verification environments is to make effective use of the OOP paradigm. When used properly, these features can greatly enhance the re-usability of testbench components.

Functional verification environments are usually complex so it is highly desirable to divide the code into smaller pieces that can be developed separately. Some general components can be recognized in Figure3.7

- Scenario level:
  - Testcases are situated on the top of the verification environment. They are implemented as a combination of additional constraints for generators, new random scenario definitions, synchronization mechanisms between stimuli, or by direct stimuli.

- Verification level:
  - The generator produces constrained random stimuli and passes them to the driver.
  - The driver receives high-level direct stimuli, usually in the form of data structures called transactions, from the generator, breaks them down into individual signal changes, supplies them on the input interfaces of the DUV, and sends their copies to the scoreboard. The driver may also inject errors or add delay parts.

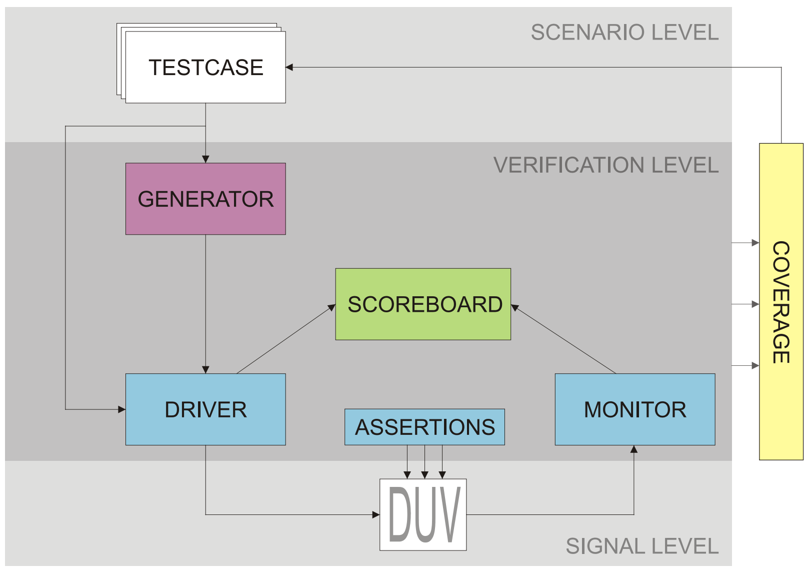


Figure 3.7: A verification environment.

- The monitor drives the output interfaces of DUV, observes signal transitions, groups them together into high-level transactions and passes them to the scoreboard.
- The scoreboard compares transactions received from the driver and from the monitor and in the case they do not match, an error is reported.
- Assertions check the validity of protocols on interfaces of the DUV. In the case of violation they directly report the failing assertion.

- Signal level:
  - DUV is connected to the verification level components through a set of signals. This layer provides pin name abstraction to enable the whole verification environment to be used unmodified with different implementations of the same DUV.

Verification environments and test cases should access the design only through the verification level and never access the signal level unless absolutely necessary.

3.1.4 Writing Verification Tests

After the testbench is applied to the DUV it is the time for validating the DUV. By analyzing coverage reports, new tests are written to cover holes in coverage using two different approaches. The first method captures a new run of the simulation with a different seed of the generator or tailors constraints on input stimuli (creating many unique input sequences with constrained-random stimuli generation). The other approach represents creating direct tests. This iterative process can be seen in Figure 3.8.

3.1.5 Analysis of Coverage

Coverage tools gather information during a simulation and post-process them in order to produce a coverage report. After analyzing both functional and structural coverage reports, new tests are written to reach uncovered areas of the design until a sufficient level of coverage is achieved.

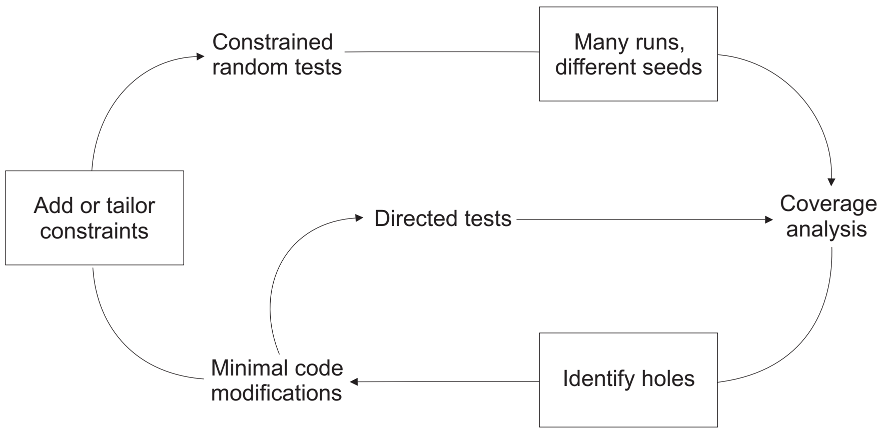

Constrained random tests
Many runs, different seeds
Add or tailor constraints
Directed tests
Coverage analysis
Minimal code modifications
Identify holes

Figure 3.8: Analysis of coverage.

3.2 Verification Methodologies

The success of verification does not depend only on the verification language. The target is to create an applicable verification methodology that uses common approach and creates highly interoperable verification environments and components. An effective methodology must address the main verification challenges: productivity and quality. Coverage-driven verification coupled with constrained-random stimuli generation are currently becoming the main principles of verification methodologies.

Various methodologies were developed with collaboration of different companies as shown in Figure 3.9, a description of three most popular follows.

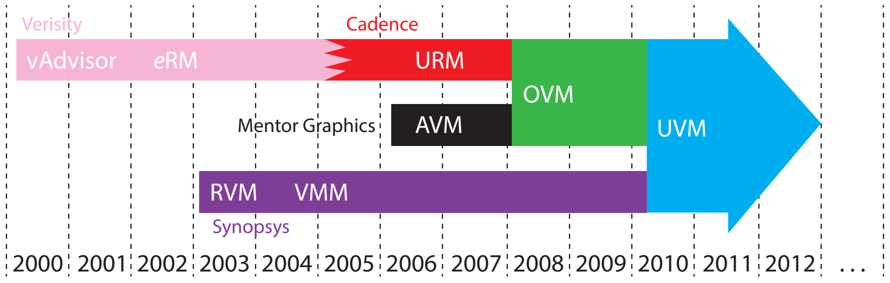


Figure 3.9: The evolution of verification methodologies.

- **Verification Methodology Manual (VMM)** [8] — was co-authored by verification experts from ARM and Synopsys. VMM's techniques were originally developed for use with the OpenVera language and were extended in 2005 for SystemVerilog. VMM has been successfully used to verify a wide range of hardware designs, from networking devices to processors. It describes how to build scalable, predictable and reusable verification environments. Very useful is the set of base classes for data, environment and utilities for managing log files and interprocess communication. VMM is focused on the principles of coverage-driven verification and follows the layered testbench architecture to take full advantage of the automation when each layer provides a set of services to the upper layers while abstracting them from lower-level details.

- **Open Verification Methodology (OVM)** [31] — this methodology is the result of a joint development between Cadence and Mentor Graphics to facilitate true SystemVerilog interoperability with a standard library and a proven methodology. The class library provides building blocks which are helpful for fast development of well-constructed and reusable verification components and test environments in SystemVerilog. The OVM class library contains:
  - component classes for building testbench components,
  - reporting classes for logging,
  - factory for object substitution,
  - synchronization classes for managing concurrent processes,
  - sequencer and sequence classes for generating realistic stimuli.
- **Universal Verification Methodology (UVM)** [69] — is a state of the art methodology that extends OVM. In Figure 3.10, the architecture of the UVM testbench is shown. The basic components that were presented in Section 3.1.3 are recognizable and some additional components are present.

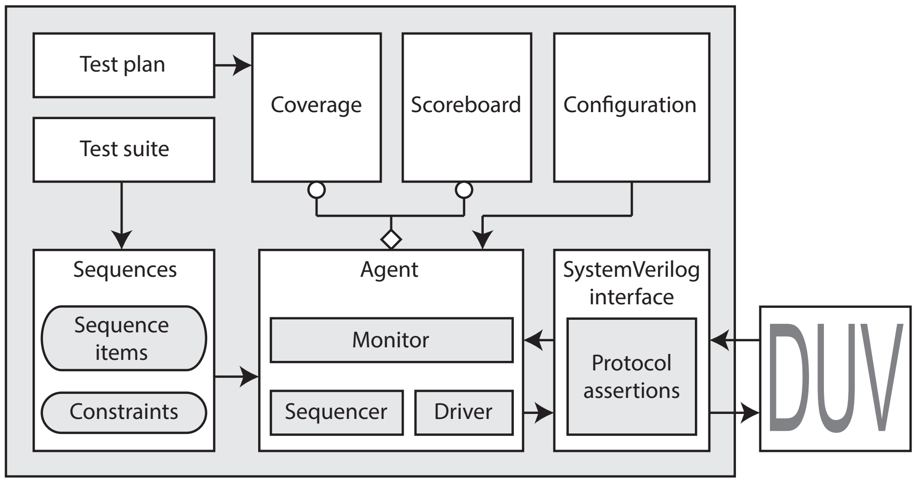


Figure 3.10: The UVM verification environment.

The *Test plan* contains all the test cases that will be evaluated during verification. Every test case (*Test suite*) consists of several *Sequences* that encapsulate input stimuli (*Sequence items*). There, the generator of pseudo-random stimuli can be utilized. Therefore, *Constraints* can be present here which restrict the generated data. Sequence items are propagated to the unit called *Sequencer* and from this unit to the *Driver* that drives input ports of DUV. Output ports of DUV are monitored by the *Monitor* unit. Sequencer, Driver and Monitor are grouped together in the structure called *Agent*. The purpose of Agents is to tie the components that are logically bounded to some *SystemVerilog virtual interface* which corresponds to one or more real interfaces of DUV (a virtual interface logically encapsulates input and output signals of DUV). If the DUV contains several interfaces of the same type, more Agents of the same type are instantiated in the verification environment as well as more virtual interfaces. *Assertions* that check the validity of protocols are also present here. Furthermore, *Coverage* monitors are instantiated in the verification environment and they measure user-defined functional properties. Of course, the most important part of testbenches is *Scoreboard* which implements the reference functionality. It can be either implemented in SystemVerilog or can contain a reference model in C/C++. UVM also adds advanced *Configuration* options to the verification environments, so all components are configured systematically and hierarchically.

Phasing is another very important part of UVM. Despite phasing was firstly introduced in OVM, UVM moves it to the next level as it is more elaborated and more efficient. The common phases are the set of functions and tasks that all UVM components execute together. In other words, all UVM components are always synchronized with respect to the common phases. All the common phases are illustrated in Figure 3.11.

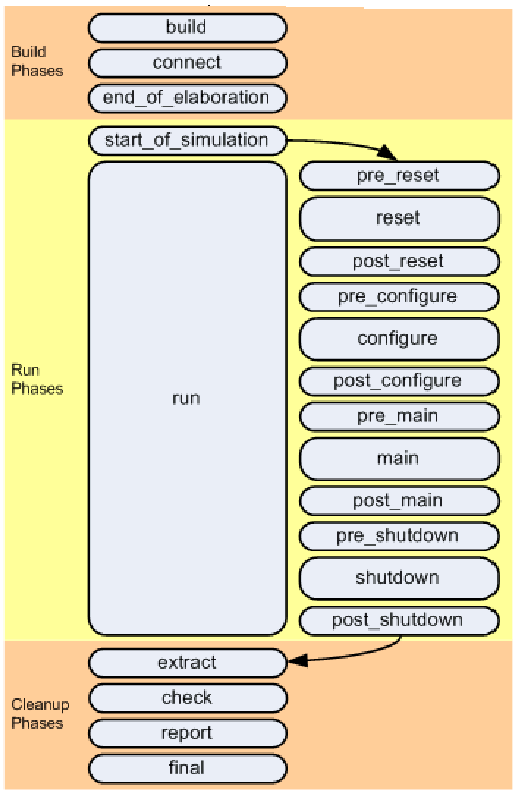


Figure 3.11: The UVM common phases. Source: http://www.soccentral.com/

3.3 Chapter Overview

In this chapter, the functional verification approach was introduced as it is the main topic of this Ph.D. thesis. Functional verification environments are mostly written in the SystemVerilog language, therefore, features of this language were described as well as the steps how to build effective and reusable verification environments according to the verification methodologies (OVM, UVM).

Chapter 4

Evolutionary Computing

This chapter familiarizes the reader with the basics about Evolutionary Computing (EC). The reason for incorporating this chapter is that EC will be used as an optimization tool in the techniques proposed in this Ph.D. thesis and therefore, it is reasonable to understand the essential terms.

The main principles of EC and a brief history of this field are presented. Afterwards, a general scheme that forms the common basis for all the variants of Evolutionary Algorithms (EA) is outlined and the main components of EAs are discussed, explaining their role and relating issues of terminology. Further, general issues concerning operation of EAs are discussed, they are placed in a broader context and their relationship with other global optimization techniques is explained. An important part is the description of the most widely known type of EA: the Genetic Algorithm (GA), because it will be frequently referred in the thesis. Finally, the issue of setting the values of various parameters of GA is discussed as it is crucial for good performance. The theoretical background for this chapter is taken from [26, 73, 67].

4.1 Main Principles of Evolutionary Computing

Evolutionary computing is a computer science research area. As the name indicates, it is a special way of computing, which draws inspiration from the process of Darwin‘s natural evolution [21]. Let us consider natural evolution simply as follows. A given environment is filled with a population of individuals that compete for survival and reproduction. The *fitness* of these individuals - determined by the environment - relates to how well they succeed in achieving these goals, i.e., it represents their chances of survival and multiplying. In the context of a stochastic problem solving process, we have a collection of candidate solutions. Their quality (that is how well they solve the problem) determines the chance that they will be kept and used as seeds for constructing further candidate solutions (Table 4.1). This phenomenon is also known as the **survival of the fittest**.

Table 4.1: The basic evolutionary computing metaphor linking natural evolution to problem solving (source [26]).

Evolution		**Problem solving**
Environment	$\longrightarrow$	Problem
Individual	$\longrightarrow$	Candidate solution
Fitness	$\longrightarrow$	Quality

4.2 Brief History

Surprisingly enough, the idea of applying Darwinian principles to automated problem solving dates back to the forties, long before the breakthrough of computers [29]. As early as 1948, Turing proposed "genetical or evolutionary search", and by 1962 Bremermann had actually executed computer experiments on "optimizing through evolution and recombination". During the 1960s, three different implementations of the basic idea were developed in different places. In the USA, Fogel, Owens, and Walsh introduced *evolutionary programming* [30], while Holland called his method a *genetic algorithm* [43, 44]. Meanwhile, in Germany, Rechenberg and Schwefel invented *evolution strategies* [66, 72]. For about 15 years, these areas were developing separately; but since the early 1990s they have been viewed as different representatives of one technology that has come to be known as *evolutionary computing* [4, 5]. In the early 1990s a fourth stream following the general ideas emerged. It was called the *genetic programming* [52, 53] and was introduced by Koza. The contemporary terminology denotes the whole field by evolutionary computing, the algorithms evolved are termed as *evolutionary algorithms*, and it considers evolutionary programming, evolution strategies, genetic algorithms, and genetic programming as sub-areas belonging to the corresponding algorithm variants.

Computation in the second half of the 20th century has created a rapidly growing demand for problem solving automation. The growth rate of the research and development capacity has not kept pace with these needs. Hence, the time available for thorough problem analysis and tailored algorithm design has been, and still is, decreasing. A parallel trend has been the increase in the complexity of problems to be solved. These two trends, and the constraint of limited capacity, imply an urgent need for robust algorithms with satisfactory performance. That is, there is a need for algorithms that are applicable to a wide range of problems, do not need much tailoring for specific problems, and deliver good (not necessarily optimal but acceptable) solutions within acceptable time. Evolutionary algorithms do all this, and therefore, provide an answer to the challenge of deploying automated solution methods for more and more problems, which are more and more complex, in less and less time.

4.3 Main Components of Evolutionary Algorithms

As the history of the field indicates, many different variants of evolutionary algorithms exist. The common underlying idea behind all these variants is the same: given a population of individuals within some environment that has limited resources, competition for those resources caused by natural selection and the survival of the fittest. During evolution, this is reflected by continuous improvement of the fitness in the population.

There are two fundamental forces that form the basis of evolutionary systems:

- Variation operators (*recombination* and *mutation*) create necessary diversity within the population, and thereby facilitate novelty.
- Selection acts as a force increasing the mean quality of solutions in the population.

It should be noted that many components of the evolutionary process are *stochastic*. It means that randomness plays a considerable role here. Thus, although during selection fitter individuals have a higher chance of being selected as parents than the less fit ones, there is a chance that even the weak individuals can become parents or can survive. During the recombination process, the choice of which pieces from the parents will be recombined is made at random. Similarly for mutation, the choice of which pieces will be changed within a candidate solution and of the new pieces to replace them is made randomly.

The general pseudo-code of an evolutionary algorithm is given in Algorithm 1.

Algorithm 1: The pseudo-code of EA.

```
Function EA:
    INITIALIZE population with random candidate solutions;
    EVALUATE each candidate;
    while termination condition is not satisfied do
        SELECT parents;
        RECOMBINE pairs of parents;
        MUTATE the resulting offspring;
        EVALUATE new candidates;
        SELECT individuals for the next generation;
    end
```

The various dialects of EAs that were mentioned previously all follow these general steps, differing only in technical details. In particular, the representation of a candidate solution is often used to characterize different streams. Typically the representation (i.e. the data structure encoding a candidate solution) has the form of: string over a finite alphabet in genetic algorithms, real-values vectors in evolution strategies, finite state machines in classical evolutionary programming, and trees in genetic programing. The origin of these differences is mainly historical. Technically, one representation might be preferable to others if it matches the given problem better; that is, it makes the encoding of candidate solutions easier or more natural. For instance, when solving a satisfiability problem with n logical variables, the straightforward choice is to use the bit-strings of length n, hence the appropriate EA would be the genetic algorithm. To evolve a computer program that can play checkers, trees are well suited, thus a genetic programing approach is likely. It is important to note that the recombination and mutation operators must match the given representation. Thus, for instance, in genetic programing the recombination operator works on trees, while in GAs it operates on strings. In contrast to variation operators, the selection process only takes fitness information into account, and so it works independently from the choice of representation. Therefore, differences between the selection mechanism commonly applied in each stream are a matter of tradition rather than of technical necessity.

The most important components of EAs are:

- Representation (definition of individuals).
- Evaluation function (or fitness function).
- Population.
- Parent selection mechanism.
- Variation operators, recombination and mutation.
- Survivor selection mechanism (replacement).

To create a complete algorithm it is necessary to specify each of these components and to define the initialization procedure and a termination condition.

4.3.1 Representation (Definition of Individuals)

The first step in defining an EA is to link the ”real world” to the ”EA world” that is, to set up a bridge between the original problem context and the problem-solving space where evolution takes place. Objects forming possible solutions within the original problem context are referred to as *phenotypes*, while their encoding that is, individuals within the EA, are called *genotypes*. This first design step is commonly called *representation*, as it amounts to specifying a mapping from the phenotypes onto a set of genotypes that are said to represent them. Within the EC literature, many synonyms can be found for naming the elements of these two spaces:

- On the side of the original problem context, the terms *candidate solution*, *phenotype*, and *individual* are all used to denote points in the space of possible solutions. This space itself is commonly called the *phenotype space*.
- On the side of the EA, the terms *genotype*, *chromosome*, and again *individual* are used to denote points in the space where the evolutionary search actually takes place. This space is often called the *genotype space*.

4.3.2 Evaluation Function (Fitness Function)

The role of the *evaluation function* is to represent the requirements the population should adapt to. It forms the basis for selection, so it facilitates improvements. More accurately, it defines what improvement means. From the problem-solving perspective, it represents the task to be solved in the evolutionary context. Technically, it is a function or procedure that assigns a quality measure to genotypes.

The evaluation function is commonly called the *fitness function* in EC. Quite often, the original problem to be solved by an EA is an optimization problem. In this case, the name *objective function* is often used in the original problem context.

4.3.3 Population

The role of the *population* is to hold (the representation of) possible solutions. A population is a multi-set of genotypes. The population forms the unit of evolution. Individuals are static objects that do not change or adapt, it is the population that does. Given a representation, defining a population may be as simple as specifying how many individuals are in it, that is, setting the population size.

In contrast to variation operators that act on one or more parent individuals, the selection operators (parent selection and survivor selection) work at the population level. In general, they take the whole current population into account, and choices are always made relative to what is currently present. For instance, the best individual of a given population is chosen to seed the next generation, or the worst individual of a given population is chosen to be replaced by a new one. In almost all EA applications the population size is constant and does not change during the evolutionary search.

The *diversity* of a population is a measure of the number of different solutions present. No single measure for diversity exists. Typically people might refer to the number of different fitness values present, the number of different phenotypes present, or the number of different genotypes.

4.3.4 Parent Selection Mechanism

The role of *parent selection* or *mating selection* is to distinguish among individuals based on their quality and in particular, to allow the better individuals to become parents for the next generation. An individual is a *parent*, if it has been selected to undergo variation in order to create offspring.

Together with the survivor selection mechanism, parent selection is responsible for pushing quality improvements. In EC, parent selection is typically probabilistic. Thus, high-quality individuals have more chance of becoming parents than those with low quality. Nevertheless, low-quality individuals are often given a small but positive chance, otherwise, the whole search could become too greedy and get stuck in a local optimum.

4.3.5 Variation Operators

The role of *variation operators* is to create new individuals from the old ones. In the corresponding phenotype space this amounts to generating new candidate solutions. Variation operators in EC are divided into two types based on their arity (number of objects they take as inputs).

1. **Mutation**. A unary variation operator is commonly called *mutation*. It is applied to one genotype and delivers a modified mutant, the *child* or *offspring*. A mutation operator is always stochastic: its output - the child - depends on the outcomes of a series of random choices. In general, mutation is supposed to cause a random, unbiased change. The role of mutation has historically been different in various EC dialects. Thus, for instance, in genetic programming it is often not used at all, whereas in genetic algorithms it has traditionally been seen as a background operator used to fill the gene pool with ”fresh blood”, and in evolutionary programming it is the sole variation operator responsible for the whole search work. It is worth noting that variation operators form the evolutionary implementation of elementary steps within the search space. Generating a child amounts to stepping to a new point in this space. From this perspective, mutation has a theoretical role as well: it can guarantee that the space is connected. There are theorems which state that an EA will (given sufficient time) discover the global optimum of a given problem. The simplest way to satisfy this condition is to allow the mutation operator to ”jump” everywhere.

2. **Recombination**. A binary variation operator is called *recombination* or *crossover*. As the names indicate, such an operator merges information from two parent genotypes into one or two offspring genotypes. Like mutation, recombination is a stochastic operator: the choices of what parts of each parent are combined and how this is done depend on random drawings. Again, the role of recombination differs between EC dialects: in genetic programming it is often the only variation operator, and in genetic algorithms it is seen as the main search operator, whereas in evolutionary programming it is never used. The principle behind recombination is simple - by mating two individuals with different but desirable features, we can produce an offspring that combines both of those features. Evolutionary algorithms create a number of offspring by random recombination, and we hope that while some will have undesirable combination of traits and most may be no better or worse than their parents, some will have improved characteristics. Recombination operators in EAs are usually applied probabilistically, that is with a nonzero chance of not being performed.

4.3.6 Survivor Selection Mechanism (Replacement)

The role of *survivor selection* or *environmental selection* is to distinguish among individuals based on their quality. In that, it is similar to parent selection, but it is used in a different stage of the evolutionary cycle. A choice has to be made about which individuals will be allowed to the next generation. This decision is often based on their fitness values favoring those with higher quality, although the concept of age is also frequently used. In contrast to the parent selection, which is typically stochastic, survivor selection is often deterministic and it is called *replacement* or the replacement strategy.

4.3.7 Initialization

Initialization is kept simple in most EA applications, the first population is seeded by randomly generated individuals. In principle, problem-specific heuristics can be used in this step to create an initial population with higher fitness. Whether this is worth the extra computational effort or not depends very much on the application at hand (see Section 2.5 in [26]).

4.3.8 Termination Condition

We can distinguish two cases of a suitable *termination condition*. If the problem has a known optimal fitness level, probably coming from a known optimum of the given objective function, then in an ideal world our stopping condition would be the discovery of a solution with this fitness value. However, EAs are stochastic and mostly there are no guarantees of reaching such an optimum, so this condition might never become satisfied and the algorithm may never stop. Therefore, we must extend this condition with one that certainly stops the algorithm. The following options are commonly used for this purpose:

1. The maximally allowed CPU time elapses.
2. The total number of fitness evaluations reaches a given limit.
3. The fitness improvement remains under a threshold value for a given period of time (i.e., for a number of generations or fitness evaluations).
4. The population diversity drops under a given threshold.

Technically, the actual termination criterion in such cases is a disjunction: an optimum value hit *or* a condition x satisfied.

4.4 Evolutionary Computing and Global Optimization

Let us illustrate the power of the evolutionary approach to automated problem solving by a number of application examples from various areas. To position these and other applications, let us sketch a system analysis perspective to problems. From this perspective, we identify three main components of a working system: inputs, outputs, and the internal model considering these two.

- In an *optimization* problem the model is known together with the desired output (or its description), and the task is to find the input(s) leading to this output.
- In a *modeling* or *system identification* problem, corresponding sets of inputs and outputs are known and a model of the system is sought that delivers the correct output for each known input.
- In a *simulation* problem we know the system model and some inputs and we need to compute the outputs corresponding to these inputs.

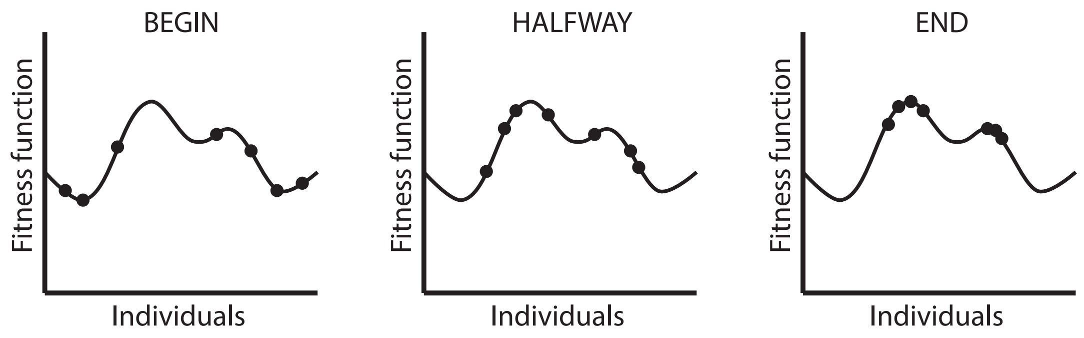


Figure 4.1: Typical progress of an EA illustrated in terms of population distribution. Source: [26].

Figure 4.1 shows three main stages of the evolutionary search for suitable candidate solutions of the above-mentioned problems, showing how the candidates might typically be distributed in the beginning, somewhere halfway, and at the end of the evolution.

In the first phase, directly after initialization, the individuals are randomly spread over the whole search space (Figure 4.1, left). After only a few generations this distribution changes: because of selection and variation operators, the population abandons low-fitness regions and starts to ”climb” the hills (Figure 4.1, middle). Later, the whole population is concentrated around a few peeks, some of which may be suboptimal (Figure 4.1, right). In principle, it is possible that the population might climb the ”wrong” hill, leaving all of the individuals positioned around a local but not global optimum.

As the Figure 4.2 indicates, the progress in terms of fitness increase in the first half of the evolution run (X) is significantly greater than in the second half (Y). This provides a general suggestion that it might not be worth allowing very long runs as they unlikely result in a better solution quality.

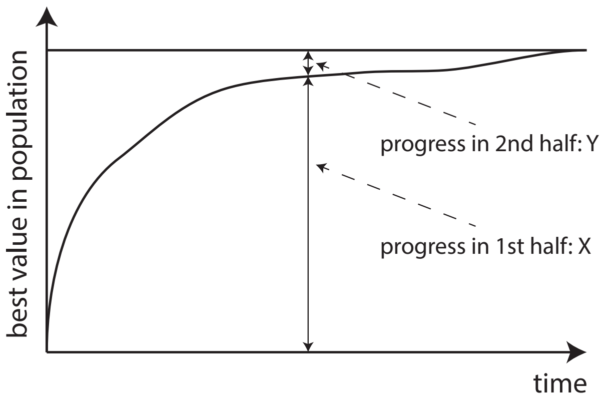


Figure 4.2: Illustration of why long runs might not be worth performing. Source: [26].

Although there is no universally accepted definition of what the terms mean, these distinct phases of the search process are often categorized in terms of *exploration* (the generation of new individuals in as yet untested regions of the search space), and *exploitation* (the concentration of the search in the vicinity of known good solutions). Evolutionary search processes are often referred to in terms of a trade-off between exploration and exploitation. Too much of the former can lead to inefficient search and too much of the latter can lead to a propensity to focus the search too quickly (see [26] for a good discussion of these issues).

In the further text of this section we will focus more on the optimization problem. Of course, EAs are not the only optimization technique known so let us explain where EAs fall into the general class of optimization methods, and why they are of increasing interest.

