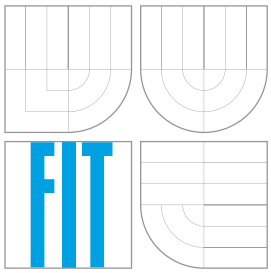
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



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

ROZŠÍŘENÝ ABSTRAKT DIZERTAČNÍ PRÁCE
EXTENDED ABSTRACT OF A PHD THESIS

AUTOR PRÁCE — Ing. MARCELA ŠIMKOVÁ
AUTHOR

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

BRNO 2015

Klíčová slova

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

Keywords

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

The original of the thesis is available in the library of Faculty of Information Technology, Brno University of Technology, Czech Republic.

Contents

Chapter 1

Introduction

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 1.1 illustrates how many participants of the Wilson Research Group Functional Verification Study [16], which is a blind study supported by Mentor Graphics and conducted regularly by Wilson Research Group, 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.

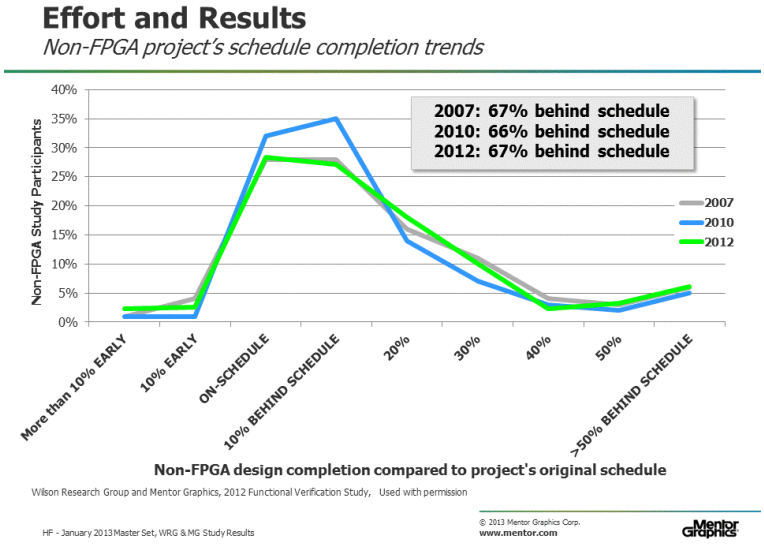


Figure 1.1: Source [16]: Project's Schedule Completion Trends.

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 and isolate a next error,
- creating sufficient tests to verify the whole design and manage the verification process.

In Figure 1.2, 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.

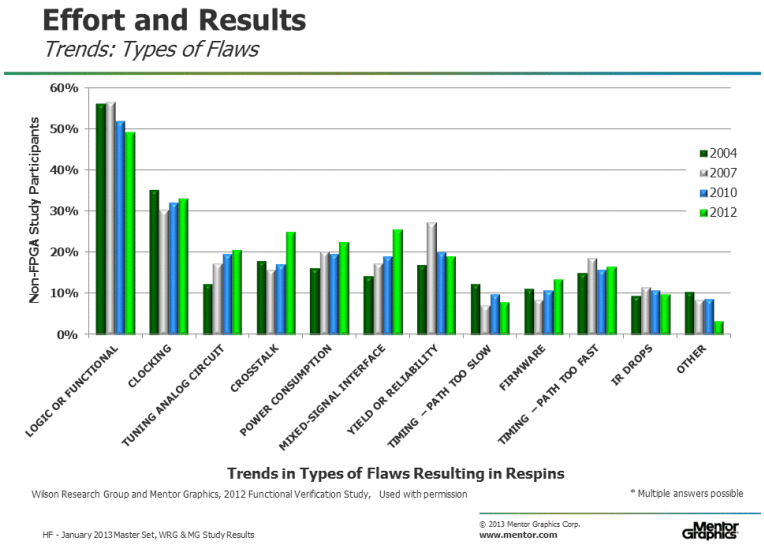


Figure 1.2: Source [16]: Types of flaws.

1.1 Thesis Contribution

We decided to focus our research on a pre-silicon verification approach called 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) [15], Universal Verification Methodology (UVM) [28]).

In the thesis, every of the challenges mentioned in Chapter 1 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.

Chapter 2

Functional Verification in SystemVerilog

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.

- **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.

