Verification of System-on-Chips using Genetic Evolutionary Test Techniques from a Software Applications Perspective

Adriel Cheng

B.E. (Hons)

A thesis submitted in fulfilment of the requirements for the degree of Doctor of Philosophy

THE UNIVERSITY OF ADELAIDE

Faculty of Engineering, Computer and Mathematical Sciences

School of Electrical and Electronic Engineering

September 2009

CHAPTER 1. INTRODUCTION

1.1 The need for hardware design verification

Ensuring the correctness of very-large-scale-integrated (VLSI) hardware designs is essential for the successful deployment of any device using integrated circuit (IC) semiconductor chips. A defective IC design embedded in the end product is extremely costly to rectify after it has been fabricated. This thesis addresses the current gaps and verification challenges posed by VLSI semiconductor designs.

Semiconductor fabrication plants require significant investments in the order of hundreds of millions of dollars to establish. A single silicon fabrication cycle of a VLSI design alone costs millions of dollars. Hence, a VLSI design project must be confident its design is bug-free and will function after production. First time silicon failures require expensive silicon re-fabrication. Besides the immediate costs of re-fabrications, other hidden costs must be accounted for, such as the engineering effort and time required to fix, re-design, integrate, and verify the revised chip design.

Such unnecessary expenditures can be minimised if design verification was conducted thoroughly to begin with. The Intel floating point divisor bug case study [Int94a, Int94b, Mol95, Wir94] underlined design verification as an essential phase of a design project's lifecycle. Design verification requires significant engineering investment upfront to guarantee the design is free of bugs before committing to fabrication. Pumping large amounts of engineering resource alone however, is insufficient. Establishing an effective design verification (and test) methodology is crucial to prevent any slipping through of costly design errors to the fabrication stage.

In addition to effective verification strategies, the verification process must be conducted efficiently to avoid any impact on time-to-market of the end-product encapsulating the IC design. Late shipment of ICs is financially detrimental to the design team and their clients, especially in the highly competitive environment of bringing ones product to customers first. Besides financial penalties, if an IC is employed in safety critical applications, design errors and hardware failures cannot be afforded under any circumstances.

Formulating and devising an effective and streamlined verification methodology, along with associated test creation and verification effectiveness measures are critical, and forms the focus of this thesis.

1.2 Scope and focus of the research

It is important to define and distinguish the meaning and differences between verification and testing within the scope of VLSI IC design projects for this thesis. During development of an IC, to ensure a functionally correct and defect free design, two main techniques are employed, (1) design verification and (2) physical (or manufacturing) testing.

The aim of design verification is to demonstrate the functional correctness of a hardware design according to its intended design specifications. Design verification is primarily undertaken at the pre-silicon stage before chip fabrication. Verification of the IC design against its specifications is usually carried out on a high level abstracted software based model of the design, or directly against the register transfer level (RTL) or gate level hardware design code; before it is synthesised to lower level formats suitable for fabrication. The purpose of design verification is to expose both functional and logical bugs arising from designer errors.

Physical testing is carried out at the post-silicon stage after the IC design has been fabricated onto silicon wafers. The aim of physical testing is to uncover any physical defects on the fabricated IC chip. Examples of physical defects include masking or metallisation errors, and other IC layout or fabrication problems

The research in this thesis focuses on the pre-silicon design verification domain. The research scope was limited to design verification for a number of reasons. ICs fail due to various causes of errors throughout the project cycle. Some of these errors include mixed-signal problems, power or timing issues, and manufacturing or fabrication imperfections like masking or metallisation defects. Significantly, the most common reason for chip failure is attributed to logical or functional flaws, which accounts for 75% of failures according to the IC verification study conducted by Collett International Research in 2004 [Col04]. This is an increase from 2000 [Col00] and 2002 [Col02] whereby the amount of silicon re-fabrication caused by this type of errors was 47% and 71% respectively; and reflects the continuing lag in verification capabilities currently and for the foreseeable future.

Furthermore, the functional flaws can be narrowed down to a number of sources as follows.

- Design errors that were undetected because certain design units or rare operating scenarios were not exercised (referred to as corner cases).
- Externally introduced errors in the design arising from the use of erroneous or unreliable third party design libraries or modules.

- Specification errors such as incorrectly interpreted or incomplete specifications.

The above categories of errors are the responsibility of the design verification phase and should be detected before fabrication. Given that more than half of IC failures are due to these functional errors implies a lack of effective design verification strategies. The statistics and their subsequent conclusions from such IC design and verification studies are the inspiration behind our research in the design verification domain, rather physical testing. Significant improvements to IC fabrication success rates can be achieved if better design verification techniques are developed and adopted.

Establishing functional correctness at the pre-silicon level also provides other advantages. The first benefit is the lower cost associated with bug fixes. Design errors detected early in the design cycle require less resource to rectify, and the desired fixes are easier to implement. Fixing the design simply involves modifying the hardware design code. At the post-silicon level, the IC components, wire connections, and interfaces are fixed in silicon. Any bug fixes would require costly re-fabrications.

Debugging and diagnosing the root cause of an error at the design verification phase is also more efficient because verification engineers can access and monitor the internal signals of the IC via software tools. A fabricated IC only provides access to the primary inputs and output pins of the chip; the characteristics of internal signals and state elements are unknown making erroneous behaviours harder to analyse.

Despite expensive bug fixes and difficult failure diagnosis, the test throughput offered by a fabricated IC at the post-silicon level can be order of magnitude greater than that of design verification. Design verification requires software simulation of the IC design; whereas a fabricated IC design can be tested at its intended operating speed. Regardless, design verification remains the critical phase where design errors can be detected, diagnosed, and fixed earlier; and in a more effective and least costly manner. It provides the best opportunities for finding bugs and to enhance the quality of the chip design. The importance and benefits of design verification call for this research thesis to focus primarily in this domain.

Besides verification and testing, validation is another term occasionally used in literature within this field of research. In this thesis, validation is considered a separate phase and a different activity from design verification or post-silicon testing. Specifically, validation refers to the process of checking that the specifications of a design have been captured and formulated correctly. That is, do the specifications for a design satisfy the application needs and criteria for the purposes and usages of the

IC in its eventual end product? Validation ensures the design specifications meet the needs of the clients of the IC chip. Validation is beyond the scope of the research in this thesis.

Figure 1.1 shows a typical flow for a chip design process indicating when validation, design verification, and physical testing are conducted. In parallel with the hardware flow, the software development process of the chip is carried out as well. In this thesis, unless otherwise specified, the terms *verification*, *testing*, and *validation* are used interchangeably within a design verification context only. Testing or validation shall refer to the application of tests for pre-fabrication design verification, not post-silicon testing or design specification checking of the client's IC requirements.

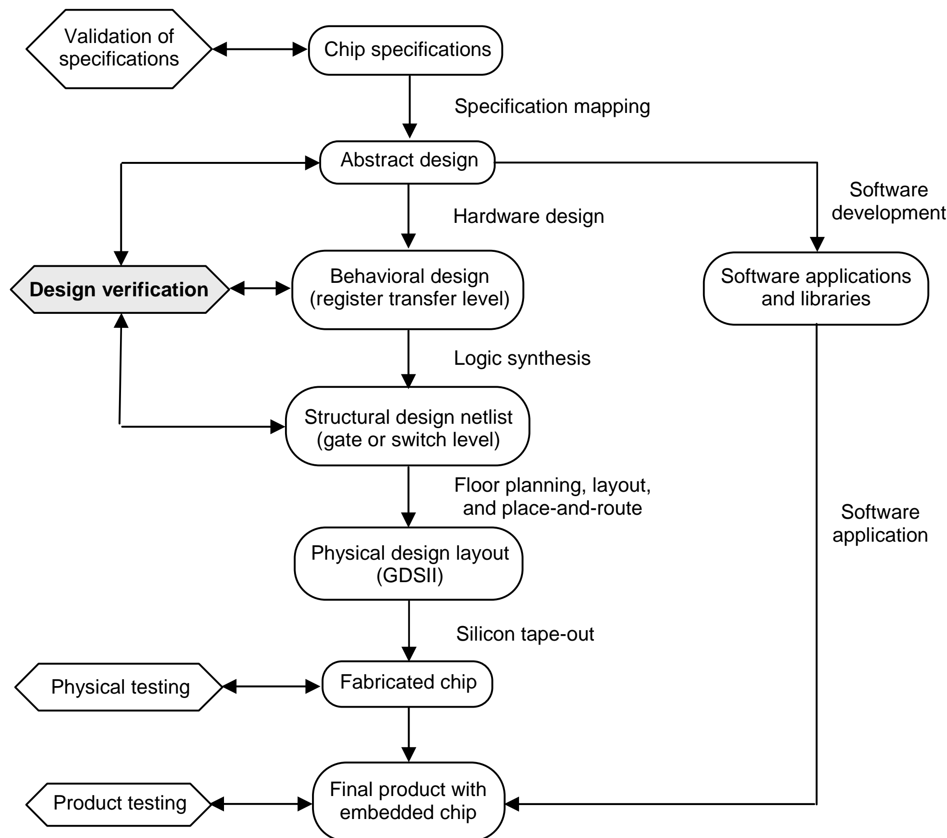


Figure 1.1 Typical chip design flow

1.3 The hardware design verification problem

Verification of hardware designs has become increasingly difficult. Today, it is a significant bottleneck in modern design flows, and this trend is continuing to worsen. Currently, hardware design verification requires as much resources (i.e., engineering effort, time, and monetary costs) as design and implementation of the IC itself. On a typical design project, up to 70% of project resources may be

expended for design verification [Ber03], taking up a significant portion of the project lifecycle. The difficulty in design verification is due to faster, lower powered and more complex designs that produce many more complicated behaviours. The enablement of such high performance chip designs can be attributed to a number of factors.

In the last two decades, extensive research and development in VLSI design methods has provided new design technologies and concepts; such as high level hardware design languages, behavioural and structural synthesis, automated transistor circuit layout/partitioning, and micro-architectural innovations such as pipelining and superscalar architectures. Designers today use these new and enhanced design techniques in addition to computer-aided-design (CAD) tools; many of which are encapsulated in industrial software tool packages. Using various design methodology flows [Ber03, KB02], and aided at every step by CAD tools, larger more sophisticated designs can be created more readily than before, and at lower design cycle times.

At the same time, the number of transistors in an IC chip has increased rapidly according to Moore's law. Improved fabrication and semiconductor material technology allow designers to exploit the increased density available from smaller chip feature sizes to fit more gates and functionality onto a single chip. The flow-on effect is a doubling in operating speed and the number of design blocks that can be embedded onto a chip every two years. This has led to a new generation of IC chip designs, known as the system-on-a-chip (SoC).

SoCs are entire systems implemented on a single IC to perform numerous functions that would have required multiple individual ICs interacting with each other on a printed circuit board. Instead, the SoC performs the equivalent functions faster on a much smaller silicon die. Traditionally, an SoC integrates a main microprocessor core and a number of peripheral core modules embedded on the same chip. Such individual cores are usually designed by different design teams or re-used from previous internal design projects to build the new SoC. Alternatively, they may even be imported from other external design vendors. This allows SoCs to be designed with faster turn-around time, and implement more functionality using an extensive range of readily available design components.

Regardless of their source, the design cores on an SoC are often created independently by different means, hence, their functional correctness cannot simply be assumed. These cores must be verified both independently and within the new system when working with each other. This adds further verification requirements and complexities for the SoC.

Design verification of stand-alone hardware design units already presents many difficulties for verification teams. However, the increased complexities in SoC functionalities mean design bugs are now even harder to find and detect. Compared with previous single core ICs, an SoC brings together many intricate devices into one integrated system, and executes multiple and concurrent processes interacting with each other. This causes an explosive number of SoC circuit conditions, burying some design bugs deep in the logic cone that can only be detected under certain scenarios. The scalability of SoCs and increasing popularity of multi-core designs imply that the size and number of design blocks will continue to increase. This creates larger designs for any IC based on the SoC design paradigm. Traditional verification methods alone are simply inadequate for SoCs.

Current design verification techniques are unable to keep up and handle the growing complexity of many hardware (including SoC) designs. The state of the verification gap is captured in Figure 1.2 [Col04], showing mismatch between verification capabilities against chip design innovations and chip fabrication is increasing rapidly.

NOTE:
This figure is included on page 6 of the print copy of the thesis held in the University of Adelaide Library.

Figure 1.2 Design and verification gaps (source: Collet International Study 2004)

Comparing quantitatively, given similar resources, the productivity from a design team can be estimated at around 170 gates per day, whilst a verification team can only verify 100 gates in the same time. The effectiveness of design verification has steadily degraded. In 2002, only 39% of first time chip design fabrication was successful. In comparison, the first time silicon success rate in 2000 was 50% [Col00]. This indicates significant decline in the quality and effectiveness of SoC verification has persisted for some time.

Given the growing popularity of SoCs, new tools and methodologies must be developed that enable SoC verification goals to be fulfilled using less resources. This thesis proposes new solutions and extends previous verification research at the micro-architectural level into the SoC domain, to contain and reduce the verification bottleneck.

1.4 Design verification in context

Design verification is carried out using two main techniques, formal verification and more commonly, simulation based verification. The scope of our research lies within the simulation based domain, although techniques from formal verification are studied and adapted for use in our methods.

Testing a hardware design via simulation requires (1) test generation of a comprehensive set of tests to stimulate the design, and (2) coverage measurements during simulation to report the effectiveness of the verification process. The design verification strategy is then enhanced by coupling both test generation and coverage measurements together to form a coverage driven verification approach.

1.4.1 Test generation

The aim of test generation is to create test cases that initiate and verify different functional behaviours on the hardware design. In simulation based verification, test cases are usually created between the assembly instruction and lower signal pins levels; using assembler instructions test programs or low level input stimulus vectors to make up the functional test cases [CCRS03a, CGW+95, FFS+01, HMK96, SA98]. These tests can be created to verify smaller individual modules or the overall design. Given the greater complexities of SoCs today, a significant portion of this thesis is dedicated to examining usage and development of automatic and algorithmic test generators to create higher level tests.

Creating quality test suites that are effective at exposing design bugs is an ongoing challenge for verification teams. At the same time, attaining an accurate measure of verification progress and comprehensiveness of the test suite is still an open problem. A successful verification strategy requires suitable measurements to monitor the effectiveness of test generation and how much test simulation is sufficient. Coverage is one such measurement.

1.4.2 Verification coverage

Coverage is a measure of design verification quality. It evaluates how comprehensively a test suite has exercised the hardware design. The greater size and complexities from an SoC demands extensive resources on verification be expended during the project lifecycle. However, how much testing is sufficient, and when can a design team be confident design bugs have been eliminated? Coverage addresses these issues by measuring and determining what portions of the design has been tested. This enables design managers to quantitatively monitor verification progress, identify what has or has not been tested, and estimate the remaining effort needed to cover the remaining design to achieve validation goals. Section 2.3 of Chapter 2 provides a comprehensive treatment of current coverage solutions. In this thesis, a functional coverage method is proposed for our SoC verification methodology, in order to assess and quantify the success of our verification approach.

1.4.3 Coverage driven verification

An increasingly popular design verification methodology that combines test generation and coverage measurement is to employ coverage driven verification (CDV). The goal of CDV is to gain maximum coverage – hence maximise the likelihood of detecting bugs – using minimal test resources. CDV involves measuring the coverage provided from an existing test suite previously simulated on the hardware design. The coverage measured is then analysed and useful information is extracted to guide future verification of the design [BGH+99, FUZ04, FZ03, UY99]. Specifically, the coverage measuring process reports which areas of the design or what functional behaviours of the design have or have not been verified. Using this information, to further enhance coverage, the verification process is guided toward untested, critical, or continual stimulation of previously beneficial areas of the design space.

The test generator is directed by coverage data from previous test simulations to create new tests that focus on desired design functions that need verification. Coverage data from current testing is periodically fed back to the test generator to drive future verification. The process iterates continually providing an effective and efficient verification scheme, attaining maximal coverage as quickly as possible. Given that test generation is often the process which uses the coverage data to determine what additional kinds of functions are to be tested, CDV is also commonly referred to as coverage driven test generation as well.

CDV can be conducted manually by verification engineers. However, the process of extracting useful information from the coverage measurement data can be tedious and time consuming. Automatic feedback of coverage for CDV is required. The thesis will investigate solutions and propose an automated CDV scheme that couples the test generation research work and coverage together.

1.5 Background of the research

The design verification research conducted in this thesis was driven by a number of factors. Design verification is essential for successful SoCs, yet it remains a significant bottleneck for many projects. For this reason, design verification presents an important and active research area in the IC community. Many researchers from both academic institutions and industry have tried to tackle these verification challenges.

At Freescale Semiconductor, design verification is a critical phase of their semiconductor engineering flow. Extensive resources including dedicated verification teams are assigned to certify the functional correctness of chip designs prior to chip fabrication. Initially, the research undertaken in this thesis was partially driven by verification projects from Freescale. However, our aim is to conduct verification research that provides techniques and tools not just for Freescale, but for the wider IC verification community.

At Freescale Semiconductor, an effective strategy for design verification was to employ application based testing for stimulating the SoC. Using this technique, numerous verification projects was able to attain high bug detection rates and extensive design coverage compared with conventional logic based simulation testing.

The application based approach has been used on various SoC designs at Freescale since 2002. For instance, the technique was previously used to verify high performance network communication SoCs [Fre04]. During verification of network SoCs – intended for use within high speed and large bandwidth telecommunication data network routers – the application based approach was employed to compliment existing conventional verification methods. Specifically, for an accelerated gigabit Ethernet controller design block, software application testing, and other manually directed and random based tests were used. Throughout the verification phase, the software application technique consistently outperformed other methods. The software application technique maintained greater error detection rate and uncovered more design bugs overall.

Figure 1.3 shows the much superior bug detection capabilities of the software application approach compared with two other test platforms based on randomisation. The x-axis represents the project lifecycle time for each verification project undertaking different test methods on the network based SoC. The graph results show the application based approach was able to detect bugs earlier, and uncovers greater number of bugs in less time compared to the random based methods.

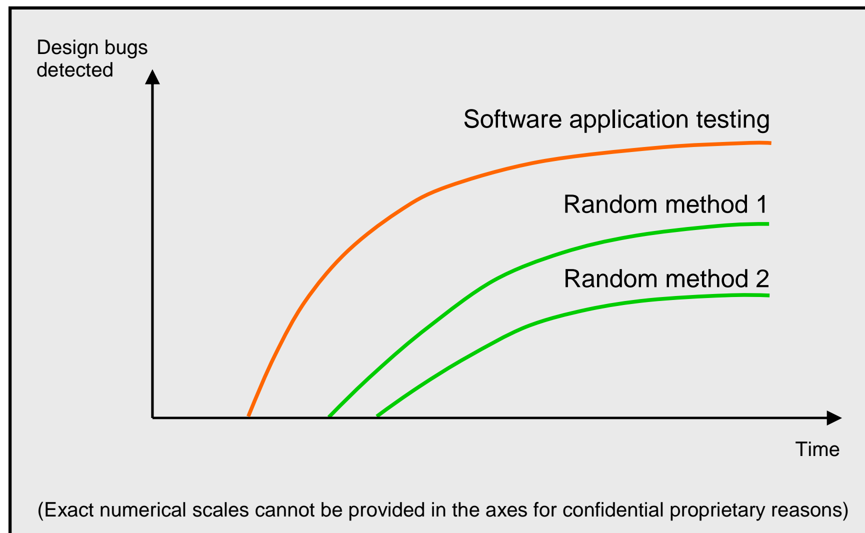


Figure 1.3 Bug detection rates of different verification methods applied to Freescale network based SoC

The advantage of the application based technique stems from the use of real-life application scenarios rather than individually devised or randomly created artificial test scenarios. The important design functionalities that will be exercised most commonly by application end-users of the SoC are tested well before fabrication. Critical design bugs will be detected at the earliest pre-silicon verification stages from the RTL design level.