In [74], the problem of optimization is formulated as the problem of searching a global extreme of the function:

$$f : A \rightarrow \mathbb{R}, \tag{4.1}$$

where the domain is the Cartesian product of closed, often variously restricted intervals $[a_i, b_i]$ usually in the real, integer or binary domain. If the objective is to find a *global minimum* (minimization of an argument), then such $x^* \in A$ should be find that for each $x \in A$, $f(x^*) \leq f(x)$. If the objective is maximization of an argument, such $x^* \in A$ should be find that for each $x \in A$, $f(x^*) \geq f(x)$. The members of A are called candidate solutions and A represents a search space of candidate solutions. Function f is called the cost function, it does not have to be continuous, to have a derivation or to be fully defined. It represents a black box that can always return a value for a given argument. Because f is not convex usually, it is appropriate to define a *local extreme*. The local minimum is such x^- that for every $x \in A$ in the neighborhood of x^- defined by the parameter δ, $f(x^-) \leq f(x)$. The local maximum is defined in the similar way.

A number of *deterministic* algorithms exist that, if allowed to run to completion, are guaranteed to find x^*. The simplest example is, of course, complete enumeration of all the solutions in A, which can take an exponentially long time as the number of variables increases.

Another class of search methods is known as *heuristics*. These may be thought of as sets of rules for deciding which potential solution out of A should next be generated and tested. If randomization is extensively utilized in the selection, these algorithms are called the *heuristic stochastic* algorithms. The examples of them are random search, simulated annealing, hill climbing, swarm algorithms, as well as evolutionary algorithms. More informations about them can be found in [11],[76].

Random search algorithm generates a candidate solution $a \in A$ randomly in each step. This algorithm remembers the candidate solution a, if its cost $f(a)$ is better than the cost of the best up to now solution. The computation ends when a desired solution is found or when the limit of iterations is reached. The Algorithm 2 demonstrates the pseudo-code of the random search that aims at finding the maximum cost (the maximum of function f).

Algorithm 2: The pseudo-code of the random search.

```
Function RANDOM SEARCH:
    INITIALIZE the best cost with minimal value;
    while termination condition is not satisfied do
        GENERATE a new random candidate solution $a$;
        EVALUATE the cost $f(a)$;
        if the cost $f(a) >$ the best cost then
            UPDATE the best cost by $f(a)$ and SAVE the candidate $a$;
        end
    end
```

Because of the stochastic nature of this algorithm, it is necessary to run the random search several times with a different seed of the pseudo-number generator in order to gain statistically important data. Nevertheless, this algorithm is weak for solving real world problems, because it lacks strategy and does not exploit the knowledge gained during the computation.

Local search algorithms (simulated annealing, hill climbing, greedy) are iterative algorithms that start with an arbitrary candidate solution $a \in A$ to a problem, then attempt to find a better candidate solution in the neighborhood of a by the local exploration. In order to evaluate the cost of candidate solutions in the neighborhood of a, it is necessary to precisely define the neighborhood $U(a)$ using the so-called *cardinality constant* C, $C = |U(a)|$. In the next step, a candidate solution b with the best cost $f(b)$ is selected from $U(a)$ and serves as a starting point in the next step of the algorithm. The resulting pseudo-code of the local search is specified in Algorithm 3 (the hill climbing algorithm):

Algorithm 3: The pseudo-code of the local search.

```
Function LOCAL SEARCH:
    GENERATE a new random candidate solution $a$;
    EVALUATE the cost of $f(a)$;
    INITIALIZE the best cost with the cost $f(a)$;
    while termination condition is not satisfied do
        GENERATE neighbors of the candidate $a$;
        EVALUATE the cost of all neighbors;
        SELECT the the best candidate $b$ from the neighborhood $U(a)$ according to the
        cost;
        if the cost of $f(b) >$ the best cost then
            UPDATE the best cost by $f(b)$;
            REPLACE the candidate $a$ by $b$ ($a := b$)
        end
    end
```

The chance that algorithm succeeds grows with the growing cardinality C. However, also the computational overhead is bigger. The disadvantage of this algorithm is that it often reaches a local extreme and the best possible solution is not found.

There are numerous features of EAs that distinguish them from local search and random search algorithms, relating principally to their use of population. The population provides the algorithm with a means of defining nonuniform *probability distribution function* (p.d.f.) governing the generation of new candidate solutions from A. This p.d.f. function reflects possible interactions between points in A which are currently represented in the population. The interactions arise from the recombination of partial solutions from two or more members of the population (parents). This potentially complex p.d.f. contrasts with the globally uniform distribution function of blind random search, and the locally uniform distribution used by many other stochastic algorithms such as simulated annealing and various hill-climbing algorithms.

According to [73], any search strategy can be viewed as utilizing one or more operators which produce new candidate solutions from those previously visited in the search space. Effective search space algorithms should balance between two apparently antagonistic goals:

1. to exploit the neighborhood of the best up to now solution,
2. to explore also uninspected areas of the search space.

While the local search algorithms like hill climbing focus mainly on the neighborhood of the best available solution, the random search moves through the whole state space but without exploiting promising areas of the search space. From this simple comparison, the evolutionary algorithms seem to be the best choice as they can meet both goals simultaneously. The ability of EAs to maintain a diverse set of points provides not only a means of escaping from local optima by mutation, but also a means of coping with large and discontinuous search spaces.

But each search space is different and even identical spaces can appear very different under different representations and evaluation functions. So there is no way to choose a single search method that can serve well in all cases [57].

This can be proven mathematically. The so-called No Free Lunch (NFL) theorem states that if a sufficiently inclusive class of problems is considered, no search method can outperform an enumeration [91]. A more general variant of the NFL theorem [79] shows that over the ensemble of all representations of one space with another, all algorithms perform identically on any reasonable measure of performance. This is a fundamental limitation of any search method. However, there are many possible improvements from a practical point of view. Primarily, it is the *domain knowledge* that has to be employed in some way in the process of effective problem-specific algorithm design.

4.5 Genetic Algorithms

Genetic Algorithm (GA) is the most widely known type of EA. A special attention is devoted to GA in this thesis, because it will be further used for solving optimization problems.

Representation of Individuals. GA uses mostly the binary representation, so the genotype consists of a string of binary digits - a bit string. For a particular application we have to decide how long the string should be and how we will interpret it to produce a phenotype. In choosing the genotype-phenotype mapping for a specific problem, one has to make sure that the encoding allows that all possible bit strings denote a valid solution to the given problem and that vice-versa, all possible solutions can be represented. Other types of representations are: integer, floating-point or permutation representation.

Mutation. For binary representations, the most common mutation operator allows each bit to flip (i.e., from 1 to 0 or 0 to 1) with a small probability p_m. This type of mutation is also called *uniform*. The actual number of values changed is thus not fixed, but depends on the sequence of random values drawn, so for encoding of length L, on average $L.p_m$ values will be changed. Other type of mutation operator is *nonuniform* mutation, where e.g. a Gaussian probability distribution on different bits is used or the *swap* mutation that randomly picks two positions in the string and swap their values.

Recombination. Recombination is considered to be one of the most important features in GAs. It is commonly applied probabilistically with a crossover rate p_c, which value is usually in the range [0.5, 1.0]. Mostly two parents are selected and then a random variable is drawn from [0,1) and compared to p_c. If the value is lower, two offspring are created via recombination of two parents; otherwise they are created asexually, i.e., by copying the parents. Three standard forms of recombination are used for binary representations. The *one-point* crossover generates randomly a position in the bit string in which both parents exchange their tails. The *N-point* crossover generates n positions, both parents are broken in these positions and then the offspring are created by taking alternative segments from the two parents. The *uniform* crossover treats each gene (an element of chromosome) separately and makes a random choice about exchange.

Population Models. Two different population models are typically used in GA: the *generational* model and the *steady-state* model. In the generational model, we begin with a population of size μ in each generation, from which a mating pool of μ parents is selected. Next, $\lambda(=\mu)$ offspring are created from the mating pool by the application of variation operators, and evaluated. After each generation, the whole population is replaced by its offspring, which is called the "next generation". In the steady-state model, the entire population is not changed at once, but rather a part of it. In this case, $\lambda(<\mu)$ old individuals are replaced by λ new ones, the offspring. The percentage of the population that is replaced is called the *generation gap*, and is equal to λ/μ.

Parent Selection. In the *fitness proportional* selection, the probability that an individual f_i is selected for mating is $fi/\Sigma_{j=1}^{\mu} f_j$. It means that the selection probability depends on the *absolute* fitness value of the individual compared to the *absolute* fitness values of the rest of the population. The *rank-based* selection sorts the population on the basis of fitness and then allocates selection probabilities to individuals according to their rank, rather than according to their actual fitness values. The mapping from rank number to selection probability is arbitrary and can be done for example, linearly or exponentially decreasing. But the sum over the population of the probabilities must be unified.

The mating pool of parents is sampled from the selection probability distribution, but in general, it will not accurately reflect it. The simplest way of achieving this sampling is known as the *roulette wheel* algorithm. Conceptually this is the same as spinning a one-armed roulette wheel, where the sizes of the holes reflect the selection probabilities. A random number is then generated uniformly from [0,1]. This random number fits to some hole in the wheel and thus identifies the corresponding parent. The *tournament selection* algorithm does not require any global knowledge of the population. It randomly picks k individuals and selects the best one from them according to their fitness values.

Survivor Selection. From a set of μ parents and λ offspring it is necessary to produce a set of μ individuals for the next generation. The selection is usually made according to the age of individuals (each individual exists in the population for the same number of iterations) or their fitness. Very common scenarios are following: all parents are replaced by offspring, the best parents according to their fitness values remain in the population and others are replaced by offspring, or just one best parent is propagated to the next population (*elitism*).

4.6 Parameter Control in Genetic Algorithms

A simple GA might be defined by stating it will use binary representation, uniform crossover, bit-flip mutation, tournament selection, and generational replacement. For a full specification, however, further details have to be specified, for instance, the population size, the probability of mutation p_m, the probability of crossover p_c, and the tournament size. These data - called the strategy parameters - complete the definition of EA and are necessary to produce an executable version. The values of these parameters greatly determine whether the algorithm will find an optimal or near-optimal solution and whether it will find such a solution effectively.

Globally, we distinguish two major forms of setting parameter values: **parameter tuning** and **parameter control**. By parameter tuning we mean the commonly practiced approach that amounts to finding good values for the parameters before the run of the algorithm and then running the algorithm using these values, which remain fixed during the run. The parameter control is an alternative which starts a run with initial parameter values that are changed during the run.

Parameter tuning is a typical approach to algorithm design. Such tuning is done by experimenting with different values and selecting those that give the best results on the test problems at hand. However the number of possible parameters and their different values mean that this is a very time-consuming activity.

The technical drawbacks to parameter tuning based on experimentation can be summarized as follows:

- Parameters are not independent, but trying all different combinations systematically is practically impossible.
- The process of parameter tuning is time consuming, even if parameters are optimized one by one, regardless of their interactions.
- For a given problem, the selected parameter values are not necessarily optimal, even if the effort made for setting them was significant.

During the history of EAs, considerable effort has been devoted to finding parameter values (for a given type of EA, such as GAs) that were good for a number of test problems [**?**]. Unfortunately, it was shown that specific problems (problem types) require specific setup for satisfactory performance. There are also theoretical arguments that any quest for generally good EA, thus generally good parameter settings, is lost a priori, c.f. the discussion of the No Free Lunch theorem in section 9.

Another drawback of the parameter tuning approach is how we define it: finding good values for the parameters before the run of the algorithm and then running the algorithm using these values, which remain fixed during the run. However, a run of an EA is a dynamic, adaptive process. The use of rigid parameters that do not change their values is thus in contrast to this spirit. Additionally, it is intuitively obvious, and has been empirically and theoretically demonstrated, that different values of parameters might be optimal at different stages of the evolutionary process [3, 23, 42].

4.7 Chapter Overview

In this chapter, the evolutionary computing and especially the genetic algorithm were described. The reason is that GA is used as the main optimization tool in the techniques proposed in this thesis and for their proper working it is important to understand the meaning of all the GA parameters and their settings.

Chapter 5

Functional Verification State-of-the-art and Related Work

To isolate the topic of the Ph.D. thesis, the international conferences, published work and literature contributing to the state-of-the-art of functional verification are examined and appraised. From these, some verification topics are selected and the related work connected to them is analyzed in more details.

5.1 Conferences and Published Work

Several conferences that are devoted to or at least contain special sessions on verification and validation currently exist. According to our knowledge, the following belong to the most world-wide recognized: *Haifa Verification Conference* (HVC), *Microprocessor Test and Verification* conference (MTV), *International Conference on Computer Aided Verification* (CAV), *Design Automation Conference* (DAC), and *Design, Automation and Test in Europe* conference (DATE).

Haifa Verification Conference. HVC is a conference organized by the IBM Research Lab in Haifa, Israel and is dedicated to the state-of the art and the state-of-the-practice in verification and testing. The conference provides a forum for researchers and practitioners from academia and industry to share their work, exchange ideas, and discuss the future directions of testing and verification for hardware, software, and complex hybrid systems. The common underlying goal of these techniques is to ensure the correct functionality and performance of complex systems.

The main topics that HVC targets in the last few years are formal verification, mainly the model checking methods, their improvement and modifications for verifying more complex systems, generation of tests for improving code and functional coverage, fast error localization, and System-on-Chip (SoC) verification, which is currently the main challenge in the industrial domain as SoCs are becoming more and more complex.

Microprocessor Test and Verification Conference. The purpose of MTV conference is to bring researchers and practitioners from the fields of verification and test together to exchange innovative ideas and to develop new methodologies to solve the difficult challenges in various processor and SoC design environments. In the past few years, some work has been done on exploiting techniques from the test area to solve problems in the verification area and vice versa.

The main areas of interest include validation of microprocessors and SoCs, performance testing, high-level test generation for functional verification and best practices, acceleration and emulation techniques, silicon debugging (including the JTAG debugging), low-power verification, formal techniques and their applications (e.g. pipeline verification), coverage improvements, equivalence checking, timing verification techniques, design for testability and verifiability, security verification, regression testing.

International Conference on Computer Aided Verification. CAV conference is dedicated to the advancements of the theory and practice of computer-aided formal analysis methods for hardware and software systems. The conference covers the spectrum from theoretical results to concrete applications, with the emphasis on practical verification tools and the algorithms and techniques that are needed for their implementation. CAV considers it vital to continue spurring advances in hardware and software verification while expanding to new domains such as biological systems and computer security.

At this conference, the main topics are: algorithms and tools for verifying models and implementations, hardware verification techniques, abstraction techniques, program analysis and software verification, decision procedures and solvers for verification, mathematical and logical foundations of practical verification tools, algorithms and tools for system synthesis, or formal models and methods for biological systems.

Design Automation Conference. DAC is recognized as the premier conference for design and automation of electronic systems. Members are from a diverse worldwide community of more than 1,000 organizations that attend each year, represented by system designers and architects, logic and circuit designers, verification engineers, CAD managers, senior managers and executives, and researchers and academicians from leading universities. Close to 300 technical presentations and sessions selected by a committee of electronic design experts offer information on recent developments and trends, management practices and new products, methodologies and technologies. A highlight of DAC is its exhibition and suite area with approximately 200 of the leading and emerging companies in Electronic Design Automation (EDA), Intellectual Property (IP), Embedded Systems and Software, Automotive Systems and Software, Design Services, and Security. The conference is sponsored by the Association for Computing Machinery (ACM), the Electronic Design Automation Consortium (EDA Consortium), and the Institute of Electrical and Electronics Engineers (IEEE), and is supported by ACM Special Interest Group on Design Automation (SIGDA).

Regarding verification, the main topics in 2015 included: low power IPs and verification, optimization of regression flows, requirement-driven verification, re-usability of verification testbenches, acceleration of simulation, SoC verification, post-silicon diagnosis, trends and challenges in functional verification (multiple clock and power domains, security, hardware-software interaction), analog-mixed signal verification, verification in SystemC.

Design, Automation and Test in Europe Conference. The DATE conference addresses all aspects of research into technologies for electronic and (embedded) systems engineering. It covers the design process, verification and test, and tools for design automation of electronic products ranging from integrated circuits to distributed large-scale systems. This includes both hardware and embedded software design issues. The conference scope also includes the elaboration of design requirements and new architectures for challenging application fields such as telecommunication, wireless communications, multimedia, health-care and automotive systems. Persons involved in innovative industrial designs are particularly encouraged to submit papers to foster the feedback from design to research. Panels, hot-topic sessions and embedded tutorials highlight and inform about emerging topics.

A special track of this conference called Design, Methods and Tools contains two verification areas: simulation-based verification and formal verification and specification techniques. The simulation-based verification covers acceleration-driven and emulation-based validation, post-silicon verification, on-line checkers and runtime verification at different levels, diagnosing and debugging solutions, automated assertions and monitor generation for verification, multi-domain simulation techniques, validation of cyber-physical systems, SoCs and emerging architectures. The formal verification and specification techniques cover equivalence checking, model checking, symbolic simulation, theorem proving, abstraction and decomposition techniques, technologies supporting formal verification, semi-formal verification techniques, formal verification of IPs, SoCs, cores, real-time and embedded systems, integration of verification into design flows, challenges of multi-cores, both as verification targets and as verification host platforms.

Verification Horizons. The Verification Horizons newsletter can be considered as a very relevant publication that reflects the state-of-the art in the functional verification domain. It is issued three times per year by the Mentor Graphics company and discusses the ad-hoc topics in many verification areas. When examining the 2015 printouts, the topics are devoted to the verification of airborne and automotive electronics with respect to the safety standards DO-254 and ISO 26262, improving localization of errors with assertions, reuse of UVM verification environments and verification IPs, hardware emulation and acceleration of RTL simulation, regression testing, power-aware verification, or coverage improvements.

5.2 Related Work

From the topics of the relevant conferences, several topics were identified that form the related area of this Ph.D. thesis. They are described in the following text.

5.2.1 Acceleration of RTL Simulation using FPGA Accelerators and Emulators

Simulation-based pre-silicon verification approaches, including functional verification, provide verifiers and designers with a great opportunity to inspect the internal behavior of a running system and comfort while debugging a failing component, checking assertions or performing coverage analysis. The values of internal signals are easily observable with arbitrary depth of history and it is easy to introduce precisely timed events to the verified component. In general, pre-silicon verification approaches have improved significantly during the last decade and a lot of new techniques, tools and verification methodologies have been developed. One of the negative consequences of Moore‘s law, which claims that the number of transistors on integrated circuits doubles approximately every two years, is that it is necessary to verify more and more complex systems. However, as the performance of computer processor cores has reached its limits (and the overall performance of computer processors is currently increased mainly by placing multiple cores in a single chip), software simulators of logical circuits cannot benefit from this increase much because the simulation task is difficult to parallelize.

Because of this limit, even with a high effort devoted to the pre-silicon verification, some previously uncovered functional errors can be recognized only after the system is manufactured. The reasons why these errors had not been found in the pre-silicon stage are, e.g. because the errors may appear after several hours of operation of the target device (a condition which may take years or even centuries to reproduce in a simulator), or because the errors might have been introduced by the synthesizer or caused by the discrepancy between the behavior of a resource and the behavior of its simulation model.

In order to eliminate as many remaining bugs as possible before the target device is fabricated, verification is currently applied even in the post-silicon phase of the development cycle, in the real hardware environment. Unfortunately, not many techniques standard for simulation can be directly used in post-silicon verification, as these techniques heavily rely on perfect observability of internal signals of the system, while in post-silicon, the observation of a system is often limited to the use of logic analyzers, oscilloscopes, etc.

In recent years, several approaches that addressed the issue of pre-silicon verification performance appeared. The first approach discussed in [41, 50, 51] translates VHDL or Verilog testbenches, which contain not directly synthesizable behavioral constructs, using advanced synthesis techniques into the synthesizable subset of the corresponding language. Note that these techniques are limited since some of the non-synthesizable constructs, such as reading from a file or evaluation of recursive functions, still cannot be synthesized. With the advent of higher-level HVLs for writing testbenches, with SystemVerilog being the most prominent, automatic synthesis of testbenches that use advanced features, such as constrained-random stimulus generation, coverage-driven and assertion-based verification, has become even more infeasible. However, soon after the introduction of HVLs, several transaction-based methodologies emerged, e.g. SystemVerilog-based VMM, OVM, and UVM. These methodologies use higher-level of abstraction and group sequences of stimuli applied to the DUV into transactions. Transactions are sent to drivers that decode them and apply proper stimuli to the DUV. Since drivers can usually be written using synthesizable constructs, it is possible to locate them in a hardware accelerator. In the further text, four categories of hardware accelerators are mentioned, their overview is taken from [59].

The first category of hardware accelerators is built from small custom processors mapped on FPGA devices that are interconnected by other FPGA devices. Each FPGA processor executes a small portion of the total simulation of DUV. All the processors execute in parallel, thereby increasing the speed of simulation. These accelerators run much faster than RTL software simulators. Speeds of 75.000 to 100.000 clocks per cycle are possible. But it is still not the maximum speed of the real product, because it is limited by the FPGA technology and complicated interconnections. Nevertheless, accelerators usually retain all the debugging capabilities, including output of the waveform. However, they are much more expensive than software simulators (10 - 20 x). When considering accelerating verification, the FPGA accelerator is utilized for running DUV and a synthesizable part of the verification testbench. The remaining part of the testbench runs in the simulation environment on a CPU. Both parts then communicate through simple channels. However, depending on the speed differences between the software simulator and the hardware accelerator, the accelerator will typically be waiting for the software part of the testbench to execute, and therefore, the entire process will run only as fast as the software simulator.

An example of such accelerator is Palladium from the Cadence company [12]. A similar approach was introduced by Huang *et al* [46]; their proposal is to place the verified system with the necessary verification components in an FPGA, and in addition, to provide limited observability of the DUV signals. Nevertheless, to the best of our knowledge, there is currently still no available working implementation based on their proposal.

To further increase the speed of accelerators, some companies have used fast ASIC processors to execute the simulation. The processors are highly optimized and run at very high speed, and there are hundreds or thousands of them in a grid. These accelerators have much higher speed, usually 100.000 to 500.000 clocks per cycle, but this speed comes at tremendous costs (millions of dollars). When connecting to the software part of the verification testbench there are the same limitation as in the previous category.

The third category of accelerators are hardware emulators based on large FPGAs that allow to observe the behavior of the verified system in a real environment. An emulator builds on the flexibility of the FPGA device and takes the reconfigurability of the FPGA one step further from a single chip to an entire system. The emulator uses a number of circuit boards containing multiple FPGA devices, where all the devices configurations are used to implement the entire system - some are used for implementing the logic, the memory, signal routing and others implement the debug circuitry and signal visibility mechanisms. The emulator uses special software to synthesize the HDL design to logic gates. This architecture runs fast because all the logic is executed in parallel and if the design fits into the emulator properly, the speed of the real environment can be achieved. Full visibility into the design is available so the designer can examine any signal at any time. But of course, these emulators are very expensive and getting a design into an emulator is very challenging task, mainly because of design partitioning and compilation. They are provided by major companies that focus on tools for hardware simulation and verification. The examples of such emulators are Mentor Graphics' Veloce2 technology [35] and Cadence's TBA [13]. Nevertheless, utilizing some HVL testbenches in emulation is not typical.

The last category is FPGA prototyping. An FPGA prototype uses multiple FPGA devices to implement the design. The design is automatically or manually partitioned into blocks. These blocks are mapped using standard FPGA design software into standard FPGA devices. This approach differs from that of the emulator in the fact that the designers usually build a custom board to interconnect the FPGA devices instead of using FPGA interconnect devices. FPGA prototypes can approach the speed of the real device, they can be tested with a huge amount of data from the real environment. They are relatively inexpensive, depending on the number of the FPGA devices used. The disadvantages of using this approach is difficult partitioning of the design and the lack of internal visibility (only capturing data into the on-chip RAM and reading them through the JTAG interface is typical).

5.2.2 Automated Generation and Reuse of Verification Environments and IPs

The current embedded systems, such as Systems-on-Chip (SoCs), Multi-Processor Systems-on-Chip (MPSoCs) or equipment for Internet of Things (IoT), are more and more complex. They usually consist of one or more processors (either General Purpose Processors (GPPs) or Application Specific Instruction-set Processors (ASIPs)), controllers (e.g.m DMA), buses, bridges between two buses and various types of peripherals. Verification of their functionality is quite difficult because most of the verification approaches like formal verification, assertion-based verification or functional verification work well at the unit level, but for SoCs, they do not scale well. The reason is not only in the complexity of these systems but also the fact that software embedded into processors must be often taken into account [24, 6].

Therefore, because of the complexity, it seems to be the current state-of-the-art in SoCs verification to come with a verification solution that is adjusted to SoCs (digital vs. analogy, verification IPs, graph-based IP connections, etc.), or their application domain (e.g. multimedia, DSP applications, smart devices, etc.) and is often connected to the development tool of these systems. For example Breker [10] introduces a graph-based approach to functional verification. With graphs, users capture the IP level scenarios as nodes and connections make the SoC level scenario. Cadence [14], Synopsys [82] or Mentor Graphics [36] provide verification IPs for more than 40 communication protocols and 60 memory interfaces in order to facilitate SoC verification. These verification IPs can be connected to the existing verification environments and they focus on checking functionality of specific bus connections in the DUV. Duolog [25] focuses on IP integration problems and generates automatically OVM/UVM verification environments from interface-based executable specification.

When focusing on automated generation of verification environments in general (not necessary only for SoCs), it seems to be very beneficial because of different factors. Manual writing of verification environments is a very time-demanding process and automated generation significantly reduces this overhead. The main point is that the basic components of verification environments (at least those written with respect to the verification methodologies like UVM) have almost the same structure so they can be pre-generated and just adjusted for specific scenarios and specific DUVs. Moreover, by automated generation, it is possible to avoid integration and interconnection problems which are quite often present in manually written testbenches when various bus protocols and IPs are used. One example of such generator is Pioneer NTB from Synopsys [80] which enables to generate SystemVerilog and OpenVera testbenches for different hardware designs written in VHDL or Verilog with inbuilt support for third-party simulators, constrained-random stimulus generation, assertions and coverage. Another example is SystemVerilog Frameworks Template Generator (SVF-TG) [92] which assists in creating and maintaining verification environments in SystemVerilog according to the UVM.

5.2.3 Coverage Improvement Techniques

For complex systems like processors or controllers, reaching coverage closure represents a daunting task and a clue how to do this is not defined yet [58]. Maybe that is the reason why some verification teams still check coverage holes and prepare direct tests to cover such holes manually [89]. To target this issue, new methods for automating CDV have to be developed. The generation of appropriate scenarios can be driven by an intelligent algorithm that adjusts constraints of the pseudo-random generator automatically according to the coverage results gained from verification.

Several solutions already exist, e.g. those based on machine learning techniques. In [28], Bayesian networks are applied to CDV problem. In the first step, a training set of stimuli is used to learn the parameters of a Bayesian network that models the relationship between coverage and generated stimuli. In the second step, the Bayesian network is used to provide the most probable stimuli that would lead to a given coverage task.

In [89], a tool called StressTest is introduced. The engine uses closed-loop feedback techniques to transform the internal Markov model (used for generating stimuli) into one that effectively covers the user-defined points of interest. This approach is targeted to verification of microprocessors and requires an engineering team to provide a template describing interface protocols.

The authors in [38] present a method for automated generation of simulation inputs based on the analysis of the HDL description and the path coverage feedback. This method utilizes constraint solving using the word-level SAT.

A recent work of [39] introduces expressive functional coverage metrics which can be extracted from the DUV reference model at a high level of abstraction. After defining a couple of assertions (property checkers), a reduced Finite State Machine (FSM) is extracted from the golden model. The reduced FSM represents the related functionality to the defined assertion. Moreover, the coverage is defined over states and transitions of the reduced FSM to ensure that each assertion is activated through many paths. However, the work of [39] cannot be applied to complex designs where the state explosion problem may arise even for the exploration of the reduced FSM.

Some of the other related solutions are inbuilt in proprietary industry tools like inFact from Mentor Graphics [34] or VCS from Synopsys [81]. Unfortunately, the producers of these tools are usually not willing to reveal the techniques their tools use to achieve the high level of coverage.

For the purposes of this thesis, mainly the approaches based on evolutionary algorithms were studied. In general, evolutionary algorithms can be applied in software as well as hardware domain for improving coverage statistics. Very often, they are used for tuning the pre-generated tests and after the evolution, the best candidates are applied. In the other cases, they assist in the generation process itself in order to find suitable test scenarios.

For the verification of software systems, an evolutionary algorithm was applied for example in the work of [78], but without any application on a real world system.

The work of [27] proposed a method based on simple genetic algorithm for guiding simulation using normal random input sequences to improve coverage count of property checkers related to the design functionality. The genetic algorithm optimizes the parameters that characterize the normal random distribution (i.e., mean and standard deviation). This work is able to target only one property at a time rather than a group of properties, and it is not able to describe sharp constraints on random inputs. Experimental results on RTL designs have shown that the pure random simulation can achieve the same coverage rate but almost with double number of simulation vectors in the case of using genetic algorithm.

Evolutionary algorithm (genetic algorithm and genetic programming) have been also used for the purpose of processor validation and to improve the coverage rate based on ad-hoc metrics. For instance, in [9], the authors proposed to use a genetic algorithm to generate biased random instruction for microprocessor architectural verification. They used ad-hoc metrics that utilizes specific buffers (like store queue, branch issue buffer, etc.) for the PowerPC architecture. The approach, however, is limited only to microprocessor verification.