- **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. 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 [22]. 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.

  Achieving *coverage closure* means provoking the occurrence of each (or some threshold) of the measurable properties [22]. 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 [22] 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 [22] or Chapter 29 in IEEE SystemVerilog standard [1]. Typical structural coverage metrics are *toggle coverage*, *code coverage* and *finite State Machine coverage*.

2.1 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 [6] we introduce the main steps of a verification process.

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 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.

Verification Plan. A verification plan contains a description of features which need to be exercised and techniques and tools which should be used to achieve the specific goals. 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.

  b) **Constrained-random stimuli generation tests** — a more efficient way to verify complex designs thoroughly is with constrained-random stimuli generation. 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. 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. 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.

  b) **Scoreboarding** — a scoreboard is used to dynamically predict the response of the device. Stimulus applied to the DUV 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. 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).

  c) **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. Some utilities can perform post-simulation comparison. In both cases the response can be checked at the cycle-by-cycle or the transaction level with reordering.

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

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.

2.2 Verification Methodologies

Various methodologies were developed with collaboration of different companies:

- **Verification Methodology Manual (VMM)** [6] — 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.
- **Open Verification Methodology (OVM)** [15] — 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.
- **Universal Verification Methodology (UVM)** [28] — is a state of the art methodology that extends OVM. In Figure 2.1, the architecture of the UVM testbench is shown.

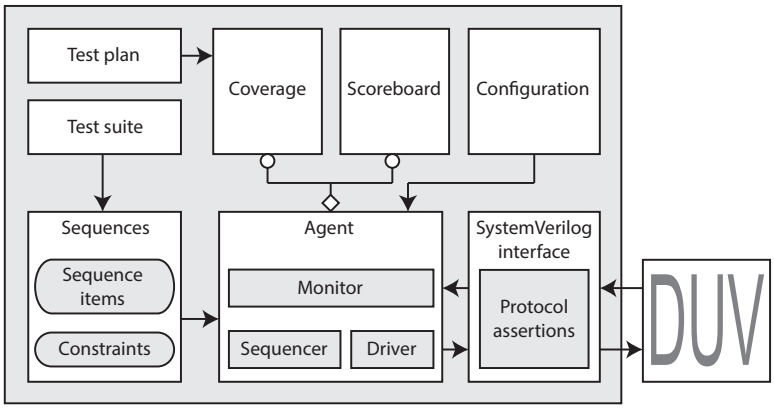


Figure 2.1: The UVM verification environment.

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 measure user-defined functional properties. Of course, the most important part of testbenches is *Scoreboard* which implements the reference functionality.

Chapter 3

Evolutionary Computing

This chapter familiarizes the reader with the basics about Evolutionary Computing (EC). The reason for incorporating the chapter about Evolutionary Computing (EC) is that it 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 theoretical background for this chapter is taken from [12, 30, 27].

3.1 Main Principles of Evolutionary Computing

Evolutionary computing is a computer science research area. It draws inspiration from the process of Darwin‘s natural evolution [11]. 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. This phenomenon is also known as the **survival of the fittest**.

3.1.1 Brief History

The idea of applying Darwinian principles to automated problem solving dates back to the forties, long before the breakthrough of computers [13]. 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* [14], while Holland called his method a *genetic algorithm* [17, 18]. Meanwhile, in Germany, Rechenberg and Schwefel invented *evolution strategies* [26, 29]. 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* [20, 21] 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.

3.1.2 Evolutionary Computing and Global Optimization

Figure 3.1 shows three main stages of the evolutionary search for suitable candidate solutions of the optimization problems, showing how the candidates might typically be distributed in the beginning, somewhere halfway, and at the end of the evolution.

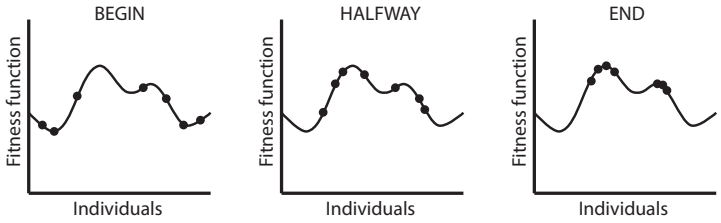


Figure 3.1: Typical progress of an EA illustrated in terms of population distribution. Source: [12].