Despite the successes and proven capability of the application based technique on these projects, the application style verification process was still conducted in a semi ad-hoc manner. Additionally, the development and usage of these application type tests were carried out manually. Research was needed to investigate and extend this application based method into a structured methodology; which can be applied effectively and efficiently for any SoC design project, not just within Freescale but for the general SoC design industry. The research conducted in this thesis began as a collaborative project with Freesacale. The goal was to develop a verification process based on the software application approach that is reusable and automated; in order to facilitate faster test cycle times and more comprehensive SoC design verifications overall.

The methodology proposed in this thesis was developed into a *software application level verification methodology*. The research work builds on the successes and experience from these verification projects. The aim of our proposed methodology is to formally define the application test technique. We develop the verification technique into an automated flow, and extend its capabilities in the areas of test generation and coverage.

The methodology from this thesis operates at the pre-silicon stage, employing high-level software application tests composed of *software code segments*. These code segments are extracted from various application use-cases and act as building blocks to form a variety of different test programs. The test programs applied onto the SoC will be based on the same functionalities needed by real-life application programs.

The thesis will define and describe our verification concepts such as the application code segment building blocks, the test generation techniques employed, and the coverage methods for our application methodology. In addition, a solution shall be proposed and demonstrated for automated coverage feedback testing using our methodology. The design verification gaps of the research conducted in the thesis are described next.

1.6 Research gaps

1.6.1 Design verification gap

The majority of verification methodologies today concentrate principally on the hardware characteristics of a design. Traditional methods like formal verification and simulation based methods are regularly applied to individual design modules to verify them at the signal interface level. Once integrated into an SoC, transaction level verification is employed to perform system-wide testing [BCG+00, RPS01]. However, little consideration is given to the requirements and interactions of the software that will eventually run on the SoC, even though the operations invoked by software will be most critical for the SoC customers and users.

The additional complexities and size of an SoC based design demand new test strategies to counter this verification gap, from both a hardware and software perspective. One research goal is to address this gap by proposing a different design verification scheme that examines the common and critical design functions that will be used by the application software when the SoC is embedded for real-life operations. The intention is to bring a holistic approach to SoC design verification by using software to

verify SoC design hardware. At the same time, this approach verifies the interactions between SoC software and hardware functions.

Our approach is to adopt software application testing at the early pre-silicon design phase using RTL simulation of the SoC design. This endorses early detection of bugs and less costly fixes on the SoC design. However, this approach lacks a structured methodology, whereby a defined sequence of steps to conduct the testing technique is currently missing. Despite its effectiveness, tests are still created manually in an ad-hoc fashion targeting various design functions in an irregular manner.

Furthermore, the use of application based testing for hardware designs has traditionally been applied at the post-silicon stage; when a hardware implementation of the SoC design is available to execute complete or large application programs within suitable timeframes. Simulation of application based tests is limited to much smaller software application code only, which can be managed by the simulator. Full application programs or operating systems are too large for RTL simulations; they would require too many simulation cycles which is impractical. To address these issues, the research proposes a methodology for testing a wide range of functionalities covered by real-life applications. The technique is to extract from these applications the test building blocks that can be used to automatically create systematic sets of tests, which are efficient in size suitable for RTL simulations.

Within the scope of this application based design verification research, two verification challenges are undertaken: test generation and coverage.

1.6.2 Test generation gap

Automation test generation gap

The creation of a single software application test program requires much thought and effort even from verification engineers with adequate knowledge of the SoC. This can often be time consuming. Automatic and algorithmic based test generation is a solution. The use of assembler instructions or lower level signals based stimulus is largely limited when used for parameterisation of test cases for automatic test generation. In our methodology, it is appropriate and beneficial to perform test generation at the software application level. During early developments of our methodology, we developed an automatic test generation based on randomisation to address this deficiency. The automated test generator made use of the application code building blocks extracted from real-life applications to build and parameterise many different varieties of tests; by selecting and chaining together different sequences of these test building blocks to compose a test.

Algorithmic test generation gap

Our first attempt at automatic test generation could be enhanced further. Similar to other automated test generators, the test generation creates many different variants of tests quickly using a 'brute force' random approach in the hope that a large number of tests will cover sufficient portions of the required test space. The downside is that this could be inefficient and is often wasteful of verification resources. The coverage attained saturates even with increasing tests.

Many automatic test generators employ randomisation. Whilst this provides a variety of tests, it does not guarantee full coverage or ensure all the important design functions are tested. Even worst, the randomness of tests may result in identical design functionalities being tested repeatedly and other design behaviours completely unverified. The problem with random based automatic test generators is the lack of algorithmic or optimisation procedures during its test creation process. Besides automation, the test set should be created in a strategic manner employing other test information like coverage results.

To this end, our research in test generation investigated and proposed the use of genetic algorithms and evolutionary strategies in automatic test generators to create software application test programs. The revised automatic test generator is more selective in the types of code building blocks chosen to create test programs, and can employ useful sequences of these code segments based on the coverage gained. The incorporation of genetic evolutionary methods in our test generation work facilitates a form of CDV usually desired in simulation based verification.

Test generation input parameters selection gap

As part of the test generation research, two other difficulties were also tackled. First, test generators are predominately controlled by input parameters that influence the test creation process, and ultimately the level of verification effectiveness. These parameters are powerful mechanisms that manipulate the types of tests generated. However, selection of values for these parameters is often ad-hoc, either chosen without much consideration (which can be detrimental to verification) or employing a refining approach from empirical results. Conducting such calibration experiments for the test generator is inefficient as it takes up valuable verification time and resources. To address this gap, we propose and develop an analytical approach to model the test generation for proper selection of input test generation parameters.

Multi-objective test generation gap

Second, in most test generators, only a single goal is established to drive the test creation process. For example, the success of a generated test suite is usually measured by certain type of coverage metric, and the test generator's aim is to maximise the SoC's coverage with its tests. However, a verification process imposes various requirements and must cater for multiple goals or constraints. There could be multiple coverage metrics, other test performance criteria, or verification resource processing or memory capacity limits to consider. Therefore, the test generation process should be driven by multiple objectives to provide more effective testing that satisfies greater scope of verification requirements.

We address this problem by extending our genetic evolutionary test generation process to operate with multiple objectives. For instance, the test generator can be driven by multiple types of coverage metric goals and also strives to minimise test sizes to achieve as high coverage as possible; subject to limitations from available memory to hold tests for simulation in any given verification platform.

1.6.3 Coverage gap

For our test methodology, the generated test suite must be accurately quantified. In application based testing, the goal is to exercise and verify design functionalities. Classical coverage measuring alone is insufficient because they do not report any useful information on what functionalities were tested. Traditional coverage metrics were often used to assess the effectiveness of covering various hardware design code implementation features, and were intended for verifications that used low level test stimulus. For different verification methodologies such as our software application approach, these coverage measures provide some guidance and certain useful information, but do not reflect the true intention of our verification test suite. They are employed to ensure the verification approach is sound and as effective as earlier techniques.

Instead, a new functional coverage measuring method is needed for the software application testing level, and to quantify the types of application functionalities tested by our verification methodology. Functional coverage identifies the design behaviours tested and quantifies the percentage of design functions validated. Current functional coverage techniques are mainly applicable for individual design modules only. They suffer from a state space explosion phenomenon (similar to that in formal verification) and measuring inefficiency. This is because the number of functional coverage points needed to represent a functional behaviour grows exponentially as the design size (and complexity)

increases. For verification of entire systems, the number of functional coverage points and functional scenarios that must be monitored becomes unmanageable.

A disparity exists between our software application testing and an efficient functional coverage method capable of handling the coverage requirements from an SoC. The functional coverage solution in this thesis is inspired from abstraction and graph based trajectory checking from the formal verification domain of *symbolic trajectory evaluation*. This solution reduces the functional coverage space and requirements of the measurement process.

1.7 Research overview

The three key areas of our research are: (i) the software application level verification methodology (SALVEM), (ii) test generation, and (iii) coverage. Figure 1.4 shows the three research domains and the technologies studied for application in each research area of this thesis. The three key research areas are interconnected and come together to provide a design verification solution that addresses important challenges in pre-silicon SoC testing.

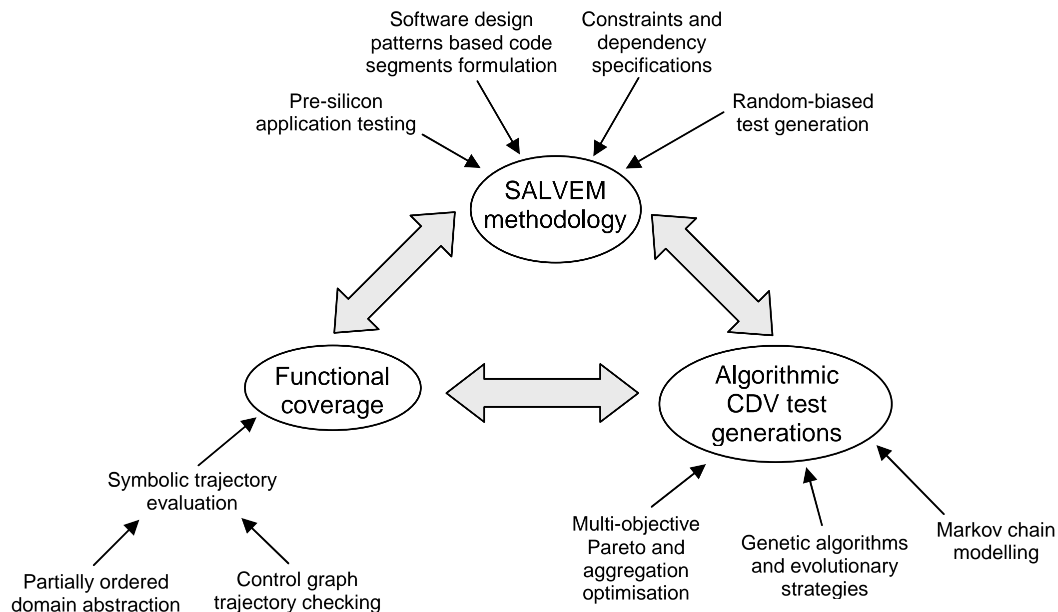


Figure 1.4 Research topics studied and technologies applied

1.8 Research objectives and contributions

Our research work and contributions in the design verification field complements existing verification techniques, rather than replace them. No silver bullet exists for conducting hardware design verification or uncovering design bugs. Many verification methods address different requirements during the extensive process of certifying design correctness. We limit the scope of our design verification research primarily within the domain of RTL simulation based verification. The goals of our research toward the design verification problem are to ensure correct operation of an SoC design, and maximise the likelihood of detecting system design errors before chip fabrication.

The research addresses the verification problems of the previous section. To achieve this, the objectives and associated contributions of the research thesis are as follows.

- Devise and formalise a methodology to apply software test programs based on real-life SoC applications at the early stages of RTL design simulations. The resulting verification methodology is named, software application level verification methodology (SALVEM).
- Develop a test generator that automatically creates software test programs efficiently according to our software application verification methodology.
- Facilitate effective and efficient testing of SoCs using our verification methodology by employing a coverage driven verification scheme. Genetic algorithms and evolutionary strategies are assimilated into the test generator to realise the coverage driven approach.
- Develop a Markov chain based analytical technique to model and examine genetic evolutionary test generation processes; in order to select test generation input parameters.
- Devise a multi-objective genetic evolutionary test generation technique that creates tests toward satisfying multiple verification goals. The multi-objective process optimises the suite of test programs to achieve best test coverage of several metric types using least resources (e.g., minimal test sizes).
- Derive a functional coverage method to estimate the effectiveness of our methodology and test comprehensiveness of our test generations. The resultant functional coverage method is able to handle SoC design sizes and measures SoC functionalities invoked from software application test programs.
- Prove the concept, feasibility and effectiveness of our design verification research by applying the methodology, test generations and coverage techniques to SoC designs. The Nios SoC [Alt03] is used predominately, along with a digital signal processor that was used as a case study for applying our CDV test generation.

The research material outputs consist of a complete verification tool suite encapsulating the software application verification methodology to test SoCs at the RTL simulation level. The tool suite includes:

- A library of application code segment building blocks and device drivers, which is used to compose test programs for the Nios SoC and digital signal processing SoC.
- Constrained-biased random test generator to create tests automatically.
- Algorithmic single and multiple objectives genetic evolutionary test generators to facilitate coverage driven test creations, and simultaneous optimisation of multiple verification goals.
- Procedural outline and examples of Markov chain designs that model sub processes of genetic evolutionary test creation; to be used for analysing and selecting input test generation parameters.
- Functional coverage measuring tool.

The peer-reviewed published papers include:

- A. Cheng, A. Parashkevov, and C.C. Lim, "Coverage Measurement for Software Application Level verification using Symbolic Trajectory Evaluation Techniques," in 2nd IEEE International Workshop on Electronic Design, Test & Applications (DELTA2004). Perth, Australia: IEEE Computer Society, 2004, pp. 237-242.
- A. Cheng, A. Parashkevov, and C.C. Lim, "Verifying System-on-Chips at the Software Application Level," in IFIP-WG 10.5 Very Large Scale Integration System-on-Chip (VLSI-SoC'05). Perth, Australia: 2005, ISBN 07298-0610-3, pp. 586-591.
- A. Cheng, A. Parashkevov, and C.C. Lim, "A Software Test Program Generator for Verifying System-on-Chips," in 10th IEEE International High Level Design Validation and Test Workshop 2005 (HLDVT'05). Napa Valley, California, USA: IEEE Computer, 2005, pp. 79-86.
- A. Cheng, A. Parashkevov, and C.C. Lim, "Coverage Measurement for Software Application Testing Using Partially Ordered Domains and Symbolic Trajectory Evaluation Techniques," in 3rd IEEE International Workshop on Electronic Design, Test & Applications (DELTA2006). Kuala Lumpur, Malaysia: IEEE Computer Society, 2006, pp. 481-487.
- A. Cheng, A. Parashkevov, and C.C. Lim, "Analysis and Optimization of Attribute Combinations Coverage for Software Application Testing," in IEEK International SoC Design Conference (ISOCC2006). Seoul, South Korea: IEEK SoC Society, 2006, pp. 482-485.
- A. Cheng and C.C. Lim, "System Test Generation for System-on-Chips using Genetic Evolutionary Methods," in 7th International Conference on Optimization: Techniques and Applications (ICOTA7). Kobe, Japan: Universal Academic Press, Inc., 2007, pp. 1-12.
- A. Cheng, C.C. Lim, Y. Sun, H. He, Z. Zhou, and T. Lei, "Using Genetic Evolutionary Software Application Testing to Verify A DSP SoC," in 4th IEEE International Workshop on Electronic Design, Test & Applications (DELTA2008). Hong Kong: IEEE Computer Society, 2008, pp. 20-25.

- A. Cheng and C.C. Lim, "Multi-Objective Genetic Test Generation for System-on-Chip Hardware Verification," in First World Congress on Global Optimization in Engineering & Sciences (WCGO2009). Zhangjiajie, China: 2009.
- A. Cheng and C.C. Lim, "Parameterise Genetic Evolutionary Test Generation with Markov Models," in Proceedings of the 4th International Conference on Optimization and Control with Applications (OCA2009). Harbin, China: 2009, pp. 41-49.
- A. Cheng and C.C. Lim, "Markov Modelling and Parameterisation of Genetic Evolutionary Test Generation," submitted to the Journal of Global Optimzation. 2009.
- A. Cheng and C.C. Lim, "Design Verification of System-on-Chips Driven by Multiple Test Objectives," submitted to the IEEE Transaction on Computer Aided Design. 2009.

1.9 Overview of thesis

Following this introductory chapter, Chapter 2 provides an overview of the prominent research that has been conducted by both academia and industry groups in the area of design verification. In particular, this chapter surveys the previous bodies of research in hardware verification methodologies, test generation techniques, and verification coverage solutions at the pre-silicon stages.

After Chapter 2, the remainder of the thesis is structured into three main sections: (1) our software application level verification methodology (SALVEM) for design verification, (2) the genetic evolutionary test generation techniques that facilitate coverage driven verification and optimise multi-objective verification goals, and (3) the functional coverage solution for our verification methodology. Figure 1.5 outlines the thesis flow.

In Chapter 3, the software application level verification methodology is introduced. In particular, the concepts of code segment building blocks that are extracted from real-life SoC usages and employed to compose various code sequences of software application test programs are explained. As part of Chapter 3, the SALVEM implementation is extended by developing a random test generator to automatically create software application test programs, and facilitate manual coverage driven verifications. The test generator composes random sequences of code segments in a random biased manner to demonstrate automatic test generation in our verification methodology. Note that the random test generation work is described as a case study fully in Appendix D as well.