In [20], genetic programming rather than genetic algorithm is used to evolve a sequence of valid assembly programs for testing pipelined processor in order to improve coverage rate for user defined metrics. The test program generator utilizes a directed acyclic graph for representing the flow of an assembly program, and an instruction library for describing the assembly syntax. Afterwards. the same authors in [71] described the application of genetic algorithm and the tool called microGP for speed-path verification in the post-silicon processors testing. As a complementary approach for processor verification seems to be the work of [33], where the genetic algorithm is applied for automated generation of stimuli for verification of a specific programs loaded to an application-specific processor.

The work of [70] proposes an approach to automatically generate proper directives for random test generators in order to activate multiple functional coverage points and to enhance the overall coverage rate. In contrast to the classical blind random simulation, an enhanced genetic algorithm procedure is defined that performs the optimization of CDV over domains of the system inputs. The proposed algorithm, which is called the Cell-based Genetic Algorithm, incorporates specific representation and specific genetic operators designed for CDV. Rather than considering the input domain as a single unit, it splits it into a sequence of cells (subsets of the whole input's domain) which provide representations of the random generator constraints. The algorithm automatically optimizes the widths, heights and distribution of these cells over the whole inputs domains with the aim of enhancing the effectiveness of using stimuli generation. The efficiency of this approach was illustrated on a set of designs modeled in SystemC.

Genetic algorithms have been used for many other verification and coverage problems. For instance, [32] addresses the exploration of large state spaces. This work is based on Binary Decision Diagrams (BDDs) and hence is restricted to simple Boolean assertions. Genetic algorithms have been used with code coverage metrics to achieve high coverage rate of the structural testing. As an example, the work of [49] illustrate the use of genetic algorithm in order to achieve full branch coverage. Furthermore, genetic algorithms have been used for Automatic Test Pattern Generation (ATPG) problems in order to improve the detection of manufacturing and fabrication defects [56]. Finally, [16] presents a genetic algorithm based approach to solve the problem of SoC chip level testing. Particularly, the algorithm optimizes the test scheduling and test access mechanism partition for SoC in order to minimize the testing time which in return reduces the cost of testing. This approach shows a superior performance to the heuristic approaches proposed in the literature.

5.2.4 Building Effective Regression Suites

The main purpose of various optimization techniques targeted to regression testing is to remove redundancy from the original suite of stimuli/tests that can be either prepared manually or automatically generated, for example, in the process of functional verification. A great advantage of such regression suites is that they are usually smaller or are evaluated much faster than the original set of stimuli and at the same time, they cover the functionality of the system very well. Therefore, they are used for checking the correctness of the system when slight changes in the implementation are made, when a new version or a patch of the same system is introduced or just for testing before a regular release is planned. One of the main reasons for regression testing is to determine whether a change in one part of the system did not affected the other parts. Many techniques for preparing optimal regression suites exist, we focused mainly on those that optimize stimuli taken from verification.

The web-based platform FoCuS [40] provides regression suite generation, optimization and management. Their regression modeling language enables customizing the size and the contents of the regression suites based on the important characteristics of the system. It takes all simulated tests from verification and optimizes them using several algorithms to the smallest possible test suite. The density of the regression suite is measured as its functional coverage. Meteor [54] coverage analysis and regression generation tools developed at IBM Research Lab in Haifa provide a greedy regression test suite generation feature. Any test that hits one or more of the pre-defined coverage metrics is added to the regression suite. However, more precise optimization is not used. When testing software programs, authors in [55, 68] focus on different optimization problem called the test case prioritization. They apply optimization algorithms like greedy, meta-heuristic or evolutionary for ordering test cases so that the most beneficial (with respect to the code coverage) are executed first.

5.3 Chapter Overview

This chapter discussed the functional verification state-of-the-art and the available related work in the field of optimization of functional verification processes. Many literature and research papers were examined and evaluated and on their basis we formulated our research goals that are outlined in Chapter 6.

Chapter 6

Goals of the Ph.D. Thesis

This Ph.D. thesis focuses on finding new optimization techniques (in comparison to the state-of-the-art methods) that will improve various processes of functional verification. The attention is mainly paid to these goals:

1. To speed-up the implementation of UVM-based verification environments by the automated pre-generation of its components from the high-level specification of the verified circuit. In this way, the manual intervention of verification engineers will be restricted only to specific UVM components, like preparing verification scenarios or implementing reference models.
2. To eliminate the simulation overhead inbuilt in UVM-based functional verification by using an affordable and flexible FPGA accelerator. Flexibility will be achieved by moving various parts of the UVM testbench into the accelerator.
3. To automate and optimize reaching coverage closure in coverage-driven verification by a well-tuned algorithm in order to meet all verification goals as soon as possible. The algorithm will be running in the background of the verification process and drive verification to the unexplored areas of the coverage search space.
4. To optimize the set of verification stimuli and to reuse them also in the further phases of the development cycle, for example, during regression testing and fault testing.

All the formulated goals and proposed techniques need to be evaluated by extensive experimental results. Therefore, DUVs of various complexity were selected for verification, ranging from a simple arithmetic-logic unit to a RISC processor.

6.1 Expected Theoretical Results

As for the first goal, automated pre-generation of basic components should significantly reduce time devoted to preparing verification environments, because the main UVM components will be generated in the order of seconds. Of course, some parts of the UVM verification environment are quite hard to be automatically generated, like the definition of verification scenarios (valid and also erroneous scenarios) as well as the reference models, so it is expected that these parts would have to be manually adjusted for some DUVs.

The graph in Figure 6.1 illustrates the increase of coverage depending on the number of verified stimuli. The blue curve represents the approximation of the current state, when a non-driven pseudo-random stimuli generation is used. The red curve represents the state expected after acceleration and optimization. The speed-up should be achieved by accelerating simulation using the FPGA accelerator and the reaching of maximum coverage should be gained by the intelligent testbench automation.

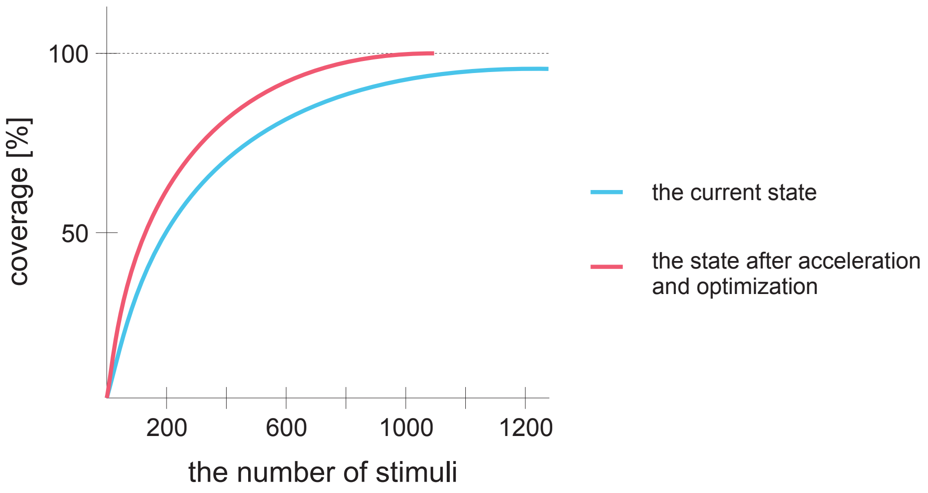


Figure 6.1: Acceleration of convergence to the maximum coverage.

In the last goal, mainly the overhead of using specific tools for preparing regression tests should be eliminated. The precondition is that verification is very often used in the development cycle of hardware systems and it can be reused also in further phases of this cycle, for example, during the regression testing or in the post-silicon fault testing. The original set of verification stimuli can be just optimized (mainly the redundancy introduced by randomness can be reduced, while preserving the same level of coverage) and then reused together with the verification testbench or its part. The expectation is that the optimized set will be running much faster and at the same time, it will check all the key functions.

In the following chapters of this Ph.D. thesis, all the proposed optimization techniques are described in detail together with all the corresponding experimental results. In particular, a separate chapter is devoted to every optimization technique.

Chapter 7

FPGA-based Acceleration of Functional Verification

Building upon our experience with different verification approaches and existing studies dealing with acceleration issues, in 2011 we introduced **HAVEN** (**H**ardware-**A**ccelerated **V**erification **EN**vironment), an open framework that exploits the inherent parallelism of hardware systems to accelerate their functional verification by moving the verified system together with several necessary components of the verification environment to FPGA. To provide advanced level of debugging capabilities, the framework adopts some formal techniques (assertion-based verification) and functional verification techniques (constrained-random stimulus generation, self-checking mechanisms) and enables partial signal observability to achieve appropriate debugging visibility while running in the FPGA. HAVEN is freely available and *open source* [85] so it can be used by academy projects and small companies without dependency on expensive accelerators or emulators mentioned in 5.2.1.

7.1 First Version of HAVEN

The first version of HAVEN was introduced in 2011 in the master thesis [75] and the basic principles of verification acceleration were described in the technical paper on Haifa Verification Conference [86].

7.1.1 Design of Acceleration Framework

HAVEN framework is based on the SystemVerilog language and allows users to run either the *non-accelerated* or the *accelerated* version of the same testbench with a cycle-accurate time behavior. The non-accelerated version runs entirely in the RTL simulator, while the accelerated version uses an FPGA to accelerate the verification runs. Providing these two versions allows to use the framework efficiently in different stages of the design flow, starting with debugging base system functions in a simulator to stress testing with millions of stimuli using hardware acceleration. After creating the basic verification environment, switching between these two versions is as easy as changing a single parameter of the verification.

The non-accelerated version of the framework presents a similar approach to functional verification that is commonly used in verification methodologies. This version is highly efficient in the initial phase of the verification process when testing basic system functionality with a small number of stimuli (up to thousands). In this phase it is desirable to have a quick access to the values of all signals of the system and to monitor the verification progress in a simulator. Coverage statistics (code coverage, functional coverage) provide a feedback about the state space exploration and allows the user to arrange constrained-random test cases properly to achieve even higher level of coverage. Despite all these advantages, the application of the non-accelerated version is very inefficient for the verification of complex systems and/or large number of stimuli. The rising complexity of verified hardware systems increases the time of simulation and also memory requirements on the storage of detailed simulation runs.

The accelerated version of the framework moves the DUV to a verification environment in the FPGA. This scenario is depicted in Figure 7.1. As the simulation takes the biggest portion of verification time, this approach may yield a significant acceleration of the overall process. Complex systems can be verified very quickly and with much higher number of stimuli (in the order of millions and more). Behavioral parts of the testbench, such as planning of test sequences, generation of constrained-random stimuli, and scoreboarding, remain in the software simulator. This partitioning is possible because the generic nature of currently prevalent verification methodologies (OVM, UVM), and transaction-based communication among their subcomponents enable to transparently move some of these components to a specialized hardware, while maintaining good readability for verification engineers.

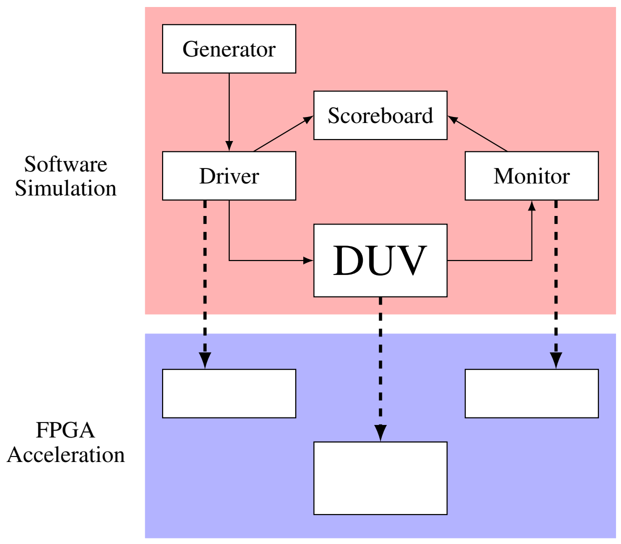


Figure 7.1: Moving some part of the verification testbench from software to hardware.

As the whole framework is implemented in compliance with the paradigm of object-oriented programming, the reuse of verification components (thanks to the mechanisms of encapsulation, inheritance and polymorphism) may lead to a considerable increase in productivity. The framework offers a library of prepared basic and extended verification components (classes) that are organized into packages. Thanks to these, assembling packages with new user-defined components or components inherited from the base classes can be easily done.

The architecture of the accelerated version of the HAVEN framework is illustrated in Fig. 7.2. The verified hardware system (DUV) is synthesized and placed into the FPGA. *Testcases*, written by the user, hold parameters such as settings of generics of the system, the number of tested stimuli encapsulated by transactions, or options for the generator of random stimuli. *Generator* produces constrained-random stimuli, which are typically random data and random delays generated in the ranges specified in the Testcase. *Scoreboard* dynamically predicts the response of the DUV and compares it with received output transactions. The remaining blocks were designed for the purpose of acceleration and they are described in detail below. They usually consist of two parts, a hardware component in the FPGA and its software counterpart in the simulator, which communicate together using a generic protocol. The communication with the FPGA is mediated over SystemVerilog's DPI that enables cooperation between SystemVerilog and C code that calls proper system functions. The hardware components were carefully designed to maximize their performance and minimize FPGA resources consumption.

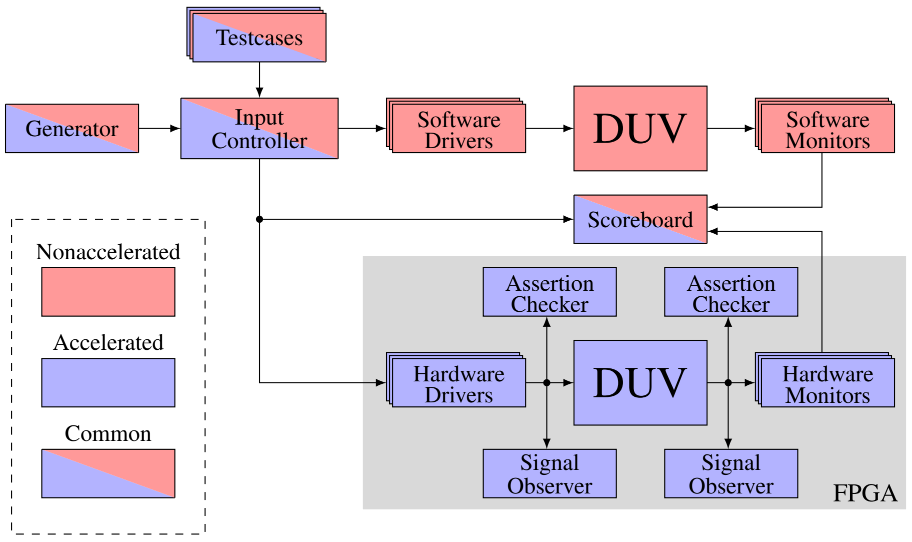


Figure 7.2: The architecture of the accelerated version of HAVEN.

Driver and Monitor. The software parts of both *Driver* and *Monitor* can be used independently of the hardware parts in the non-accelerated version of HAVEN when the system runs in a simulator. In such a case, the software part of Driver breaks data in input transactions down into individual signal changes and supplies them on the assigned input interface of the simulated DUV. The copy of the input transaction is sent to Scoreboard. The software part of Monitor drives the simulated DUV's output interfaces, observes signal transitions, groups them together into high-level output transactions and also passes them to Scoreboard for comparison.

In the accelerated version of HAVEN it is necessary to attach corresponding hardware parts of Driver and Monitor to existing software parts. The main purpose of hardware parts is to manage input and output interfaces of the synthesized DUV which is in this version situated in the FPGA.

Assertion Checker. SystemVerilog Assertions (SVA) is a standardized language for the specification of linear-time temporal properties of systems. Being a core part of SystemVerilog makes SVA easy to use for assertion-based verification of hardware systems. During the verification of a system, it is often useful to provide SVA formula describing correct behavior of interfaces of system subcomponents, so that any violation of an interface protocol during a verification run is captured and reported. Moreover, for standard interfaces, such as PCI, AHB, AXI, or USB, the packages with their description (either in SVA or in another assertion language) are already available.

Due to its linear-time nature, any SVA formula can be effectively transformed into a Büchi automaton, which is in turn easily synthesizable into a finite-state machine in an FPGA, as shown in [22]. HAVEN allows the user to connect various *Assertion Checkers* to the verified system.

An Assertion Checker represents one or more assertions and whenever a violation of any of the assertions is detected, a special packet with information about the nature of the violation is sent to the software environment for further analysis of the error.

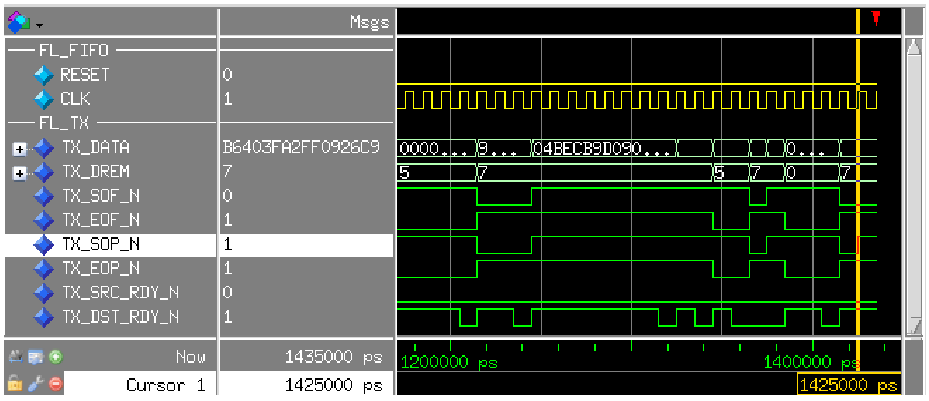
Signal Observer. When functional verification of a system detects an error, it is often convenient to observe internal states of the verified system in order to localize the source of the erroneous behavior. While this is easy when the system runs in a simulator, observing internal states of a system in an FPGA is not directly possible. Therefore, HAVEN provides *Signal Observer*, a component that monitors values of signals in the system during a verification run. The values are stored into the standardized Value Change Dump (VCD) format and can be inspected using any compliant waveform viewer, e.g., ModelSim or GTKWave (in the former case, the file with the dump needs to be converted into a ModelSim internal format by `vcd2wlf`, a tool provided in the ModelSim distribution). For common interfaces, it is easily possible to define specialized components derived from Signal Observer such that the output waveform contains correctly named and grouped signals.

Platform. HAVEN is built upon NetCOPE [65], a platform for developing various applications in FPGAs. The version of NetCOPE from 2011 is free and open-source, the newest versions are provided by the Invea-Tech company [47]. NetCOPE provides abstraction over the type of the FPGA and the used acceleration card by defining a uniform interface for data transfers between the FPGA and the CPU. Although the focus of NetCOPE is primarily on network applications, it was successfully used for our purpose as well. Moreover, because NetCOPE provides a uniform interface over several protocols, its use makes it very easy to change the framework to use Ethernet or other supported communication protocol for data transfers instead of a system bus without any change to the verification environment itself.

Cycle-Accurate Behaviour. Our goal is to obtain the same time behavior of verification runs in the non-accelerated and the accelerated version of the framework. The reason for this is clear: when a bug occurs in the accelerated version, it is possible to run the non-accelerated version with the same failing verification scenario and explore the origin of the failure in detail in the perfect debugging environment of a simulator. Simply placing the DUV in the FPGA is not an option as the transfer of data through the system bus may be delayed, thus yielding behavior different from the one obtained from the simulator. We solved this issue by placing the DUV into a separate clock domain and enabling/disabling the clock signal for this domain depending on the state of buffers for the input and output transactions. Thus the run of the DUV in the FPGA is guaranteed to result in the same waveform as the run in the simulator.

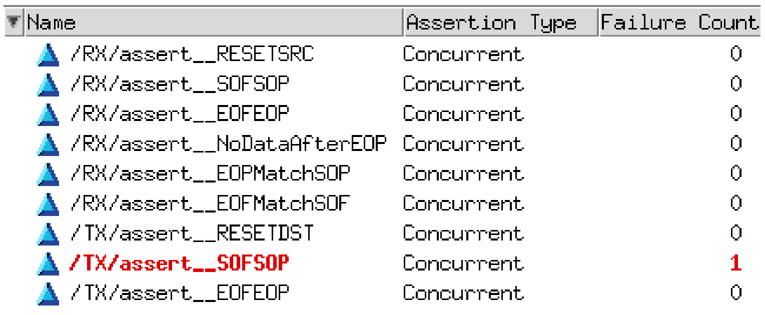
Error Detection. The framework monitors two types of errors: assertion failures and conflicts in Scoreboard such as missing or corrupted data or incorrect order of received transactions. If a bug is detected, the framework provides a short report about the nature of the failure, the simulation time when it occurred and the number of the received transactions which caused the inconsistency in Scoreboard. An example of error detection follows.

Thanks to the assertion analysis, any detected discrepancy in the observed behavior results in an error, which is reported near the origin of the functional defect. In the non-accelerated version of HAVEN it is recommended to take advantage of all possibilities offered by a simulator to explore the issue. In ModelSim, an assertion error is directly detectable from the simulation waveform as illustrated in Fig. 7.3. The origin of the failure can be examined in detail in the assertions window (Fig. 7.4).



	Msgs
FL_FIFO	
RESET	0
CLK	1
FL_TX	
TX_DATA	B6403FA2FF0926C9
TX_DREM	7
TX_SOF_N	0
TX_EOF_N	1
TX_SOP_N	1
TX_EOP_N	1
TX_SRC_RDY_N	0
TX_DST_RDY_N	1
Now	1435000 ps
Cursor 1	1425000 ps

Figure 7.3: Assertion failure in the waveform of the ModelSim simulator.



Name	Assertion Type	Failure Count
/RX/assert__RESETSRC	Concurrent	0
/RX/assert__SOFSOP	Concurrent	0
/RX/assert__EOFEOP	Concurrent	0
/RX/assert__NoDataAfterEOP	Concurrent	0
/RX/assert__EOPMatchSOP	Concurrent	0
/RX/assert__EOFMatchSOF	Concurrent	0
/TX/assert__RESETDST	Concurrent	0
/TX/assert__SOFSOP	Concurrent	1
/TX/assert__EOFEOP	Concurrent	0

Figure 7.4: The list of assertions in the assertions window of the ModelSim simulator.

Independently of the simulator and even in the accelerated version, HAVEN prints a short report to the transcript file with the description of circumstances that led to the assertion failure, the time of the detected discrepancy and the sequence number of the affected transaction. In the accelerated version of HAVEN this information is provided by Assertion Checker. The presented example demonstrates an erroneous situation during the verification of the FIFO component with a deliberately inserted bug. The reports obtained from the non-accelerated version (Fig. 7.5) and from the accelerated version (Fig. 7.6) of HAVEN show that the failure is invoked by the violation of the communication protocol called FrameLink (see description in Chapter 12) when the signal `SOF_N` is not active at the same time as the signal `SOP_N`, which means that there is some data that does not belong to any part of the FrameLink frame (which is forbidden). Note that the reported times of failures differ because whereas the simulator reports an assertion failure as soon as it happens, the created hardware Assertion Checker reports all assertion failures only once for each erroneous frame. But the sequence numbers of corrupted transactions fit in both versions, in the non-accelerated version 25 items were successfully compared in Scoreboard, so the 26th is faulty and in the accelerated version directly the 26th erroneous transaction is reported.

```
############ TEST CASE 1 ############
#
# START TIME: Tue Aug 23 13:54:43 CEST 2011
# ** Error: TX_SOF_N is not active in the same time as TX_SOP_N.
#    Time: 1435 ns Started: 1435 ns  Scope: testbench.TX File:
# --------------------------------------------------------------
# --  TRANSACTION TABLE
# --------------------------------------------------------------
# Size:             28
# Items added:      53
# Items removed:    25
# --------------------------------------------------------------
```

Figure 7.5: The assertion report from the non-accelerated version of HAVEN.

```
############ TEST CASE 1 ############
#
# START TIME: Tue Aug 23 14:57:11 CEST 2011
#
# !!!!!! Assertion error !!!!!!
# Violation of FrameLink protocol at checker:     170
# Time of violation:                             1530 ns
# Violated transaction #:                          26
# Tue Aut 23 14:57:12 CEST 2011
#
# ------------ ASSERTION REPORT ------------
# TX FrameLink Assertion Error: SOF_N without SOP_N
# TX FrameLink Assertion Error: Data between EOP_N and SOP_N
# TX FrameLink Assertion Error: No EOP_N before SOP_N
# --------------------------------------------------------------
```

Figure 7.6: The assertion report from the accelerated version of HAVEN.

For debugging purposes, the accelerated version of HAVEN includes the Signal Observer to ensure adequate signal visibility. In Figure 7.7, the comparison of the waveform obtained from the simulation of the non-accelerated version and the waveform received from the Signal Observer in the accelerated version is available.

7.1.2 Experimental Results

We performed a set of experiments using the COMBOv2 LXT155 acceleration card [64] equipped with the Xilinx Virtex-5 FPGA in a server with two quad-core Intel Xeon E5420@2.50 GHz processors and 10 GiB of RAM. The data throughput between the acceleration card and the CPU was measured to be over 10 Gbps for this configuration. We used Mentor Graphics' ModelSim SE-64 6.6a as the SystemVerilog interpreter and in the case of the non-accelerated version also as the DUV simulator. Unfortunately, we were not able to compare HAVEN to other solutions for acceleration of functional verification, because these are mostly not freely available commercial products.

We evaluated the performance of HAVEN on two hardware components that use FrameLink (a communication protocol described in Appendix 12) as their input and output interface: a simple First In First Out (FIFO) buffer and a Hash Generator (HGEN) which computes the hash value of input data using the Bob Jenkins's Lookup2 hash algorithm [48]. In order to fully exploit the capabilities of the accelerated version of HAVEN it is necessary to verify a complex system. For this purpose we also built systems with 2, 4, 8, and 16 HGEN units working in parallel. We focused on verification of a large number of very short data transactions (1–36 B).

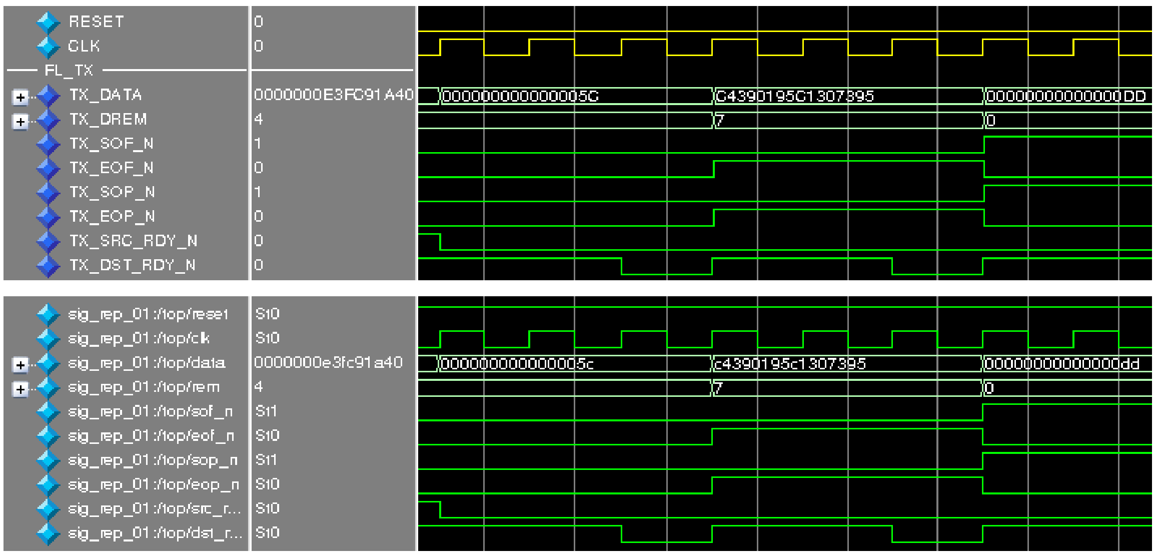


Figure 7.7: The simulation waveform (upper part) compared to the waveform obtained from Signal Observer (lower part).

The results of our experiments are given in the following tables. Table 7.1 compares the times of the whole verification runs of the non-accelerated version (**NAV**) to the times of runs of the accelerated version (**AV**) and the acceleration ratio (**Acc**). During the experiments, we observed that a considerable amount of time is taken by generating stimuli, therefore, we also measured the times of verification runs without the time of stimuli generation, as this value is the same for both the accelerated and the non-accelerated version. These results are given in Table 7.2. Fig. 7.8 shows the relation between the complexity of the verified component and the acceleration ratio (without the time of stimuli generation).

Table 7.1: Acceleration of verification including the time of stimuli generation.

Trans.	**FIFO**			**HGEN**		
	NAV [s]	**AV [s]**	**Acc [-]**	**NAV [s]**	**AV [s]**	**Acc [-]**
50,000	49	26	1.885	176	37	4.757
100,000	99	52	1.904	353	74	4.770
200,000	197	104	1.894	706	149	4.738
500,000	492	258	1.907	1,760	378	4.656
Trans.	**HGENx2**			**HGENx4**		
	NAV [s]	**AV [s]**	**Acc [-]**	**NAV [s]**	**AV [s]**	**Acc [-]**
50,000	446	37	12.054	614	37	16.595
100,000	897	74	12.122	1,241	74	16.770
200,000	1,795	149	12.047	2,461	148	16.628
500,000	5,181	379	13.670	6,979	378	18.463
Trans.	**HGENx8**			**HGENx16**		
	NAV [s]	**AV [s]**	**Acc [-]**	**NAV [s]**	**AV [s]**	**Acc [-]**
50,000	1,318	37	35.622	5,281	37	142.730
100,000	2,680	75	35.733	10,591	74	143.122
200,000	5,282	149	35.450	21,148	149	141.933
500,000	13,538	377	35.910	53,248	377	141.241

Table 7.2: Acceleration of verification without the time of stimuli generation.