In the first phase, the individuals are randomly spread over the whole search space (Figure 3.1, left). After only a few generations this distribution changes, the population abandons low-fitness regions and starts to ”climb” the hills (Figure 3.1, middle). Later, the whole population is concentrated around a few peeks, some of which may be suboptimal (Figure 3.1, 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. The 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 [12].

Another class of search methods is known as *heuristics*. These may be thought of as sets of rules for deciding which potential solution 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 [7],[32].

Random search algorithm generates a candidate solution randomly in each step. This algorithm remembers the candidate solution, if its cost 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. 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 to a problem, then attempt to find a better candidate solution in the neighborhood by the local exploration. In order to evaluate the cost of candidate solutions in the neighborhood it is necessary to precisely define the neighborhood using the so-called *cardinality constant*. In the next step, a candidate solution with the best cost is selected and serves as a starting point in the next step of the algorithm. The disadvantage of this algorithm is that it often reaches a local extreme and the best possible solution is not found.

According to [30], 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.

3.2 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 candidate solution consists of a string of binary digits. For a particular application we have to decide how long the string should be and how we will interpret it. 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.

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 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 and swap their values.

Recombination. Recombination is 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 mating pool of parents is sampled from the selection probability distribution. 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*).

3.2.1 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.

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 [3]. 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 (No Free Lunch theorem [40]).

Chapter 4

Goals of the Ph.D. Thesis

The 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. When talking about expected results, in the first goal, the 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 (definition of verification scenarios, reference models) so it is expected that these parts would have to be manually adjusted for some DUVs. The speed-up (the second goal) should be achieved by accelerating simulation using the FPGA accelerator and the reaching of maximum coverage (the third goal) should be gained by the intelligent testbench automation. 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. 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.

Chapter 5

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* [35] so it can be used by academy projects and small companies without dependency on expensive accelerators or emulators.

5.1 First Version of HAVEN

The first version of HAVEN was introduced in 2011 in the master thesis [31] and the basic principles of verification acceleration were described in the paper on Haifa Verification Conference [36].

5.1.1 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 5.1. 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 thanks to the transaction-based communication among the subcomponents and this enables a transparent move of the components to a specialized hardware, while maintaining good readability for verification engineers.

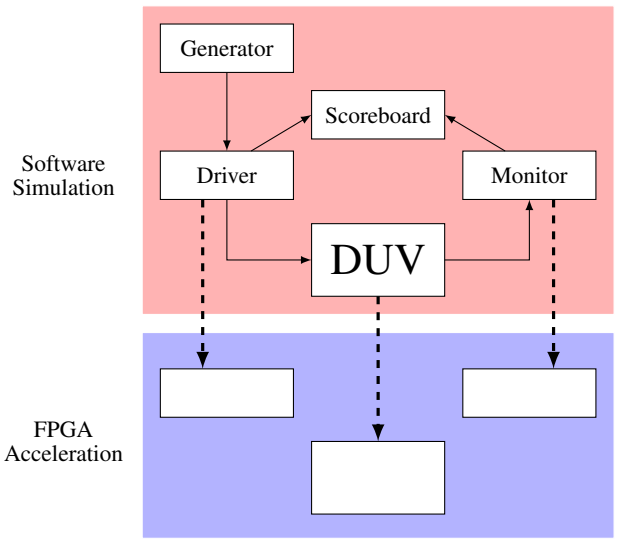


Figure 5.1: Moving some part of the verification testbench from software to hardware.

5.1.2 Experimental Results

We performed a set of experiments using the COMBOv2 LXT155 acceleration card [25] 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: 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 [19]. 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).

During the experiments, we observed that a considerable amount of time is taken by generating stimuli, therefore, we 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 Fig. 5.2 which shows the relation between the complexity of the verified component and the acceleration ratio (without the time of stimuli generation).