The study of automated coverage driven test generation processes begins with Chapter 4. Building on previous automated test generation and manually driven verification procedures in Chapter 3, genetic algorithms and evolutionary strategies are adopted into the test generation process. One of the key elements of this chapter is how the software application based test generation is encoded into a genetic

evolutionary process. Based on experimental studies, the viability and advantages of genetic evolutionary methods for automated coverage driven SALVEM testing is demonstrated. Chapter 4 also describes a case study whereby the coverage driven SALVEM verification was applied to a digital signal processor SoC

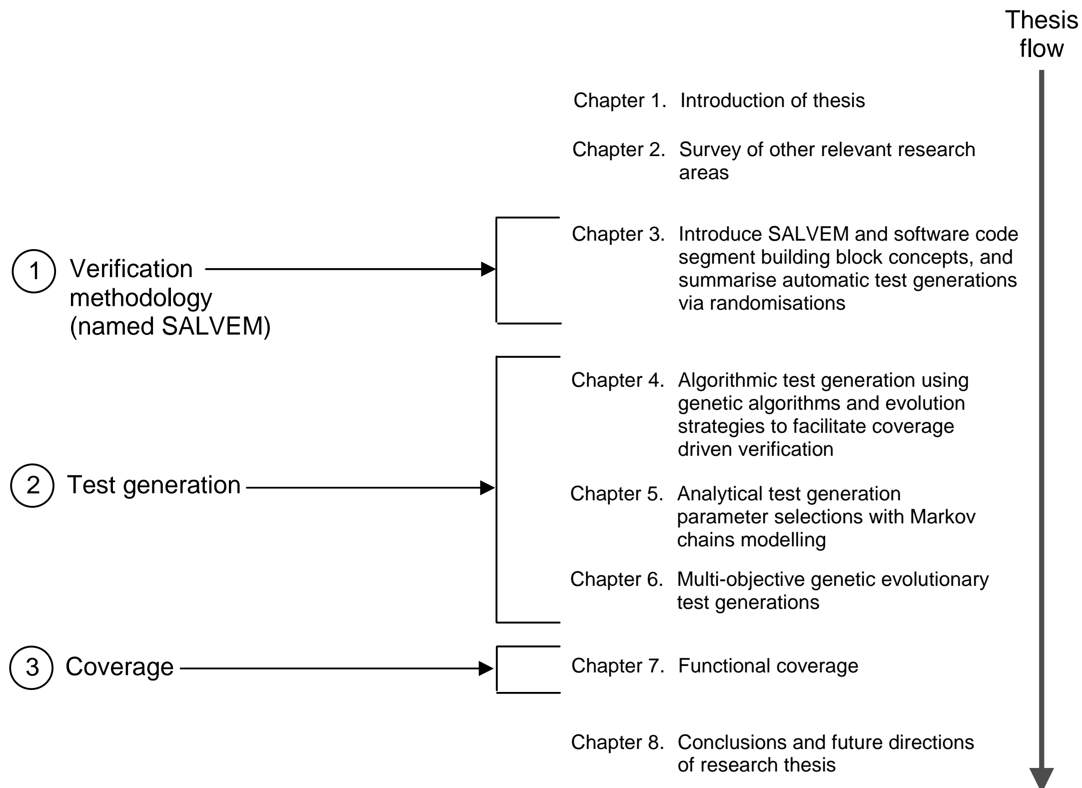


Figure 1.5 Research flow of this thesis

The success of coverage driven SALVEM testing with genetic evolutionary methods depends largely on the genetic evolutionary and other verification parameters chosen, and the internal evolutionary processes conducted. In Chapter 5, an analytical technique to study genetic evolutionary test generations is presented. Using Markov chains to model the sub processes and components of the test creation process, input test parameters are identified and chosen with values to enhance the test generation process and ensure high quality tests for effective SoC verification.

In Chapter 6, the multi-objective genetic evolutionary test generator is described. The key to our approach lies in the use of aggregation and Pareto optimality for the multi-goal orientated test creation process. Aggregation and Pareto methods are used in a novel manner to perform important phases of the genetic evolutionary process, and are the facilitator for handling multiple test objectives. The benefits of our multi-objective approach are proven by the superior verification results it acquires compared with other test methods.

Chapter 7 discusses the functional coverage solution for SALVEM, named attributes combinations coverage. The coverage measurement process and coverage quantification metric is defined in this chapter. We also describe a scheme to reduce the functional coverage test space using abstraction, and facilitating efficient coverage measuring using graph trajectory checking.

Finally, Chapter 8 presents the conclusions of this research thesis, along with other possible avenues of research that can be investigated further based upon the work in this thesis.

CHAPTER 2. LITERATURE SURVEY

The three key areas which form the foundation of this thesis are design verification methodologies, test generation, and coverage. This chapter surveys and compares prior research in each area relevant to this thesis.

2.1 Design verification methodologies

Design verification methodologies can be classified into three general categories, (i) simulation based methods, (ii) formal verification methods, or (iii) a mixture of both formal and simulation methods known as semi-formal techniques. Our design verification methodology research falls within the simulation based domain, which is surveyed in the next section. The major advancements in formal verification and semi-formal methods are highlighted in Appendix B.1, given that certain formal techniques are adapted and employed for sub-flows of our verification methodology.

2.1.1 Simulation based verification

Conventional simulation based methods operate directly on some representation of the hardware design. Usually, the design to simulate is described by hardware design languages (HDL) or other forms of software based models [Ber03, RPS01]. By applying test stimulus to the simulator, various hardware chip design operations can be simulated and the design's functional behaviours are verified. During simulation, the response from the design's primary outputs and internal state elements are checked against expected results; to ascertain if the design behaved correctly and uncover any faulty characteristics.

Conventional simulation is considered an incomplete verification method because they can only consider a subset of a design's entire behavioural space. The sheer complexity of hardware designs today, in particular system-on-chips (SoC), implies an exponential number of circuit states and control paths. Exhaustive coverage of all design behaviours is not possible given current simulation computational capabilities [Ber03]. The tests applied for simulation must be selective in the types of design functionalities it exercises to ensure adequate verification. The methodology devised in this thesis focuses on critical design functionalities that are widely used by real-life applications.

Simulation based techniques remain the most popular form of design verification because it is relatively easy to set up and use. Many simulation tools are readily available today [Cad, Men, Syn]. These tools are all compatible with common HDLs such as Verilog or VHDL [Ber03, RPS01], and can even handle higher level behavioural models of the design, described in traditional software languages (e.g. ANSI-C/C++) or software for hardware extended languages such as SystemC [OSCI, RPS01] Additionally, other useful debugging tools such as signal waveform or design schematic viewers, and coverage measurers are often integrated into the simulation framework. This makes simulation highly attractive to first time users. Furthermore, simulation techniques scale better with increasing design sizes compared with formal verification methods.

For these reasons, the verification methodology devised in this thesis was chosen to operate within the simulation domain. It addresses certain simulation shortcomings by creating and applying tests that targets hardware behaviours from the viewpoint of the design's eventual usage. The verification strategy is to exercise the design with higher level software application functions using simulation early in the design cycle. Rather than blindly simulate any possible design behaviours, we focus on validating the important hardware operations that are used by eventual applications of the chip design.

A survey of relevant simulation based verification schemes is given in the remainder of this section, comparing against our proposed application driven methodology. In particular, our survey focuses on the system testing domain, which governs our verification methodology. Unlike unit or block testing, the goal of system testing is to verify the interactions between individual design blocks within the system, and ensure correctness of system-wide functions throughout the entire chip design [Ber03, RPS01]. Appendix B.2 describes unit, block, and system testing in general. A form of system testing which bears some similarities to our verification technique is transaction based verification, which we discuss next.

Transaction based verification

Transaction based verification is an implementation of the system test concepts [BCG$^+$00, RPS01]. Instead of manipulating conventional 0 or 1 logical signals at the design pins and interconnects, testing is conducted in terms of transactions. A transaction can be an individual or collection of tasks that perform some high level data or control transfer between the components in a design, testbench, or simulation environment. The transaction can be simple reads or writes of memory data units, a series of error handling interactions, or transferring of communication packets throughout the entire system.

Given the complexities of tackling the entire design in system testing, it is much more practical and intuitive to test design functions using transactions instead of low level signals or waveform representations. A shortcoming of transaction verification is that the transactional tasks must be created manually from scratch, based on design implementation specific details. Extensive transactors and transaction verification modules must also be developed to trigger these transactions, along with suitable testbenches and simulation environments. Considerable set up costs and effort are required in transaction verification.

The SoC test operations invoked by our methodology for design testing can be considered similar in some aspects to that of a transaction. However, in our approach, our test cases invoke many different sequences of interacting transactional-like operations, that are automatically extracted from actual applications of the SoC during real-life usage.

An example of transaction verification is the XGEN test generator [EJN+02], which was used to verify Cell based architectures [SGK+06]. XGEN operates at a similar level to our methodology, but differs in how tests are generated. XGEN employs a specialised model of the system under verification; manually identifying the low-level components, interactions, and configurations possible in the SoC. Using randomisation and user templates, tests are created to perform transactions based on the system model. In comparison, our approach is to categorise the range of SoC applications, and break them down into *snippets* of high-level test building blocks to automatically create many different tests from these *snippets*.

The drawback with XGEN test generation is its randomisation approach whereas our methodology is flexible to employ both randomisation and genetic evolutionary algorithms, or any other test generation schemes. Our method operates at a higher level of abstraction. It uses tests composed entirely from the viewpoint of software application programs and the eventual real-life usages of the design. Based on application usage information, larger and more diverse SoC designs can be tackled.

Hardware software co-verification

Another verification scheme of interest to our research is hardware software co-verification. Co-verification tests the hardware and software functional capabilities of a chip design concurrently [Har04, Hub98, RPS01, SG00, Tur04]. Software from the software development team is tested directly on the hardware chip design. At the same time, the chip design provided undergoes verification using

software programs as test cases. Real-life stimulus inputs are applied to the design to exercise the chip through actual operating conditions. Under this approach, the software developed for the chip design is checked to ensure it can control the hardware, whilst hardware design functionalities are verified to ensure that the design's application requirements and performance specifications are met.

The goal of hardware software co-verification is to verify a design's hardware and intended operating software, in order to detect bugs earlier in the design cycle using application based testing. However, co-verification is often hindered by software test programs that cannot execute because of simulation speed and computational resource limitations. Without test simulation speed-ups, many software test programs cannot be applied.

Co-verification simulation speed-ups are usually realised by three methods, emulation, hardware acceleration, and rapid prototyping systems [Cad04, HZB+003, RPS01]. Emulation employs configurable hardware based equipment to emulate the operations of the chip design under test. In hardware acceleration, certain components of the simulation system are mapped to actual hardware devices that can mimic their operation at hardware speeds. For example, the simulation operating performance of an SoC's CPU can be accelerated by external hardware acting as a virtual processor and integrated into the simulation environment.

Rapid prototyping systems model the chip design using high level software languages entirely. Tests can be run faster but the downside is that the true hardware chip design representation is not directly tested. This set up is primarily for software development and testing of the applications that will eventually be loaded onto the chip design. For actual verification of the chip design, emulation and hardware acceleration are viable options. However, expensive hardware test equipment is needed to facilitate such approaches.

Our verification methodology shares similar goals as co-verification and also other software based self-test techniques. That is, to apply application based testing early in the design cycle using software test programs. Additionally, our method is applied directly to the chip design and avoids the need for expensive hardware test equipment. It employs a novel technique of breaking down applications into building blocks and recomposing them into effective test programs that are shorter, more manageable, and can be simulated by conventional means. Our test programs are made up of many different application test building blocks. A wider range of application functions are verified compared with using a set of standalone software programs from software development teams.

Coverage driven verification

One other verification strategy worth examining is coverage driven verification (CDV). During simulation based verifications, many tests are applied to verify the hardware design. The large number of tests can be loosely guided by the verification team, and may even be random if such test generators are employed. Even if a large number of tests are generated, the design space verified can be unreliable with possible pockets of unverified design holes. The verification team has no accurate measure of how much of the design was verified, and where to focus their efforts next.

The strategy in coverage driven verification addresses these deficiencies by,

1. determining the quality of verification – i.e., how comprehensively a design has been verified; and
2. directing the verification toward areas in the design that needs to be tested – i.e., ensure verification is thorough.

The goal of CDV is to achieve maximum coverage, and hence increase the probability of detecting errors, but using minimal test resources. CDV can be applied using two different approaches, CDV by construction [GFL+96, HYHD95], or CDV by feedback [FZ03, NMUZ03]. We survey CDV by construction in Appendix B.3, and discuss CDV by feedback here because it is the conventional method more commonly employed by the verification industry.

CDV by feedback relies on coverage information measured during test simulation to enhance and refine the verification test suite for future testing. The goal is to increase likelihood of bug detection by extending coverage of the design to previously unverified behaviours; or continue to stimulate previously beneficial regions of the test space that provided high coverage – this allows useful deviations from the same types of SoC operations that were effective at bug detection.

The CDV by feedback flow is shown in Figure 2.1. The flow is an iterative cycle of test simulation and coverage measuring, coverage analysis, and creation of more effective new tests for the next cycle based on previous coverage information and test creation directives. Whilst the CDV flow can be conducted manually, this is often a painstaking and complex process for engineers. Full automation of CDV remains a major gap in design verification.

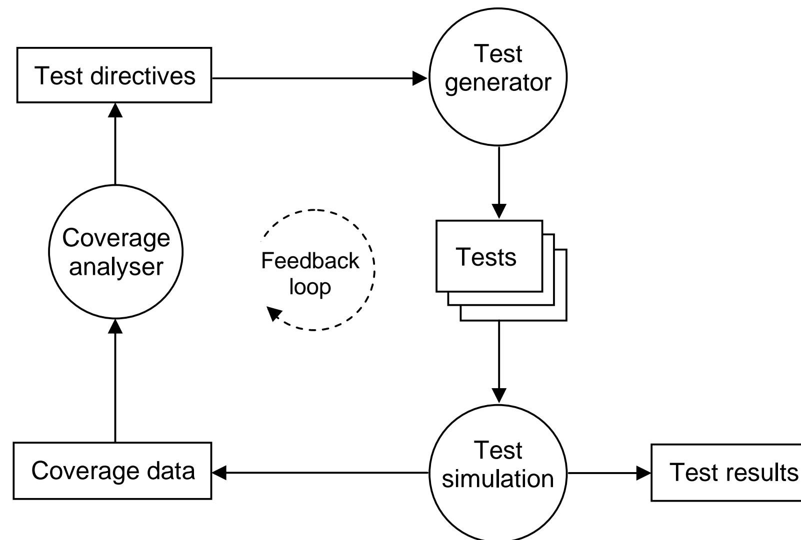


Figure 2.1 Coverage driven verification by feedback flow

A number of techniques have been proposed to automate CDV. In [NMUZ03], Nativ et al. present an integrated microarchitecture test generation and simulation framework. Their framework uses a rules module to deduce un-simulated coverage tasks and automatically apply stimuli to cover them. The rules module is a database of simulation directives that are mapped to specific coverage tasks. Despite CDV automation, the disadvantage with their approach is their reliance on experienced engineers with in-depth design knowledge to create a rules module for each design. The scalability of their approach was not discussed. For larger designs, applying tests using low-level signal stimulus would not be practical given the size of the rules module.

Fine and Ziv also devised a CDV flow to verify microprocessor units [FZ03]. Their CDV flow is modelled as a Bayesian network, and learning algorithms are used to train the CDV process. Initially, the relationships between coverage information and test generator directives are examined to tune the Bayesian network parameters. Statistical methods are then employed to identify the best test generator directives from the network to cover unverified tasks. Their technique was able to test pipelining functions and mainframe storage control units, even attaining higher coverage than manual CDV. The capacity of this method for larger designs is not clear however, as experiments were limited to unit-level testing of certain processor modules only. Furthermore, no performance results were given as to the efficiency of their automation, or how much overhead the learning and statistical processes imposes.

The GOTCHA coverage driven test generator by Benjamin et al. [BGH⁺99] operates on a finite state machine (FSM) based model of the design under test. The GOTCHA tool uses Murϕ model checking to extract formal test specifications from the FSM. These specifications act as directives to the test

generator to exercise untested portions of the FSM during simulations. The advantage of using a model checking approach is its ability to automatically differentiate reachable and unreachable coverage space. But this approach suffers the same scalability drawbacks as earlier methods discussed above, and requires specialised formal verification domain knowledge to configure the model checker for use in other designs.

Other variants of CDV by feedback methods have also been proposed. Tasiran et al. [TFC+01] extended their observability based coverage method (Section 2.3.3) with random biased simulation and formal verification (FV) techniques to implement a CDV flow. Their method applies approximation analysis techniques on a Markov chain model of the design. The approach taken by Hekmatpour and Coulter [HC03] is not strictly an automated CDV process but a collection of useful methods that is applied at various verification stages. These methods use coverage data to calibrate and perform fine-tuning of random test generators to influence creation of tests that converges toward corner cases and complex test scenarios. Gluska [Glu03, Glu06] proposes coverage oriented as opposed to coverage driven verification. The strategy is to apply random testing at the beginning of verification to quickly detect simple and easy-to-find bugs. Once bug detection rates levels out, functional coverage is then used to aid test efforts. This approach is simplistic yet effective for saving valuable test generation and coverage resources at the start of verification. However, no detail is given regarding how coverage was used to direct the test generator.

The CDV methods outlined in this section have been devised for and experimented on specific types of hardware design, namely microprocessors or their sub-units. Their capacity for larger and more complex hardware designs is largely unproven. Additionally, some of these methods employ complicated semi-formal methods that are difficult to apply and reuse on other designs without prior domain knowledge or experience.

In our verification methodology, we propose an automated CDV approach that is not tightly coupled to the type of design under test, and able to handle SoCs. The strength of our CDV approach lies with our test generation process, and use of the code segment building blocks to compose our tests using genetic algorithm and evolutionary strategy.

2.1.2 Design verification methodology summary

The verification methods surveyed in this section provide no suitable solutions for explicit testing of SoC designs early in the pre-silicon stage, using application functionalities which are critical for eventual usages of the SoC. Besides co-verification, no application driven testing is conducted. Even

then, co-verification is only possible just before fabrication with the aid of hardware testing equipment. Our verification methodology addresses this problem, applying application based tests using CDV at the behavioural simulation stage.

2.2 Test generation techniques

Test generation provides the stimulus that drives a hardware design to verify the design and uncover bugs. The creation of test case stimulus is challenging because identifying test stimulus that exercises desired design functionalities is not straightforward, and different chip designs present different functional scenarios to verify. A survey of test generators pertinent to our automated and algorithmic test creation methods is given in the following sections.

2.2.1 Random test generation techniques

Random test generators are the most common form of automated test generation methods because they are easy to set up and able to generate many tests quickly. Random test generation can usually cover up to 80% of the design space quickly. For example, Taylor et al. [TQB+98] reported that 79% of bugs in the DEC Alpha design were uncovered by randomised tests. In [SA98], the VERTIS test generator uses randomised instruction sequences to conduct design verification and self-testing of functional behaviours. The study of random test generators has been led by the IBM verification research group at Haifa in Israel. Their random test program generator (RTPG) tool was one of the earliest random test generator tools. It was used to verify the IBM System/6000 reduced instruction set computer (RISC) design [ABD+91], but user control was limited.

Typically, random tests exercise many simple test scenarios to find non-complicated and obvious bugs during early stages of random generation. But given enough time and resources, more complex corner cases could be derived by random generations. Over time, the likelihood of exercising more complex interactions slowly increases. Indeed, putting aside simulation speed and testing resource considerations, if the random test generator could be run forever, all possible design scenarios could eventually be realised to provide 100% test coverage. The other benefit of random test generators is their ability to create unexpected test scenarios which a human engineer would not have normally conceived.

Despite their simplicity, random testing becomes ineffective as the design size increases [Abr90]. Additionally, randomness can cause holes in the design space that are not exercised or only covered with low probability. Another side effect with random tests lies in debugging difficulties compared with manually created tests. Additional effort is needed to identify and analyse the various interactions invoked by the random test generator.

Despite such shortcomings, the continued popularity and benefits of random test generators have driven further research in this test creation strategy. The areas of developments include model-based testing, user biasing, test templates, and constraint solvers.

Model-based test generators [AGL+95, FAL99] operate on a model of the chip design under verification. In this way, the tool is not tied to any one particular implementation of a design, and may be reused to verify other designs. In biasing, test generation parameters (e.g. those in [AAF+04]) that shape the random test set are supplied with various values to steer testing toward desired design scenarios.