Trans.	**FIFO**			**HGEN**		
	NAV [s]	**AV [s]**	**Acc [-]**	**NAV [s]**	**AV [s]**	**Acc [-]**
50,000	24	1	24.000	144	5	28.800
100,000	49	2	24.500	289	10	28.900
200,000	97	4	24.250	578	21	27.524
500,000	242	8	30.250	1,440	58	24.828

Trans.	**HGENx2**			**HGENx4**		
	NAV [s]	**AV [s]**	**Acc [-]**	**NAV [s]**	**AV [s]**	**Acc [-]**
50,000	414	5	82.800	582	5	116.400
100,000	833	10	83.300	1,177	10	117.700
200,000	1,667	21	79.381	2,333	20	116.650
500,000	4,861	59	82.390	6,659	58	114.810

Trans.	**HGENx8**			**HGENx16**		
	NAV [s]	**AV [s]**	**Acc [-]**	**NAV [s]**	**AV [s]**	**Acc [-]**
50,000	1,286	5	257.200	5,249	5	1,049.80
100,000	2,616	11	237.818	10,527	10	1,052.70
200,000	5,154	21	245.429	21,020	21	1,000.95
500,000	13,218	57	231.895	52,928	57	928.56

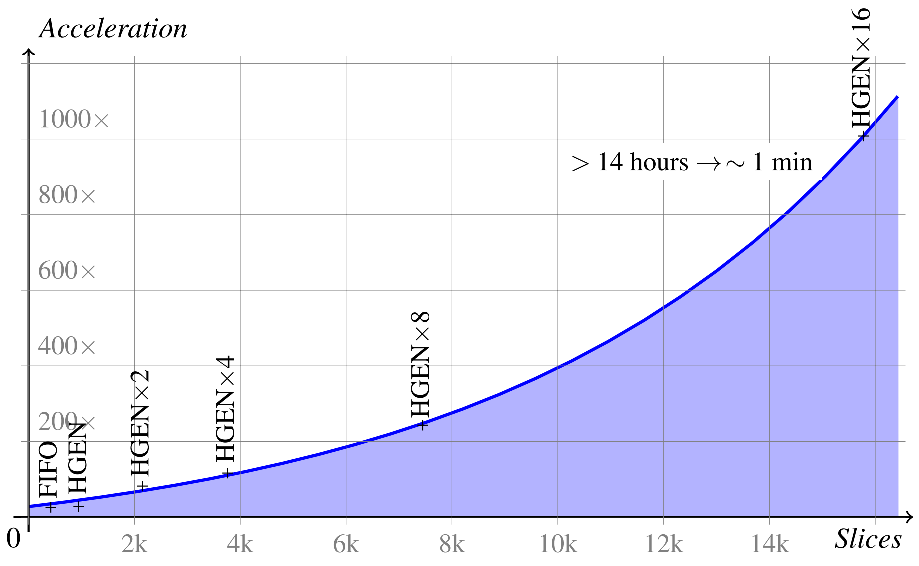


Figure 7.8: Relation between the acceleration ratio and the complexity of the verified component.

Table 7.3 summarizes the number of Virtex-5 slices used by the verification core of the accelerated version with the verified component (the column **Slices**) and the total number of occupied slices of the FPGA together with NetCOPE (the column **Total slices**); the total number of slices of the used FPGA (Xilinx Virtex-5 XC5VLX155T) is 24,320. The column **Build time** gives the time it took to generate the firmware for the FPGA. It can be observed that this time increases significantly as the total resource consumption approaches the capacity of the FPGA. The computed *break-even* number of transactions, which is, loosely speaking, the number of transactions for which the acceleration starts to be beneficial, is further given in the column **B-E Transactions**. Formally, this number is defined as the number $trans_{be}$ as can be seen in Equation 7.1:

$$\frac{trans_{be}}{\text{trans\_per\_sec}(NAV)} = build\_time + \frac{trans_{be}}{\text{trans\_per\_sec}(AV)} \tag{7.1}$$

where $build\_time$ is the build time of the firmware in seconds and $\text{trans\_per\_sec}(AV)$ and trans_per_sec (NAV) are the average numbers of transactions processed in a second by the accelerated and the non-accelerated version respectively. It is easy to deduce Equation 7.2 for computing $trans_{be}$:

$$trans_{be} = build\_time \cdot \frac{\text{trans\_per\_sec}(AV) \cdot \text{trans\_per\_sec}(NAV)}{\text{trans\_per\_sec}(AV) - \text{trans\_per\_sec}(NAV)} \tag{7.2}$$

Lastly, the column **B-E time** gives the time at which the break-even number of transactions is reached (i.e., the time of the run of the non-accelerated version).

Table 7.3: Properties of verified components.

Component	Slices		Total slices		Build time [s]	B-E trans.	B-E time [s]
FIFO	420	(1.7 %)	9,362	(38.5 %)	1,473	3,116,000	3,078
HGEN	947	(3.9 %)	9,787	(40.2 %)	1,724	622,000	2,188
HGENx2	2,152	(8.8 %)	11,315	(46.5 %)	1,895	222,000	2,061
HGENx4	3,762	(15.4 %)	12,938	(53.2 %)	2,340	196,000	2,486
HGENx8	7,448	(30.6 %)	16,304	(67.0 %)	3,390	131,000	3,488
HGENx16	15,778	(64.9 %)	22,096	(90.9 %)	7,909	75,000	7,965

In Fig. 7.9 we give the graph of the amount of time saved for given number of transactions when the accelerated version is used, which is again computed from the average number of transactions processed per second. Note that this time depends linearly on the number of tested transactions. Also note that the crosspoint with the x axis is the break-even number of transactions. Fig. 7.10 shows the amount of additional transactions that could be verified in the given time when the acceleration is used. The graphs in these figures show values that include the build time of the firmware, which we consider fair for the case when the system is being verified after some change in its internals. However, if the tests run in parallel, the overhead of building the firmware decreases. Moreover, for the case when only new test scenarios are added, the build time overhead does not apply and the acceleration is advantageous from the very beginning.

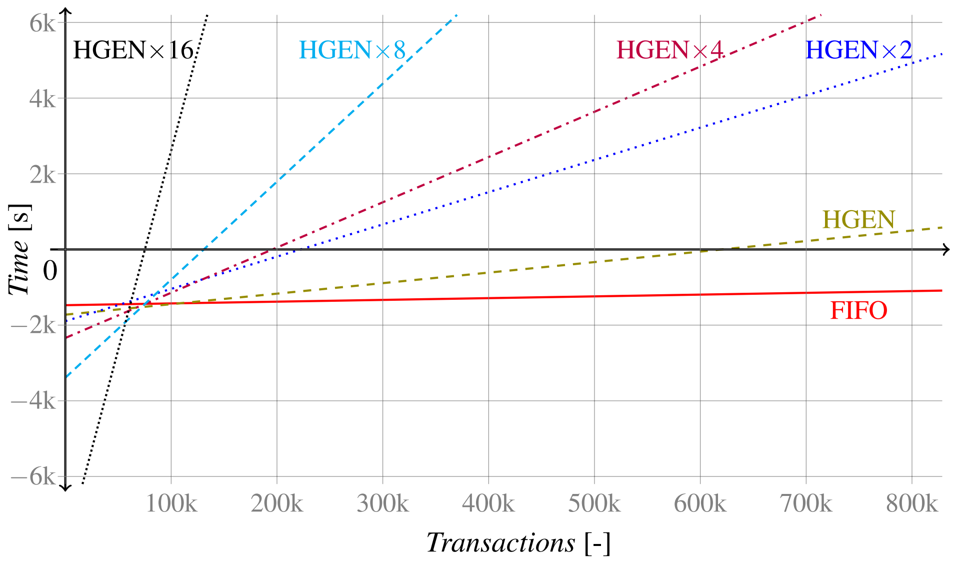


Figure 7.9: The time saved for verification of given number of transactions for the accelerated version.

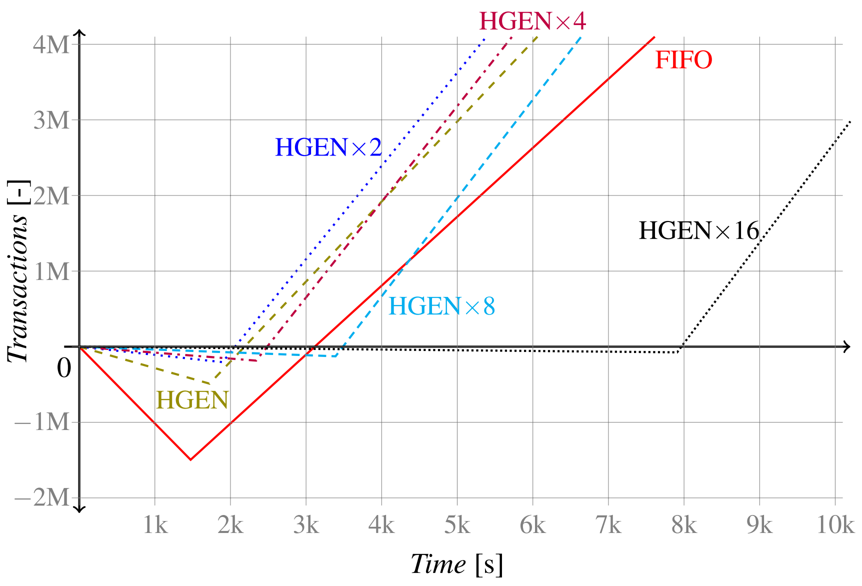


Figure 7.10: The number of additional transactions verified for the given time for the accelerated version.

7.2 Second Version of HAVEN

The second version of HAVEN which contains even more acceleration scenarios was introduced in 2012 in the technical paper on Haifa Verification Conference [84].

7.2.1 Design of Acceleration Framework

In this version, we further extended HAVEN with hardware acceleration of the remaining parts of the verification environment. This enables the user to choose from several different architectures which are evaluated and compared (in the following text, the term testbed will be used for representing one architecture). We have shown that each architecture provides a different trade-off between the comfort of verification and the degree of acceleration. Using the highest degree of acceleration, we were able to achieve the speed-up in the order of hundreds of thousands when verifying the most complex system (which consumed over 90% of the FPGA resources) while still being able to employ assertion and coverage analysis.

The new features added to HAVEN support seamless transition from the pre-silicon to the post-silicon verification using several architectures of the verification testbed. The user can start with the pure software version of the functional verification environment to debug the base system functions and discover the main bulk of errors (it is the original non-accelerated version). Later, when the simulation cannot find any new bugs in a reasonable time, the user can start to incrementally move some parts of the verification environment from software to hardware, with each step obtaining a different trade-off between the acceleration ratio and the debugging comfort.

During the evaluation, we observed that the main performance bottlenecks were generation of constrained-random stimuli, maintaining transactions in the scoreboard and comparing them to the outputs of the DUV. Therefore, we extended HAVEN with even better support for hardware acceleration by providing hardware implementations of the following components:

Hardware Generator. The Hardware Generator consists of a random number generator (we used the fast hardware implementation of the Mersenne Twister [45] which provides a random vector in each clock cycle) and an adapter to the desired format with a constraint solver. The generator seed as well as parameters of generated stimuli can be set from the simulator using a configuration interface. When a stimulus is generated it is encapsulated into the form of high-level transaction.

Hardware Scoreboard. The Hardware Scoreboard is a component that selects data sent from hardware monitors corresponding to output transactions from the DUV and performs comparison of these data from several interfaces. Any discrepancy in the received data is reported to the user.

Transfer Function. Hardware implementation of the Transfer Function depends on the verified component and can be performed in several ways. For components with already existing reference hardware implementations, we can use this as the transfer function (this use case may be suitable e.g. for regression testing). In the case only a software implementation of the transfer function is available, it is possible to use a soft processor core (e.g. MicroBlaze [93]) and run the transfer function as a program on the processor. If the transfer function takes a long time to be evaluated, a block of processor cores working in parallel may be used.

Coverage Monitor. In order to be able to guarantee reaching coverage closure in larger designs, the Coverage Monitor may be used to check whether given points of the DUV's state space have been covered. The component is connected to the wires which are to be checked and periodically sends the information about triggered cover points to the simulator. This information is reported to the user so that he/she knows which cover points have not been triggered. The user can in turn e.g. write direct tests or change settings of the generator to target these points. Since this component uses a register for every cover point, it is recommended for monitoring coverage of so far not covered points only.

7.2.2 Testbed Architectures of HAVEN

In this section we show how the components introduced in the second version of HAVEN may be assembled with the components of the first version in order to create several different testbed architectures, each suitable for a different use case and a different phase of the overall verification process. We start our description with the non-accelerated version running solely in the simulator and proceed by moving components of the verification environment into hardware in several steps.

Software version (`SW-FULL`). All components of the verification environment are in the software simulator (Fig. 7.11). The Software Generator produces stimuli encapsulated into input transactions which are propagated to the Software Driver and further supplied on the input interface of the DUV. Copies of transactions are sent to the Software Scoreboard where the expected output is computed using a reference transfer function. The Software Monitor drives the output interface of the DUV and sends received output transactions also to the Software Scoreboard to be compared to the expected ones.

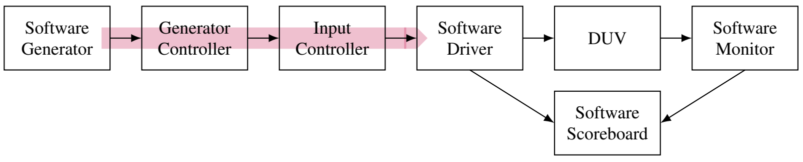


Figure 7.11: Software version (**`SW-FULL`**).

Hardware Generator version (`HW-GEN`). The architecture is similar to the **`SW-FULL`** version with the exception of the Hardware Generator and the Constraint Solver, which are placed in the FPGA and send generated stimuli in the form of transactions to the simulator.

Hardware DUV version (`HW-DUV`). In this architecture (Fig. 7.12), the Software Generator sends input transactions to the verification environment in hardware. In addition, a copy of every transaction is passed to the Software Scoreboard for further comparison. The Hardware Driver and the Hardware Monitor fulfill the same functions as their counterparts in the **`SW-FULL`** version, but they drive the input and output interfaces of the DUV running in the FPGA. The output transactions produced by the DUV are directed from the Hardware Monitor to the Software Scoreboard.

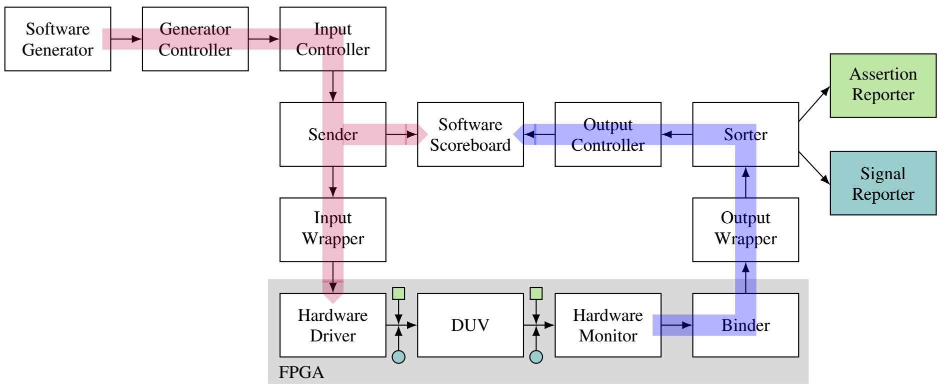


Figure 7.12: Hardware DUV version (**`HW-DUV`**).

Hardware Generator and DUV version (`HW-GEN-DUV`). This architecture is similar to the **`HW-DUV`** version but the generator is in hardware, as in **`HW-GEN`**.

Hardware version (`HW-FULL`). All core components of the verification environment in this architecture (Fig. 7.13) reside in the FPGA. The components in the software only set constraints for the Constraint Solver and report assertion failures, coverage statistics, or display waveforms of signals from hardware components.

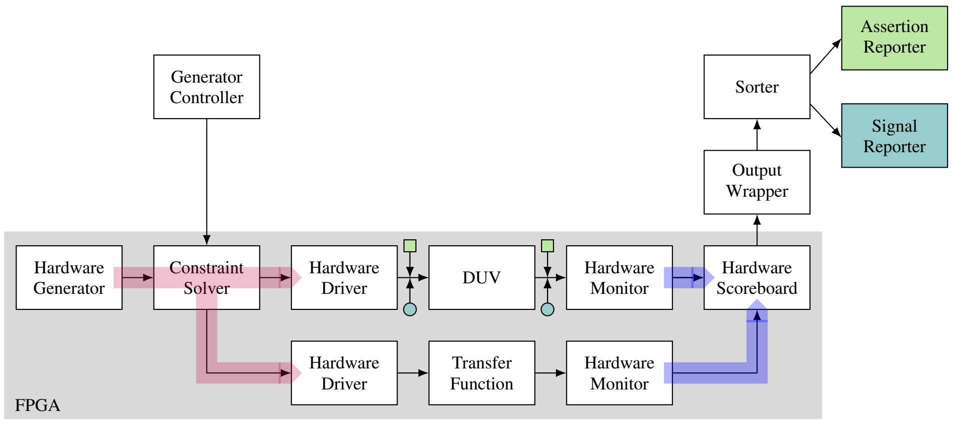


Figure 7.13: Hardware version (**`HW-FULL`**).

For those architectures of the HAVEN testbed that place the DUV into the FPGA (**`HW-DUV`**, **`HW-GEN-DUV`**, **`HW-FULL`**), it is possible to use the following optional components in hardware:

Assertion Checkers (illustrated by squares in figures) detect assertion violations of the DUV in hardware and report them to Assertion Reporters in the simulator, which in turn display them to the user.

Signal Observers (illustrated by circles in figures) store values of signals in hardware and send them to Signal Reporters in the simulator to be displayed as waveforms.

Coverage Monitors check coverage as described in Section 7.2.1.

7.2.3 Experimental Results

We performed a set of experiments using an acceleration card with the Xilinx Virtex-5 FPGA supporting fast communication through the PCIe bus in a PC with two quad-core Intel Xeon E5620@2.40 GHz processors and 24 GiB of RAM, and Mentor Graphics' ModelSim SE-64 10.0c as the simulator. We evaluated the performance of the architectures of HAVEN presented in the previous section on several hardware components: a simple FIFO buffer and several versions of a hash generator (HGEN) which computes the hash value of input data, each version with a different level of parallelism (2, 4, 8, and 16 units connected in parallel).

For each of the components, Table 7.4 gives the wall-clock time it took to verify the component for 100,000 input transactions for each architecture of the HAVEN testbed (because of issues with precise measurements of the time for the **`HW-FULL`** architecture, for this case we measured the time it took to verify the component for 1,000,000,000 input v and computed the average time for 100,000 transactions). Table 7.5 in turn shows the acceleration ratio of each of the architectures of the HAVEN testbed against the **`SW-FULL`** architecture.

We can observe several facts from the experiments. First, they confirm that the time of simulation (**SW-FULL**) increases with the complexity of DUV, so it is not feasible to simulate complex designs for large numbers of stimuli. Second, we can observe that it is not reasonable to use the simulator with hardware acceleration of the stimuli generator only (**HW-GEN**), at least for simple input protocols, which is the case of our protocol. In this case, the overhead of communication with the accelerator is too high. However, for the case when the DUV is also in hardware (**HW-GEN-DUV**), hardware generation of stimuli is (with the exception of the FIFO unit) advantageous compared to software generation (**HW-DUV**). Lastly, we can observe that the major speed-up of the hardware version (**HW-FULL**) makes this version preferable to be used for very large amounts of stimuli, e.g. when trying to reach coverage closure. Running verification of HGEN×16 for a billion stimuli, which took less than 7 minutes in this version, would take more than 21 months in the **SW-FULL** version.

Table 7.4: Results of experiments: times for verifying 100,000 transactions (in seconds).

Component	**FIFO**	**HGEN**	**HGEN×2**	**HGEN×4**	**HGEN×8**	**HGEN×16**
SW-FULL	199.	319.	1,126.	1,617.	2,539.	5,650.
HW-GEN	268.	308.	1,101.	1,984.	3,274.	7,534.
HW-DUV	65.	45.	48.	48.	48.	48.
HW-GEN-DUV	74.	22.	12.	12.	13.	13.
HW-FULL	0.0148	0.0205	0.0205	0.0239	0.0341	0.0410

Table 7.5: Results of experiments: acceleration ratios.

Component	**FIFO**	**HGEN**	**HGEN×2**	**HGEN×4**	**HGEN×8**	**HGEN×16**
HW-GEN	0.743	1.036	1.023	0.815	0.776	0.750
HW-DUV	3.062	7.089	23.458	33.688	52.896	117.708
HW-GEN-DUV	2.689	14.500	93.833	134.750	195.308	434.615
HW-FULL	13,429.	15,564.	54,925.	67,626.	74,347.	137,875.

7.3 Use-cases of HAVEN

The HAVEN framework was further used in two different scenarios: for accelerating functional verification of application-specific processors (ASIPs) [61] and as a part of the platform for testing fault-tolerant electro-mechanical systems [62].

7.3.1 FPGA Prototyping and Accelerated Verification of ASIPs

In 2014, as a continuation of our previous work, we designed and implemented a new feature for automated FPGA prototyping and accelerated verification of ASIPs and MPSoCs [61]. We realized that simulation-based verification of ASIPs and MPSoCs is valuable, but it runs very slowly when we need to evaluate thousands of embedded software applications. Therefore, we applied the accelerated version of HAVEN (the HW-DUV testbed) and moved the DUV from the software simulation into an FPGA. All other parts of the verification environment, such as loading applications/programs to ASIP, running a reference model and scoreboarding, remained in the software simulator. We were able to detect several errors using the accelerated version that were further debugged and fixed using the non-accelerated version (the SW-FULL testbed). Moreover, we identified many problems related to the placement of the ASIP into a real hardware which were not detectable in simulation.

At first, let us present how the verification environment for an MPSoC can look like and how it can be divided for the acceleration purposes. The demonstrational MPSoC consists of two ASIPs: ASIP1 and ASIP2. ASIP1 receives input data and works as a pre-processor for ASIP2, ASIP2 sends results to its output ports. Both processors share the same memory.

The non-accelerated UVM verification environment in Figure 7.14 for this MPSoC can be automatically generated in the Codasip Studio [18] together with the Reference Model. More details about the process of verification environment generation will be provided in Chapter 8. In the preparation phase of verification, programs (that will be running on processors) are loaded to the program part of the memory by the Program Loader component in the Memory Agent. During verification, data are driven to the inputs of ASIP1 by Driver of the Platform Agent and after their processing, outputs are read from ASIP2 by Monitor in the Platform Agent. Output data are automatically compared to those of the Reference Model. For more precise verification, the content of memories and register fields of the ASIPs can be compared to their counterparts in the Reference Model every time program data are processed. For reading the content of registers and the memory, the monitors in Register Agents and in the Memory Agent are used.

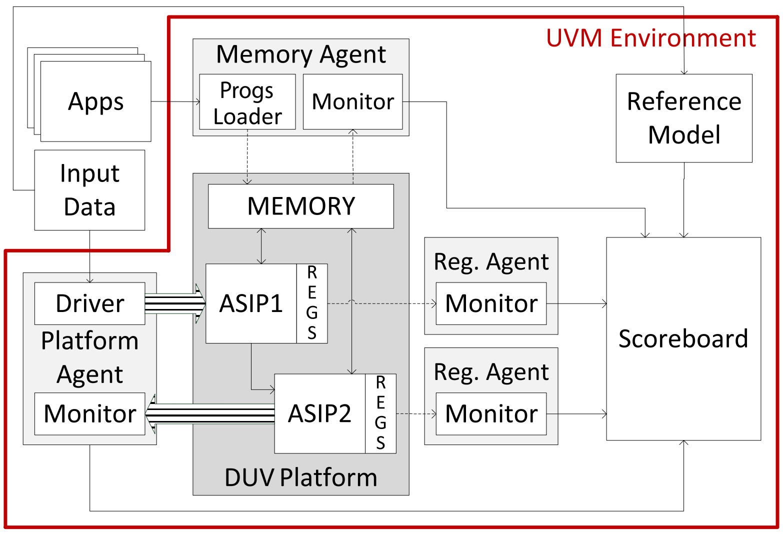


Figure 7.14: The architecture of the non-accelerated verification environment.

The accelerated verification environment in Figure 7.15 is derived from the non-accelerated version. Almost all UVM components are moved into the FPGA, except of the Reference Model and Scoreboard. In this environment, we replaced the older hardware components of HAVEN by some new ones, which are more similar to UVM testbenches by names and structures. For example, the original UVM Agents and their inbuilt components are replaced by the HW Agents. Scoreboard compares results of the Reference Model to the results of DUV (received from hardware through the Output Wrapper component). The Input Wrapper is used for sending programs to the Program Loader and for sending data to Driver of the HW Platform Agent.

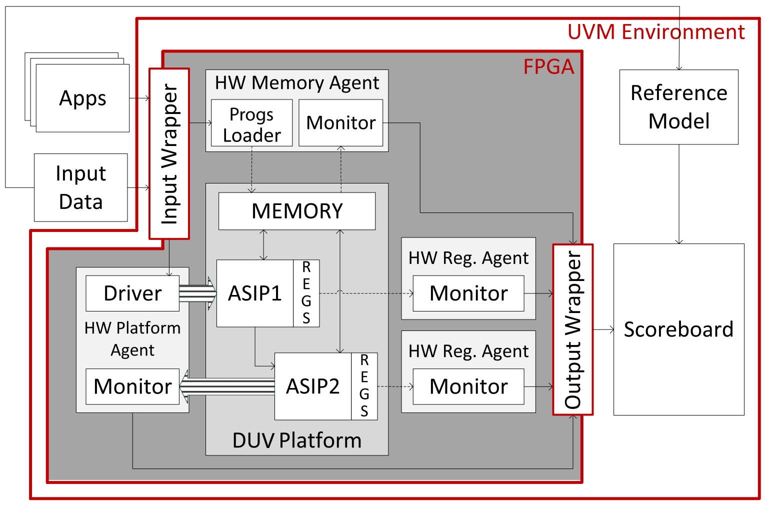


Figure 7.15: The architecture of the accelerated verification environment.

We performed experimental measures of the acceleration ratio with the DUV consisting of one ASIP called Codix RISC [17]. All experiments were running on the server with two quad-core Intel Xeon E5620@2.40 GHz processors and 24 GiB of RAM. The amount of consumed FPGA resources (slices) in the accelerated version was following: 1,428 (5.8%) for the Codix RISC processor and 1,669 (6.9%) for the hardware verification environment. Results of these experiments are depicted in Table 7.6. We have measured the runtime of the accelerated and the non-accelerated version, both for a different number of programs. Afterwards, the acceleration ratio was computed.

Table 7.6: The acceleration ratio and the run time of verification.

Number of	**Run time**		**Acceleration**
Programs [-]	**Non-accelerated [s]**	**FPGA-accelerated [s]**	**ratio [-]**
500	3458	2010	1.719
1,000	6841	3974	1.721
2,000	13634	7917	1.722
4,000	27208	15784	1.723
6,000	40845	23682	1.724
8,000	54384	31451	1.729
10,000	67965	39372	1.726

The measured values are also presented in the graphs of Figure 7.16 and 7.17. The acceleration ratio is on average 1,7x and slowly grows up with the number of the evaluated programs. Because we expected better results, we have performed some specific additional measurements and have identified candidates for further improvements.

From these preliminary experiments and some additional measures we concluded that the side of the verification environment running in simulation is still too complex and therefore, the runtime of the accelerated verification is negatively affected. The second problem is that we have executed a huge number of small programs, but only with the basic data sets. For precise verification, every program should be evaluated with more transaction data. In the case of extensive data processing the computation burden will be higher, so the acceleration will be more beneficial. Furthermore, we plan to examine the benefits of other HAVEN testbeds for the verification of ASIPs.

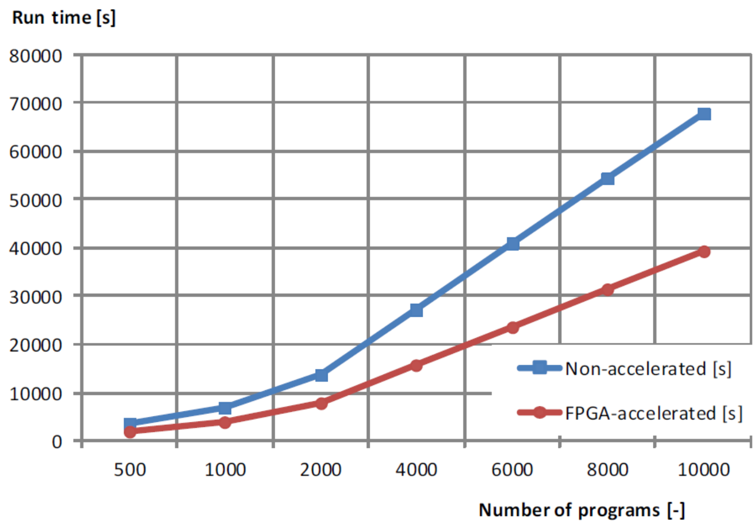


Figure 7.16: The relation between the runtime of the non-accelerated and the accelerated version.