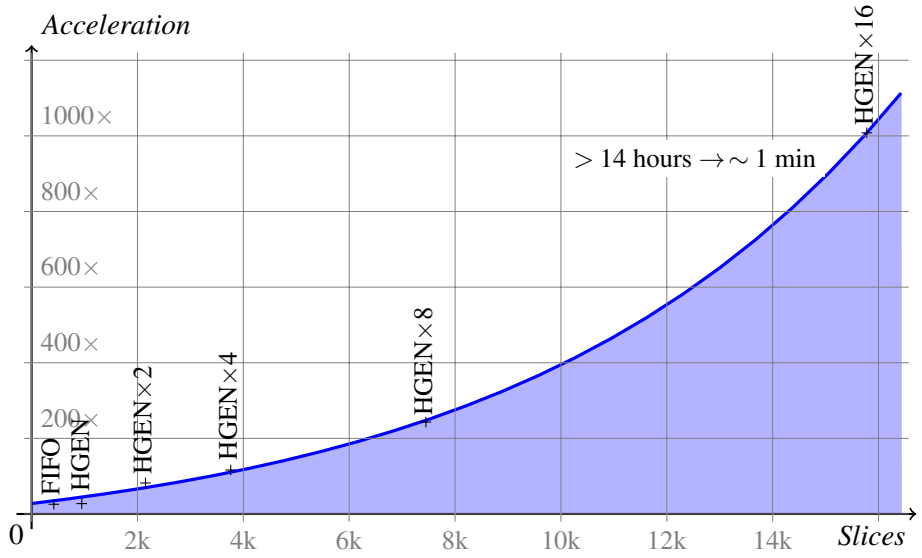


Figure 5.2: Relation between the acceleration ratio and the complexity of the verified component.

Table 5.1 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 5.1:

$$\frac{trans_{be}}{\text{trans\_per\_sec}(NAV)} = build\_time + \frac{trans_{be}}{\text{trans\_per\_sec}(AV)} \tag{5.1}$$

where $build\_time$ is the build time of the firmware in seconds and $\text{trans\_per\_sec}(AV)$ and $\text{trans\_per\_sec}(NAV)$ 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 5.2 for computing $trans_{be}$:

$$trans_{be} = build\_time \cdot \frac{\text{trans\_per\_sec}(AV) \cdot \text{trans\_per\_sec}(NAV)}{\text{trans\_per\_sec}(AV) - \text{trans\_per\_sec}(NAV)} \tag{5.2}$$

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 5.1: 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

5.2 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 [34].

5.2.1 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.

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 other parts of the verification environment from software to hardware (generator of stimuli, scoreboard, transfer function, coverage monitor), with each step obtaining a different trade-off between the acceleration ratio and the debugging comfort.

5.2.2 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.

Software version (`SW-FULL`). All components of the verification environment (stimuli generator, scoreboard) are in the software simulator (Fig. 5.3).

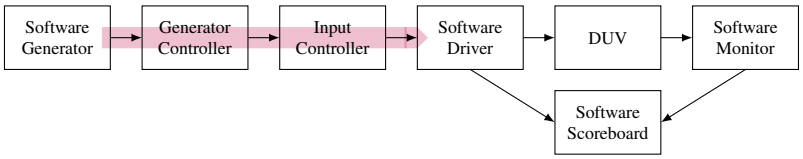


Figure 5.3: Software version (`SW-FULL`).

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. 5.4), 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.

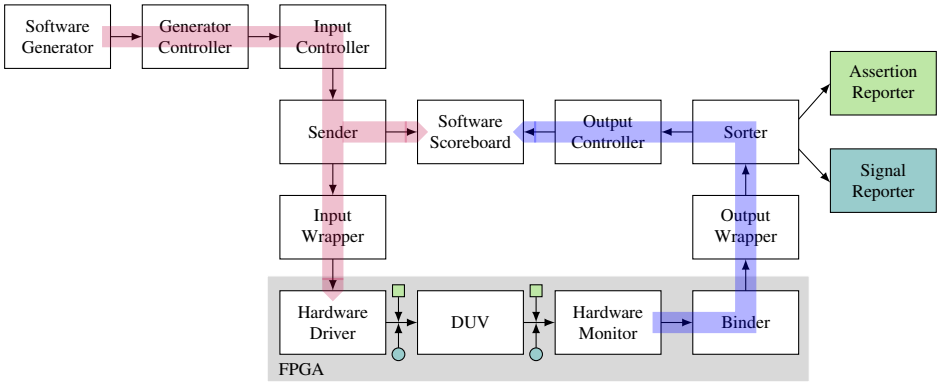


Figure 5.4: Hardware DUV version (**`HW-DUV`**).

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. 5.5) 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.