Template enhanced testing [EJN+02, WBA05] also provide an inlet for users to influence randomly generated tests. In this approach, tool users lay down the basic and minimal framework of a test, specifying only various elements of the test that ensure certain types of interactions or scenarios are exercised. The remaining characteristics of the test will be supplemented by random values from the test generator.

Increasingly, random test generators are being enhanced with formal verification techniques, such as adopting constraint solving capabilities to facilitate semi-formal test generations [AAF+03, AAF+04, ABPZ03]. By encoding the random test generation process, and design and test parameters as constraints, certain test generation decisions can be made more effectively by solving these constraints; whilst the remaining test creation process will still rely on randomisation. Appendix B.4.1 reviews in more detail the model-based, biasing, templates, and constraints randomisation research domains, comparing these domains to our research.

In our test generation research, we extend random test generation and its test creation components such as biasing and templates for the proposed software application level verification methodology. Chapter 3 will provide a full account of the methodology.

2.2.2 Genetic and evolutionary algorithms for test generators

Employing genetic and evolutionary algorithms (GEA) [BHS97, Fog94, MHM97, Mic94, Mic96] for design verification, or even post-silicon automatic test pattern generation (ATPG) is a promising approach to test generation. In GEA test generators, tests are considered similar to living organisms that evolve to produce new tests in the same way as biological evolutionary processes. GEA test creation uses the previously generated suite of tests to produce new tests. These new tests, if effective from a verification context, are then added into the current test suite so future tests can be produced based on these new tests. The GEA process is iterative, generating new tests by evolving the previous test suite over many cycles.

For conventional simulation based verification, GEA has been used to create tests to verify hardware designs at various levels of the design flow, between pre-silicon verification to post-silicon chip testing. For example, Samarah et al. devised a scheme whereby tests described in their cell based structures were optimised using genetic algorithms, and then applied to various design modules described by SystemC [SHTK06]. Yu et al. used a genetic algorithm based test generator to create sequences of input signals for simple circuit designs that can be reused at the register transfer level (RTL) and gate levels [YFFR02]. In [LLR+00], GEA is used to automate generation of test benches for simulation-based validation of RTL designs. And finally, test signal sequences for RTL simulation are created in an evolutionary manner using the ARPIA tool in [CCRS01b].

Combining pure randomisation based test generators with GEA can also be effective. Previously, GEA was used for the generation of assembler instruction test programs that tests processor designs [ASR+04, CCRS03a, CCS03, SBF97, Squ05]. In [SBF97], Smith et al. showed that GEA can be used to create variable length assembler programs to validate cache access arbitration units on processors. Corno et al. extended this approach further and adopted a mixed genetic and evolutionary technique for generating complete assembler instruction test programs [CCRS02c]. Their approach uses a GEA test generator named μGP [CCS03], to generate tests that can verify entire microprocessor cores. They have proven their technique for a number of processor designs [CCRS02a, CCRS02c, CCRS03a, CCRS03b, CCS03, CS03, CSRS04a, CSRS04b, Squ05]. μGP generated test suites was able to achieve improved line and toggle coverage compared with other types of random based test generators or manual test creation efforts.

Despite μGP's successes, simulation was the main bottleneck in the evolutionary test generation flow. Subsequently, Sanchez applied μGP for hardware accelerated testing to address this issue [SSR+05,

SSV04a, SSV04b]. However, the drawback with his solution was the availability and costs associated with the hardware equipment needed to attain test acceleration.

Like other GEA based test generators at the RTL or gate level, μGP is solely focused for microprocessor design or miscellaneous modular design testing. The sequences of GEA influenced assembler instructions created are mostly effective for stressing the processor core only. The instructions themselves cannot initiate system wide transactions to test an SoC design. The assembler test programs are only optimised to test the micro-architectural design features, e.g. arithmetic and logic, load/store, pipeline, or cache units on a processor.

In the past, research on GEA hardware verification has predominately concentrated on ATPG for large sequential circuit designs or assembler instructions based test programs for microprocessor verification. Despite their benefits in other verification domains, adopting GEA for system-level testing of SoC designs is an area that has not been investigated; especially at the software application test level. This is a gap which our test generation research addresses.

We propose to combine GEA with the SoC system based test infrastructure of our verification methodology, to conduct testing at higher pre-silicon verification levels. We implement GEA as a facilitator for CDV. Despite its application for test generation, to the best of our knowledge, GEA test generation for software application level testing in a CDV flow has not been proposed. GEA was mostly applied for low-level assembler instruction test generation and post-silicon ATPG previously. Appendix B.4.3 surveys other related areas of verification and test research that also employ GEA, for example, ATPG or built-in-self-test methods in the post-silicon physical test domain. For completeness of the test generation literature survey, besides GEA, Appendix B.4.2 summarises other prominent types of algorithmic based test generations partially related to our work.

Despite existing benefits of GEA for test generation, our literature survey led us to identify a number of additional areas where further improvements for GEA test creation was possible. The two areas of research interest are, (i) multi-objective test generation, and (ii) test generation input parameter selections. We discuss multi-objective test creations first.

2.2.3 Multi-objective optimisation and genetic evolutionary test generations

Multi-objective test generation is an area rarely focused upon at the higher pre-silicon design verification level. Usually, a test generation process is driven by a single objective, such as the test coverage acquired or number of design bugs detected. We extend our GEA test generation research into the multi-objective domain, taking into account more than one verification goal. The aim of multi-

objective test generation is to create tests such that multiple objective goals of the verification undergo an optimisation process (e.g. via multi-objective GEA). The multi-objective test generation not only applies at our software application verification level, but is ideally suitable for conducting testing at other levels.

In multi-objective GEA test generations, or indeed any multi-objective process in general, the challenge is how the multiple objectives are handled. In single objective GEA, the aim is simply to achieve best performance results for an individual goal. The performance of the GEA process is measured by a quantitative value called the *fitness* value; which is associated with the objective. The greater the fitness value acquired from each test derived by the GEA process, the more superior results acquired for that objective. Extending GEA to a multi-objective domain, the management of more than one objective and their fitness values distinguishes the GEA optimisation process from one another. Handling of multi-objectives during the GEA flow is paramount to the success of the test generation process. With this in mind, our survey focuses on current strategies commonly employed for representing and manipulating multiple objectives in the GEA optimisation process.

Aggregation optimisation strategy

Aggregation combines the fitness values of multiple objectives together into a single fitness value. A range of aggregation methods have been proposed to combine multiple values, each with varying success. For example, fitness values can be added based on differing objective priorities assigned by the user, or simply summed together and averaged out amongst the number of objectives.

The reverse approach can also be used, whereby fitness values are included based on a penalty function. The lower the performance attained for an objective, or if the objective violates any constraints, the less influence that objective will have toward the final aggregated fitness value.

More elaborate aggregation functions have also been employed. Jakob et al. [JGSB92] assigned specially devised weights to objectives and summed their fitness values accordingly. Hajela and Lin [HL92] also extended similar weighting and fitness combination techniques. Goal attainment combines fitness values depending on how certain objectives satisfy certain goals [WM93]. And finally, various ranking systems for objectives and their associated fitness values have been implemented to produce numerous amalgamations of the final fitness value [Coe99].

Whilst it is easy to apply, the downside with aggregation is that certain objectives may dominate the GEA optimisation process. By simply combining the multiple objectives' fitness together, detection of any low performing objective fitness values can be easily lost within the aggregation process; optimisation for that objective may cease or be given lower priority such that poor fitness result for that objective will not be identified until too late into the GEA process.

Pareto optimisation strategy

Pareto optimising strategies for fitness assignment and GEA test selection into the next GEA evolution was first proposed by Goldberg et al. [Gol89]. Their approach was to sort and rank all the test solutions into different Pareto optimal subsets of tests, known as *fronts* [SD84]. Extensions of Goldberg's method have continued. These include ranking the solutions within each front based on the number of other solution it dominates [FF93]. The goal was to provide more fine-grained sorting. Also, Pareto strategies supplemented with additional objective priority or constraint information has also been proposed. However, these methods all require user intervention, which is discouraged for our verification methodology because automated test generation is desired. Other schemes that extend Pareto techniques significantly have also been proposed, for instance, the strength Pareto approach in [SMK00, ZT99]. Another method involves tournament selection when establishing the Pareto dominance of solutions. The drawback with this is that much larger populations are needed to provide a suitable sample set of solutions for the tournament [HNG94].

Non-aggregate and non-Pareto optimisation strategies

The remaining class of multi-objective GEA belong to methods that do not employ aggregation or Pareto based methods. For example, Schaffer's vector evaluated genetic algorithm (VEGA) [Sch84, Sch85] handle objectives individually by identifying groups of individuals within the population that perform best for an objective. The selection of solutions for the next population is then formed by intermixing solutions from each of these groups. Whilst VEGA is able to identify and preserve solutions that cater for each objective, partitioning and intermixing of solutions effectively amounts to averaging the fitness values of objectives. This produces GEA populations of solutions that are optimised with objectives that attain the best average fitness only.

Other non-aggregation and non-Pareto methods involve tournament selection schemes. For example, pairs of solutions were compared according to a randomly selected objective. Then, tournament selection is performed according to that objective to determine the wining solution into the next GEA population. This method is also equivalent to averaging the fitness values depending on which objective was chosen for the tournament selection [BT80].

Overall, non-aggregate and non-Pareto methods are not suitable for the needs of our multi-objective test generation. Rather than optimisation using average coverage fitness or test size, our test generation requires creation of tests to achieve the best coverage using the most efficient test size. For more detailed survey of non-aggregation and non-Pareto methods the reader is referred to [Coe99, TKK96].

Multi-objective GEA test generation summary

Based on the current methods available for multi-objective GEA, our approach presented in Chapter 6 extends current research by combining both aggregation and Pareto optimal methods to exploit their strength and compensate for their shortcomings. This produces a unique test selection and multi-focal goal management process for our multi-objective GEA test generation. Our devised method ensures fairness between all test objectives, and maintains creation of tests that are most favourable for each objective so that overall verification goals are satisfied.

2.2.4 Test generation parameter selections

During the process of conducting test generation research and this literature survey, it became clear that user input parameters to a test generation process is critical to the types of tests created, and their effectiveness toward verification quality. For instance, the number and size of tests, the priorities given to test certain design elements, or the assigned processing resources and duration for performing test creation. These factors all affect the kinds of functionalities that can be embedded into tests; including how such testing of design functions are invoked in the verification environment. However, in many of the test generators surveyed, it was obvious that not much consideration had been afforded as to how the test generation parameters and their values were chosen.

In the research literature, the procedure in which parameters and values were selected was commonly ignored, unspecified, or conducted ad-hoc based on intuition. In other cases, parameters were selected based on empirical results from certain pre-verification experiments or calibration of the verification

system [CCRS03a, CCS03, CSRS04b]. Therefore, an important focus and extension of our test generation research is to devise an analytical solution to examine our GEA test generation techniques. Our method allows for appropriate test generation parameters to be chosen that will benefit the verification process significantly.

2.2.5 Test generation summary

Compared with random-only test generators, the algorithmic test generators – such as those incorporating genetic evolutionary methods in Section 2.2.2 – are better at exercising complex corner case scenarios, which enhances coverage attainment levels and testing efficiency as well. The downside is that many of these test generators are highly customised for a specific design type or verification flow. In addition, effective usage of these test generators often requires sound knowledge and some level of expertise in the algorithmic domains which the test generators are based on. Previously, many test generators focus on microprocessor verification despite the increasing number of SoC designs. These test generators operate only at the microarchitectural or signal pin stimulus level, and create tests as assembler instruction programs or test vectors. There is a lack of system level test generators employing higher level software application programs to verify design functions.

To address this problem and enhance automated test generation further in our methodology, we first extend our automated randomised test generation to verify entire SoC designs at the software application level. Next, we enhance our test creation processes further, by employing algorithmic strategies such as GEA, multi-objective GEA optimisation for satisfying various verification goals concurrently, and an analytical approach toward selecting appropriate test generation input parameters with best values.

2.3 Coverage measuring

In design verification, coverage is used to determine the quality of testing and manage verification resources. Coverage measures the completeness of a test suite and reports how effective it was in verifying the design. Quantitative coverage metrics are often employed to monitor the progress of verification and estimate the remaining effort needed to achieve validation goals. A coverage measurement result that satisfies verification objectives provides confidence to the project team that

the design has been sufficiently tested and is ready for fabrication. Coverage can also be used to direct testing in a CDV flow as previously discussed in Section 2.1.1.

Establishing the criteria for measuring verification completeness and functional correctness of the design is difficult. Wang et al. [WHY03] established requirements which a good coverage solution must demonstrate. These requirements, (1) *Accountability*, (2) *Coverability*, and (3) *Efficiency*, are useful for evaluating the effectiveness of coverage methods, and are described as follows.

1. *Accountability* – each coverage test event must be counted once only; this ensures accurate and non-inflated coverage results.
   A coverage test event is an event that occurs during simulation as a result of testing, and differs depending on the coverage method. According to Moundanos [MAH98], the quantitative definition of a coverage metric is the ratio of exercised coverage events over the total number of possible events.

2. *Coverability* – high coverage (>95%) must be an attainable and practical goal.
   A metric that cannot progress toward and acquire high coverage percentages regardless of how many tests are executed and the effort expended by the verification team, is ineffective for estimating verification progress.

3. *Efficiency* – coverage measurement should incur low overhead, and not adversely affect the overall verification process.

In addition, to facilitate a CDV flow, the coverage method must be able to provide appropriate feedback information to guide the test generator.

Given similarities between design verification with software testing and post-silicon physical testing, many coverage methods for design verification can be traced back to the software domain or physical test origins. Despite their applicability and widespread use, the effectiveness of these coverage methods for design verification varies. The following sections provide a survey of common verification coverage methods currently in use today, including those adopted from software and physical testing.

2.3.1 Design bugs and error modelling

The goal of design verification is to uncover design bugs and prevent their propagation onto the fabrication stage. Therefore, the measure of the design bugs detected during verification could be adopted naturally for coverage purposes.

A promising approach is to examine and adapt fault modelling methods from the physical hardware testing domain. In physical testing, generic fault models such as node value stuck-at faults, or signal paths and propagation delay faults can be identified throughout the circuitry of a chip design [Cro99]. These fault models account for common imperfections in the manufacturing process such as doping, masking, or metallisation fabrication defects. Full coverage of these faults ensures all possible hardware failures from the fabrication process are tested. The method is highly effectively for physical testing because it captures the entire set of hardware faults.

For design verification, similar design error models that can encapsulate all the possible design bugs are needed. Unfortunately, the classification of such error models is difficult. Chip design is a highly complicated process, and many types of errors may be committed at any stage due to any number of possible reasons. Each design is different and design bugs are inherently dependant on the designers' personal skills and experience.

Campenhout [Cam99] tackled error modelling for functional verification of microprocessors. His error models are closely coupled to physical testing fault models, and can be deployed for test generation and error checking, not just coverage. However, Campenhout's error models are derived solely from empirical data attained from many design projects undertaken at his research department. Whilst these error models have been demonstrated to be highly effectively for verifying pipelined microprocessors from those projects, there are certain dependencies and concerns with its wider applicability. Given their reliance on empirical data, the usefulness of these error models degrades over time as advances in design architectures and methods bring about new families of design bugs that would not have been encountered previously.

Velev also analysed and classified different types of design bugs from formal verification of pipelined and superscalar processors [Vel03]. Though it is unclear how effective error models from these categories of bugs would apply for simulation based verification, or other types of system based designs. In [ZH00], the definitions and incorrect assignments of data variables in the design's HDL code are employed as a form of error modelling and coverage measuring. In [KS94], design errors are modelled from logical functions that cause changes in the primary output response if such functions were mistakenly altered in any way. Unfortunately, the theoretical analysis conducted to identify these

functions operates only on low-level circuitry and would not scale well with large designs. In [KS92], the design errors devised include instantiations of incorrect gates or design modules in the RTL design code.

For design error modelling, whilst accountability and efficiency is not an issue, coverability is problematic because it is extremely difficult to devise a complete set of error models to ascertain the total number of design bugs possible in the design beforehand. Unlike physical fault models which are based on solid foundation from manufacturing defects, design error models cannot follow any guaranteed type of design bugs. The errors that eventually led to design bugs are too diverse. Identifying and defining generic error models that can cover all the design bugs across all types of design types is not feasible.

Without knowing the possible types and total number of design bugs, no measure of progress or percentage of bugs uncovered can be reported making this method ineffective. In the past, it was proposed that design projects monitor bug detection rates and only permit tape-out when there have been no new bugs in the design after long periods of test simulations [Cla91]. However, the appropriate period of time to wait once the last bug has been detected remains an open issue as well.

Our coverage method does not rely on error modelling in any way. Hence, it does not suffer from the same shortcomings as the above methods. In our method, the only instance of modelling involves the functional behaviours invoked by our software application test programs. Such modelling is via the measurable attributes of the functional test operations from our code segment test building blocks, and can be identified fully.

2.3.2 Code based coverage

Behavioural RTL hardware designs are described using hardware design languages (HDL) similar to programming languages for software development. Given their similarities, code coverage techniques from software testing domains are often employed to measure coverage for hardware designs [Bei90, UZ02, UZ98]. In code coverage, the syntactical characteristics of a design's hardware code description are examined to measure verification completeness [Ber03, RPS01, WT95].

Code coverage is a measure of verification comprehensiveness that is low in complexity. They provide a preliminary assessment of the status of verification. For design projects today, conducting code coverage is a prerequisite for design verification. The goal of code coverage is to ensure correctness in

basic functional quality of the chip design. Usually, code coverage goals must be achieved before other more specialised testing and sophisticated coverage measures are employed to target complex corner case scenarios.

Code coverage is also incorporated in many verification tool packages [Cad, Men, Syn], because it is simple to set up. To conduct code coverage measurements, the design's HDL code is instrumented with measurement variables that are associated with design code elements. During testing, if any applicable code element is exercised, the associated variable is updated. Code instrumentation ensures accountability in code coverage.

Despite its popularity, code coverage alone is not always sufficient for verification. Even if 100% code coverage can be achieved, given the concurrent nature of hardware designs, full code coverage do not always represent sufficient testing of design scenarios and behaviours to uncover all possible bugs.

The most common types of code coverage in use are today, (1) line (or statement) coverage, (2) toggle coverage, and (3) conditional (branch and path) coverage. These code coverage variants are closely related in terms of the information they measure, but each one provides unique perspectives into how the design was tested. Together, they can be combined to direct creation of tests that comprehensively exercise all syntactical and semantic aspects of the design's HDL code.

Line coverage

Line based code coverage monitors which lines of HDL statements in the design were exercised during testing. Line coverage is the simplest form of code coverage and incurs low overhead. Each HDL statement is instrumented with a counter that counts the number times that the statement is exercised. Despite its efficiency, line coverage lacks useful test information. For example, it is not clear whether multiple hardware operations were conducted independently, simultaneously, or otherwise, by examining line coverage. Even if full coverability can be satisfied, exercising all lines in the design code does not account for the different modes in which each statement can be exercised. Toggle and conditional code coverage extends line coverage to capture the data and control characteristics of the design's code description.