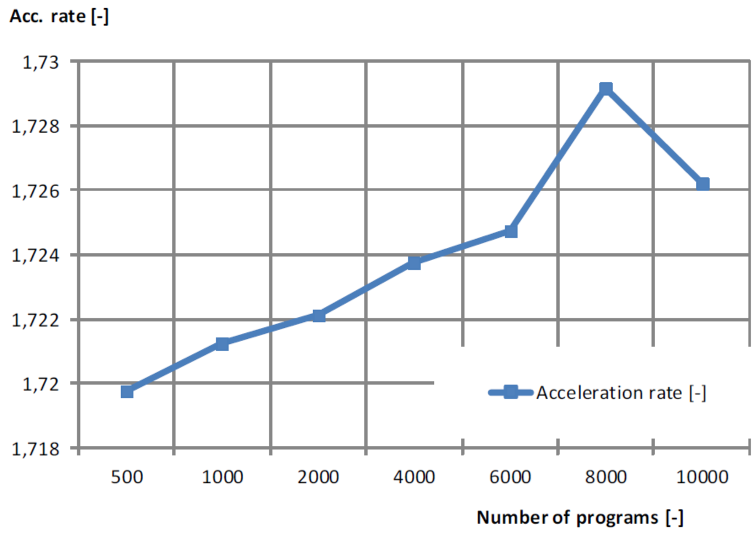


Figure 7.17: The relation between the acceleration ratio and the number of processor programs.

7.4 Evaluation Platform for Testing Fault-Tolerance Methodologies in Electro-mechanical Applications

At this point, it is important to mention the difference between verification and testing hardware systems against faults. Verification is mostly the part of the pre-silicon development and aims at discovering design errors. Testing against faults is usually done after verification, with real hardware representation of the system (e.g. FPGA) and aims at checking whether a specific hardware architecture is resilient to faults that may originate in extreme environments (like radiation, high temperature or high pressure).

Nevertheless, while testing resilience against faults (they are usually artificially injected into the system) we must be sure that if a failure occurs it is really caused by the injected fault and not by some design error still present in the system. In 2014, we introduced an evaluation platform for testing various fault-tolerance methodologies in electro-mechanical (EM) applications [62]. Our demonstration application was represented by a robotic system, in which the robot controller was the electronic part and the sensors and moving structures were the mechanical parts.

For extensive checking of the behaviour of EM systems placed into the evaluation platform, we needed to examine various scenarios. At the beginning of the experiments, we just manually prepared few mazes for the robot, we injected several faults into its controller and monitored the movement of the robot through the maze after every injected fault. However, the manual check is always difficult as it requires a full control from the user. Moreover, it was not feasible to do the experiments for more mazes and for every possible fault. Therefore, we decided to apply automation. We proposed a 3-phased approach, generally applicable for many EC systems, that is illustrated in Figure 7.18.

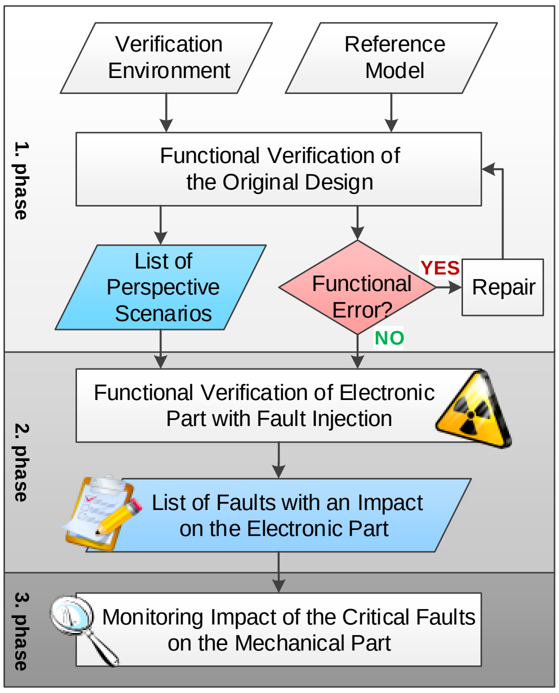

1. phase
Verification Environment
Reference Model
Functional Verification of the Original Design
List of Perspective Scenarios
Functional Error?
YES
Repair
NO
2. phase
Functional Verification of Electronic Part with Fault Injection
List of Faults with an Impact on the Electronic Part
3. phase
Monitoring Impact of the Critical Faults on the Mechanical Part

Figure 7.18: The flow of phases in the fault-tolerant systems verification.

1. In the first phase, the verification environment in SystemVerilog and the reference model for the electronic part (in our case the robot controller) must be created. We decided to use the reference model implemented in the C++ language. In this phase, the regular simulation-based functional verification is utilized, where the VHDL/Verilog description of the electronic part is used as the DUV.

   In our scenario, it was also necessary to connect the environment, where the mechanical parts of the robot were simulated. So for clarification, we used two simulation environments: functional verification was running in the RTL simulation environment and the mechanical robot was simulated in a separate simulation environment (we used the Player/Stage robot simulation, further referred only as the robot simulation). When the robot moves through the maze, information from sensors about the position and barriers is provided from the robot simulation to the verification environment. This information represents an input for DUV and for the reference model. After computation, the outputs from the DUV are automatically compared to the outputs of the reference model and they also represent the inputs that are propagated to the robot simulation (they represent the new moving coordinates for the robot).

Thus, the output of the DUV stimulates the movement of the mechanical part of the robot in the simulated maze. In this way, it is possible to evaluate many mazes and explore various verification scenarios very effectively and automatically, without any manual intervention.

The main output of the first phase is a claim whether the electronic controller works correctly as specified or not. It is very important because we have to be sure that the controller does not contain any functional errors in its implementation. It is also important to point out that in this phase we acquire a set of verification scenarios (in our case, different mazes with different start and end positions for robot movements) that will be re-used in the next phase.

2. The second phase consists of the verification running in the FPGA with the verification scenarios obtained from the previous phase. It is guaranteed for these scenarios that if no artificial faults are injected into the system the electronic part always behaves correctly. After a fault is injected, each of these scenarios is repeated (according to the number of injected faults). The result of this phase is a list of faults which causes a discrepancy on the output of the electronic controller for these specific verification scenarios. These faults will be examined in detail in the next phase where three possible outcomes can arise: (1) The output from the DUV and from the reference model is the same and an error did not appear. (2) The output is not identical but despite this, the system did not failed (e.g. the robot has completed its mission and reached the end position in the maze). (3) The output is not identical and at the same time, the system failed. This last outcome is the most serious one and it will require a thorough analysis of the problem.

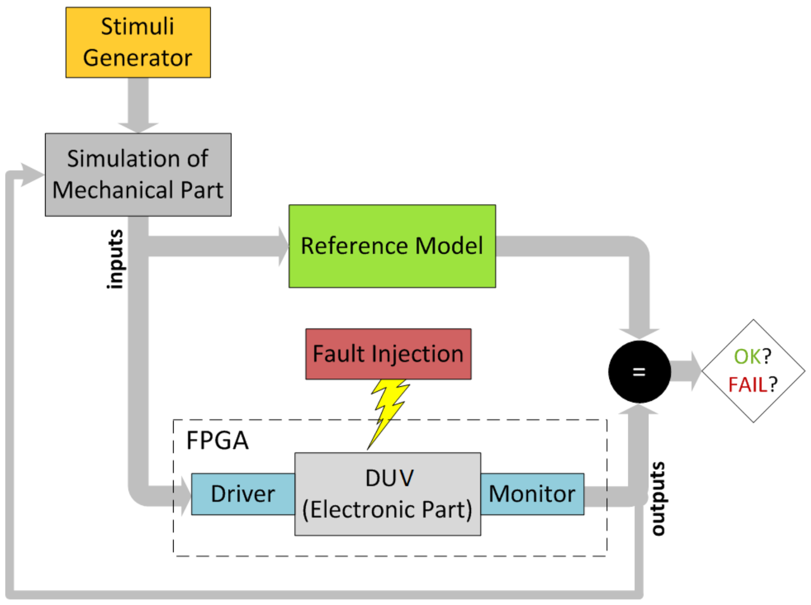


Figure 7.19: The functional verification involvement in our platform with the fault injection.

In order to be able to inject faults into the FPGA while performing functional verification in the second phase, we must carry out verification directly in the FPGA (not just in the simulation as in the first phase). For this purpose, we utilized the HAVEN framework. The extension of the evaluation platform with the support of functional verification is shown in Figure 7.19. The DUV (e.g. the robot controller) is placed into the FPGA. As the reference model, the C++ implementation from the first phase can be used, or the same VHDL implementation as is used as the DUV but without the injected faults can be considered. The Fault Injector is a unit that differentiates the platform architecture from the basic architecture of HAVEN and it is used for the artificial injection of faults into the FPGA.

3. The analysis of the faults which affected badly the mechanical part is the task for the third phase. In this phase, we will examine the faults that caused the failure of the system in more details. This activity can be carried out manually, or using the verification environment from the first phase.

Until now, we have used the platform only for the experiments with the robot system but the results seem to be very promising. In the future work, we will perform more complex measures and we will define a methodology how to harden the critical parts of the system (isolated in the third phase) against faults effectively in order to avoid all critical outcomes.

7.5 Main Contributions of HAVEN

In this chapter, the HAVEN framework for acceleration of functional verification runs was introduced together with many experimental results in various applications. To conclude this chapter, let us present the main contributions of the framework.

- The acceleration ratio over 100,000 x can be achieved for complex systems.
- Not only simple testbenches, but also highly complex UVM-based verification environments written in SystemVerilog can be accelerated.
- It is possible to use different testbed architectures of HAVEN, they represent a trade-off between the acceleration ratio and internal signal visibility of DUV.
- The acceleration board is cheaper in comparison to commercial solutions as it is built from achievable FPGA boards.
- HAVEN is freely available an open-source.

Chapter 8

Automated Generation of UVM Verification Environments

During the last decades, different verification methodologies have been proposed in order to build highly reusable and scalable functional verification environments. In this thesis, mainly the OVM and UVM methodologies were introduced in Section 3.2 which both provide a free libraries of basic classes and examples implemented in SystemVerilog that can be reused among several projects. These basic classes are usually just extended or modified for specific DUV purposes.

When focusing on effectiveness while implementing functional verification environments we realized that the reuse of testbench parts can significantly shorten the whole verification process. Therefore, in addition to using OVM and UVM libraries, in 2013 we devised an innovative technique how to shorten the implementation phase even more with the automated pre-generation of specific parts of the OVM/UVM verification testbenches (those tightly connected to specific DUV architectures) according to the high-level description of a circuit [88].

As the core of current embedded systems is usually formed by one or more processors, we decided to verify Application Specific Instruction-set Processors (ASIPs) in our experiments. In ASIPs, it is necessary to test and verify significantly bigger portion of logic, tricky timing behaviour and specific corner cases in a defined time schedule, so they represent a good working example. Nevertheless, it is important to point out that this approach is applicable in general, so it can be used also in the development cycle of other kinds of hardware systems.

8.1 Codasip Studio

As a development environment we utilized the Codasip Studio from the Codasip company [18]. As the main description language it uses architecture description language called CodAL, which is based on the C language and has been developed by the Codasip company in the cooperation with the Brno University of Technology, Faculty of Information Technology. All mainstream types of processor architectures such as RISC, CISC or VLIW can be described in this language, therefore, we consider it as the high-level specification language.

The CodAL language allows two kinds of descriptions. In the early stage of the design space exploration a designer creates only the instruction-set of ASIP (*the instruction-accurate description*). It contains information about instruction decoders, the semantics of instructions and resources of the processor. Using this description, programming tools such as a C/C++ compiler and simulation tools can be properly generated. The C/C++ compiler is based on the open-source Low Level Virtual Machine (LLVM) platform [2].

As soon as the instruction-set is stabilized a designer can add information about processor micro-architecture (*the cycle-accurate description*) which allows generating other programming tools like the cycle-accurate simulators and the HDL representation of the processor (in VHDL, Verilog or SystemC). As a result, two high-level models of the same processor on different level of abstraction exist.

The instruction-accurate description can be transformed into several formal models which are used for capturing particular features of a processor. Formal models which are used in our solution are *decoding trees* in the case of instruction decoders and *abstract syntax trees* in the case of semantics of instructions [63]. All formal models are optimized and then normalized into the abstract syntax tree representation that can be transformed automatically into different implementations (mainly in C/C++ languages). The generated code together with the additional code (it represents resources of processor such as registers or memories) forms the instruction-set simulator.

It is important to point out that in our generated verification environments, we took the instruction-accurate description as a golden (reference) model and the HDL representation generated from the cycle-accurate description is verified against it. The reason why the HDL is considered as DUV is that it represents the final stage of the pre-silicon processor development. At the time the golden model is generated, also the connections to the verification environment are established via DPI in SystemVerilog. Automated generation of golden models reduces the time needed for implementation of verification environments significantly. Of course, a designer can always rewrite or complement the golden model manually.

The cycle-accurate description of a processor can be transformed into the same formal models as in the case of the instruction-accurate description. Besides them, the processor micro-architecture is captured using *activation graphs*. In the case of the cycle-accurate description, the formal models are normalized into the *component* representation. Each component represents either a construction in the CodAL language such as arithmetic operators or processor resources or it represents an entity at the higher level of abstraction such as the instruction decoder or a functional unit. Two fundamental ideas are present in this model, (*i*) components can communicate with each other using ports and (*ii*) components can exist within each other. In this way, component representation is closely related to HDLs and serves as an input for the HDL generator as well as for the generator of verification environments.

For a better comprehension of the previous text, the idea is summarized once again in Figure 8.1. Codasip Studio works with the instruction and the cycle-accurate description of a processor and specific tools are generated from these descriptions. The highlighted parts are used during the verification process. It should be noted that the presented idea is generally applicable and is not restricted only to the Codasip Studio. Verification environments generated from the formal models are thoroughly described in the following section.

8.2 Functional Verification Environments for Processors

In order to comfortably debug and verify ASIPs designed in the Codasip Studio as fast as possible and do not waste time with implementation tasks, we designed a special feature allowing automated pre-generation of OVM/UVM verification environments for every processor. In this way we can highly reuse the specification model provided in the CodAL language and all intermediate representations of the processor for comprehensive generation of all units.

Our main strategy for building robust verification environments is to comply with principles of verification methodologies like OVM or UVM (they are depicted in Figure 8.2). We have fulfilled this task in the following way:

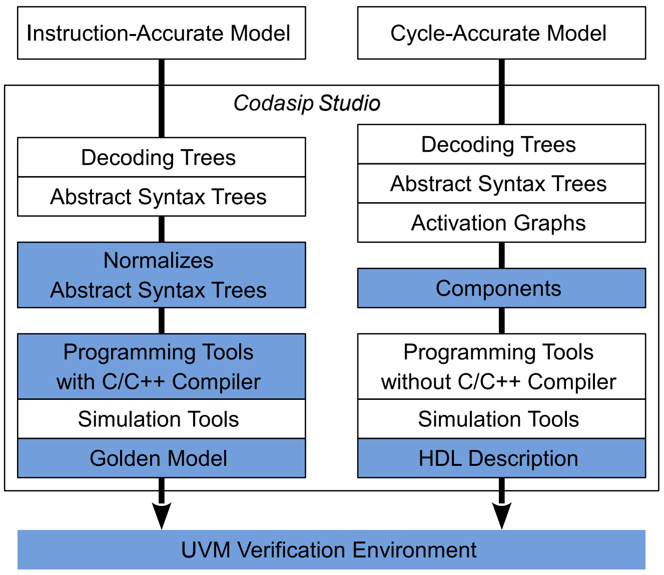


Figure 8.1: Verification flow in the Codasip Studio.

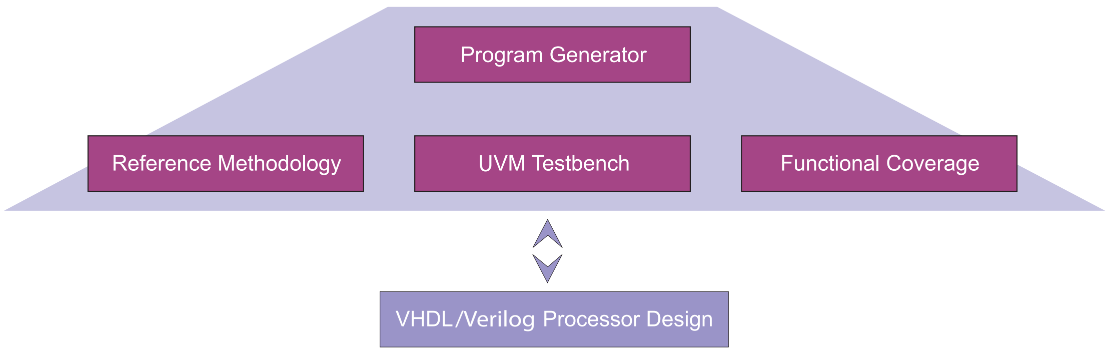


Figure 8.2: Verification methodology.

1. **OVM/UVM Testbench.** We support automated generation of object-oriented testbench environments created with compliance to open, standard and widely used OVM and UVM methodologies.

2. **Program Generator.** During verification we need to trigger architectural and micro-architectural events defined in the verification plan and ensure that all corner cases and interesting scenarios are exercised and bugs hidden in them are exposed. For achieving the high level of coverage closure of every design of processor it is possible to utilize either a generator of simple programs in some third-party tool or already prepared set of benchmark programs.

3. **Reference Methodology.** A significant benefit of our approach is gained by automated creation of golden models. We realized that it is possible to reuse formal models of the instruction-accurate description of the processor at the higher level of abstraction and generate C/C++ representation of these models in the form of reference functions which are prepared for every instruction of the processor. Moreover, we are able to generate SystemVerilog encapsulations, so the designer can write his/her own golden model with the advantage

of the pre-generated connection to other parts of the verification environment.

4. **Functional Coverage.** According to the high-level description of the processor and the low-level representation of the same processor in HDL, we are able to automatically extract interesting coverage scenarios and pre-generate coverage points for comprehensive checking of functionality and complex behaviour of the processor. Of course, it is highly recommended to users to add some specific coverage points manually. Nevertheless, the built-in coverage methodology allows measuring the progress towards the verification goals much faster.

In addition, interconnection with a third-party RTL simulator e.g. ModelSim from Mentor Graphics allows us to implicitly support assertion analysis, code coverage analysis and signals visibility during all verification runs. An example of generated UVM testbench for two ASIP processors was already shown in Figure 7.14 in Section 7.3.1 together with the description of the main components.

8.3 Experimental Results

In this section, the results of our solution are provided. We generated verification environments for two processors. The first one is the 16-bits low-power DSP (Harward architecture) called Codix Stream. The second one is the 32-bit high performance processor (Von Neumann architecture) called Codix RISC. Detailed information about them can be found in [17]. We used Mentor Graphics ModelSim SE-64 10.0b as the SystemVerilog interpreter and the DUV simulator. Testing programs from benchmarks such as *EEMBS* and *MiBench* or test-suites such as *full-retval-gcctestuite* and *perrenial testsuite* were utilized during verification. The Xilinx WebPack ISE was used for synthesis. All experiments were performed using the *Intel Core* 2 *Quad* processor with 2.8 GHz, 1333 MHz FSB and 8GB of RAM running on 64-bit *Linux* based operating system.

Table 8.1 expresses the size of processors in terms of required *Look-Up Tables* (LUTs) and *Flip-Flops* (FFs) on the Xilinx *Virtex5* FPGA board. Other columns contain information about the number of tracked instructions and the time in seconds needed for generation of SystemVerilog verification environment and all reference functions inside the golden models (Generation Time). In addition, the number of lines of program code for every verification environment is provided (Code Lines). A designer typically needs around *fourteen days* in order to create basics of the verification environment (without generation of proper stimuli, checking coverage results, etc.), so the automated generation saves the project time significantly.

Table 8.1: Measured Results.

Processor	LUTs/FF (Virtex5)	Tracked Instructions	Generation Time [s]	Code Lines
Codix Stream	1411/436	60	12	2871
Codix RISC	1860/560	123	26	3586

Table 8.2 provides information about the verification runtime and results. As Codix Stream is a low-power DSP processor, some programs had to be omitted during experiments because of their size (e.g. programs using standard C library). Therefore, the number of programs is not the same as in the case of Codix RISC. Of course, the verification runtime depends on the number of tested programs and if the program is compiled with no optimization the runtime is significantly longer.

The coverage statistics in Table 8.3 can show which units of the processor have been appropriately checked. As can be seen, the instruction-set functional coverage reaches only around fifty

Table 8.2: Runtime statistics.

Processor	Programs	Runtime [min]
Codix Stream	636	28
Codix RISC	1634	96

percent for both processors (i.e. a half of instructions were executed). The low percentage is caused by the fact that selected programs from benchmarks did not use specific constructions which would invoke specific instructions. On the other hand, all processor register files were fully tested (100% Register File coverage). This means that *read* and *write* instructions were performed from/to every single address in register files. The functional coverage of memories represents coverage of control signals in memory controllers. Besides functional coverage, ModelSim simulator provides also code coverage statistics like branch, statement, conditions and expression coverage. According to the code coverage analysis we were able to identify several parts of the source code which were not executed by our testing programs and therefore we had to improve our testing set and explore all the coverage holes carefully.

Table 8.3: Coverage statistics.

Processor	Code Coverage [%]				Functional Coverage [%]		
	Branch	Statement	Conditions	Expression	Instr-Set	Reg. File	Memories
Codix Stream	87.0	99.1	62.3	58.1	51.2	100	87.5
Codix RISC	92.1	99.2	70.4	79.4	44.7	100	71.5

Of course, the main purpose of verification is to find bugs and thanks to our pre-generated verification environment we were able to target this issue successfully. We discovered several well-hidden bugs located mainly in the C/C++ compiler or in the description of a processor. One of them was present in the data hazard handling when the compiler did not respect a data hazard between read and write operation to the register file. Another bugs caused jumping to incorrectly stored addresses and one bug was introduced by adding a new instruction into the Codix RISC processor description. The designer accidentally added a structural hazard into the execution stage of the pipeline.

8.4 Main Contributions of Automated Generation

The experimental results show that the automatic generation is fast and robust and we were able to find several crucial bugs during the processors design. The generation process saves the implementation time rapidly, as it is possible to generate the complete verification environment in the order of seconds.

Nevertheless, from the experiments we realized that the coverage rates must be increased. Therefore, during the last two years we were developing a universal generator of random assembly programs that is able to automatically extract all the necessary information about the ASIP from the instruction-accurate and the cycle-accurate models, like the format of all the instructions, latencies, ordering restrictions, etc. and to generate valid streams of instructions. Using this feature, we were able to further increase all coverage metrics for both processors over 90%. Some preliminary experiments were published in [15].

Chapter 9

Automation and Optimization of Coverage-driven Verification

In Coverage-Driven Verification (CDV), the search space (coverage space) of possible solutions is defined by different coverage metrics (they were specified in Chapter 3). In particular, the n-dimensional coverage space is defined by n various coverage metrics. Because CDV is mainly a manual task, it is quite challenging to check all of the coverage properties in a reasonable time. Therefore, new methods for automating this process have to be developed.

In this chapter, the manual CDV is described and a reasoning why and how it should be automated is elaborated. Furthermore, our solution for automation and optimization of CDV which is based on evolutionary computing [87] is introduced. In comparison with the standard CDV that utilizes the random search, using this method, the convergence to the maximum coverage is much faster, fewer input stimuli are used and no manual effort is required from the user. Moreover, the optimization is targeted to the verification process itself without the dependence on the circuit that is verified.

9.1 Manual CDV

The process of manual CDV is illustrated in Figure 9.1. Functional verification runs in a standard way. Stimuli are applied on the inputs of DUV and on the inputs of the reference model. A verification engineer usually changes the constraints of the pseudo-random generator or prepares direct tests according to the coverage analysis that is provided after the previous verification run is completed. Changing of constraints reflects the need to cover as many coverage holes as possible. This iteration process continues until the satisfactory coverage is achieved.

9.2 Automated CDV

If the search space of all measurable properties defined by the coverage metrics is so big that it cannot be explored by manual constraining of the pseudo-random generator, it is necessary to apply some kind of automation. The related work described in 5.2.3 already mentioned approaches that aim at this goal. They are based on Bayesian networks, Markov models or formal models. They try to substitute the verification engineer by an intelligent algorithm that adjusts constraints of the pseudo-random generator automatically according to the coverage results from verification.

When speculating which algorithms are suitable for effective exploring the coverage state space and finding good candidate solutions in CDV, we did a following reasoning. As the randomization is

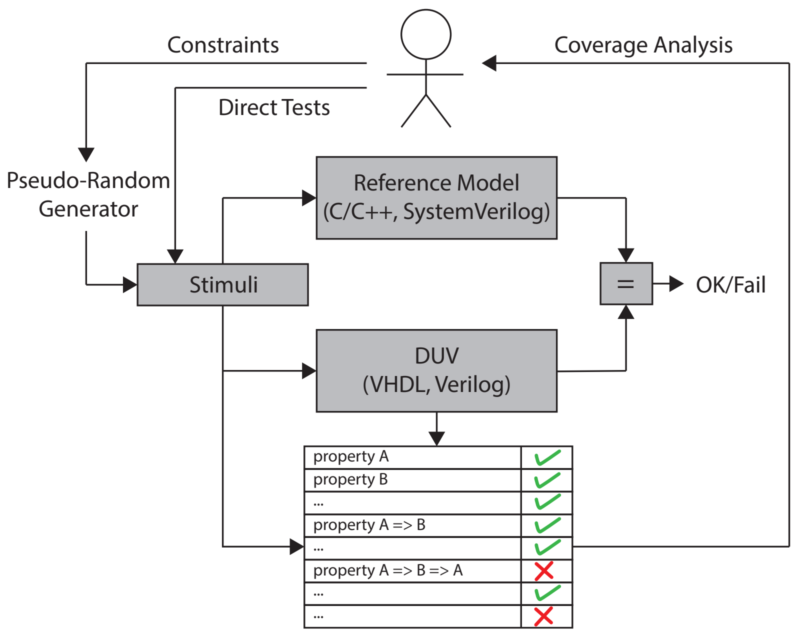


Figure 9.1: The manual coverage-driven verification.

natively used in functional verification, the heuristic stochastic algorithms may be selected, because randomization plays a considerable role in their evaluation. Heuristic algorithms were already described in Section 4.4 and from their simple comparison, the evolutionary algorithms seem to be most beneficial for our work.

9.2.1 Automated CDV Driven by Genetic Algorithm

For CDV automation and optimization, we decided to use one of the evolutionary algorithms called the Genetic Algorithm (GA). GA fits best to this problem as it utilizes both genetic operators (crossover and mutation) and its candidate solutions are encoded as chromosomes with a constant length of bit strings (*chromosome* is a coding representation of a candidate solution). At this point, it is important to mention that the result of GA is typically the one best chromosome that was evolved through several generations. In some cases, GA serves just as an optimizer of specific processes and its aim is not to find the best solution but only to preserve and employ the domain knowledge. This is exactly what we need in CDV as we want to optimize the process of functional verification continuously and to utilize the domain knowledge about the reached level of coverage. Figure 9.2 demonstrates how GA-driven verification works and the following text explains, how it differs from the basic GA.

In our proposal, every candidate solution is represented by a chromosome that encodes constraints (restrictions) for the pseudo-random stimuli generator (step 1). These constraints define probabilities of values that can be set on the inputs of DUV. To get an idea about probability constraints, see Figure 9.3. For a better comprehension, see two case studies in Section 9.5.

Initial population of chromosomes is created randomly. In particular, it means that the probabilities are set randomly. For evaluation of every chromosome, we instantiate the DUV and all the UVM components within the UVM verification environment, make the proper connections between modules, and then invoke verification in the RTL simulator. According to the constraints

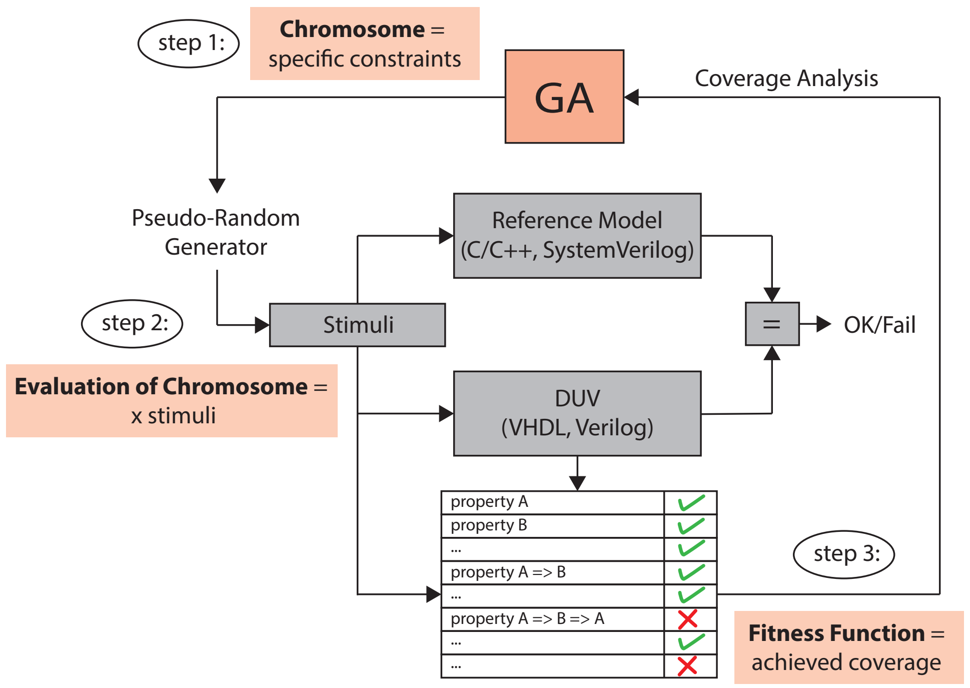


Figure 9.2: The automated coverage-driven verification driven by a genetic algorithm.