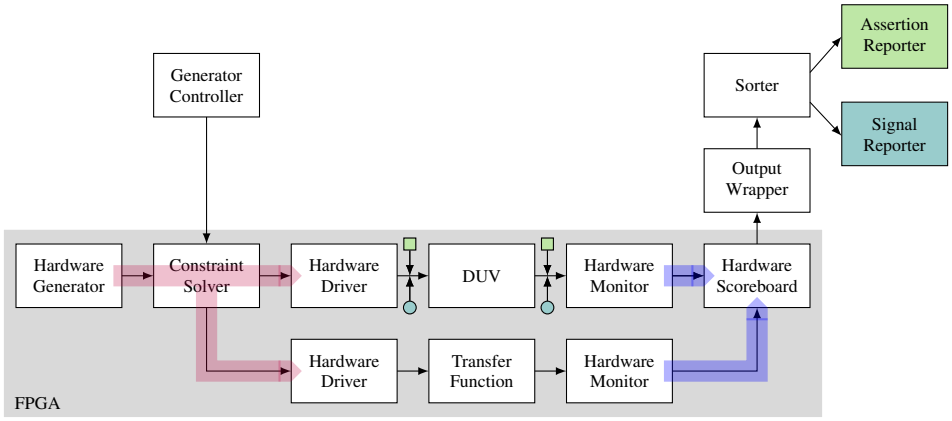


Figure 5.5: Hardware version (**`HW-FULL`**).

5.2.3 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 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). Table 5.2 shows the acceleration ratio of each of the architectures of the HAVEN testbed against the **SW-FULL** architecture.

Table 5.2: 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.

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 protocols like ours. 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.

5.3 Use-cases of HAVEN

The HAVEN framework was further used in two different scenarios: for accelerating functional verification of application-specific processors (ASIPs) [23] and as a part of the platform for testing fault-tolerant electro-mechanical systems [24].

5.4 Main Contributions of HAVEN

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 complex UVM-based verification environments 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 built from achievable FPGAs is cheaper than commercial solutions.
- HAVEN is freely available an open-source.

Chapter 6

Automated Generation of UVM Verification Environments

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 reusable 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 [38].

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.

6.1 Codasip Studio

As a development environment we utilized the Codasip Studio from the Codasip company [10]. 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.

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.

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.

6.2 Functional Verification Environments for Processors

In order to comfortably debug and verify ASIPs designed in the Codasip Studio as fast as possible, we designed a special feature allowing automated pre-generation of OVM/UVM verification environments for every processor.

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.** 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.

6.3 Experimental Results

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 [9]. 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 testsuites such as *full-retval-gcctestuite* and *perrenial testsuite* were utilized during verification.

Table 6.1 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 6.1: 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 6.2 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.

Table 6.2: Runtime statistics.

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

The coverage statistics in Table 6.3 show that the instruction-set functional coverage reaches only around fifty 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). 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.

Table 6.3: 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 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.

6.4 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 further increase all coverage metrics for both processors over 90%. Some preliminary experiments were published in [8].

Chapter 7

Automation and Optimization of Coverage-driven Verification

In this chapter, our solution for automation and optimization of CDV which is based on evolutionary computing [37] 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.

7.1 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. 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 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 3.1.2 and from their simple comparison, the evolutionary algorithms seem to be most beneficial for our work.

7.1.1 Automated CDV Driven by Genetic Algorithm

For CDV automation and optimization, we decided to use 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). 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 7.1 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 7.2. For a better comprehension, see two case studies in Section 7.2.