Toggle coverage

Toggle coverage measures whether design code state elements have been exercised with sufficient data values, and how often these elements are exercised. In actual hardware design, these toggling code elements generally map to design state elements or storage nodes. Toggling such design nodes

frequently with diverse values ensures the functional logic manipulated by these nodes is thoroughly tested. The assumption from toggle coverage is that a greater range of state element values exercised by nodal elements results in more hardware operations executed.

Full toggle coverability requires all state elements to transition through all possible values. As design size increases, applying toggle coverage brings with it scalability issues. The number of nodal state elements and values to toggle becomes intractable. Therefore, more selective toggling criteria must be imposed to reduce the toggle coverage space. For example, toggling boundary or other interesting values only. Like line coverage, accountability and efficiency criteria are easily satisfied by toggle coverage.

Conditional coverage

Conditional branch and path coverage checks the traversal of control flow paths through the design code during testing. The coverage method requires each conditional code element to assert true so that every possible path from these branch points in the design code is taken at least once. For instance, conditional coverage requires each branch decision at the HDL's 'case' or 'if' constructs to be chosen. Whilst line coverage measures what lines were exercised only, conditional coverage identifies the sequential control flow from where these design code statements are exercised.

Conditional coverage is the most effective form of code coverage because it captures control flow information. However, full coverage is very difficult to attain. There are many conditional and branch points in a design, hence the number of control paths to traverse increases exponentially with design size. Also, greater computational resources are needed to track and monitor conditional control paths, hence efficiency is lower compared with line or toggle coverage.

Code coverage summary

For our verification methodology and test generation experiments, we employ code coverage because it can be easily adapted for use in our design verifications research. Whilst it may not be the most compatible form of coverage for our software application test methods, it can still provide highly useful information and feedback regarding the effectiveness of our approach. Once feasibility and potential benefits of our techniques have been confirmed by code coverage, we devised functional coverage methods that focus on the hardware functionalities exercised by our software application test strategy.

Appendix B.5 summarises other variants of code coverage along assertion coverage; which is becoming more popular, but is not directly relevant to our functional coverage method. We include such additional coverage methods for completeness of this literature survey.

2.3.3 Observability tag coverage

Observability based code coverage metric (OCCOM) is an extension of conventional code coverage that focuses on the observability of design errors. The code coverage methods described in Section 2.3.2 relies on the activation of design code elements. These techniques concentrate on controllability of coverage events within the design code, however observability is ignored. In [DGK96], Devadas et al. demonstrate that exercising HDL design code alone can be insufficient, and the design code should not be considered covered. Their proposition is based on the fact that a block of defective code exercised by testing, cannot be regarded as covered unless incorrect responses are observed at the outputs. OCCOM is effective because it ensures functional behaviours that contain potential errors are properly exercised.

Devadas implemented OCCOM by associating a *tag* [DGK96] with design code statements, nodes, and branching paths that may contain errors. During testing, if erroneous design code is exercised, the associated tag is propagated through the circuit to an appropriate node where the effect of the error can be observed. Tag propagation is conducted using a novel scheme employing D-calculus rules. The coverage metric is defined as the percentage of tags that is propagated to an observable state, over the total number of tags injected into the HDL design code description. Following Devadas's work, Fallah et al. refined OCCOM by implementing propagation rules that enable more efficient tag propagation and coverage computation [FAD02, FDK01, FDK98].

Despite the efforts of Devadas and Fallah, the major disadvantage with OCCOM is its inefficiency. Besides tag propagation overhead, OCCOM requires the design code to be modified beforehand into a form suitable for injecting and propagating tags; which is troublesome and difficult to apply for any verification methodology (such as our software application test method).

In our coverage scheme, hardware design test observability is implicitly provided by directly measuring the hardware functionalities exercised; and in some cases, checking functional outcomes against expected behaviours by examining the design elements which are affected by our application tests. In addition, our coverage method is designed for software application verification (and any

functional verification approach in general). It does not require any prior HDL code conversion or manipulation of the design.

2.3.4 Finite state machine coverage

Besides design code, coverage can be measured against a finite state machine (FSM) or equivalent graph based model of the hardware design [BH99, HDA95, Ho96, LJ99, PSK94]. The FSM encapsulates design behaviours in terms of functional states and sequence of transitions between these states. During testing, the states visited and transitions traversed are recorded. Coverage is measured by the ratio of exercised states (or transitions) over the total number of states (or transitions) in the FSM.

FSM coverage offers a more accurate representation of how thorough a design has been verified because it measures the functional behaviours exercised. It is widely supported in many EDA tools [Men, Syn], and the FSM is often extended for other design verification usages such as test generation [HYHD95, MAH96, SA98, VK95]. FSM coverage is accountable but efficiency depends on the size and complexity of the FSM.

The primary limitation with FSM coverage is achieving full coverage. Given the size of modern hardware designs where thousands of register state elements are common, it is often impractical to model all states and transitions. Even if a model can be extracted, it would be too large to exhaustively exercise the entire FSM, violating the coverability criterion. Therefore, smaller FSMs are usually created by abstracting out datapath information. Such FSMs focus on modelling the control flow of the design instead, as most design errors relate to control logic [HDA95]. Ho has successfully shown that design bugs are more prone to the functional control interactions as well [Ho96].

In [HDA95] and [MAH98], Hoskote, Moundanos and Abraham devised a method to extract an extracted control flow machine (ECFM) of a design. The ECFM is a FSM that captures the control behaviours of a design’s functional operations exclusively, and can be extracted from any design. Coverage is measured by the proportion of the ECFM exercised. Moundanos extended this method further by defining the event sequence coverage metric (ESCM) based on a design’s ECFM [MA98]. The ESCM measures sequences of control events in the ECFM that must be exercised.

To handle even larger designs and satisfy coverability, Ho proposed FSM abstraction methods using a divide-and-conquer bottom-up approach, where the control design space is partitioned into smaller

FSMs interacting with each other [Ho96]. An incremental strategy is then employed to achieve high coverage for these smaller FSMs before larger and more complex interacting FSMs are introduced. In [LJ01], Liu and Jou reduce the size of their FSM by combining states (and transitions) that describe similar behaviours into a single but equivalent semantic state. Their semantic finite state machine (SFSM) can be applied to larger designs but is limited to certain design styles only. Bergmann devised a method called *projection directed state exploration* where only relevant portions of the design graph model that leads to interesting states are explored [BH99].

Despite these abstraction orientated counter measures, FSM coverage will still suffer from the state explosion phenomenon regardless. Hardware design size and complexity are still growing, so FSM coverage will require other forms of graph reduction methods to ensure full state and transition reachability remains practical.

Like FSM coverage, our functional coverage method employs graph techniques. However, our method does not suffer from the same graph coverability problems as FSM coverage. This is because we do not model the hardware design under test as a graph representation, and thus avoids design graph scalability issues. Instead, we graph the possible sequence of code segment building blocks that can compose test programs. The graph size is contained directly by the set of building blocks employed, which is intended to be minimal but still provide diverse range of application test functions.

The emphasis on measuring control flow design behaviours modelled in the FSM has led to FSM coverage being referred to as functional coverage in some literature [HDA95, KN96, MAH98]. In this thesis, we do not regard FSM coverage as functional coverage. We discuss functional coverage in its proper context next.

2.3.5 Functional coverage

One main drawback with the coverage methods discussed so far is that they concentrate on low-level hardware characteristics such as pin signal value transitions, or exercising various syntactical elements of the design code. Functional coverage focuses on measuring the functional operations tested in a hardware design [FZ03, GHO+98, LMUZ02].

These functional operations are usually captured by user defined models. The aim of these models is to describe critical and interesting design functions that need to be verified. Each of these functional operations can be associated with a set of attributes. Attributes can be variables or parameters of various hardware design elements that control how the design behaves or indicate what operations are

performed. Each attribute holds certain values. During test simulation, the realisation of particular combinations of attribute values demonstrates certain operations were performed and tested. The functional coverage metric is quantitatively defined as the percentage of attribute combinations exercised, out of all possible combinations.

Functional coverage models have been adapted for use in both the hardware and software domains by Ur, Fine, and Farchi [FU99, FZ03, UZ02, UZ98]. In their method, tasks are identified in which their design under test must perform. A functional task is considered exercised when the correct sequence of attribute values is observed from the simulation trace. Their functional coverage model is therefore the set of pre-identified coverage tasks. The goal of their approach is to achieve cross-product realisation of all combinations of attributes' values.

Functional coverage can be used to measure a variety of design properties. For example, in [GHO+98], a functional coverage model was developed to test if assembler instructions were correctly propagated through a microprocessor pipeline without conflicts or excessive delays. The coverage model consisted of two attributes, the first and second instructions supplied into the pipeline. To attain full coverage, tests were developed that executes all combinations of any first instruction followed immediately by any second instruction.

In [LMUZ02], to verify the floating point functionality of an instruction set specification, the coverage model was developed in terms of attributes such as instruction opcodes, operands, and result values. For instance, the coverage model can ensure all floating point instructions, using all possible operand values, producing all possible results, in various rounding off modes, are tested.

The Meteor and COMET functional coverage tools [GHO+98, NMUZ03] are two realisations of functional coverage research. In [FKL99], Fournier et al. made use of the Genesys test generator [AGL+95] and COMET tool to demonstrate how functional coverage can be used to create a test suite for validating PowerPC based architectural designs. Functional coverage was also employed for CDV by Intel on their Banias and Merom processors [Glu03, Glu06]. In [Ziv03], functional and assertion coverage are combined by attaching auxiliary variables to temporal assertions. These variables act as attributes to facilitate the use of functional coverage models for assertion coverage measuring. This approach is more efficient because the functional coverage model does not impose as much computational overhead as conventional assertion monitoring. By sharing common attributes between similar assertions, the number of assertions needed for a design is reduced as well.

The benefit in functional coverage is that the functional operations that must be verified are directly measured to determine if they have been tested. The downside is the potential blow-out in the size of the functional coverage model. The coverage model is dependant on the number of attributes and attributes' values. If too large a coverage model is created, it would be impossible to cover all the functional behaviours.

For larger designs, the number of coverage tasks also increases significantly, and there is overhead involved in identifying and monitoring these tasks and their associated attributes. Wang et al.'s coverability criterion [WHY03] would be violated. Therefore, the number of coverage tasks and attributes in the coverage model are chosen to take into consideration the available verification resources. Otherwise, methods to contain the model or conduct coverage measuring more effectively must be applied.

For example, in most coverage models, it is unnecessary to exercise all combinations because certain combinations of attribute values are considered illegal, cannot be realised by the design. In [LMUZ02], Lachish et al. created restrictions to describe these illegal combinations, reducing the coverage model size to attain full coverability.

Lachish et al. also reduced the size of the coverage model by grouping together attribute combinations that cover similar functional operations [LMUZ02]. These attribute combinations are considered equivalent and represented as a single coverage task instead. The concept of grouping combinations was extended further by extracting *quasi* and *query holes* to identify sets of uncovered functionalities quickly [AMF+05]. Asaf et al. [AMZ04] define the concept of coverage *views* onto the functional coverage model to reduce the number of coverage tasks. Coverage views combine and simplify vast amounts of functional coverage data. Each view reports only the relevant functions that are of interest to the user for verification.

In [LMUZ02], in order to conduct measurement more efficiently, Lachish et al. propose an incremental strategy, whereby a small number of attributes is chosen initially to obtain high coverage, before adding new attributes and exercising additional functional behaviours. The effectiveness of this method depends on selecting which attributes to maximise their coverage first. This ensures more critical attributes are not neglected. The difficulty with this approach lies with prioritising the attributes.

Functional coverage summary

In our coverage method, we also adopt a functional coverage strategy; but we devise and employ attribute combinations of specific hardware design elements which are specially verified by our software application test methodology only. We extend the functional coverage method to the SoC design domain to handle system-wide verifications. The containment of our equivalent functional coverage model is facilitated by abstraction of the individual attributes and their combinations. The abstraction method adapts formal techniques from symbolic trajectory evaluation.

2.3.6 Coverage summary

Each coverage method described in this section offers different feedback for assessing verification effectiveness. In a typical verification project, various coverage methods are applied to compliment one another to provide more comprehensive assessment of verification overall. This is also the approach undertaken for our verification methodology. Despite this, certain types of tests and verification methods would operate more productively if a compatible coverage solution was applied. Given the software application test nature of our verification research, functional coverage is the suitable solution we eventually focused upon.

Unlike functional coverage of microprocessor designs discussed in Section 2.3.5 and in [FKL99, GHO^{+}98, LMUZ02, NMUZ03], the SoC designs verified by our methodology are larger and complex. Also, previous functional coverage methods employ a *greedy* strategy, whereby many attributes and attributes' values are exercised. Blindly adopting a cross-product approach for SoC testing is not feasible. In our functional coverage, we ensure our coverage model is not larger than necessary by adopting a minimalist approach, where only the important or interesting attribute values are considered.

The problem with overly large functional coverage models is fundamentally the same as the state-space explosion phenomenon in FSM coverage and formal verification (FV). Therefore, our coverage method employs abstraction techniques from the FV field of symbolic trajectory evaluation. We describe the functional coverage method in Chapter 7.

CHAPTER 3. SOFTWARE APPLICATION LEVEL VERIFICATION METHODOLOGY

This chapter introduces our software application level verification methodology to test system-on-chip hardware designs. We describe the goals, concepts and methods of the methodology. In particular, what kinds of hardware functionalities are verified, how these test functions are invoked, and the technique to create test stimulus via software test programs. The methodology provides the platform for conducting further test generation and coverage research in this thesis.

3.1 Introduction

The software application level verification methodology (SALVEM) is a methodology specifically devised to facilitate application based testing of system-on-chip (SoC) hardware designs. The methodology simulates software test programs derived from eventual usages of the SoC. It describes a system for conducting pre-silicon simulation-based verification testing at the behavioural level. The methodology applies test stimulus designed to verify functionalities that are commonly used by real-life applications of the SoC. SALVEM defines the components and methods for setting up the verification platform, creating the software application-based test programs, and executing test program simulations.

SALVEM was initially motivated by verification projects at Freescale Semiconductors, which the thesis author was a project member of. It was observed that the use of application-like tests for simulation could trigger critical and previously unthought of bugs for detection. Compared with conventional test methods, the application based approach could uncover certain design errors earlier in the verification phase. However, the application driven method was carried out manually in an ad-hoc manner because of the different designs under verification and their complexities. Further research was needed to formalise and automate the application based approach into an effective and efficient methodology that can be applied to SoC designs.

A survey was conducted to find similar application based verification methods. As far as we are aware, no formalised method exists for application based verification of designs at the pre-silicon stage using simulation; hence the need for SALVEM. As discussed in Section 2.1.1 Chapter 2, the only methods operating with a similar strategy to our application test method are transaction based verification

[BCG+00, EJN+02, RPS01] and hardware software co-verification [RPS01, SG00, Tur04]. In comparison with these methods, SALVEM complements other conventional verification at lower levels, and tests design functions that will be commonly used by real-life applications of the chip design in its eventual operating environment. Such design functionalities are considered the most important, and should be tested earlier in the design process rather than later. SALVEM provides verification from the application use-case perspective to directly target these specific design behaviours.

Rather than hardware verification, co-verification is sometimes more useful for developing and debugging the chip design's software application programs before the actual fabricated chip is available. For instance, if emulation is used for simulation speed-up, this implies the design would have undergone significant pre-silicon simulation testing already; otherwise it would not be mature enough for use on such hardware test equipments and for software development. Additionally, co-verification with soft prototypes verifies a design supplemented with software models of the hardware. In contrast, SALVEM verifies the actual hardware behavioural description of the design during early stages of the design cycle in its entirety. The focus is solely on uncovering hardware design bugs, and no expensive hardware test equipment is needed for test simulation speed-up.

The primary objective of SALVEM is to facilitate a methodology so as to perform application based testing early in the design cycle, and verify the SoC design at the register transfer level (RTL). The verification must ensure a large percentage and wide range of application functions are tested. SALVEM must also be portable. Setting up and applying the verification methodology to verify other SoC designs should not be difficult. Additionally, elements of the verification method such as test generation should be reusable at other levels of verifications if required, e.g., creating tests for post-silicon physical testing or even co-verification.

To satisfy these requirements, the test cases applied must be devised strategically. Conceptually, our technique aims to identify and analyse application use-cases and existing application software of the design. This is followed by breaking down these applications into modular test building blocks, and then recomposing the building blocks to form many different types of software test programs of appropriate sizes for simulation. This concept forms the basis of the verification methodology, and is the key to our test creation process in SALVEM.

The aim of this chapter is to describe the SALVEM methodology for achieving the verification objectives outlined above. The chapter is organised as follows.

Chapter overview

The next section describes the concepts and procedural flows of SALVEM. Section 3.3 outlines how the SALVEM verification test platform is configured for executing different application based tests. Sections 3.4 and 3.5 explains the software test building blocks we denote as *snippets* for creating SALVEM test programs. This is followed by a description of the software test program creation process using these building blocks in Section 3.6. Section 3.7 conducts experimentation to analyse manually created test programs and verifications facilitated by SALVEM. Before concluding, Section 3.8 summarises a random test generation case study using SALVEM to automatically create SALVEM test programs. The full details of this case study are described in Appendix D.

3.2 Concepts and methods in SALVEM

3.2.1 Origins of SALVEM strategy

SALVEM originated from three key observations. First, software has taken up an increasingly influential role in the operations of today's hardware designs. Nowadays, hardware designs, in particular SoCs, contain much greater amount of software driven components. The hardware operations performed by a chip design are largely controlled by software applications and this trend is likely to continue [RPS01]. With this in mind, we examined the relationships between the SoC hardware and software domains, and how it can be applied for verification.

Our SoC verification framework is based on the use of embedded software applications to excite interactions across multiple software and hardware layers of the SoC architecture. Figure 3.1 shows the architecture levels of an SoC. It defines distinct boundaries across which operations invoked by software applications interacts with each other at higher levels, and eventually, requests for executing chip design functions are propagated down to the hardware level In a typical SoC, user software applications communicate with hardware modules (peripherals or processor units) through these multiple layers of abstraction. These applications are typical of real-life software run on the SoC when used in customers' end products. The hardware functionalities invoked are deemed most common and critical. The goal is to verify functional behaviours of the SoC via user-developed software applications down to the functional hardware implementation level.