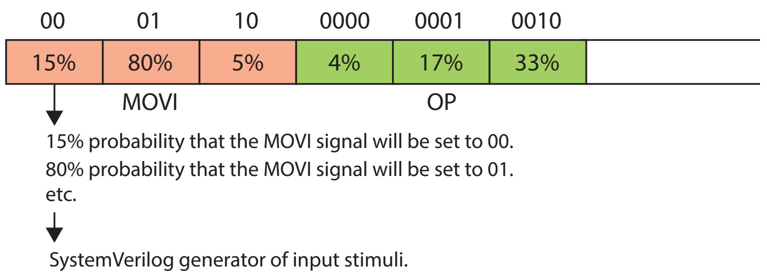

00
01
10
0000
0001
0010
15%
80%
5%
4%
17%
33%
MOVI
OP
15% probability that the MOVI signal will be set to 00.
80% probability that the MOVI signal will be set to 01.
etc.
SystemVerilog generator of input stimuli.

Figure 9.3: Probability constraints encoded in the chromosome.

in the chromosome, the generator produces a set of input stimuli that are applied to the inputs of the DUV during verification (step 2). Using these stimuli, specific properties are verified and it is reflected by the coverage measurement, how well it is done. The coverage status is used to compute the fitness function using which the quality of the chromosome is evaluated (step 3).

The best chromosomes are propagated to the next generation using elitism. It can be specified using the `ELITISM` parameter, how many best chromosomes will be passed to the next generation. Other chromosomes that participate in the next generation are represented by the offsprings of the chromosomes in the current population that were created by the genetic operators crossover and mutation. It is determined by the selection algorithm which chromosomes participate in making offsprings. Two selection algorithms can be used, the roulette selection and the tournament selection, the one that is used is defined by the test parameter `SELECTION`. Two genetic operators are applied. The first one is the two-points crossover, where the part of chromosome between two points is switched between two parent chromosomes. The second genetic operator is mutation, which is in our case represented by inversion of particular bits in the chromosome. The probability of crossover is defined by the test parameter `CROSSOVER_PROB` and the maximum number of mutations and the probability of mutation are defined by the test parameters `MUTATIONS_MAX` and `MUTATION_PROB`.

The GA optimization terminates when the threshold number of generations is reached, or when a desired coverage is achieved.

A pseudo-code of the genetic algorithm follows in Algorithm 4. Chromosomes encode the evolved constraints for the pseudo-random generator and the fitness function is expressed by the achieved level of coverage.

Algorithm 4: The proposed GA for CDV optimization.

```
Function genetic algorithm(number of generations, stimuli number):
    $t = 0$;
    $P(t)$ = create initial random population;
    evaluate population(P(t), stimuli number)
    while ((coverage! =100) or (t<number of generations)) do
        $Q(t)$ = select parents from$P(t)$;
        $Q'(t)$ = create new chromosomes from $Q(t)$ by crossover and mutation;
        evaluate population(Q'(t), stimuli number)
        $P(t+1)$ = select chromosomes to the next population from $P(t)$ and $Q'(t)$;
        $t = t + 1$;
    end
Function evaluate population(P, stimuli number):
    $c = 0$;
    while (c<population size(P)) do
        $G(c)$ = generate stimuli for chromosome $c$;
        evaluate coverage of $G(c)$ in verification and compute fitness;
        $c = c + 1$;
    end
```

A great advantage of this method is in its circuit-independency. It means that optimization focuses on reaching coverage closure of the defined coverage metrics, but only these metrics are dependent on the circuit that is verified. In other words, GA only integrates coverage metrics to the definition of the optimization task, but this task is the same for every verified circuit. Therefore, this method is generally applicable for functional verification of any circuit.

9.3 Integration into UVM

The GA-driven approach is provided as an extension of the basic functional verification environment prepared according to the UVM, following the principle of the object oriented programming (OOP). Our aim was to integrate the GA components effectively so the interference to the standard architecture of UVM is minimal. The Figure 9.4 highlights the components/classes that are added to the standard UVM environment and are implemented in the SystemVerilog language.

The basic UVM class for the definition of candidate solutions is called *Chromosome* class and contains an array with probabilities, the fitness function definition and the definition of genetic operators crossover and mutation. From this class, more specific chromosomes can be derived. It depends always on the circuit that is verified, items in the probability array are concretized and some further constraints can be added.

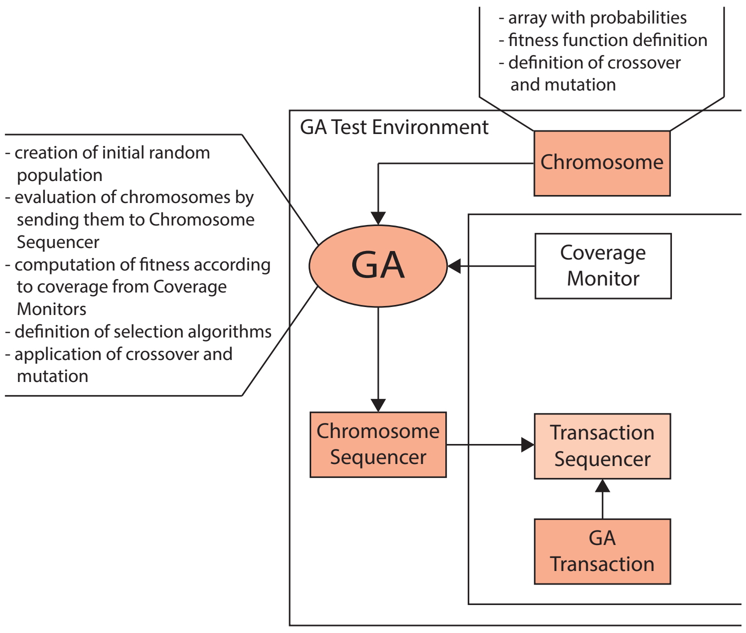

- array with probabilities
- fitness function definition
- definition of crossover and mutation
GA Test Environment
Chromosome
- creation of initial random population
- evaluation of chromosomes by sending them to Chromosome Sequencer
- computation of fitness according to coverage from Coverage Monitors
- definition of selection algorithms
- application of crossover and mutation
GA
Coverage Monitor
Chromosome Sequencer
Transaction Sequencer
GA Transaction

Figure 9.4: Extension of the UVM verification environment by GA classes.

The *GA Test* contains the core of the algorithm (marked *GA* in Figure 9.4). It is responsible for creating a random initial population (probabilities in the probability array are generated randomly) of chromosomes. Afterwards, it initiates the evaluation of chromosomes in simulation (an RTL simulator is typically launched for this purpose, e.g. ModelSim from Mentor Graphics), so every chromosome is sent to the *Chromosome Sequencer*. The *Chromosome Sequencer* sends chromosomes to the *Transaction Sequencer* that subsequently generates input stimuli (united by constructs called transactions) according to the constraints encoded in these chromosomes. The structure of transactions is defined by the *GA Transaction* class. Note that other parts of the standard UVM verification environment are not affected. When the simulation for one chromosome ends, the coverage is returned to *GA* by *Coverage Monitors*. *GA* uses coverage information for the computation of fitness. When all chromosomes are evaluated in this way, *GA* starts to prepare a new population. At first, selection of parents is performed. Afterwards, the genetic operators crossover and mutation are used to create new chromosomes. In this way, all chromosomes in a new population are prepared. The GA-driven optimization ends when the threshold number of generations is achieved, or when a desired coverage is reached.

For pseudo-random generation of initial populations, the inbuilt pseudo-random generator of UVM/System Verilog [90] is used. For generating input stimuli that are applied to DUV, the same generator or some proprietary generator can be used. We utilize our own universal generator [15] that is based on the Merssene Twister algorithm.

9.4 Tuning Parameters

GA needs to have all its parameters well tuned. For different optimization tasks, the values of these parameters can differ and must be found at first. We conducted several experiments to find their proper values for CDV optimization task.

From the implementation point of view, GA parameters are described in the SystemVerilog package class. Except of already mentioned parameters like `SELECTION`, `CROSSOVER_PROB`, `MUTA-`

TION_MAX, MUTATION_PROB, and ELITISM, there are parameters like GENERATIONS which determines the number of evaluated generations of chromosomes, POPULATION_SIZE which determines the number of chromosomes in every population, and SEED that sets the seed for the pseudo-random generator. Setting of these parameters affects the convergence to a proper optimal solution. However, some of these parameters have dominant effects on the GA performance while others have minor effects and can be kept constant for all the experiments.

We believe that the population size and the number of generations are the most important parameters that determine the amount of information stored and searched by the GA. Our rule of thumb is to use a population size proportional to the coverage search space and the complexity of DUV. However, we must consider also a real bottleneck in the GA performance which is the time that is required for the evaluation phase of the chromosomes in the RTL simulation. It is significantly longer than the time required for producing new generations of chromosomes using the GA operators. The reason is that in order to evaluate each potential solution, we need to stimulate the DUV for a number of simulation cycles depending on the complexity of the verified circuit as well as the complexity of the coverage task. Therefore, the population size and the number of generations must be selected wisely.

As can be seen in the experiments for ALU in Section 9.5.1, we decided to set the population size up to 20 chromosomes, and the number of generations up to 40. The number of generations was not higher than 40, because we were able to achieve 100% coverage for this number of generations. Despite ALU is a simple circuit, the coverage space is quite big and we generated a lot of stimuli per one chromosome during verification. The exact number of stimuli was computed from the average number of stimuli in the random search (AVG_RS) and from the number of generations (GENERATIONS) according to Equation 9.1. The reason for this setting is that we wanted to challenge the GA in achieving better results by restricting the number of stimuli which are applied to DUV per generation. The overall number of stimuli is then computed as GENERATIONS * STIMULI_NUMBER and we wanted to have it significantly lower than the average number of stimuli applied in the random search.

$$\texttt{STIMULI\_NUMBER} = \frac{\frac{\texttt{AVG\_RS}}{2}}{\texttt{GENERATIONS}}. \tag{9.1}$$

For the RISC processor, the population size was set up to 20 chromosomes, and the number of generations up to 100. The reason is that we evaluated the processor just for one program per chromosome during verification, because one program contains a significant number of instructions, even up to 1000.

After running empirical experiments for ALU, we believe that the probabilities of genetic operators crossover and mutation can be fixed during most experiments. We determined the optimal values for CROSSOVER_PROB and MUTATION_MAX that were later used also for the verification of the processor. In this way we want to demonstrate that setting the GA parameters is not a bottleneck when another system is verified. Or at least, we want to share our settings, which were beneficial for both of our experimental circuits which are quite different in the matter of complexity.

Other parameters of GA were set according to the results of the empirical study. When using the tournament selection, it is important to define a suitable size of tournament by the parameter TOURNAMENT_SIZE. We decided to apply Equation 9.2.

$$\texttt{TOURNAMENT\_SIZE} = 25\% \text{ of } \texttt{POPULATION\_SIZE}. \tag{9.2}$$

During mutation, it is specified at first, how many bits will be mutated in the chromosome (inverted). Afterwards, the mutation probability helps to determine if the inversion really happens or not. The maximal number of mutations is defined by the parameter MUTATION_MAX and is determined using the bit width of chromosomes (CHR_BW) by Equation 9.3.

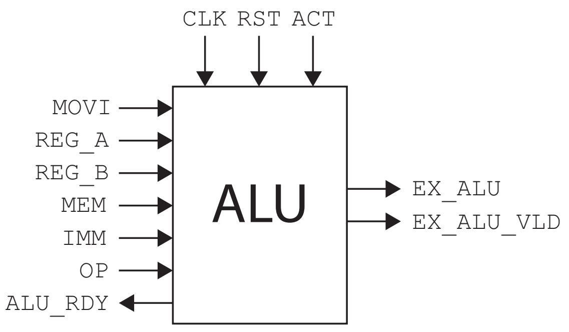
$$\texttt{MUTATION\_MAX} = 10\%\ \texttt{CHR\_BW}. \tag{9.3}$$

9.5 Experimental Results

In this section, we illustrate the effectiveness of the proposed GA for the automation of CDV and for the optimization of reaching coverage closure in the functional verification of two designs: the sequential Arithmetic-Logic Unit (ALU) with Booth multiplier and the RISC processor called Codix RISC [17]. We used the VHDL language to model DUV. For creating verification environments we use SystemVerilog language and UVM.

9.5.1 Arithmetic-logic Unit

As the first evaluation circuit, the arithmetic-logic unit (ALU) was selected. The block diagram of ALU and the description of its signals are provided in Figure 9.5.



- CLK, RST, ACT (in): the clock, reset, activation signal.
- REG_A (in): the first operand for every operation.
- MOVI (in): the selection signal, according to its value the second operand is picked either from data memory (MEM), register (REG_B), or as an immediate value (IMM).
- OP (in): the selected operation (16 options supported).
- ALU_RDY, EX_ALU, EX_ALU_RDY (out): output ALU signals.

Figure 9.5: The demonstration circuit - ALU.

For ALU, we focused on functional coverage metric. We were able to define 1989 functional properties by the standard means of SystemVerilog language. The aim of verification is to check all of these properties, so to achieve 100% functional coverage.

Two examples of functional coverage scenarios are depicted in Figure 9.6. The first one expresses all interesting data values of REG_A (all zeros, all ones, small values, big values, and others). The second one catches all sequences of possible operations (e.g. plus after minus, plus after multiplication, multiplication after multiplication, etc.).

The UVM verification environment for ALU is shown in Figure 9.7. The chromosomes for ALU are specified in the *ALU Chromosome* class. The evaluation of chromosomes is performed in simulation, the *Chromosome Sequencer* sends chromosomes to the *Transaction Sequencer* that subsequently generates STIMULI_NUMBER input stimuli (transactions) according to the constraints encoded in chromosomes. The structure of transactions is defined by the *GA ALU Transaction* class.

At this point, three experiments with ALU are described. As our goal is to show how effective the GA-driven search is in the process of CDV, we decided to compare its results with the basic random search and the constrained random search. They are shortly described in the following paragraphs.

```
reg_A: coverpoint register_A {
  bins zeros = {0};
  bins ones = {255};
  bins small_values = {[1:15]};
  bins big_values = {[240:254]};
  bins other_values = default;
}

op_after_op: coverpoint operation {
  bins op_after_op[] = ([0:$] => [0:$]);
}
```

Figure 9.6: Functional coverage scenarios for ALU.

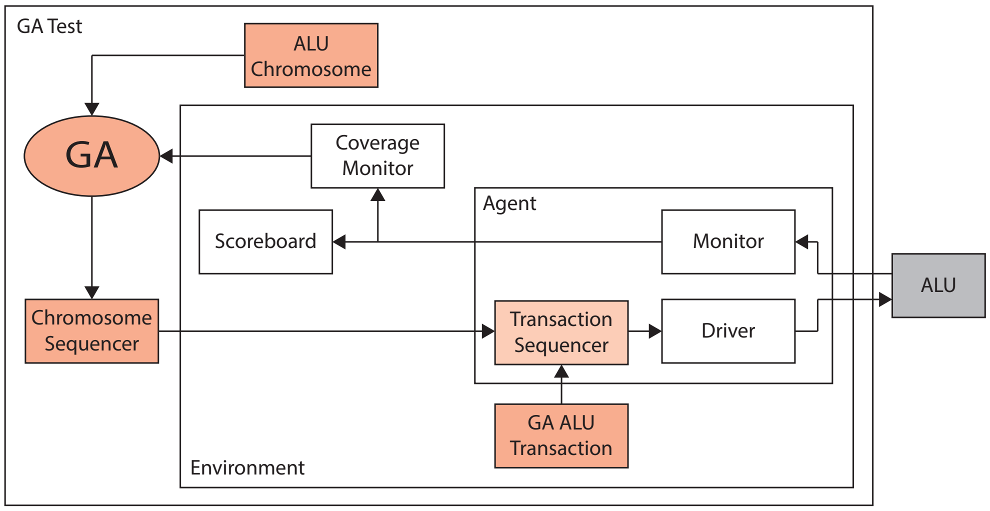


Figure 9.7: The UVM verification environment with GA classes for ALU.

In the GA-driven search, probability constraints for the pseudo-random generator are encoded in the chromosome. At first, input signals of ALU are divided into two categories: data and control. All possible values of control signals are specified (every control sequence is important and need to be checked). In case of ALU, these signals are: `RST`, `ACT`, `MOVI`, `OP`. In case of data signals, ranges of all possible values are selected (as these possible values can be reduced to ”interesting” ranges using approximation). In ALU, these signals are: `REG_A`, `REG_B`, `MEM`, `IMM`. Afterwards, probabilities are defined for every value of control signals and for every range of data signals. For example, the input signal `MOVI` can have three valid values (00, 10, 01). In the chromosome, for every of them a number is specified that defines a probability with which these values are generated as input of `MOVI`. For illustration, see Figure 9.8. Probabilities in the initial population of chromosomes are created randomly.

The basic random search does not specify probability constraints for generating stimuli. Instead, they are generated randomly. This approach represents the standard concept that is used in functional verification [90]. However, it can take a very long time to cover all of the properties, because without the coverage feedback, the generator produces stimuli that cover some properties repeatedly. It is based more on the idea: the longer the verification runs, the higher is the chance that there are no errors left in the state space (or, at least, the most critical ones). So the pseudo-random generator produces valid stimuli until the satisfying level of coverage is reached. Of course, it can take a very long time to cover all properties by this approach. Without the coverage feedback, the

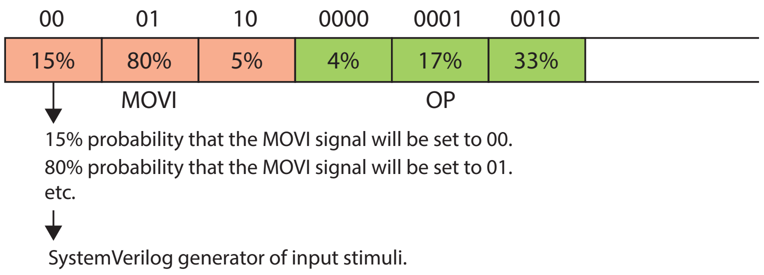

00
01
10
0000
0001
0010
15%
80%
5%
4%
17%
33%
MOVI
OP
15% probability that the MOVI signal will be set to 00.
80% probability that the MOVI signal will be set to 01.
etc.
SystemVerilog generator of input stimuli.

Figure 9.8: Probability constraints encoded in the chromosome.

generator produces stimuli that cover the same properties all over again.

The constrained random search uses probabilities for constraining the stimuli generation as the GA approach does but these probabilities are generated randomly. It means that good constraints are not remembered and propagated further. By this approach, we want to show that there is a difference when probabilities are driven by GA and when they are random.

The graphs in Figure 9.9 and Figure 9.10 show the comparison of the basic random search, the constrained random search and the GA-driven search. The first graph compares average values from 20 different measures for every kind of search, the second graph compares maximum values. In both graphs, the x axis represents the number of the required input stimuli while the y axis represents the achieved level of coverage of functional properties for ALU.

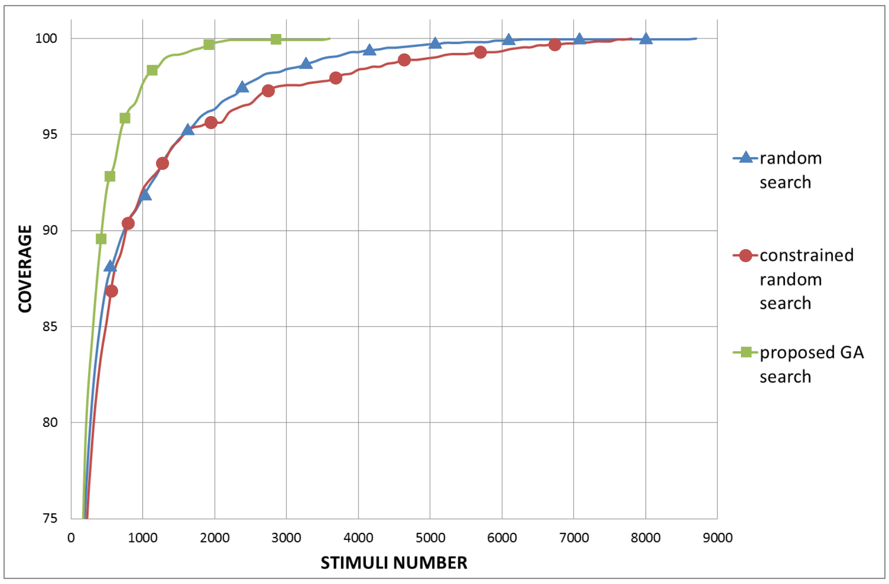


Figure 9.9: The comparison of average values from the basic random search, the constrained random search and the GA-driven search.

Parameters of GA in the graphs were set according to the results from the empirical study as follows. The probability of crossover was set to 80%, and the probability of mutation to 60%. The maximum number of mutations was set to 67 (see equation 9.3), tournament size to 5 (20% of the population size, see equation 9.2), elitism to 1 (1 best chromosome is propagated), stimuli number per chromosome to 171 (see equation 9.1, the average number of stimuli for random search is 5133). The size of the population was set to 10, the number of generations to 20. These are the settings

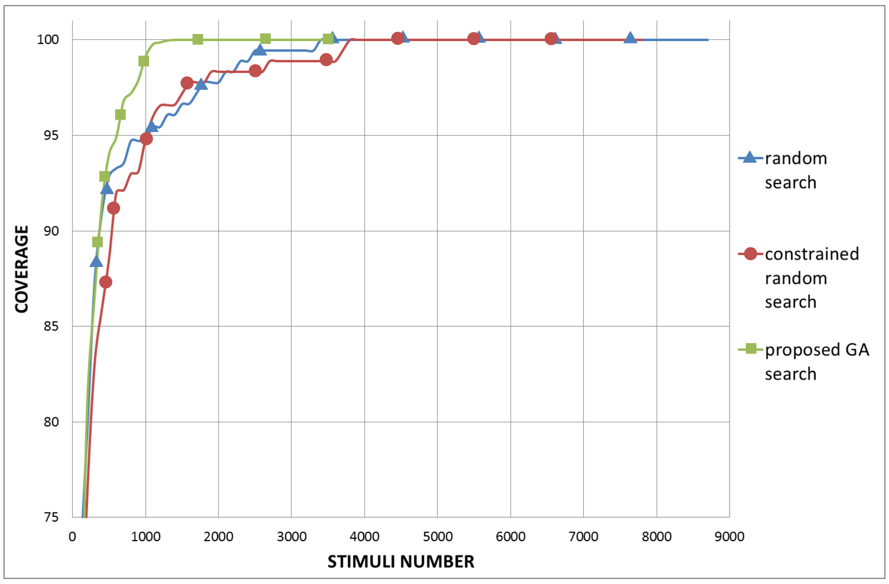


Figure 9.10: The comparison of maximum values from the basic random search, the constrained random search and the GA-driven search.

that were rated as the best ones from our empirical study. In this study, we changed the size of the population (5, 10, 15, 20) and the number of generations (10, 15, 20, 25, 30, 35, 40) in order to see their effect on the speed of learning.

It can be seen that GA achieves much better results than both random approaches. The convergence to the maximum coverage is significantly faster and the number of required stimuli is lower. It can be stated that for ALU, GA drives the generation of input stimuli successfully.

For running the experiments, we used Intel Core i5 CPU 3.33 GHz, 8 GB of RAM and the ModelSim simulator.

9.5.2 Codix RISC Processor

Coverage metrics that are tracked for Codix RISC processor in verification are: code coverage (statement, branch, expression, FSM), functional coverage (requests, status and responses on the bus interface), and instruction coverage (the complete instruction set and sequences of instructions).

Two examples of instruction coverage scenarios are depicted in Figure 9.11. The first one expresses that all 49 instructions of the instruction set should be covered during verification. The second one reflects all sequences of interesting sequences of instructions (e.g. arithmetic and logic instructions, jumps, etc.).

In the standard process of Codix RISC verification, different test programs are evaluated. At first, benchmark programs are used. Then, random programs that are generated from the universal generator [15] are applied. This generator works as follows. It uses two formal models for generating random programs. The first model describes the instruction set of the processor. The second model describes basic constraints that are important for generating valid programs for the processor. These basic constraints can drive initialization of registers, they can restrict sequences of instructions for jumps, they can define positions of labels, start and final sections, etc.

In the described standard process, there is no feedback provided to the generator about the achieved coverage. In the random approach, the generator just produces random programs and to

```
cvp_instructions : coverpoint m_transaction_h.m_instruction{
  illegal_bins unknown =
  {codix_risc_decoder::UNKNOWN};
  option.weight = 5;
}

inst_after_inst: coverpoint m_transaction_h.m_instruction {
  bins inst_after_inst[] = ([0:$] => [0:$]);

  // illegal or unimportant sequences are excluded
  illegal_bins unknown_trans1 =
  ([0:$] => codix_risc_decoder::UNKNOWN);
  illegal_bins unknown_trans2 =
  (codix_risc_decoder::UNKNOWN => [0:$]);
  ...
}
```

Figure 9.11: Instruction coverage scenarios for Codix RISC.

achieve reasonable coverage, considerable amount of them must be applied to the processor during verification. To optimize this approach, we incorporated the GA algorithm that reads feedback about the achieved coverage from verification and adds some additional constraints to the generator. In particular, constraints that are added restrict the size of programs (100 - 1000 instructions) and define the probability with which every instruction is generated to the program. All in all, the generator works with the original basic set of constraints, plus with the probability and size constraints which values are modified by the GA. Figure 9.12 illustrates how the size and probability constraints are encoded in the structure of chromosome and how they are processed by the generator.

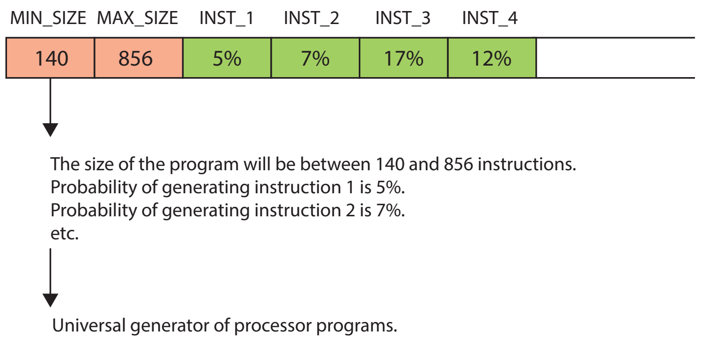

MIN_SIZE
MAX_SIZE
INST_1
INST_2
INST_3
INST_4
140
856
5%
7%
17%
12%
The size of the program will be between 140 and 856 instructions.
Probability of generating instruction 1 is 5%.
Probability of generating instruction 2 is 7%.
etc.
Universal generator of processor programs.

Figure 9.12: The structure of the chromosome for the Codix RISC processor.

Again, the UVM environment is extended by the components of the GA (Figure 9.13). The *GA test* represents the core of the algorithm. It produces chromosomes with encoded constraints that are propagated to the generator through the *Chromosome Sequencer*. The generator then generates one program per one chromosome. This is a difference in comparison with ALU, because in ALU, more stimuli were generated according to the settings in the chromosome and then propagated through the *Transaction Sequencer* and the *Driver* to the DUV. For the Codix RISC processor, generated programs are directly loaded to the program memory of the processor using the *Program Loader* component and the verification is started immediately.

In the experiments, we compared the effectiveness of the GA optimization to two standard approaches. In the first approach, the benchmark programs are used for the verification of the pro-

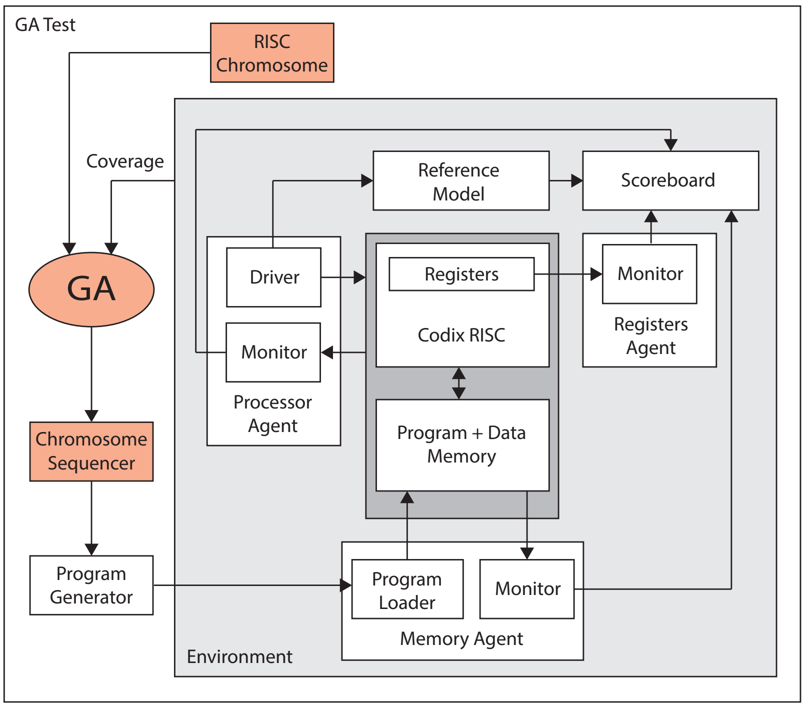


Figure 9.13: The UVM verification environment with GA classes for the Codix RISC processor.

cessor. In the second approach, the universal generator of random programs is used. The proposed GA-driven approach also uses the universal generator of random programs, but adds additional constraints to the generator encoded in the chromosome. The following graphs demonstrate the results of the experiments. The first graph in Figure 9.14 compares average values from 20 different measures for every approach (using various seeds), the second graph in Figure 9.15 compares maximum values. In both graphs, the x-axis represents the number of evaluated programs on the Codix RISC processor and the y-axis the achieved total coverage.