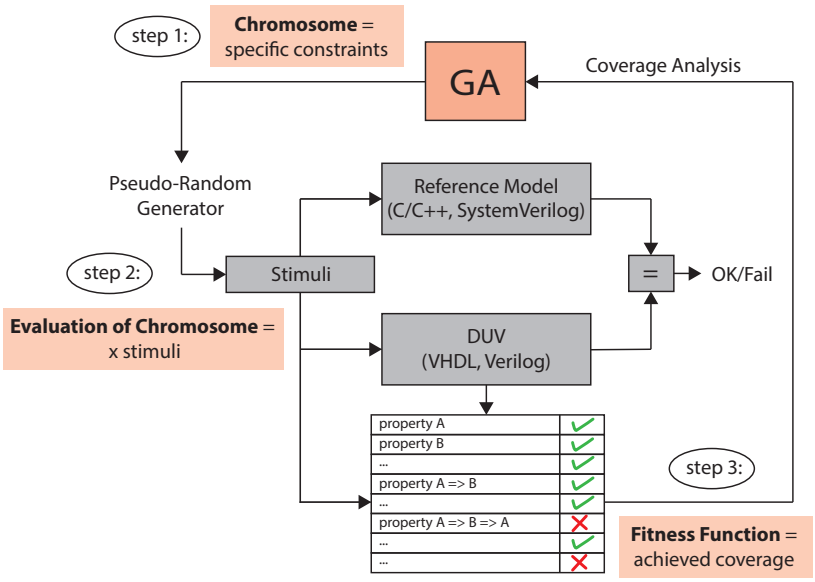


Figure 7.1: The automated coverage-driven verification driven by a genetic algorithm.

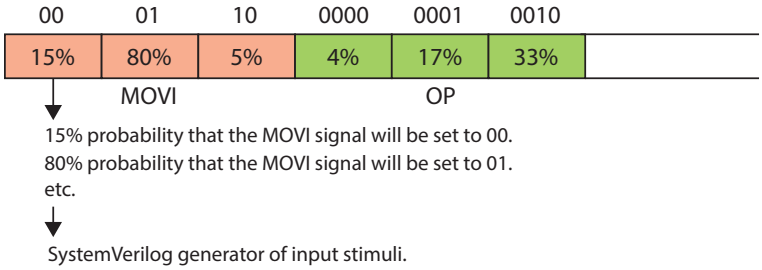


Figure 7.2: Probability constraints encoded in the chromosome.

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 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.

7.1.2 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.

For pseudo-random generation of initial populations, the inbuilt pseudo-random generator of UVM/System Verilog [39] 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 [8] that is based on the Merssene Twister algorithm.

7.1.3 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.

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 7.2.1, 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 7.1. 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 in the random search.

$$\texttt{STIMULI\_NUMBER} = \frac{\frac{\texttt{AVG\_RS}}{2}}{\texttt{GENERATIONS}}. \tag{7.1}$$

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 for one program per chromosome, because it 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 7.2.

$$\texttt{TOURNAMENT\_SIZE} = 25\% \text{ of } \texttt{POPULATION\_SIZE}. \tag{7.2}$$

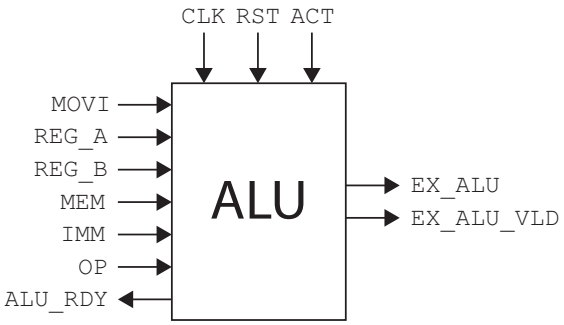
During mutation, it is specified at first, how many bits will be mutated in the chromosome (inverted). Afterwards, the mutation probability helps to determine if the inversion really happens or not. The maximal number of mutations is defined by the parameter MUTATION_MAX and is determined using the bit width of chromosomes (CHR_BW) by Equation 7.3.

$$\texttt{MUTATION\_MAX} = 10\% \ \texttt{CHR\_BW}. \tag{7.3}$$

7.2 Experimental Results

7.2.1 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 7.3.



- 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 7.3: 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.

We decided to compare the results of the GA_driven search with the basic random search and the constrained random search. 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 7.2. 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 [39]. However, it can take a very long time to cover all of the properties, because without the coverage feedback, the generated stimuli cover some properties repeatedly.

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 graph in Figure 7.4 shows the comparison of the basic random search, the constrained random search and the GA-driven search. It compares average values from 20 different measures for every kind of search. 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.