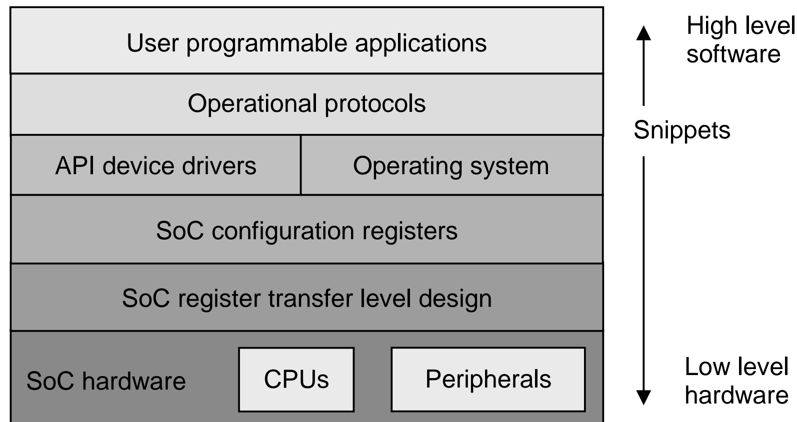


Figure 3.1 SoC software to hardware architectural abstraction layers

Our second observation is that the main hardware elements a user application has access to are usually *configuration registers* of the SoC modules. Software applications interact with peripherals by writing to and reading from their configuration registers. However, these registers are embedded at low hardware levels which user applications are unable to access directly. These registers can only be controlled through the various abstraction layers using the available device drivers and operating system (OS) routines.

To initiate hardware operations, software must bridge the layers of abstraction down to the hardware domain. To do this, software applications interface with protocols running on the OS, and call peripheral application programming interface (API) driver functions to access the required registers. Essentially, any software application test program should contain sequences of these register accesses in order to trigger various hardware operations and verify the executed hardware design functions.

Finally, we observe that software applications can be broken down into smaller sequences of tasks. Examples include tasks that initialise a peripheral or send data using an Ethernet controller. Our test methodology calls for test cases that exercise a range of interactions and functionalities throughout the architectural levels. Therefore, different combinations and sequences of these smaller tasks produces a variety of application test cases. These tasks form the fundamental test building blocks of our test programs, which we denote as *snippets*. Snippet building blocks operate just above the device driver level in Figure 3.1. The concept of snippets and its role in test generation is described in Section 3.4.

3.2.2 The SALVEM pseudo applications strategy

Based on our observations, we devise SALVEM as a verification methodology that couples the software and hardware domains together to verify across multiple architectural abstraction layers. The approach is to analyse how the SoC will be used under various potential applications of the chip, and identify the possible use-case scenarios of the SoC when it is eventually integrated into the customers' end-products. The information from these application scenarios are collated into a set of *pseudo applications*.

Pseudo applications describe the design functionalities and behaviours that are utilised by SoC use cases. They are used to configure the SALVEM system and provide test objectives that must be satisfied in order to verify the SoC. Besides SoC analysis, it is intended that pseudo applications may also be obtained from legacy or regressions testing, either from previous verification phases or other architecturally similar SoC projects.

A pseudo application of an SoC design describes a usage scenario for the SoC. Depending on the use-case scenario, each pseudo application may require different hardware environments to provide the desired interface to the SoC. The operations conducted by a pseudo application may also be broken down into smaller functions, which are captured into test building blocks (i.e. snippets) for software test generation purposes.

Figure 3.2 shows diagrammatically the SoC component layout for the example pseudo applications discussed in the remainder of this section. In Figure 3.2, a simplistic pseudo application is the direct memory access (DMA) transfer of a certain amount and type of data, from one particular source to a particular destination, and at a desired rate of transfer. If the DMA transfer involves external off-chip memory or input/output (I/O) devices, the SoC testbench environment must be configured appropriately for such a setup. The operations carried out for the DMA transfer can also be dissected into smaller and modular functions, such as initialisation of the DMA device (and source or destination device if required), execution and monitoring of the data transfer, and termination of the data transfer.

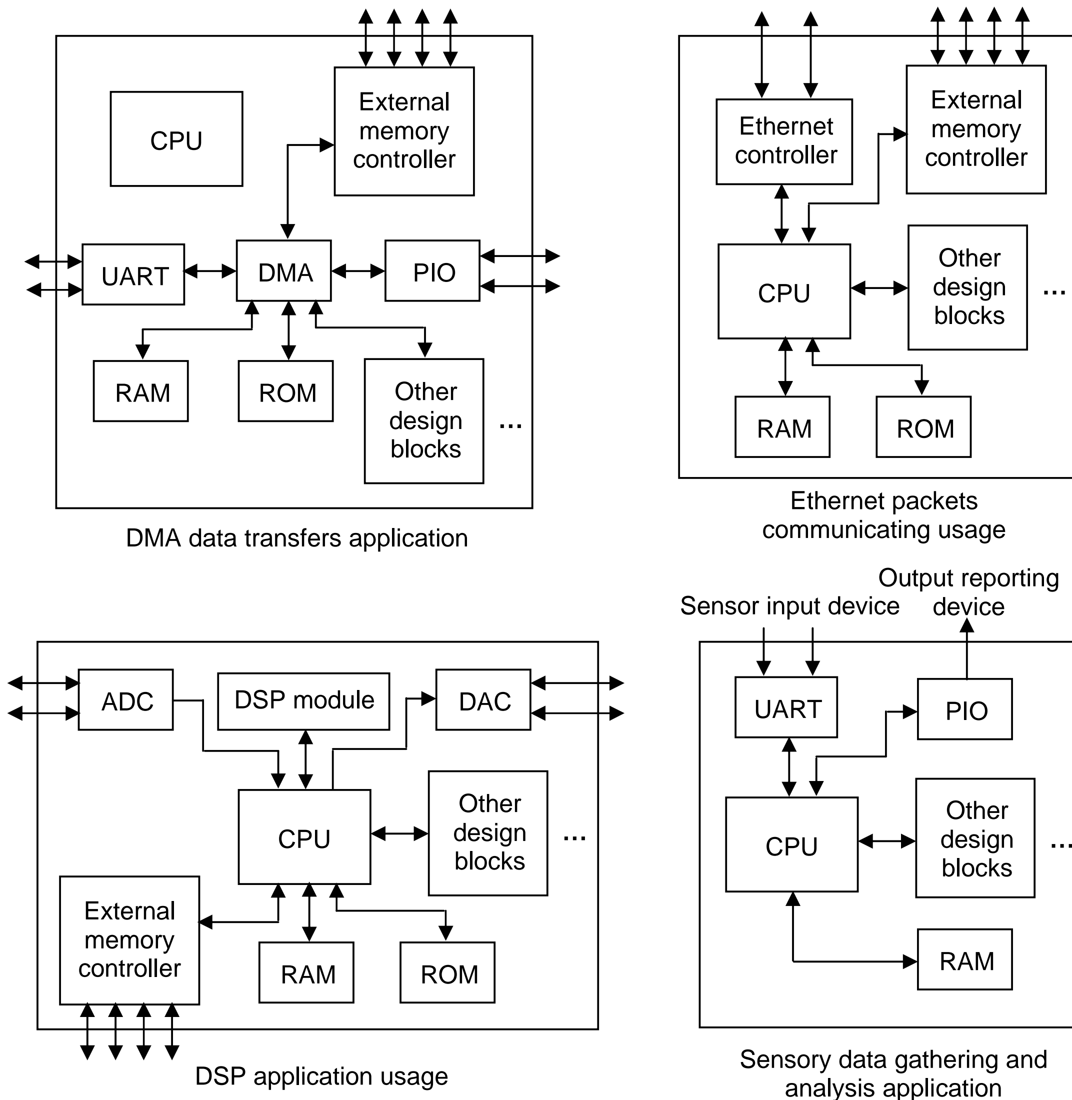


Figure 3.2 SoC application usage examples

Other more involved pseudo applications include communications of data traffic through an Ethernet device under a transmission control protocol and internet protocol (TCP/IP) framework. Once again, testbench infrastructure is needed to simulate sending and receiving of data packets to and from the SoC. The operations needed to perform Ethernet transfers can also be broken down into lower level functions, such as establishing connection with the external environment under some form of handshaking protocol, setting up and running the Ethernet controller device, or preparation of the data packet with correct headers, error checksums, and so forth.

Digital signal processing (DSP) pseudo functions are also common. In this use case, the operations broken down consists of initiating analogue to digital conversion (ADC), manipulation and storage of large signal data sets, performing mathematical or filtering processing, and digital to analogue conversion (DAC) to send processed signal data out again. Other examples of pseudo applications are

infrared sensor applications that communicate sensory data with the SoC via serial universal asynchronous receive/transmit (UART) ports, or security encryption/decryption data operations. The SoC can also be used as a simple controlling unit to manage various devices such as automotive engine control units, avionics, or equipment like mobile phones or photocopiers. In [SS04], a photocopier application scenario previously devised for the Nios SoC is described.

The concept of pseudo applications described in this section form the basis from which SALVEM creates test scenarios to perform application driven verifications. We outline this procedure next.

3.2.3 High level SALVEM flow

In order to verify SoCs with pseudo applications, the SALVEM procedure consists of (1) hardware configuration, and (2) software test generation. The hardware simulation environment must be configured to mimic the real-life customer product operating the SoC as specified by pseudo applications. Software test generation creates the software test programs for simulation in the hardware environment.

Figure 3.3 shows the flow of operations in our SALVEM system. Both hardware configuration and test generation are originally influenced from the same set of pseudo applications. However, hardware configuration and test generation are carried out independently, and do not influence one another so as to create different combinations of test programs and hardware configurations for verifying the SoC.

Executing pseudo application based test cases requires the SoC and simulation testbench to be set up according to test environment configurations as specified by the pseudo applications. The hardware configuration flow is responsible for this and is described fully in the next Section 3.3. Note that the hardware configured test environment can be used multiple times by many different software test programs to test the SoC, before another test environment is to be configured.

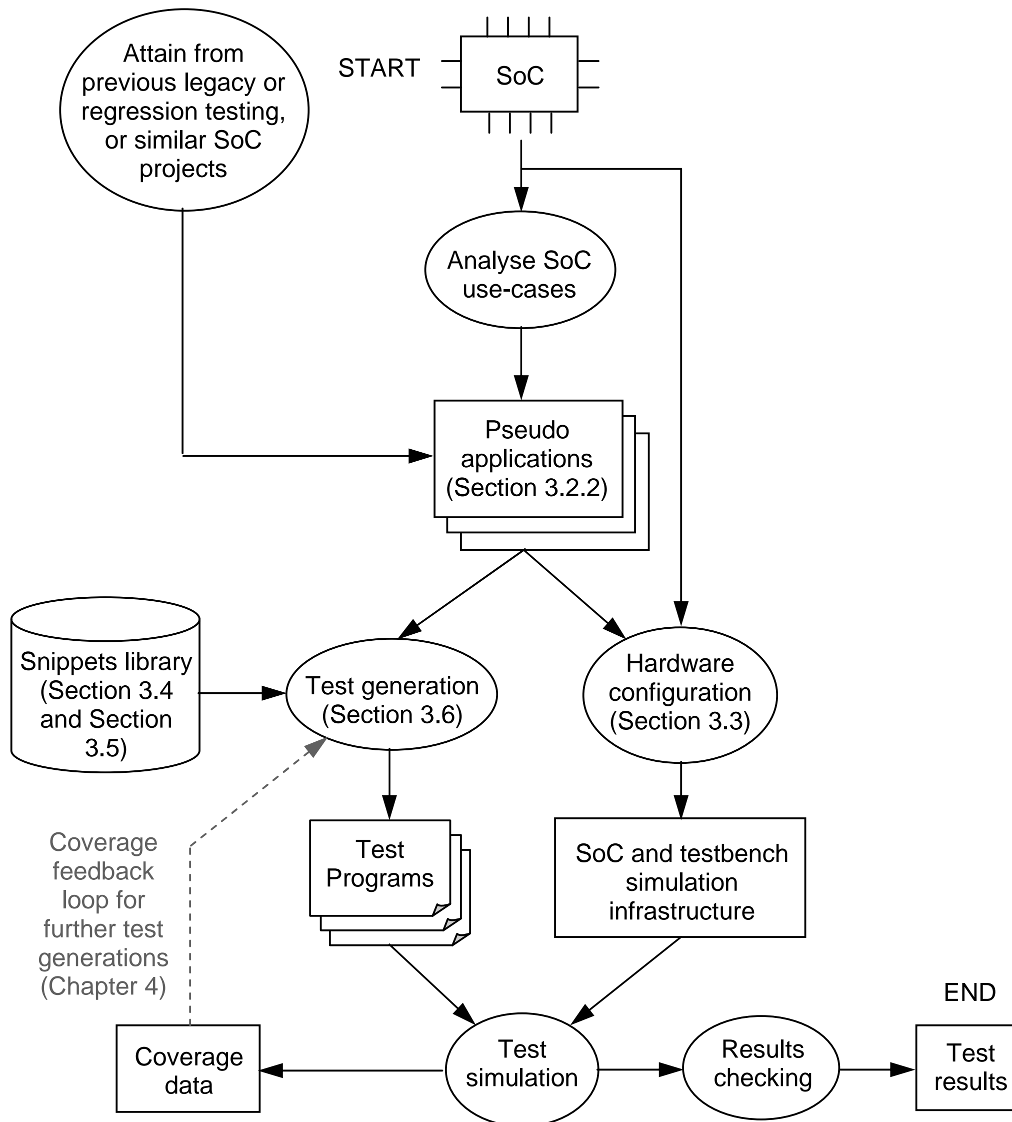


Figure 3.3 SALVEM verification flow

Our software test programs are made up of test building blocks derived from snippets of broken down functions employed by pseudo applications. Snippets are small modular pieces of software code fragments that invoke SoC modules to perform specific tasks and are detailed in Section 3.4. For any SoC, all the snippets that are extracted from pseudo applications are captured in a database we denote as the snippets library. The test creation process selects and composes various combinations of these test building block snippets to satisfy the test objectives of pseudo applications, and steer testing toward critical and desired design behaviours. Diverse test programs are created by intermixing pseudo applications' snippets from the snippets library to exercise a large range of application functions. The test creation process is described in Section 3.6.

Once the hardware configured test environment is set up, and the software test from test generation is loaded onto this test environment, test execution is carried out by the simulator. During test simulation, coverage is measured to estimate the progress and completeness of verification. Coverage information can be analysed to determine if the SoC has been tested to a sufficient level of quality, and can also facilitate a coverage driven verification flow to drive further test generations (i.e. to execute additional tests to exercise previously unverified functions or continue verifying critical and coverage enhancing portions of the SoC further). Our coverage driven mechanism is tightly coupled to software algorithmic test generation and is described in Chapter 4.

During testing, SoC behaviours are also monitored to ensure conformance with the operations initiated by application test programs and the design specifications. Results checking enable test failures to be identified and analysed faster, so errors in the SoC design, test program, or verification environment can be corrected earlier. Results checking is facilitated by both hardware configured units, and from the software snippets test building blocks, and are described in Section 3.4. Automated results checking extracts test information from the hardware configuration and software test program to validate expected results against actual test execution outputs. We begin in-depth explanation of SALVEM next, beginning with the hardware configuration process.

3.3 Hardware configuration

The goal of hardware configuration is to construct and set up an operating environment for the SoC to function within; so that test programs derived from pseudo applications can be executed similar to actual usage scenarios. The operating environment consists of various modules that are combined with the SoC design to form the simulation system under which SoC design functions are simulated and verified.

Figure 3.4 shows the flow carried out to configure the SoC operating environment. Configuration of the SoC environment begins by examining the pseudo applications intended for testing the SoC design. Depending on the types of pseudo applications executed on the SoC, various external modules to support the SoC operations carried out. The modules also mimic the environment of the SoC customers' end-product which the SoC will be embedded within. Examples of these modules include memories, I/O stimulus units, and other specialised hardware blocks that perform specific functions.

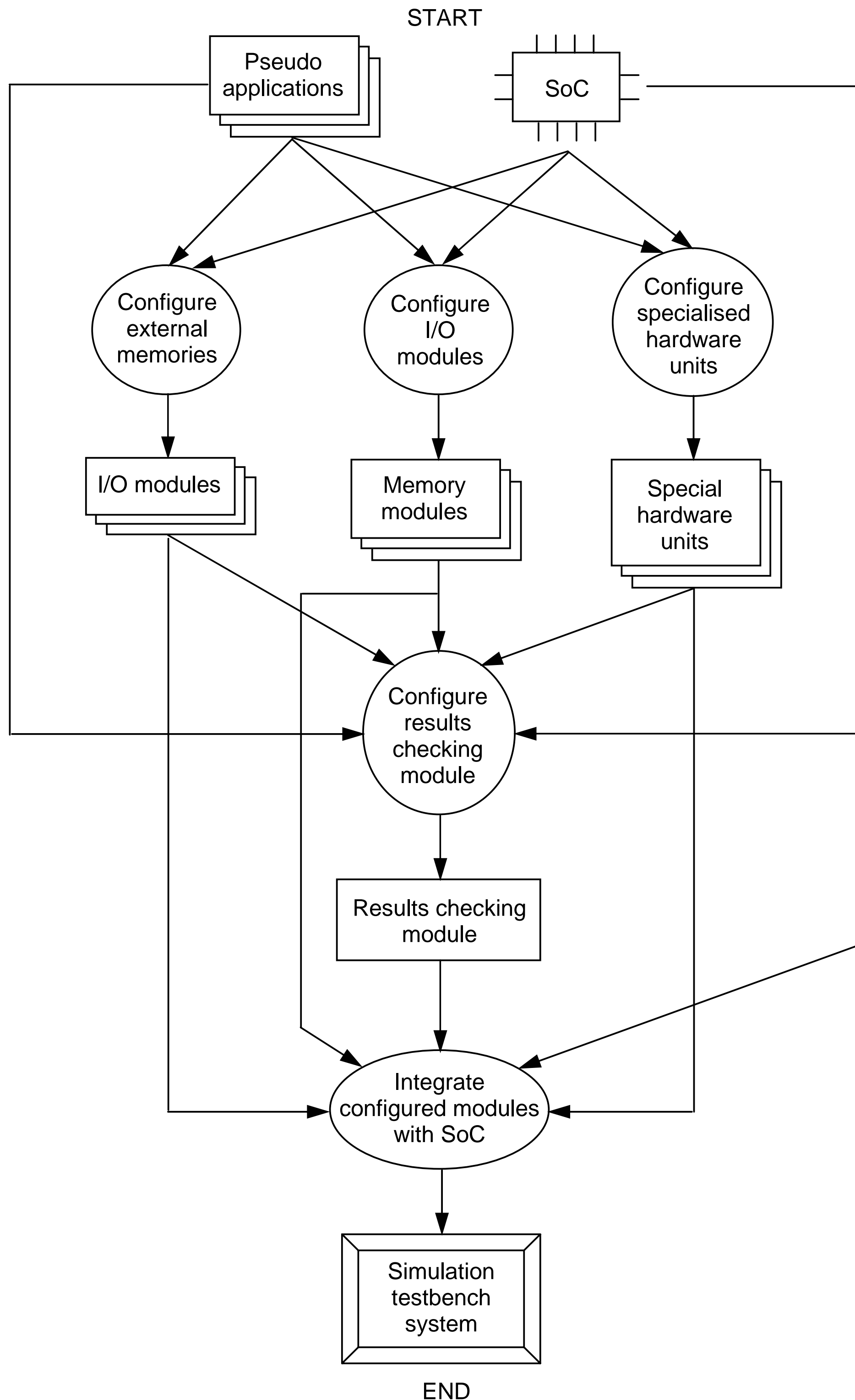


Figure 3.4 Hardware configuration flow

External SoC memories such as static random access memory (SRAM) or flash based memories are often required to supplement on-chip SoC memories. These external memory modules may be used for storing software test programs or holding large sets of temporary data during execution of data

intensive applications (e.g., DSP usages). Besides the number and types of memory modules, other low-level specifications for each memory module must be considered (e.g., the size, internal row/column data banks configuration, read/write modes, etc). The memory interface to the SoC must also be established. The hardware configuration phase is responsible for these tasks.