It can be seen that the maximum coverage that was achieved by 1000 benchmark programs is 89.2%, the average was 88%. The maximum coverage that was achieved by 1000 random programs is 97.3%, which is also the average, as every run was able to gain this value. The GA-driven random generator was producing random programs for 20, 50 and 100 generations (the 100 generations scenario is captured in the graph). The size of the population was always the same, 10 chromosomes. For the experiment with 20 generations, the maximum coverage achieved was 98.1% and the average was 97.78%. For the experiment with 50 generations, the maximum coverage achieved was 98.79% and the average was 98.72%. For the experiment with 100 generations, the maximum coverage achieved was 98.91% and the average was 98.89%. It means that for all experiments with the GA optimization, we were able to achieve better coverage. It is also important to mention other great advantage of this approach. When assembling the programs that were generated for the best chromosome in every generation, we can get a set of very few programs but with very good coverage. So for example, for the GA-optimization running 20 generations we can get the best 20 programs from every generation that are able to achieve 98.1% coverage. These programs can be further used for regression testing effectively.

The time cost of the GA optimization was as follows. Evaluating one program in simulation took in average 12.626 seconds, generating one program around 1 second and preparing new population 0.095 second. Experiments ran on Intel Core i5 CPU 3.33 GHz, 8 GB of RAM and the ModelSim simulator.

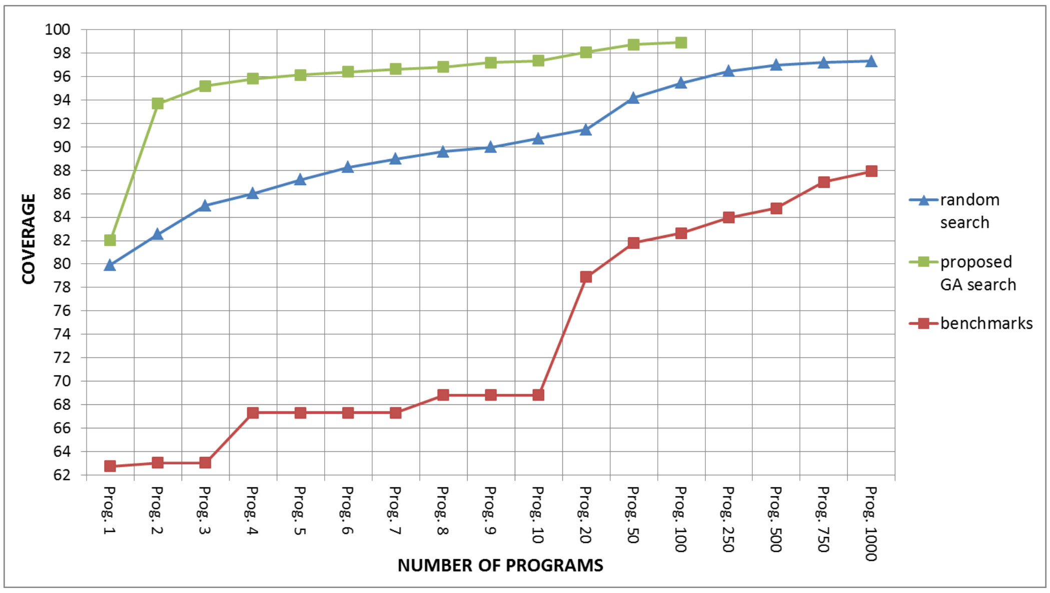


Figure 9.14: The comparison of average values from the benchmark programs, the random programs and the GA-driven generated programs.

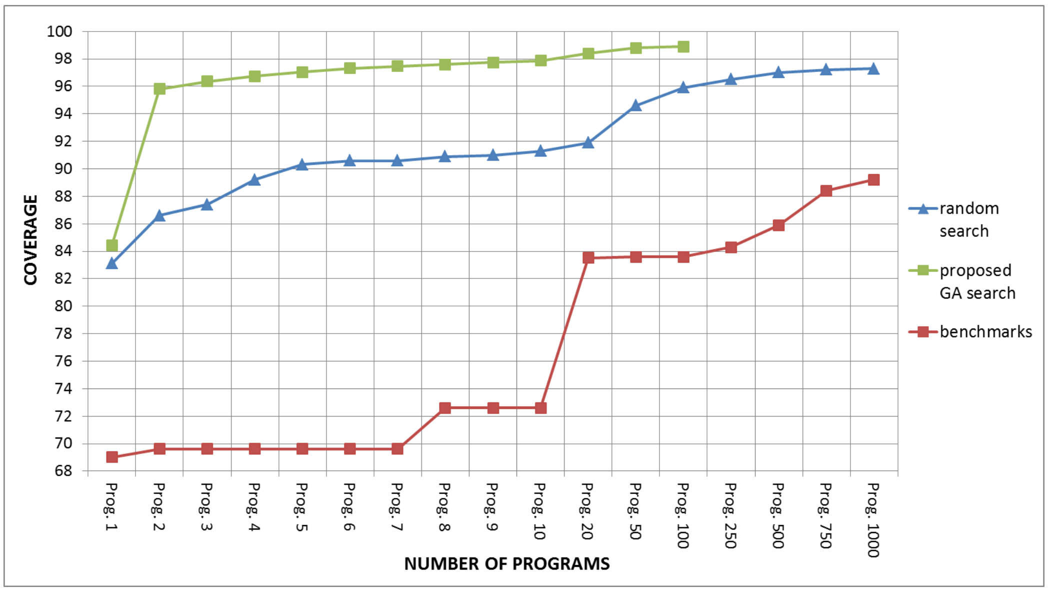


Figure 9.15: The comparison of maximum values from the benchmark programs, the random programs and the GA-driven generated programs.

9.6 Main Contributions of GA Automation and Optimization

In this chapter, the GA-driven CDV optimization was introduced. Two circuits were verified using this approach, a simple ALU and a complex RISC processor. In both cases, verification was fully automated and the achieved coverage was better than in the state-of-the-art approaches (the manual CDV, the random search).

The main contributions of the proposed method are:

1. It automates CDV (which is usually a manual process) and this markedly reduces the effort needed for preparing comprehensive verification stimuli.

2. It optimizes reaching coverage closure of measurable properties in comparison with the state-of-the-art methods and this improves the productivity rapidly.

3. It can be integrated into the standard UVM environment while the integration was optimized in a way that the interference to the UVM components is minimal.

4. GA serves unconventionally as an optimizer which is running in the background of the functional verification process. It means that in contrast to the typical application of GA, it is not determined only for searching the best candidate solution.

5. Profitable values of GA parameters were found for the CDV domain so it is not necessary to tune them for verification of various circuits.

In the future work, we will perform more experiments with elitism, as we currently use elitism set to 1, which means that only one best chromosome is propagated to the next population. Also, we will focus more on verification of ASIPs, because the proposed approach seems to be promising in this area. Thus, we will integrate it into the standard design and verification flow of ASIPs.

Chapter 10

Optimization of Regression Suites

Regression test suites are necessary to ensure that changes made to the system after bug fixes or reimplementation have not corrupted the intended functionality. Common methods of regression testing include rerunning previously completed tests and checking whether the system behavior has changed. To perform such testing effectively in the matter of time, a systematic selection of an appropriate minimum of tests is needed. The question is, how to create small and optimal regression suite for a system and be sure that it checks all the important properties of the system?

In 2015 we published a coverage-directed optimization algorithm for creating optimized regression suites from verification stimuli that were evaluated in an UVM-based verification environment [83]. The aim of the optimization was to effectively eliminate the redundancy introduced by the randomness of the generated stimuli but to preserve the coverage that was achieved in verification. Preserving the coverage guarantees that the optimized test suite will check properly all the key functions and properties of the system while running much faster and thus being more suitable for regression testing. It is important to mention that the reason why we did not focus on optimizing the random generator itself (in order not to produce so many redundant stimuli) is that in the standard process of functional verification redundancy of stimuli is a beneficial factor [90], because properties of the system are checked repeatedly. But after this phase, the redundancy is not needed anymore, so it is good to have regression tests that are effectively prepared and are running faster. Therefore, we decided to apply our optimization after the first phase of verification. Moreover, the already created verification environment can be reused for running regression tests so it is not necessary to utilize a separate approach for that purpose.

The proposed optimization algorithm is open-source, cooperates easily with the UVM-based verification and does not depend on the hardware system that is verified. Despite we were not able to compare our approach to the solutions mentioned in 5.2.4 (they either focus on different optimization problems or use confidential algorithms), we provided general improvement statistics.

10.1 Optimization Problem

The optimization problem that need to be solved during optimization of stimuli is not straightforward. Some properties to be covered can be sequential i.e. they need a specific sequence of input stimuli. Let us demonstrate this on a simple fragment of the finite-state automaton in Figure 10.1.

States of the automaton represent some verification scenarios, transitions represent processing of input stimuli. We usually select only some states to be covered (they reflect verification targets), for example, B, E, H in Figure 10.1. It can be seen that many transitions do not change the state of the automaton, for example b, l, n. Therefore, stimuli processed in these transitions are redundant.

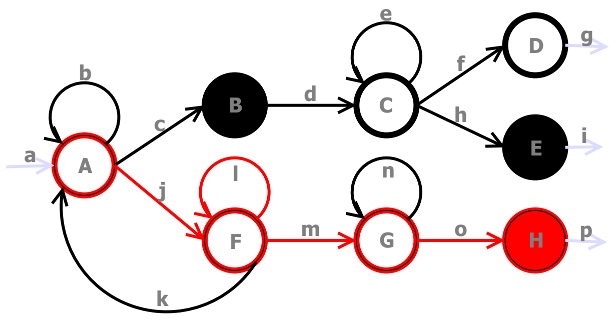


Figure 10.1: A fragment of an automaton that demonstrates sequential properties.

As the generator does not know this automaton (it can be very complex), it generates a lot of redundant stimuli. From the automaton, followings facts can be deduced:

1. Online methods building regression tests by adding just the stimulus that increased coverage cannot work. It is because one stimulus is often not enough but the whole sequence is needed. Without the automaton, we cannot determine all important stimuli.

2. The optimization that tries to remove stimuli one-by-one according to some deterministic algorithm is insufficient as well. It would either very likely suffer from huge time or space requirements or, in some special cases, it would be unable to find an optimal solution at all.

When considering these facts, an optimization algorithm with a heuristic is needed which removes more than one stimulus per one iteration so the computational overhead is reasonable. We decided to use an evolutionary algorithm for this purpose, based on the comparison of the optimization algorithms in Section 4.4.

10.2 Evolutionary Optimization of Regression Test Suites

The survey of the proposed optimization technique follows; it is divided into several steps:

1. Run the UVM-based functional verification for a selected DUV and collect stimuli until the threshold in coverage is reached.

2. Optimize the collected stimuli by the proposed technique.

3. Rerun functional verification for the optimized suite during regression testing.

10.2.1 Core of the Optimization Technique

The optimization technique incorporates GA as the main optimization tool. When adjusting the basic GA for the given problem, different lengths of candidate solutions are used (different number of stimuli is removed) and genetic operators (mutation and crossover) are customized, so they allow the length reduction. The adapted GA is designed as follows:

Initial candidate solution. An initial candidate solution contains a collected sequence of stimuli from functional verification achieving the desired coverage threshold. Initial candidate solutions differ in lengths depending on the seed used in the constraint-random generator.

Candidate solution (encoded as a chromosome). Candidate solutions are represented by sub-sequences of stimuli from the original sequence.

Mutation. Mutation is proposed in two different ways:

1. *type delete*: removes some stimuli from the regression suite. In the example of candidate solution, if the algorithm randomly chooses the j-indexes 1 and 2 then the candidate solution X in Example 10.1 would be mutated to the X' in Example 10.2.

$$X : v_1^1 v_1^2 v_1^3 v_1^4 v_1^5 \tag{10.1}$$

$$X' : v_1^3 v_1^4 v_1^5 \tag{10.2}$$

2. *type swap*: the mutual swap of two stimuli in the sequence (the ordering of stimuli is important).

Crossover. With a specified probability, sub-sequences of stimuli are swapped between two candidate solutions. The fact that the lengths of candidate solutions are not fixed and their structure is very simple, gives us an opportunity to present more liberal approach. In this approach, the number of swapped stimuli between two candidates does not need to be the same. Also, the positions of swapped stimuli can vary. Now, let us have a look at the example of the crossover. Let the two candidate solutions be sub-sequences of length 7 and 3. During the crossover, the child candidate solutions X' and Y' are created from parent candidate solutions X and Y, where v_i^j is the j-th stimulus of the i-th parent. In the given Example 10.3 the algorithm has randomly chosen to swap stimuli with j-indexes 3-5 of parent X and with j-index 2 of parent Y. The mark $|$ symbolizes the position of crossover.

$$X : v_1^1 v_1^2 | v_1^3 v_1^4 v_1^5 | v_1^6 v_1^7 \quad Y : v_2^1 | v_2^2 | v_2^3 \quad X' : v_1^1 v_1^2 v_2^2 v_1^6 v_1^7 \quad Y' : v_2^1 v_1^3 v_1^4 v_1^5 v_2^3 S \tag{10.3}$$

Fitness value. The fitness value depends on two requirements:

1. If the candidate solution does not reach the coverage threshold, it should be greatly disadvantaged, so it will not participate in the next generation.
2. The less simulation runtime the candidate solution consumes, the better, because the simulation runtime is the most important factor in the optimization.

According to these requirements, we defined the fitness function as follows. When the coverage threshold is reached, the fitness function returns the number of stimuli contained in the candidate solution as the fitness value (the lower the fitness value the better). If the coverage threshold is not achieved, a high constant value (disadvantageous) is returned.

For this purpose, the interaction of GA and the verification environment is needed (see Figure 10.2). GA runs functional verification for every newly created candidate solution (the sequence of stimuli is written to *file.txt* file). The coverage of the selected properties (coverpoints in Figure 10.2) for these stimuli is measured and returned to GA, where the fitness value is computed according to the coverage.

Selection. The tournament selection algorithm is used.

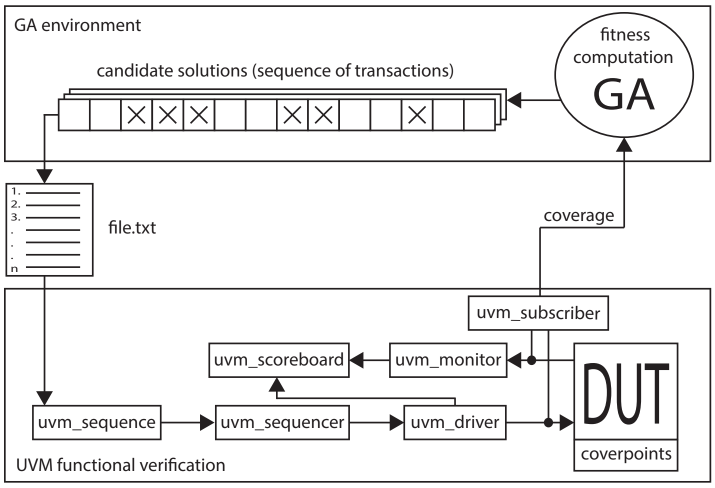


Figure 10.2: Evaluation of candidate solutions in UVM verification.

Termination condition. As a termination condition, the inability to find the new best candidate solution through several generations is considered.

Moreover, two improvements for the overall optimization process efficiency were implemented. The first improvement allows a repeated reproduction and mutation of a candidate solution until the coverage threshold is reached. Of course, just several attempts for each candidate solution are allowed. The second improvement is in the adaptation of GA. This improvement is based on the assumption that as the population evolves, creating a new offspring achieving the coverage threshold gets harder. It is caused by parents containing a small number of redundant stimuli. As a result, probabilities of genetic operators should be lowered.

Now, let us examine the presented assumption that forming a candidate solution with satisfactory coverage is more likely to fail in case of parents with better fitness values, in more detail. Given assumption relies on the following idea. Let p be the probability of *type delete* mutation. Then after application of this type of mutation on one candidate solution, about $100 \cdot p\%$ stimuli are deleted from the regression suite. Let the regression suite, we want to find, consist of m stimuli. Furthermore, let there be two candidate solutions N and O with number of stimuli n and o so that $m < n < o$. This means that the number of redundant stimuli for candidate solution N is $n - m$ and for O is $o - m$. It is obvious that $n - m < o - m$, i. e. candidate solution O is more redundant than N. Next, it can be seen that ratio of redundant stimuli to the total number of stimuli for the candidate solution N is $(n-m)/n$, for O is $(o-m)/o$ and $(n-m)/n < (o-m)/o$. When the same percentage p of stimuli is deleted in N and O, which for N means $n \cdot p$ stimuli and for O $o \cdot p$ stimuli, whereas $n \cdot p < o \cdot p$, more stimuli are deleted from the more redundant candidate solution. Probabilities p_n and p_o define that all deleted stimuli are redundant and this is described by Formulas 10.4 a 10.5. For creation of these formulas we utilize the field of combinatorics in mathematics, to be more specific combinations without repetition are used. In the given formulas, the numerator stands for the number of different combinations of redundant stimuli. The denominator represents all combinations of stimuli whether they are redundant or not. The number of elements in one combination is equal to the number of deleted stimuli.

$$p_n = \frac{\frac{(n-m)!}{(n-m-n\cdot p)!}}{\frac{n!}{(n-n\cdot p)!}} \quad (10.4)$$

$$p_o = \frac{\frac{(o-m)!}{(o-m-o\cdot p)!}}{\frac{o!}{(o-o\cdot p)!}} \quad (10.5)$$

Because $n < o$, $p_n < p_o$ and the application of *type delete* mutation without lowering its probability is getting more difficult during the following generations, similarly as finding better candidate solutions. The necessity of adaptation for other genetic operators can be shown in similar way.

According to the given assumption, we can tell that it is effective to start GA with higher probabilities of genetic operations and as the average fitness value of the population improves, the probabilities of genetic operations should be decreased. One might also want to set the minimal permitted probabilities of the genetic operations. We believe that this assumption is specific for this optimization problem.

Both presented improvements depend on each other. When the population is overfilled with candidate solutions with unsatisfactory coverage, instead of many attempts for reproduction, the probabilities of genetic operators should be lowered. As a consequence, the next generations will probably be refilled with solutions that accomplish the coverage threshold. To summarize, the final pseudo-code of the improved GA is presented in Algorithm 5.

Algorithm 5: The proposed GA for creation of optimal regression tests.

```
Obtain the initial population from verification;
Evaluate the fitness of the candidate solutions by running verification and evaluating
their coverage;
repeat
    for the number of new candidate solutions in generation do
        Select 2 parents;
        Generate a new candidate solution using crossover and mutation;
        Evaluate the fitness of the new candidate solution by running verification and
        evaluating coverage;
        while the candidate solution does not accomplish the coverage threshold or the
        number of maximal attempts for recreating was not reached do
            Recreate candidate solution;
            Evaluate the fitness of the new candidate solution by running verification
            and evaluating coverage;
        end
    end
    Create a new generation;
    if the number of candidate solutions with coverage < threshold is higher than the
    limit then
        Decrease probabilities of crossover and mutation;
    end
until the maximal number of generations was reached or better candidate solution than
its ancestors was not found during the specified number of generations;
```

10.2.2 GA Parameters

It is common in the application of GA to some domain to find tuning of its parameters in order to work well. We performed extensive experiments with various settings of GA parameters and the following seem to be beneficial: the population size set to 2, the probability of the *type delete* mutation set to 0.22, the probability of the *type swap* mutation set to 0.01, the maximal number of recreations set to 2, the ratio of adaptation set to 0.5 and a very low probability of crossover. Elitism seemed to be also beneficial; we always propagate one best candidate solution to the next generation.

10.3 Experimental Results - ALU Case Study

This case study shows the application of the proposed optimization technique to the sequential arithmetic-logic unit (ALU). However, we believe that the proposed algorithm is independent on the circuit that is verified. For every scenario we take the test suite generated from functional verification and shrink its size. And this does not depend on the characteristics of the DUV.

For ALU, we were able to define **28 coverage scenarios** with **1989 functional properties**. The aim of verification is to cover all of these properties. The aim of coverage-directed optimization is to remove stimuli which are redundant while the coverage level remains the same.

Experiments were running on Intel Xeon E5-2640, 2.5GHz processor, the UVM-based verification was executed in the ModelSim simulator from Mentor Graphics. After running the optimization process, we gained several interesting results which are summarized in Table 10.1.

The average fitness value of the initial population	5654
The fitness value of the best solution	316
The overall regression optimization [%]	94.41
The resulting coverage of the best solution [%]	100

Table 10.1: The optimization results.

The main result is that the presented technique reduced the original sequence of stimuli to the 5.59 % of its original size while preserving coverage at 100%. Figure 10.3 demonstrates the dependency between the optimization runtime and the level of optimization. Note that for more optimal solutions it is more difficult to remove another stimuli. It is important to point out that the optimization process can be stopped whenever the optimization is satisfactory (in few minutes after the started the optimization level was over 50%). On the contrary, it is also possible that even more optimal solution than the one presented here can be found if more generations were allowed or population size was bigger. But in both cases, the optimization run is longer.

10.4 Main Contributions of Regression Suites Optimization

In this chapter we presented a technique for optimization of test suites taken from functional verification in order to create optimal regression suites. For optimization we used the genetic algorithm. The technique works well in the environments where a large number of random or automatically generated stimuli are applied and the quality and progress of verification are measured based on the actual coverage of the system that is verified. The main contributions of this technique are:

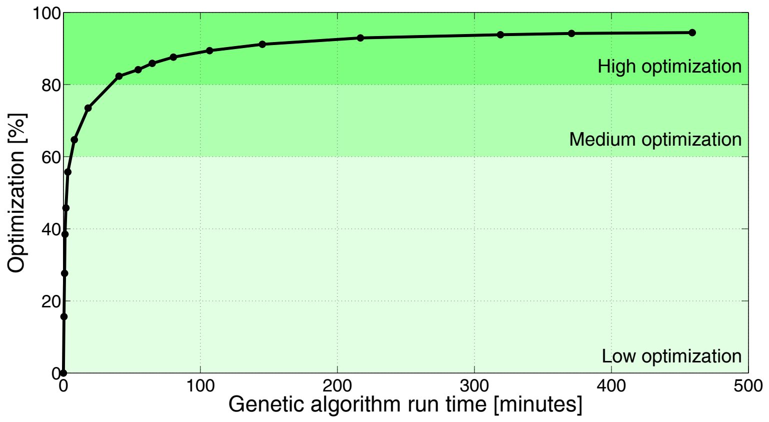


Figure 10.3: The dependency between the optimization runtime and the level of optimization.

- It eliminates the redundancy in the original suite of stimuli so the optimized suite will be running much faster in simulation.
- It preserves the same level of coverage as was achieved by the original suite of stimuli.
- It reuses already created verification environment also for running regression tests so it is not necessary to utilize a separate approach for that purpose.

Our experiments have shown that when verifying hardware systems using functional verification, the redundancy of the generated stimuli is quite high for regression testing as we were able to eliminate 94.4% of the original stimuli for the ALU test case. Furthermore, we realized that the evaluation of GA candidate solutions is very time-consuming, not because of the GA process itself but because of the necessity to launch an RTL simulator for fitness value computations. In order to reduce optimization time we intend to implement the following features in our future work:

1. To manipulate with fix-sized bitmaps that correspond to the stimuli in the sequence. 0 for stimuli that are redundant and should be removed and 1 will represent stimuli that are kept. GA will manipulate only with these bitmaps and the genetic operators remain simpler.
2. To find a way how to minimize launching of the RTL simulator when evaluating candidate solutions.
3. To evaluate candidate solutions of one population in parallel because they do not depend on each other.

Until now, we already implemented the first two features from the list, but the results were not published yet. We decided to employ the optimization technique for preparing effective verification test suites for ASIPs. Of course, ASIPs are much more complex systems than ALU is and we are not optimizing the number of input stimuli that are directly applied on the input ports of DUV, but we are optimizing the number of programs that are loaded to the program memory of an ASIP during verification.

The first feature was implemented in a way that the chromosome is represented by a bit string of a constant length, where each bit represents one program from the original suite of programs used in functional verification. The value 1 of a bit in the bit string stands for the program that remains in the regression suite and the value 0 of a bit in the bit string stands for the program that was optimized away from the regression suite.

As for the second feature, for the experiments with ASIPs we decided to eliminate the bottleneck of launching the RTL simulator when evaluating candidate solutions by following steps:

1. In the standard verification phase, the information about coverage is stored to the coverage database for every evaluated program.
2. In the optimization phase, for every program that actively participates in the candidate solution (marked by the 1 value in the bit string of chromosome), the coverage information stored in the coverage database is merged to the coverage databases of other active programs. As a result, the summarized coverage database is gained for one candidate solution. For storing and merging coverage databases, we use the QuestaSim simulation tool from Mentor Graphics.

The most important point of the presented improvement is that merging coverage databases is much faster operation than launching the simulator and evaluating the candidate solutions in simulation. Therefore, we decided to integrate this improvement into the optimization technique to be used in all further experiments.

The results of experiments have shown that the optimization technique reduced the number of programs to the 0.522 % of the original size and the resulting statement coverage remained at 97.2%, branch coverage at 94.4% and functional coverage at 98.0%. What is more important, the simulation runtime of the optimized regression suite is much shorter and was optimized by 98.1 %.

Figure 10.4 demonstrates the dependency between the achieved level of optimization of the regression suite (the y axis in the graph) and the time of optimization (the x axis in the graph). The level of optimization (O) is computed from the simulation runtime of the best up-to-now candidate solution (C) and from the simulation runtime of the original suite of programs (S) according to Equation 10.6.

$$O[\%] = (1 - C/S) \cdot 100 \tag{10.6}$$

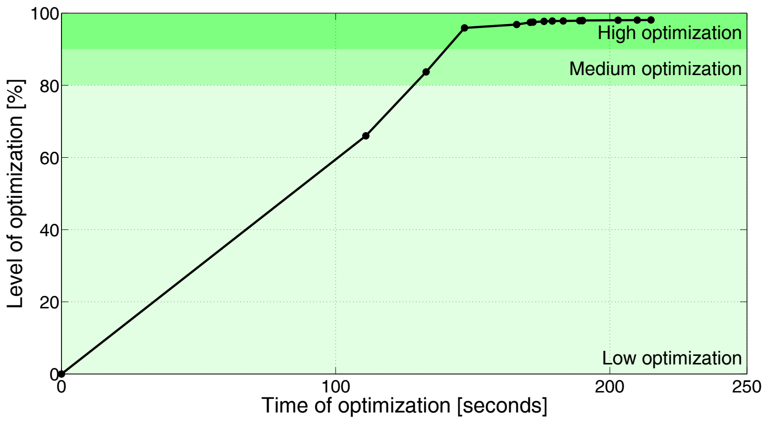


Figure 10.4: The dependency between the optimization runtime and the level of optimization.

Chapter 11

Conclusions

In this Ph.D. thesis, four optimization techniques are presented that help to optimize the process of functional verification.

The first technique targets the time bottleneck of slow simulation when verifying real-world systems. As the functional verification is based on simulation, it suffers from the fact that simulation of inherently parallel hardware systems is extremely slow when compared to the speed of real hardware. When also the verification overhead is taken into account, like preparing various test cases, measuring coverage, running the reference model, the project time expands a lot and endangers the delivery of the product to the market. Our optimization technique accelerates verification runs by moving some or all parts of the verification testbench to the FPGA accelerator. Our acceleration framework is called HAVEN and offers five testbed architectures that represent a trade-of between the speed of verification and the internal visibility of the behaviour of DUV. In practice it means that at the beginning of verification it is good to start with the testbed architecture where only DUV is moved to the FPGA and where a good signals visibility is available. Later on we recommend to move more verification components to the FPGA and evaluate millions of stimuli in order to reach all coverage goals and be sure that also corner cases are checked. In our experiments with simple FIFO buffer and complex hash generator architectures (HGEN) we have shown that very good acceleration ratios can be achieved by HAVEN (over 100,000 x for the most complex system HGENx16). As the HAVEN framework is open-source, it was later used also for accelerating verification runs of processors (ASIPs) and stands for the basic part of the platform for measuring fault resilience of electro-mechanical hardware applications.