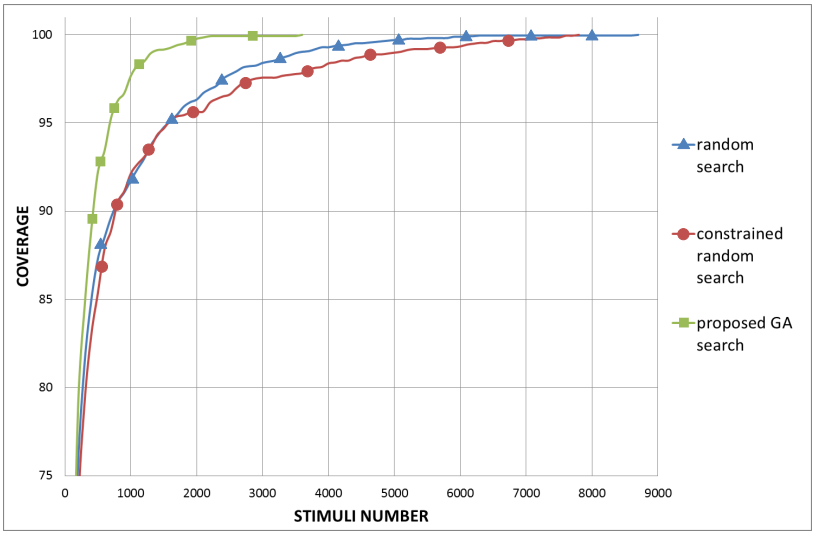


Figure 7.4: The comparison of average values from the basic random search, the constrained random search and the GA-driven search.

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 7.3), tournament size to 5 (20% of the population size, see equation 7.2), elitism to 1 (1 best chromosome is propagated).

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.

7.2.2 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).

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 [8] are applied.

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 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 7.5 illustrates how the size and probability constraints are encoded in the structure of chromosome and how they are processed by the generator.

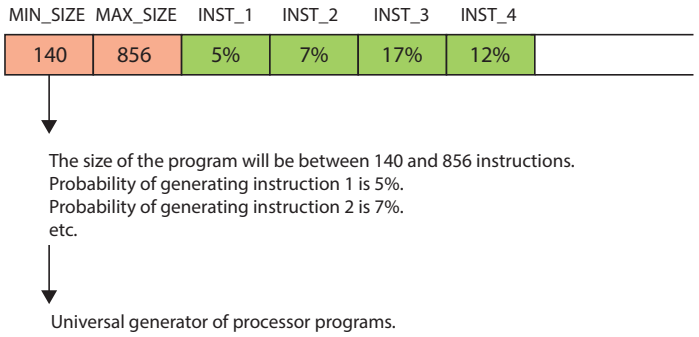


Figure 7.5: The structure of the chromosome for the Codix RISC processor.

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 processor. 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 graph in Figure 7.6 demonstrates the results of the experiments. It compares average values from 20 different measures for every approach (using various seeds)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 average coverage that was achieved by 1000 benchmark programs was 88% and 97.3% by 1000 random programs. 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 average coverage achieved was 97.78%. For the experiment with 50 generations 98.72%, for the experiment with 100 generations 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

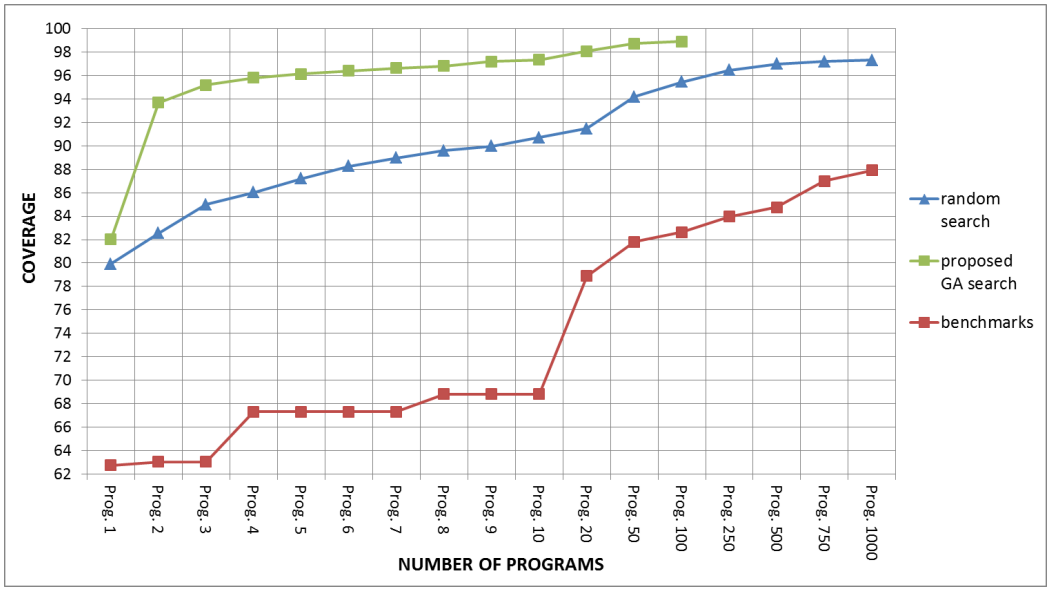


Figure 7.6: The comparison of average values from the benchmark programs, the random programs and the GA-driven generated programs.