If the SoC is equipped with I/O ports and test programs execute operations that need to communicate with the external environment, then external I/O modules must be provided. These I/O modules are responsible for handling any output signals or data from the SoC, and also supplying any required input stimulus to the SoC's I/O ports. For example, I/O modules may be created for handling serial or parallel data with UART or parallel I/O (PIO) ports on the SoC. The configuration of I/O modules ensures the correct format, size and rate of communication of such data from test program functions are supplied to the SoC under applicable protocols.

Besides memories and I/O modules, the SoC may be employed for specialised applications that require other forms of hardware modules. An example is the inclusion of memory controllers to interface the SoC with larger off-board hard disk memories. If the SoC is a network centric or security focused design, Ethernet controllers or security encrypt/decrypt units will also be required.

The other requirement during hardware configuration is to provide the necessary results validation functions in the result checking module. Under the direction of test programs, the functions performed by the SoC, external memories, I/O, or other specialised modules are examined to identify outputs that can be captured and compared against expected results. For example, the input stimulus provided by I/O modules can be analysed by the results checking module to extract expected data signals from the SoC based on the application functions to be performed. The expected sequence of data signals is then compared against actual SoC outputs. For memory transfers, the blocks of data written out from the SoC can be checked to ensure it was transferred correctly at destination locations.

When all the required external modules are created and configured, they are integrated with the SoC to form the complete simulation system. An example of the possible hardware configuration of the simulation system is shown in Figure 3.5.

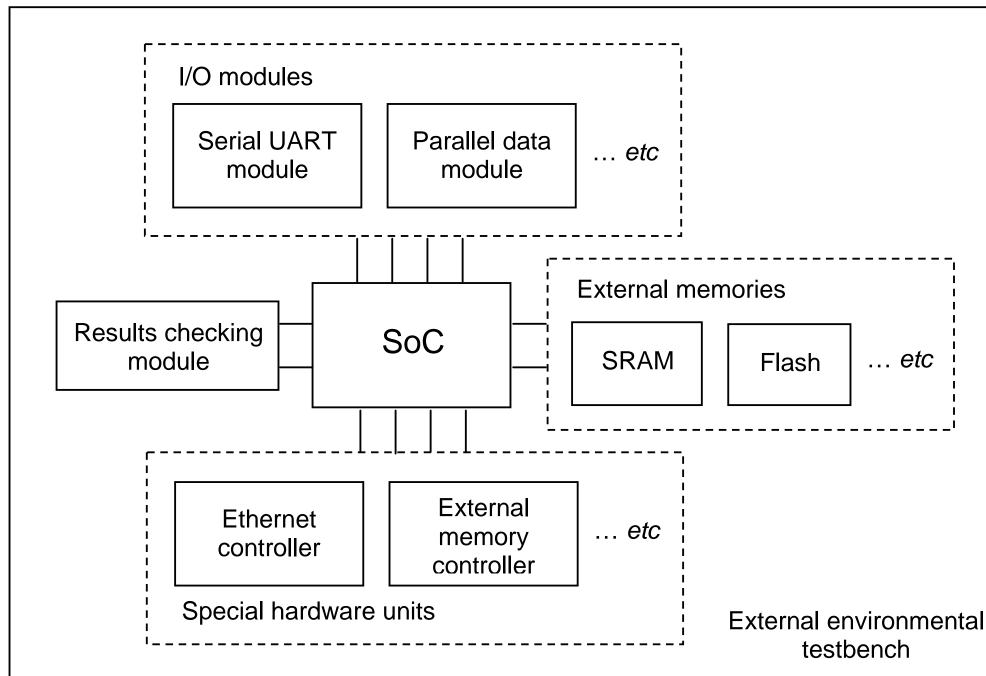


Figure 3.5 Hardware configured simulation system example

Compared to software test generation, hardware configuration is not as complex. Once each external module has been created, they can be reused and configured for additional or different application functions. Also, there is usually a small number of external environmental configurations needed by the SoC to execute different pseudo application functions. Once a particular hardware configuration is put in place, the software test generation can create many different test programs to perform a large variety of application based functions for each hardware setup.

The diversity in test programs arises from the use of different test building blocks to compose the tests. Before describing the test program creation process, these snippet test building blocks are described in the next section.

3.4 Snippets for software test creation

Conceptually, snippets are small, modular and distinct fragments of software that perform specific tasks using the processor and peripherals on the SoC. Snippets are created by examining SoC usage scenarios of pseudo applications, and breaking down these application use-cases into small modular functions. Each function performs specific tasks to achieve the goals of the overall application operation, and is extracted into a snippet.

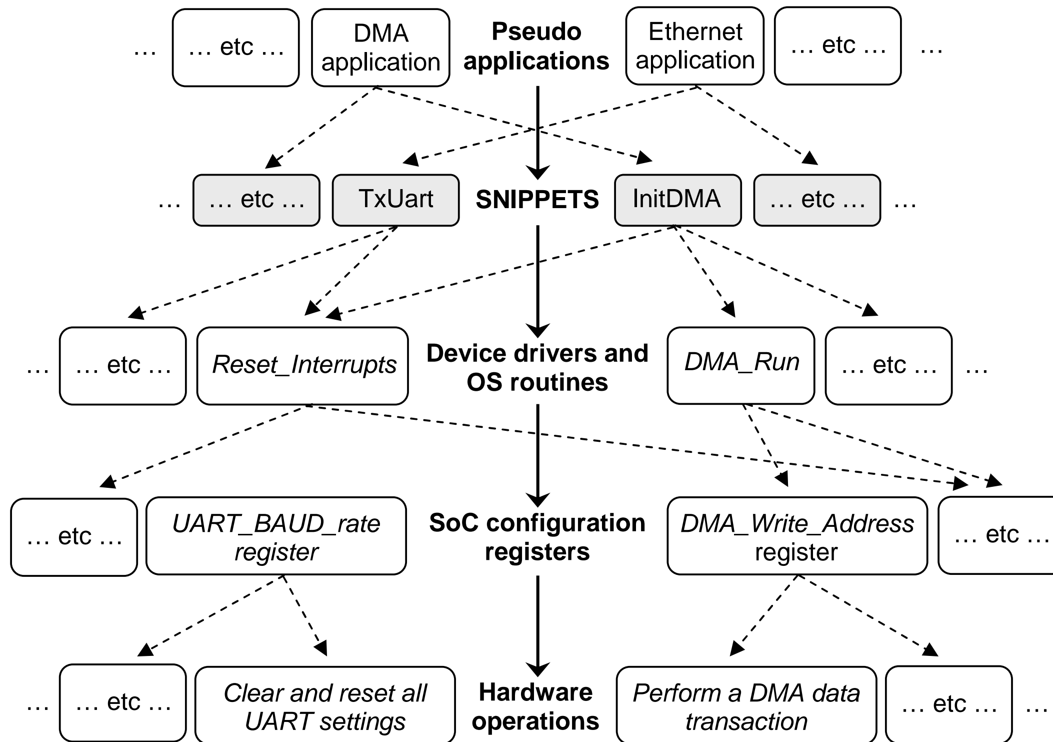
A SALVEM software test program is created by composing various sequences of these snippets together. By reusing and intermixing the functions from pseudo applications, many more test programs can be created to exercise diverse combinations of these application functions and verify design behaviours. If pseudo application programs were used solely, the functions executed would be highly rigid and the range of application driven design behaviours tested would be limited and similar each time. Our test creation strategy enables many different permutations of the original applications to be tested.

3.4.1 Operational characteristics of a snippet

The goal of snippets is to facilitate verification of design behaviours typically exercised by overall applications of the SoC. Snippet initiate hardware functions which can interact with operations from other snippets. Snippets can conduct operations on a single on-chip unit, or on multiple devices throughout the entire system. Each SoC device can be assigned a set of snippets. Such snippets test specific functionalities on their assigned SoC device but may invoke operations on other on-chip peripherals as well. The outcome is extensive individual device testing whilst at the same time, a range of system-wide transactions are verified – invoked by various combinations of snippets chosen within the test program.

In order to execute SoC device specific functions, the multiple layers of abstraction between software to hardware domains from Figure 3.1 was examined and exploited for use by snippets to invoke application based hardware operations. Specifically, we design snippet functions to operate just above the device drivers domain. Figure 3.6 is an extension of the SoC architectural layers diagram from Figure 3.1. Figure 3.6 depicts where snippets are embedded amongst the elements making up the SoC software and hardware abstraction levels. It shows how requests for software application functions are initiated and passed down into the hardware domain to exercise operations on the SoC.

At the top level, complete software application programs are described in terms of pseudo applications. These pseudo applications usually lie above the OS level for task scheduling purposes. In SALVEM, the complexities of an OS can be avoided and task execution can be managed by the test program itself; that is, the functions required by pseudo applications are broken down and invoked directly from the snippets level. These high level functions may include transfer of Ethernet data packets under a TCP/IP protocol, or copying boot-up code between flash and on-chip memories via a DMA device.



Note : The shaded boxes represent snippets, which are the focus of the SALVEM approach. The solid arrow shows the conceptual direction of interactions from high level pseudo applications to low level hardware operations. The dashed arrows connect specific invocations of components between different levels, from pseudo applications, to corresponding snippets, to specific device drivers, and etc.

Figure 3.6 Snippets operating between software and hardware SoC abstraction levels

At the snippets level, each snippet performs a specific function required by pseudo applications. The snippet initiates short and basic hardware operations, and are not restricted to requests by pseudo applications' usages only. Instead, with our software test programs, a variety of combinations of snippets' functions can be executed. As long as the sequences of snippets are composed legally within the test program for SoC execution, a larger range and mixture of the original pseudo applications will be tested.

Examples of snippet functions are reset, configuration, execution, or termination of device operations, e.g., initialising a DMA device, or monitoring and error check sum evaluation of data transfers. Other non hardware related snippet functions are also possible, e.g., expected results validation, assembling data packets according to Ethernet transfers protocols. To perform snippet functions on an SoC, various driver routines of SoC devices are called at the next level.

Below the snippets and device driver levels lies the configuration registers. Recall from Section 3.2 that the concept behind snippets is based on our observation that lower level SoC hardware operations can only be initiated by reading and writing to configuration registers. Our test programs are made up

of snippets which capture proper sequences of register accesses to trigger desired hardware operations. These registers are typically accessible from our snippet functions through device drivers only. Hence, snippets that employ a sequence of one or more calls to the drivers managing the various devices on the SoC must be created specifically.

In particular, our snippets initiate numerous calls to the driver API of the devices whose registers it must control. For example, to conduct data transfers with an UART device, snippets need to initialise the UART device by writing to registers such as the receive or transmit rate and duplex control registers. When conducting serial transfers, the snippet would need to handle data sent and received by writing to and reading from the UART data buffer registers as well.

Snippets, device drivers, and configuration registers couple the software application and hardware domains together to facilitate testing of SoC behaviours by SALVEM. By accessing SoC configuration registers, the SoC hardware design can carry out operations invoked from snippet functions directly. Snippets are required to use device drivers to execute the correct sequence of register accesses, and with correct register data so as to invoke hardware application functions from the software level. The next Section 3.4.2 shows an example sequence of register accesses for DMA device snippets.

Implementation of snippets

Implementation wise, snippets are developed in terms of individual ANSI-C software callable and parameterised functions. The operations invoked by each snippet are interfaced and invoked directly through the snippet's function call headers. In our SALVEM software test program, sequences of snippets' function calls are composed together. Snippets themselves contain low-level device driver API calls, possibly some OS routine equivalent calls for task scheduling purposes, and other ANSI-C code statements to conduct the required SoC hardware operations via configuration registers.

An important element of our snippets approach is the provision of snippet ANSI-C function parameters. These parameters manipulate and vary the SoC tasks invoked by the snippet each time. Depending on their supplied values, parameters control interactions between devices, enable self-checking, or assign priorities to processes it initiates, and influence many other snippet functions. Different sequences of snippets and variation of parameters' values are employed to generate diverse software test programs during the test creation process. As an example, in the next section, we expand upon our snippet ideology and implementation further by describing snippets for a DMA device.

3.4.2 Snippet example – the DMA snippets

In this section, we describe snippets for a DMA device. We use the DMA as an example because a DMA controller calls upon other devices to carry out numerous transfer operations, similar to a processor core but not as complex.

The device which we develop snippets for is the DMA module of the Nios SoC. Appendix A gives an overview of the Nios SoC, which is the target SoC for developing and experimenting with SALVEM and other verification research in this thesis. The Nios SoC DMA is similar to other general purpose DMA devices in other hardware designs, and share equivalent functionalities. Therefore, we are confident that the snippets developed can be easily mapped and applied for other DMA devices. Indeed, one of the goals of snippets and SALVEM is reusability. The SALVEM approach and snippets should be applicable for other SoCs of similar hardware devices and architecture. In Chapter 4, we demonstrate how Nios SoC snippets were reused for verifying another SoC.

DMA snippets design

In order to develop DMA snippets, we consider possible usages of the DMA. The primary role of a DMA is to facilitate reliable, fast and independent transferring of data whilst other SoC operations are being performed concurrently. This frees up data transfer bottlenecks in the system. The DMA alleviates responsibility from the CPU to perform these transfers so the CPU may focus on other tasks. Our DMA snippets are based on common SoC use-cases of how applications use the DMA. For instance, common DMA usages include streaming transfers between memories and I/O devices such as the UART, or transferring initialisation code to instruction executable memory.

Our DMA snippets (i) initialise DMA transfers between different source and destination devices (InitDMA snippet), (ii) execute transfers in blocking or interrupt modes (ExecDMA snippet), (iii) terminate transfers (TermDMA snippet), and (iv) self-checks for successful data transactions (CheckDMA snippet).

The DMA snippets are designed consisting of configuration register accesses. As an example, Figure 3.7 illustrates the internal control flow of a DMA initialisation and execution snippets sequence. Each oval-shaped node corresponds to DMA configuration registers accesses that collectively initialise DMA operations, initiate DMA transfer, and monitor data transactions. The registers accessed are stated within parenthesis in each node. For example, to set up a DMA transfer, the InitDMA snippet has to write to various registers to specify the source and destination of the transfer, amount to transfer, how much to transfer per transaction, and how or when the transfer can be terminated.

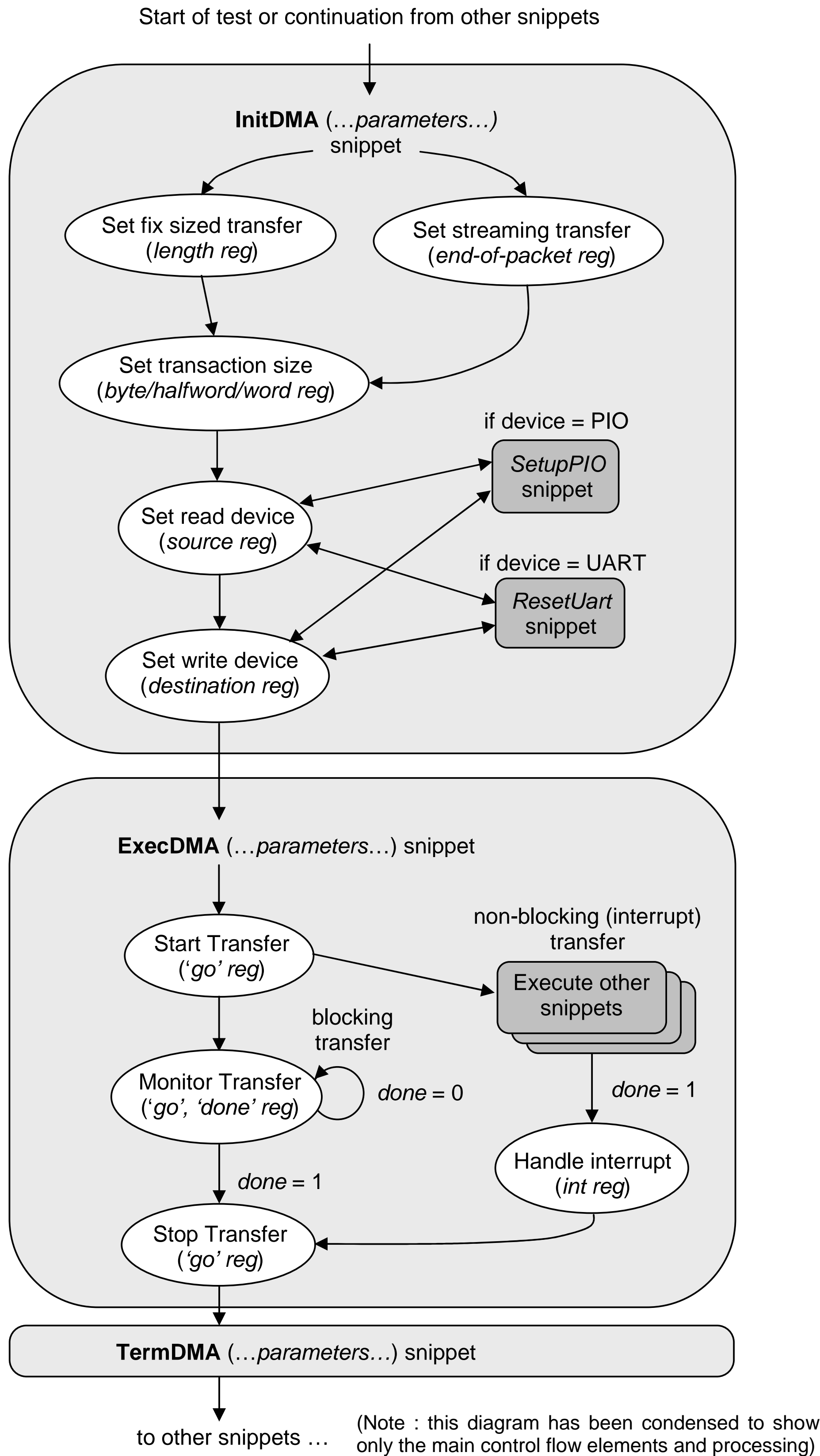


Figure 3.7 InitDMA and ExecDMA snippet control flow

For the ExecDMA snippet, the DMA transfer operation depends on the DMA register configurations conducted by InitDMA. The DMA transfer is initiated by writing to the '*go*' bit of the DMA control register. Data transferring can be conducted in non-blocking mode in the background with other system operations, relying on interrupts to handle data transfer completion. Transfers can also be conducted in blocking mode whereby the DMA operation is continuously monitored. The remaining DMA termination and correctness checking operations are performed in a similar manner using configuration register accesses to control the DMA device and associated operations.