The second optimization technique focuses on automated generation of OVM/UVM based verification environments. As we realized that many parts of the testbench components are replicated between different projects and just modified for specific DUVs, we implemented an engine that takes high-level specification of the DUV and is able to generate not only complete testbench components, but also the reference model and the interconnection between the DUV and the testbench from such specification. In this case, our experimental DUV systems were application-specific processors (ASIPs) for which the high level specification is written in the ADL language called CoDAL. Using our engine, a complete verification environment is generated in the order of seconds. When we compare it to the manual work that usually takes around two weeks, it is quite significant improvement. Despite this work was experimentally evaluated just on ASIPs, we believe that the main principles can be reused also for other systems (not only processors) whose specification can be provided at the higher level of abstraction.

The third technique is used for automation and optimization of CDV. CDV helps the verification engineers to estimate the moment when the process of functional verification can be ended, because it provides a feedback about the features that were checked in DUV. In the thesis, the mostly used coverage metrics were mentioned together with the steps how to build a comprehensive coverage model. Coverage metrics used in the model form the coverage space that needs to be explored during verification. There are several algorithm for the search-space exploration and we selected the evolutionary exploration as the most suitable one for CDV. We created an adapted genetic algorithm (this algorithm belongs to the class of evolutionary algorithms) that takes the coverage feedback from verification and prepares constraints for the pseudo-random generator which is used for generating stimuli. This whole process is running fully automatically, without any manual intervention from the user. After several generations of the evolution, the genetic algorithm aims at the unexplored areas of the coverage search space and as a consequence, the verification iterates to the coverage goals much faster then by the standard approach that utilizes the undriven pseudo-random generator. Experiments were performed on a simple ALU and on the RISC processor and in both cases, the genetic algorithm running in the background of the verification process was recognized as beneficial. For ALU, the GA-driven approach reduced the number of stimuli needed for achieving 100% coverage by more than 50%. For the RISC processor, the results were much more optimistic. 100 programs produced by the GA-driven generator were able to achieve 98.91% total coverage while 1000 programs generated by the undriven generator or programs from the benchmark sets were not able to reach this coverage level at all (they achieved 97.3% and 89.2% respectively). In addition, another benefit was identified after the experimental phase. When assembling the programs that were generated for the best chromosome in every generation, we can get a set of very few programs but with very good coverage. So for example, for the GA-optimization running 20 generations we can get the best 20 programs from every generation that are able to achieve around 98.1% coverage. These programs can be further used for regression testing effectively.

Our last optimization technique is also connected to regression testing. We devised an algorithm how to optimize the stimuli used in functional verification in order to reuse them and create an optimal regression test suite. This technique works offline after the standard verification phase. It takes all the stimuli that were either randomly generated or prepared manually during verification and reduces their number without decreasing the coverage achieved by the original set. Reduction is possible due to the redundancy that is usually introduced by the pseudo-random stimuli generator which is standardly used in functional verification. For the reduction purposes, the genetic algorithm was used. The experiments were once again performed on the ALU and on the RISC processor. The results are quite promising, in the case of ALU we reduced the original sequence of stimuli to the 5.59% of its original size and in the case of the RISC processor the number of programs was reduced to the 0.522% of its original size.

11.1 Future Work

Based on our existing work about automation and optimization of CDV, we plan to create an open-source library that extends the standard UVM library by the configurable genetic algorithm. GA will primarily serve for the intelligent testbench automation. We will also create some advanced tests considering the settings of the ELITISM parameter, because it was tested only for the simplest scenario with ELITISM set to 1 (just one best candidate solution is propagated to the next generation).

As for the second direction of our future research, we will focus on neural networks and their possible application as an optimizer in functional verification. Neural networks have been used in classification and recognition problems. Consequently, they can be used to classify subsets of the DUV input stimuli that are suitable to activate the coverage points under consideration. The neural network architecture may contain an input layer that maintains the DUV inputs, an output layer to represent the logical status for each coverage scenario, and some hidden layers according to the complexity of the CDV problem. Here, a pseudo-random generator may be used as primary test generator where the tuning of neurons weights continuously changes over time according to the coverage feedback. Backward tracing can be used to extract the useful directives for each coverage point, where the neural network works as a useful test generator.

Chapter 12

Appendix A

12.1 FrameLink

FrameLink is a synchronous point-to-point frame-oriented protocol for communication between hardware components, originally developed at the Liberouter[1] project. Communication over FrameLink consists of frames (delimited by signals `SOF_N` and `EOF_N`, which stand for *start of frame* and *end of frame* respectively), which can further be composed of several parts (delimited by signals `SOP_N` and `EOP_N`, which stand for *start of part* and *end of part* respectively). The source and the destination of one FrameLink connection need to share the same `CLOCK` and `RESET` signals. Note that all control signals are active in `0`.

The actual data transfer takes place when both the source and the destination are ready to communicate, which is signalled by their `SRC_RDY_N` and `DST_RDY_N` signals; if both of these signals are active, the source sends the data and the destination receives it. More detailed description of FrameLink signals follows (`src` means that the signal is driven by the source and `dst` means that it is driven by the destination component):

- **`DATA[DATA_SIZE-1:0]`** (`src`): Vector of signals that bear data, $\mathtt{DATA\_SIZE} \in \{8, 16, 32, 64, 128, \ldots\}$.
- **`REM[REM_SIZE-1:0]`** (`src`): Sets validity of bytes in `DATA`, i.e., defines the address of the last valid byte. This signal is valid only when the control signal `EOP_N` is active. $\mathtt{REM\_SIZE} = \log_2(\mathtt{DATA\_SIZE}/8)$.
- **`SOF_N`** and **`EOF_N`** (`src`): Delimit the start and end of a frame.
- **`SOP_N`** and **`EOP_N`** (`src`): Delimit the start and end of a frame part.
- **`SRC_RDY_N`** (`src`): Signals that the source is ready to send data.
- **`DST_RDY_N`** (`dst`): Signals that the destination is ready to receive data.

[1] `http://www.liberouter.org`

Chapter 13

Appendix B

13.1 Publications and Products

Journal Publications.

- J. Podivínský, M. Šimková, O. Čekan, and Z. Kotásek. The evaluation platform for testing fault-tolerance methodologies in electro-mechanical applications. In *Microprocessors and Microsystems*, Elsevier, 2015, doi:10.1016/j.micpro.2015.05.011.

Conference Publications.

- M. Šimková, O. Lengál, and M. Kajan. HAVEN: An Open Framework for FPGA-Accelerated Functional Verification of Hardware. In *Proc. of HVC'11*, Haifa, Israel, LNCS 7261, Springer, 2012, pp. 247–253, ISSN 0302-9743.
- M. Šimková. Acceleration of Functional Verification in the Development Cycle of Hardware Systems. In *Proc. of PAD'12*, Prague, Czech Republic, CVUT, 2012, pp. 73–78, ISBN: 978-80-01-05106-1.
- M. Šimková, O. Lengál: Towards Beneficial Hardware Acceleration in HAVEN: Evaluation of Testbed Architectures. In *Proc. of HVC'12*, Haifa, Israel, LNCS 7857, Springer, 2013, pp. 266–273, ISSN 0302-9743.
- M. Šimková, Z. Přikryl, T. Hruška, Z. Kotásek. Automated Functional Verification of Application Specific Instruction-set Processors. In *Proc. IFIP Advances in Information and Communication Technology*, Heidelberg: Springer Verlag, 2013, vol. 4, no. 403, pp. 128–138. ISSN 1868-4238.
- M. Šimková, C. Bolchini, Z. Kotásek. Analysis and Comparison of Functional Verification and ATPG for Testing Design Reliability. In *Proc. of IEEE DDECS'13*, Karlovy Vary, Czech Republic, IEEE Computer Society, 2013, pp. 275–278, ISBN 978-1-4673-6133-0.
- M. Šimková. New Methods for Increasing Efficiency and Speed of Functional Verification. In *Proc. of PAD'13*, Pilsen, Czech Republic, University of West Bohemia in Pilsen, 2013, pp. 111-116. ISBN 978-80-261-0270-0.
- J. Podivínský, M. Šimková, Z. Kotásek. Complex Control System for Testing Fault-Tolerance Methodologies. In *Proc. of MEDIAN'14*, Dresden, Germany, COST, European Cooperation in Science and Technology, 2014, pp. 24–27, ISBN 978-2-11-129175-1.

- J. Podivínský, O. Čekan, M. Šimková, Z. Kotásek. The Evaluation Platform for Testing Fault-Tolerance Methodologies in Electro-mechanical Applications. In *Proc. of Euromicro DSD'14*, Verona, Italy, IEEE Computer Society, 2014, pp. 312–319, ISBN 978-1-4799-5793-4.

- M. Šimková. Application of Evolutionary Computing for Optimization of Functional Verification. In *Proc. of PAD'14*, Liberec, Liberec University of Technology, 2014, pp. 135–140. ISBN 978-80-7494-027-9.

- M. Kekelyová, M. Šimková, Z. Kotásek, T. Hruška. Application of Evolutionary Algorithms for Optimization of Regression Suites. In *Proc. of DDECS'15*, Belgrade, Serbia, IEEE Computer Society, 2015, pp. 91–94. ISBN 978-1-4799-6779-7.

- J. Podivínský, M. Šimková, Z. Kotásek. Radiation Impact on Mechanical Application Driven by FPGA-based Controller. In *Proc. of MEDIAN'15*, Grenoble, France, COST, European Cooperation in Science and Technology, 2015, pp. 13–16.

- J. Podivínský, M. Šimková, O. Čekan, Z. Kotásek. FPGA Prototyping and Accelerated Verification of ASIPs. In *Proc. of DDECS'15*, Belgrade, Serbia, IEEE Computer Society, 2015, pp. 145–148. ISBN 978-1-4799-6779-7.

- O. Čekan, M. Šimková, Z. Kotásek. Universal Pseudo-random Generation of Assembler Codes for Processors. In *Proc. of MEDIAN'15*, Grenoble, France, COST, European Cooperation in Science and Technology, 2015, pp. 70–73.

Technical Reports.

- M. Šimková, O. Lengál, and M. Kajan. HAVEN: An Open Framework for FPGA-Accelerated Functional Verification of Hardware, FIT-TR-2011-05, Brno, Czech Republic, FIT BUT, 2011, p. 16.

- M. Šimková, O. Lengál. Towards Beneficial Hardware Acceleration in HAVEN: Evaluation of Testbed Architectures, FIT-TR-2012-03, Brno, Czech Republic, FIT BUT, 2012, p. 14.

Posters and Presentations.

- M. Šimková. HAVEN: An Open Framework for FPGA-Accelerated Functional Verification of Hardware, MEMICS'2012, Znojmo, Czech republic.

- M. Šimková. Towards Beneficial Hardware Acceleration of Functional Verification, Verifying Reliability (Dagstuhl Seminar 12341), Dagstuhl, Germany, 2012.

- M. Šimková, and J. Kaštil. Verification of Fault-tolerant Methodologies for FPGA Systems, poster at First Median COST Action 2012, Annecy, France, 2012, pp. 55–58.

Software products.

- M. Šimková, O. Lengál, and M. Kajan: HAVEN: An Open Framework for FPGA-Accelerated Functional Verification of Hardware, software, 2012.

13.2 Research Projects and Grants

- Mathematical and Engineering Approaches to Developing Reliable and Secure Concurrent and Distributed Computer Systems, GAČR, GD102/09/H042, 2009–2012, completed.
- Manufacturable and Dependable Multicore Architectures at Nanoscale, COST, IC1103, 2011–2015, running.
- Advanced recognition and presentation of multimedia data, BUT, FIT-S-11-2, 2011–2013, completed.
- Methodologies for Fault Tolerant Systems Design Development, Implementation and Verification, MEYS, LD12036, 2012–2015, running.
- Application of methods and techniques of formal verification in the design of advanced digital circuits, FRVŠ MEYS, FR1086/2013/G1, 2013, completed.
- Participant of the Brno Ph.D. Talent Scholarship Programme, 2011 - 2014, completed.
- The IT4Innovations Centre of Excellence, MŠMT, ED1.1.00/02.0070, 2011-2015, running.
- Architecture of parallel and embedded computer systems. BUT, FIT-S-14-2297, 2014–2016, running.

Bibliography

[1] IEEE Standard 1800-2005 for SystemVerilog - Unified Hardware Design, Specification, and Verification Language. IEEE, 2004.

[2] The LLVM Compiler Infrastructure Project, 2015. http://llvm.org/.

[3] Thomas Bäck. Self-Adaptation in Genetic Algorithms. In *Proceedings of the First European Conference on Artificial Life*, pages 263–271. MIT Press, 1992.

[4] Thomas Bäck, David B. Fogel, and Zbigniew Michalewicz, editors. *Evolutionary Computation 1: Basic Algorithms and Operators*. IOP Publishing Ltd., Bristol, UK, UK, 1st edition, 1999.

[5] Thomas Bäck, David B. Fogel, and Zbigniew Michalewicz, editors. *Evolutionary Computation 2: Advanced Algorithms and Operators*. IOP Publishing Ltd., Bristol, UK, UK, 2000.

[6] Rico Backasch, Christian Hochberger, Alexander Weiss, Martin Leucker, and Richard Lasslop. Runtime Verification for Multicore SoC with High-quality Trace Data. *ACM Trans. Des. Autom. Electron. Syst.*, 18(2):1–26, April 2013.

[7] Janick Bergeron. *Writing testbenches using SystemVerilog*. Springer, New York, NY, 2006.

[8] Janick Bergeron, Eduard Cerny, Alan Hunter, and Andy Nightingale. *Verification Methodology Manual for SystemVerilog*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2005.

[9] M. Bose, J. Shin, E. M. Rudnick, T. Dukes, and M. Abadir. A Genetic Approach to Automatic Bias Generation for Biased Random Instruction Generation. In *Proc. of the Congress on Evolutionary Computation*, pages 442–448. IEEE, 2001.

[10] Breker. TrekSoC, 2015. http://www.brekersystems.com/products/treksoc/.

[11] Jason Brownlee. *Clever Algorithms: Nature-Inspired Programming Recipes*. Lulu.com, 1st edition, 2011.

[12] Cadence. Palladium, 2015. http://www.cadence.com/products.

[13] Cadence. Transaction-based Acceleration (TBA), 2015. http://www.cadence.com/products.

[14] Cadence. Verification IP, 2015. http://ip.cadence.com/ipportfolio/verification-ip.

[15] Ondřej Čekan, Marcela Šimková, and Zdeněk Kotásek. Universal Pseudo-random Generation of Assembler Codes for Processors. In *Proceedings of The Third Workshop on Manufacturable and Dependable Multicore Architectures at Nanoscale*, pages 70–73. COST, European Cooperation in Science and Technology, 2015.

[16] Santanu Chattopadhyay and K. Sudarsana Reddy. Genetic Algorithm Based Test Scheduling and Test Access Mechanism Design for System-on-Chips. In *Proc. of the International Conference on VLSI Design*, pages 341–346. IEEE Computer Society, 2003.

[17] Codasip. Codasip ASIP Cores, 2015. https://www.codasip.com/products/cores/.

[18] Codasip. Codasip Studio, 2015. https://www.codasip.com/products/.

[19] Ben Cohen, Srinivasan Venkataramanan, and Ajeetha Kumari. *Using PSL/Sugar for Formal and Dynamic Verification*. VhdlCohen Publishing, 2004.

[20] F. Corno, E. Sanchez, and G. Squillero. Automatic Test Program Generation: a Case Study. In *Design and Test of Computers*, pages 102–109. IEEE, 2004.

[21] Charles Darwin. *The origin of species by means of natural selection, or, The preservation of favoured races in the struggle for life / by Charles Darwin*. John Murray London, 6th ed. with additions and corrections. edition, 1898.

[22] Sayantan Das, Rizi Mohanty, Pallab Dasgupta, and P. P. Chakrabarti. Synthesis of SystemVerilog Assertions. In *Proceedings of the Conference on Design, Automation and Test in Europe: Designers' Forum*, DATE '06, pages 70–75, 3001 Leuven, Belgium, 2006. European Design and Automation Association.

[23] Lawrence Davis. Adapting Operator Probabilities in Genetic Algorithms. In *Proceedings of the Third International Conference on Genetic Algorithms*, pages 61–69, San Francisco, CA, USA, 1989. Morgan Kaufmann Publishers Inc.

[24] R. Drechsler, C. Chevallaz, F. Fummi, A.J. Hu, R. Morad, F. Schirrmeister, and A. Goryachev. Future SoC Verification Methodology: UVM Evolution or Revolution? In *Proc. of the Conference on Design, Automation & Test in Europe*, pages 1–5. IEEE, 2014.

[25] Duolog. Tools, 2015. http://www.duolog.com/products/.

[26] Agoston E. Eiben and J. E. Smith. *Introduction to Evolutionary Computing*. SpringerVerlag, 2003.

[27] P. Faye, E. Cerny, and P. Pownall. Improved Design Verification by Random Simulation Guided by Genetic Algorithms. In *Proc. of the ICDA/APChDL Conference*, pages 456–466. IFIP World Computer Congress, 2000.

[28] Shai Fine and Avi Ziv. Coverage Directed Test Generation for Functional Verification Using Bayesian Networks. In *In Proceedings of the 40th Design Automation Conference*, pages 286–291, 2003.

[29] David B. Fogel. *Evolutionary Computation: The Fossil Record*. Wiley-IEEE Press, 1998.

[30] L.J. Fogel, A.J. Owens, and M.J. Walsh. *Artificial intelligence through simulated evolution*. Wiley, Chichester, WS, UK, 1966.

[31] Mark Glasser. *Open Verification Methodology Cookbook*. Springer, 2009.

[32] Patrice Godefroid and Sarfraz Khurshid. Exploring Very Large State Spaces using Genetic Algorithms. In *Proc. of the Tools and Algorithms for the Construction and Analysis of Systems*, pages 266–280. LNCS 2280. Springer-Verlag, 2002.

[33] O. Goloubeva, M. Sonza Reorda, and M. Violante. Automatic Generation of Validation Stimuli for Application Specific Processors. In *Proc. of the IEEE Conference on Design Automation and Test in Europe*, pages 188–193, Washington, DC, USA, 2007. IEEE Computer Society.

[34] Mentor Graphics. Infact, 2015. http://www.mentor.com/products/fv/infact/.

[35] Mentor Graphics. Veloce2, 2015. http://www.mentor.com/products/fv/emulation-systems/veloce/.

[36] Mentor Graphics. Verification IP, 2015. http://www.mentor.com/products/fv/verification-ip.

[37] Wilson Research Group. The 2014 Wilson Research Group Functional Verification Study. Published online, 2015.

[38] Yang Guo, WanXia Qu, Tun Li, and Sikun Li. Coverage Driven Test Generation Framework for RTL Functional Verification. In *Proc. of the IEEE Conference on Computer-Aided Design and Computer Graphics*, pages 321–326. IEEE, 2007.

[39] Ali Habibi, Sofiène Tahar, Amer Samarah, Donglin Li, and O. Ait Mohamed. Efficient Assertion Based Verification Using TLM. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '06, pages 106–111, 3001 Leuven, Belgium, 2006. European Design and Automation Association.

[40] Amir Hekmatpour, James Coulter, and Azadeh Salehi. FoCuS: A Dynamic Regression Suite Generation Platform for Processor Functional Verification. In Gongzhu Hu, editor, *Computers and Their Applications*, pages 373–378. ISCA, 2005.

[41] Renate Henftling, Andreas Zinn, Matthias Bauer, Martin Zambaldi, and Wolfgang Ecker. Re-Use-Centric Architecture for a Fully Accelerated Testbench Environment. In *Proc. of the Design Automation Conference*, pages 372–375. ACM, 2003.

[42] Jürgen Hesser and Reinhard Männer. Towards an Optimal Mutation Probability for Genetic Algorithms. In *Proceedings of the 1st Workshop on Parallel Problem Solving from Nature*, PPSN I, pages 23–32, London, UK, UK, 1991. Springer-Verlag.

[43] John H. Holland. Genetic Algorithms and the Optimal Allocation of Trials. *SIAM Journal on Computing*, pages 88–105, 1973.

[44] John H. Holland. *Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control and Artificial Intelligence*. MIT Press, Cambridge, MA, USA, 1992.

[45] HT-LAB. Mersenne Twister, MT32: Pseudo Random Number Generator for Xilinx FPGA., 2007. http://www.ht-lab.com/freecores/mt32/mersenne.html.

[46] Chung-Yang Huang, Yu-Fan Yin, Chih-Jen Hsu, Thomas B. Huang, and Ting-Mao Chang. SoC HW/SW Verification and Validation. In *Proc. of the Asia and South Pacific Design Automation Conference*, pages 297–300. IEEE, 2011.

[47] Invea-Tech. NetCOPE, 2015. https://www.invea.com/en/products-and-services/fpga-development-kit/netcope.

[48] Bob Jenkins. The Lookup2 hash algorithm. http://burtleburtle.net/bob/c/lookup2.c.

[49] B. F. Jones, H. H. Sthamer, and D. E. Eyres. Automatic Structural Testing Using Genetic Algorithms. *Software Engineering Journal*, 11:299 – 306, 1996.

[50] Mohammad Reza Kakoee, Mohammad Riazati, and Siamak Mohammadi. Generating RTL Synthesizable Code from Behavioral Testbenches for Hardware-Accelerated Verification. In *Proc. of the Digital System Design Conference*, pages 714–720. IEEE Computer Society, 2008.

[51] Young-Il Kim and Chong-Min Kyung. Automatic Translation of Behavioral Testbench for Fully Accelerated Simulation. In *Proceedings of the 2004 IEEE/ACM International Conference on Computer-aided Design*, ICCAD '04, pages 218–221, Washington, DC, USA, 2004. IEEE Computer Society.

[52] John R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, Cambridge, MA, USA, 1992.

[53] John R. Koza. *Genetic Programming II*. MIT Press, 55 Hayward Street, Cambridge, MA, USA, 1994.

[54] IBM Haifa Research Lab. Meteor, 2015. https://www.research.ibm.com/haifa/projects.

[55] Zheng Li, Mark Harman, and Robert M. Hierons. Search Algorithms for Regression Test Case Prioritization. *IEEE Trans. Softw. Eng.*, 33(4):225–237, April 2007.

[56] Pinaki Mazumder and Elizabeth M. Rudnick, editors. *Genetic Algorithms for VLSI Design, Layout and Test Automation*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1999.

[57] Zbigniew Michalewicz and David B. Fogel. *How to Solve It: Modern Heuristics*. Springer Berlin Heidelberg, 2004.

[58] A. Molina and O. Cadenas. Functional Verification: Approaches and Challenges. *Latin American Research*, pages 65–68, 2007.

[59] Douglas Perry and Harry Foster. *Applied Formal Verification*. McGraw-Hill, 2005.

[60] Andrew Piziali. *Functional Verification Coverage Measurement and Analysis*. Springer Publishing Company, Incorporated, 1st edition, 2007.

[61] Jakub Podivínský, Marcela Šimková, Ondřej Čekan, and Zdeněk Kotásek. FPGA Prototyping and Accelerated Verification of ASIPs. In *Proc. of the IEEE International Symposium on Design and Diagnostics of Electronic Circuits and Systems*, pages 145–148. IEEE, 2015.

[62] Jakub Podivínský, Marcela Šimková, Ondřej Čekan, and Zdeněk Kotásek. The Evaluation Platform for Testing Fault-Tolerance Methodologies in Electro-mechanical Applications. *Microprocessors and Microsystems*, 2015.

[63] Zdeněk Přikryl. Advanced Methods of Microprocessor Simulation. *Information Sciences and Technologies Bulletin of the ACM Slovakia*, 3(3):1–13, 2011.

[64] Liberouter Project. COMBO LXT Card. https://www.liberouter.org/combo-lxt/.

[65] Liberouter Project. NetCOPE. https://www.liberouter.org/technologies/netcope/.

[66] I. Rechenberg. *Evolutionsstrategie: optimierung technischer systeme nach prinzipien der biologischen evolution*. Frommann-Holzboog, 1973.

[67] Grzegorz Rozenberg, Thomas Bäck, and Joost N. Kok. *Handbook of Natural Computing*. Springer Berlin Heidelberg, 2012.

[68] Seema S. Suman. A Genetic Algorithm for Regression Test Sequence Optimization. *International Journal of Advanced Research in Computer and Communication Engineering*, pages 478–481, 2012.

[69] Ray Salemi. *The UVM Primer: A Step-by-Step Introduction to the Universal Verification Methodology*. Boston Light Press, 2013.

[70] Amer Samarah. *Automated Coverage Directed Test Generation Using a Cell-Based Genetic Algorithm*. PhD thesis, Concordia University of Montreal, Quebec, Canada, 2006.

[71] Ernesto Sanchez, Giovanni Squillero, and Alberto Paolo Tonda. Evolutionary Failing-Test Generation for Modern Microprocessors. In *Proc. of the Genetic and Evolutionary Computation Conference*, pages 225–226. ACM, USA, 2011.

[72] Hans-Paul Schwefel. *Evolution and Optimum Seeking: The Sixth Generation*. John Wiley & Sons, Inc., New York, NY, USA, 1993.

[73] L. Sekanina. *Evolvable Components: From Theory to Hardware Implementation*. Springer Berlin Heidelberg, 2004.

[74] Lukáš Sekanina, Zdeněk Vašíček, Richard Růžička, Michal Bidlo, Jiří Jaroš, and Petr Švenda. *Evoluční hardware: Od automatického generování patentovatelných invencí k sebemodifikujícím se strojům*. Edice Gerstner. Academia, 2009.

[75] Marcela Šimková. Hardware Accelerated Functional Verification. Master Thesis., 2011.

[76] James C. Spall. Stochastic optimization. In James E. Gentle, Wolfgang Härdle, and Yuichi Mori, editors, *Handbook of Computational Statistics*. Springer Berlin Heidelberg, 2012.

[77] Chris Spear. *SystemVerilog for Verification, Second Edition: A Guide to Learning the Testbench Language Features*. Springer Publishing Company, Incorporated, 2nd edition, 2008.

[78] V. Subedha and S. Sridhar. An Efficient Coverage Driven Functional Verification System based on Genetic Algorithm. *European Journal of Scientific Research*, pages 533–542, 2012.

[79] P. D. Surry. *A Prescriptive Formalism for Constructing Domain-Specific Evolutionary Algorithms*. PhD thesis, Univeristy of Edinburg, Scotland, 1998.

[80] Synopsys. Pioneer NTB, 2015. www.synopsys.com/Tools/Verification/FunctionalVerification.

[81] Synopsys. VCS, 2015. www.synopsys.com/Tools/Verification/FunctionalVerification.

[82] Synopsys. Verification IP, 2015. www.synopsys.com/Tools/Verification/FunctionalVerification.

[83] M. Šimková, M. Belešová, Z. Kotásek, and T. Hruška. Application of Evolutionary Algorithms for Regression Suites Optimization. In *Proc. of the IEEE International Symposium on Design and Diagnostics of Electronic Circuits and Systems*, pages 297–300. IEEE, 2015.

[84] M. Šimková and O. Lengál. Towards Beneficial Hardware Acceleration in HAVEN: Evaluation of Testbed Architectures. In A. Biere, A. Nahir, and T. Vos, editors, *Hardware and Software: Verification and Testing*, Lecture Notes in Computer Science, pages 266–273. Springer Berlin Heidelberg, 2013.

[85] M. Šimková, O. Lengál, and M. Kajan. HAVEN: An Open Framework for FPGA-Accelerated Functional Verification of Hardware, 2012. http://www.fit.vutbr.cz/ isimkova/haven/.

[86] M. Šimková, O. Lengál, and M. Kajan. HAVEN: An Open Framework for FPGA-Accelerated Functional Verification of Hardware. In K. Eder, J. Lourenco, and O. Shehory, editors, *Hardware and Software: Verification and Testing*, Lecture Notes in Computer Science, pages 247–253. Springer Berlin Heidelberg, 2012.

[87] Marcela Šimková and Zdeněk Kotásek. Automation and Optimization of Coverage-driven Verification. In *18th Euromicro Conference on Digital Systems Design*, pages 87–94. IEEE Computer Society, 2015.

[88] Marcela Šimková, Zdeněk Přikryl, Zdeněk Kotásek, and Tomáš Hruška. Automated Functional Verification of Application Specific Instruction-set Processors. In Gunar Schirner, Marcelo Götz, Achim Rettberg, Mauro C. Zanella, and Franz J. Rammig, editors, *Embedded Systems: Design, Analysis and Verification*, volume 403 of *IFIP Advances in Information and Communication Technology*, pages 128–138. Springer Berlin Heidelberg, 2013.

[89] Ilya Wagner, Valeria Bertacco, and Todd Austin. Microprocessor Verification via Feedback-adjusted Markov Models. In *Proc. of the IEEE Conference on Transactions on Computer Aided Design*, pages 1126–1138. IEEE, 2007.

[90] Bruce Wille, John Goss, and Wolfgang Roesner. *Comprehensive Functional Verification: The Complete Industry Cycle (Systems on Silicon)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2005.

[91] D. H. Wolpert and W. G. Macready. No Free Lunch Theorems for Optimization. In *Proc. of the IEEE Conference on Evolutionary Computation*, pages 67–82. IEEE, 1997.

[92] Paradigm Works. SystemVerilog Frameworks Template Generator, 2015. http://paradigm-works.com/products/.

[93] Xilinx. MicroBlaze Soft Processor Core, 2012. http://www.xilinx.com/tools/microblaze.htm.