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.

7.3 Main Contributions of GA Automation and Optimization

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.

Chapter 8

Optimization of Regression Suites

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 [33]. 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 [39], 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.

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 8.1.

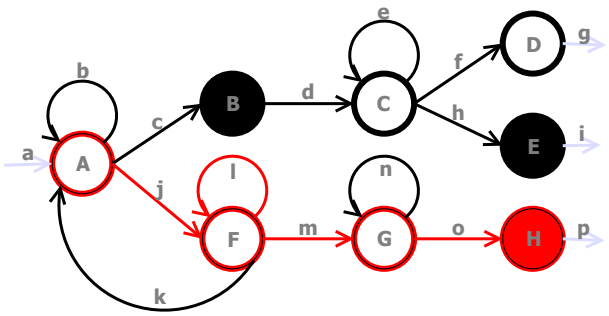


Figure 8.1: A fragment of an automaton that demonstrates sequential properties.

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 8.1. 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. 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 3.1.2.

8.1 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.

8.1.1 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. 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 solutions are represented by sub-sequences of stimuli from the original sequence. Mutation is proposed in two different ways:

1. *type delete*: removes some stimuli from the regression suite.
2. *type swap*: the mutual swap of two stimuli in the sequence (the ordering of stimuli is important).

As for 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. 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. GA runs functional verification for every newly created candidate solution. The coverage of the selected properties for these stimuli is measured and returned to GA, where the fitness value is computed according to the coverage. The tournament selection algorithm is used for selecting parents. 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.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.

8.1.2 GA Parameters

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.

8.2 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 8.1.

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 8.2 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 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 8.1: The optimization results.

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.

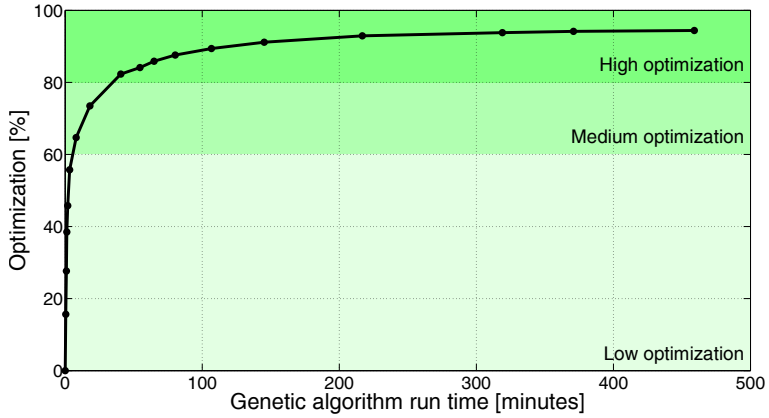


Figure 8.2: The dependency between the optimization runtime and the level of optimization.

8.3 Main Contributions of Regression Suites Optimization

The main contributions of the proposed technique are:

- 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.

Chapter 9

Conclusions

In the 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.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 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). 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.

The third technique is used for automation and optimization of CDV. There are several algorithm for the coverage-space exploration and we selected the evolutionary exploration as the most suitable one for CDV. We created an adapted genetic algorithm 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. 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).

Our last optimization technique is 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. The experiments were once again performed on the ALU. 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.

9.1 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.

9.2 Related 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.

9.3 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, David B. Fogel, and Zbigniew Michalewicz, editors. *Handbook of Evolutionary Computation*. IOP Publishing Ltd., Bristol, UK, 1st edition, 1997.

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

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

[8] 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.

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

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

[11] 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.

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

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

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

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

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

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

[18] 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.

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

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

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

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

[23] 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.

[24] 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.

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

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

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

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

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

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

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

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

[33] 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.

[34] 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.

[35] 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/.

[36] 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.

[37] 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.

[38] 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.

[39] 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.

[40] 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.