All register accesses are facilitated by device driver calls. The snippets and register access mechanism enables our high level snippets to be reused for other SoCs with similar devices. Only device specific drivers for the target SoC needs to be provided or implemented. For implementation example purposes, we continue to focus on the InitDMA; all other snippets share equivalent implementation features.

InitDMA implementation details

Table 3.1 shows the InitDMA snippet ANSI-C function header, and the subset of most influential parameters that determine how the DMA device can be initialised and used. InitDMA parameters are chosen during the test creation process to initialise the DMA with different read and write addresses, data sizes, set up interrupts, etc. Snippet parameters enable many different operations to be conducted by test programs each time the snippet is chosen for inclusion into the test. The available snippet parameters for the remaining DMA snippets are described in B7 of Appendix C.3.1.

Appendix C.3.1 at B15 shows the implementation of the InitDMA snippet for the Nios SoC in ANSI-C software code. Like other SoCs, ANSI-C is a common language for lower level software and device drivers; and cross-compilers are readily available (as is the case for the Nios SoC). The software code implementation of the snippet is based on the sequence of register accesses in Figure 3.7. Each register access is performed by calling the appropriate DMA's device driver API function. For example, the length, end packet, byte/halfword/word, and source and destination address registers accesses from Figure 3.7 correspond to the driver calls at lines 135, 141, 174, and 180 respectively in Appendix C.3.1 B15. The ANSI-C implementation of the snippet contains other device driver calls and register accesses to perform other set up operations which were not shown in Figure 3.7. Additionally, the snippet also conducts miscellaneous checks of parameter values before they are configured into the hardware registers. These checks are eased if explicit testing of error handling mechanisms is required.

Table 3.1 InitDMA snippet function header and parameters

DMA snippet function header :	
InitDMA(dmaId, rAddr, wAddr, length, uartEOPValue, transSize, intEnable, rAddrCon, wAddrCon)	
Parameters (variable name)	**Parameter type, options and values**
Transfer process ID (dmaId)	32 bit integer value
Source read address (rAddr)	32 bit memory address ranges, peripheral port addresses, memory boundary values
Destination write address (wAddr)	32 bit memory address ranges, peripheral port addresses, memory boundary values
Transfer length (length)	11 bit integer value
End-of-packet (uartEOPValue)	8 bit character value
Transaction size (transSize)	2 bit integer value to choose between byte (8 bit), halfword (16 bit), or word (32 bit) data sizes
Interrupt (intEnable)	Single bit integer boolean value
Source address increment (rAddrCon)	Single bit integer boolean value
Destination address increment (wAddrCon)	Single bit integer boolean value

All snippets are implemented as typical ANSI-C functions. For InitDMA, the parameters of this snippet (from Table 3.1) are provided as parameters into its snippet function. Whenever the snippet is chosen for inclusion into a test program, the snippet function call is simply instantiated, specifying different values for parameters in the instantiation. The DMA snippets functions are provided as part of an API library with other snippets for the SoC. In this way, the underlying snippets functionalities are abstracted away from the test programs itself, but can be easily compiled and linked in with the test and device drivers to form the final test executable binary for simulation.

The example DMA snippets described in this section demonstrates our snippets' concept and mechanisms for test creation. Later in Section 3.5, the full list of snippets developed for creating SALVEM test programs is given. Before this, we discuss how specification of snippets can be formalised in the next section. A well-defined process and guideline for describing snippets enhances their eventual effectiveness and re-usability; and ensures snippets conform to the SALVEM approach.

3.4.3 Formalisation of snippets

Introduction and background

In this section, we introduce and summarise formalisation of snippets. Formalisation is an important research component in SALVEM because it facilitates standardised snippet representations and implementations for all types of snippets. This enables snippets to be employed by different test creation methods as long as the interfaces to these snippet specifications are maintained. Formalisation of snippets provides a well-defined consistent approach for snippet creation, development, and usage in test program generation. With a uniform snippet representation, engineers can identify snippets that can be reapplied for other SoC designs, or create new snippets that integrate seamlessly with existing snippets.

The method we adopt for specifying snippets is based on the design patterns specification approach from software engineering. Design patterns [GHJV95] are software design elements that can be reused repeatedly in objected-oriented software programs to tackle different problems. Similarly, our snippets are reusable blocks of software code that are composed into different test programs to exercise and verify various functionalities on an SoC. From this perspective, snippets can be considered the design patterns needed to create programs for addressing the SoC verification problem.

The strong correlation between software design patterns and snippets was the motivation for adapting design patterns specification methods for our snippets formalisations. Our snippets of hardware test building blocks will be described in an effective and elegant manner equivalent to design pattern software test building blocks.

Our goal is to promote snippet reusability, by describing them as *templates* to further extend new snippet developments and re-usage within test programs; in the same way that design patterns support object-orientated reuse strategies. In fact, the concept of breaking down a problem and reusing design elements to create other solutions with wider range of applicability originated in other domains in architecture and civil designing [AIS+77]; whereby design elements can be generalised from existing solutions and recycled effectively as patterns for other similar problem areas. It is this notion of pattern reuse that we try to exploit with snippets for test generation and verification.

Snippet formalisation summary

Snippet formalisation is an extremely important facet and contribution to SALVEM research in this chapter, but due to the large size of material to cover, we present snippet formalisation details fully as an Appendix C instead. Appendix C describes the complete snippet formalisation guidelines, specification components required to describe a snippet, and examples of actual formalised snippet descriptions such as the DMA snippets. The formalised descriptions of these example snippets also capture the snippets' full implementation and usage information. The remainder of this section summarise the major elements making up the formalised specification of a snippet.

Software engineering design patterns are specified by four main characteristics, the pattern name, problem summary, solution, and consequences from applying the pattern. For snippet formalisations, we also analysed these characteristics to dissect and identify from them, the essential elements that must be specified in order to create a snippet.

The snippet elements that must be specified are grouped into three categories, (A) description, (B) implementation, and (C) discussion. Description information provides an overview of what the snippet does and its goals. Implementation states the necessary details in order to create and apply the snippet to verify the SoC. Discussion examines the effectiveness, consequences, and outcome of applying the snippet, including characteristics that affect how the snippet performs. We outline fully the snippet elements and information that must be given for each of these categories in Appendix C.2.

The next section summaries the entire set of snippets available to verify the SoC, whilst Appendix C.3 describes the formalisations and specifications of subset of these snippets.

3.5 The snippets library

To create comprehensive suite of tests, a library of snippets test building blocks was developed. Sequences of snippets from this library are chosen and composed together to create a SALVEM software application test program. The sequence of snippets communicates and interacts with each other executing a variety of sequential and simultaneous operations.

In addition to DMA snippets from Section 3.4.2, many other types of snippets can be created for exercising an SoC. Table 3.2 shows the other snippets and their descriptions for a typical SoC with a processor, memories, and I/O peripherals. These snippets form the snippets library.

Table 3.2 Snippets library

Snippet	Device	Function
InitDMA	DMA	Configures DMA for transfer
ExecDMA	DMA	Executes and monitors DMA transfer
TermDMA	DMA	Terminates DMA transfer
CheckDMA	DMA	Validates DMA transfer success
ResetUart	UART	Initialises UART device
TxUart	UART	Transmits serial data
RxUart	UART	Receives serial data
RxTxUart	UART	Duplex serial data transfer
RxTxNDupUart	UART	Non-duplex serial data transfer
GenRandNumSeq	CPU	Generates random number sequences on CPU
MatrixMultiply	CPU	Performs fast matrix multiply on CPU
MatrixInverse	CPU	Performs matrix inverse on CPU
Search	CPU	Performs large intensive search algorithms on CPU
Sort	CPU	Performs fast binary, bubble, etc sorts on CPU
Convolve	CPU	Performs DSP signal convolution function
DFT	CPU	Performs DSP discrete Fourier transforms
InvDFT	CPU	Performs DSP inverse discrete Fourier transforms
WriteMemory	Memory	Executes data units writing to on/off chip memories
ReadMemory	Memory	Executes data units reads from on/off chip memories
WriteReadMemory	Memory	Executes writes/reads of on/off chip memories
TestMemoryLogic	Memory	Checks correct operations of memories control logic
WriteROM	Memory	Specialised on-chip ROM testing
MissAlignedAddr	Memory	Invokes various memory error conditions handling
RestartTimer	Timer	Initialises the timer
ReadTimer	Timer	Checks correct timer operation and accuracy
SetTimerPeriodAuto Restart	Timer	Reconfigures timer parameters to restart timer operations
SetTimerPeriodNo Auto-Restart	Timer	Reconfigures timer parameters and delay timer operations
StopTimer	Timer	Terminates timer operations
SetupPIO	PIO	Initialises or clears PIO pins
ConfigPIODir	PIO	Re-configures PIO data directional pins
WritePIO	PIO	Transfers parallel data

Table 3.2 also shows the primary SoC device each snippet is responsible for. Snippets test specific functionalities on their assigned SoC device but invoke operations on other on-chip peripherals as well. For example, the DMA, UART, CPU, and memory snippets can execute sequential or concurrent SoC operations between different external and on-chip devices. This enables extensive individual device testing and verifies a range of system-wide transactions. Other administrative snippets include testbench snippets that control, check and monitor the overall simulation of the software test.

These snippets were originally developed for the Nios SoC in mind. However, they can be reused or modified for other SoC verification without too much difficulty. With this snippet library, the next section describes how a SALVEM software test program is created.

3.6 Creating SALVEM test programs from snippets

In this section, we describe how test programs are created in SALVEM, focusing on the role of snippets. We explain the basic SALVEM test creation processes, which forms the foundation for discussing automated and algorithmic test generation later in this thesis.

The software test creation process is largely dependant on the available snippets to exercise the SoC. The database of snippets that form the snippets library is the most important component of SALVEM test creation. Figure 3.8 shows the concept of SALVEM test creation. To construct a test case, (1) snippets are chosen from the snippets library, and (2) the snippets are chained together into various sequences. The different sequences of snippets composed into a test will provide a diverse range of SoC application functionalities tested by each test.

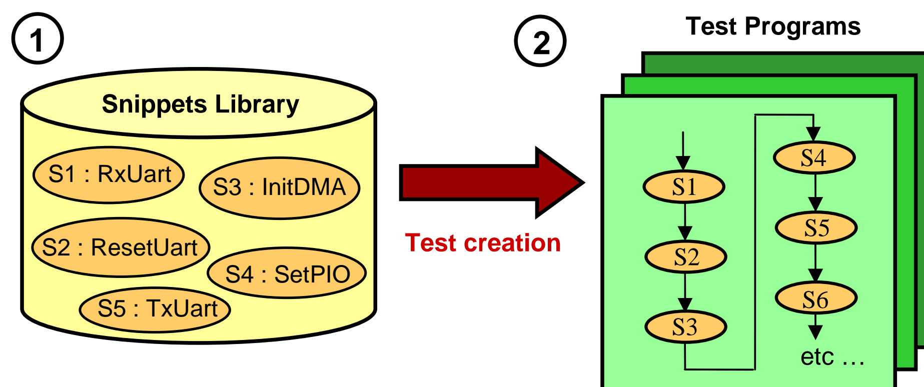


Figure 3.8 Concept of SALVEM test creation from snippets

Figure 3.9 shows a graphical representation of a typical sequence of snippets that can be composed and embedded within a test program. The representation depicts the control flow of the test governed by the snippets. The underlying detail of the test program structure is abstracted away. The sequence is intermixed with snippets that invoke different operations from various on-chip devices. The sequence consists of DMA, UART, CPU, memory, timer, and PIO specific snippets blended together to form the overall test case. These snippets communicate and interact with each other executing a variety of sequential and simultaneous operations. For example, the DMA snippet can operate whilst the PIO snippet or serial UART data transfer is carried out. The operations invoked by snippets will emulate and verify real-life system behaviours of the SoC. The snippet dependencies shown in Figure 3.9 ensure the test program produced is fit for execution on the SoC; dependencies are discussed fully in 3.6.2, whilst the dependency notations 'DEP' are prescribed in Appendix C.2.1 B9 as part of snippet specification formalisations.

Besides the variation available from the choice of snippets and snippet sequences, snippet parameters also contribute strongly to the diversity of SoC functions exercised by SALVEM tests. Table 3.1 showed the parameters for an InitDMA snippet to configure different types of DMA transfers. During test creation, different parameter values will be supplied to each snippet every time they are inserted into a test. This facilitates different variants of SoC tasks performed by the snippet each time. Parameters manipulate the internal device operations a snippet will invoke, whilst the different sequences of snippets chosen into a test control the overall test flow.

To verify an SoC comprehensively, sequences of snippets and snippet parameters are intermixed as much as possible. However, our SALVEM tests must be composed according to various test generator and SoC architectural rules. During the test creation process, the selection of snippets, snippet insertion into existing snippet sequences, and assignment of snippet parameters, are govern by test creation attributes such as constraints, dependencies, and user influences (i.e. user biasing or test creation templates). Test creation attributes are essential and form part of the formalised snippets specification requirements in Appendix C.2. We describe constraints, dependencies, and user influences next.

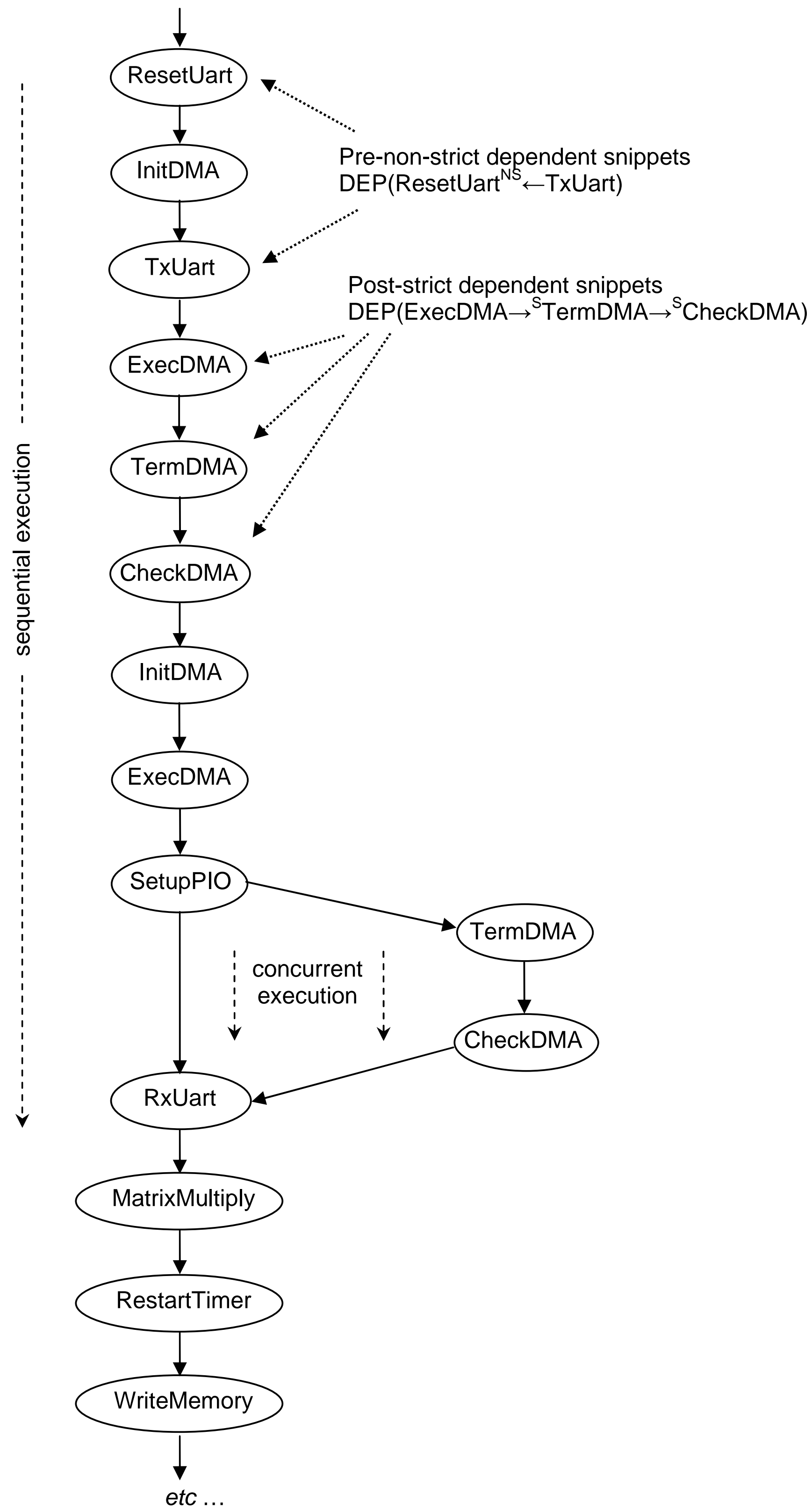


Figure 3.9 Graphical representation of example test program created from snippets

3.6.1 Constraints during test creation

Constraints are required to prevent illegal conflicts or erroneous conditions to be invoked by test programs unintentionally. For example, snippet parameters' choices are constrained to prevent DMA operations between devices when these same devices are already in use and cannot be shared. Any snippet operation that writes to read-only portions of memories must also be restricted from doing so. Besides restrictions, constraints may also impose other requirements. For example, DMA transfers involving 8 bit I/O ports require the transaction unit size for each transfer operation be set to byte sized. If serial or I/O operations require interrupts, the interrupt priorities and routines must be set up suitably.

The test creation process must adhere to these constraints to produce executable tests. For illegal conditions that do not cause critical failures in test executions, these constraints may be lessened to test certain error handling capabilities of the SoC. The formalisation of constraint specifications and full description of constraints for SALVEM verification is provided in Appendix C.2 B8.

3.6.2 Snippet dependencies during test creation

Besides snippet constraints, the selection and insertion of snippets into tests must be conducted under dependency requirements. In general, our test creation process shall attempt to select any arbitrary snippet into the snippet sequences of the test. However, certain snippets carry dependencies that require other *pre* or *post* snippets to be chosen and inserted into the test's snippet sequences beforehand. Dependencies specify allowable combinations of snippets that can be composed to form software tests. We specify dependency rules to ensure only executable snippet sequences are generated.

SALVEM defines two types of dependencies. A strict dependency requires a dependent snippet to be executed immediately before (or after) the target snippet. A non-strict dependency implies the dependent snippet can be generated amongst other snippets before (or after) the target snippet.

For example, let us consider the UART transfer snippets. Before initiating any UART transfers, the ResetUart snippet that sets up the UART must be created and inserted into the test beforehand. This dependency is considered pre-non-strict because the ResetUart dependency snippet does not have to be invoked immediately before the UART transfer snippets. An example of post-strict dependency involves the DMA snippets. When executing blocking DMA transfers, the TermDMA and CheckDMA snippets must follow immediately after the ExecDMA snippet. Examples of these

dependencies were illustrated in Figure 3.9. Dependency rules can apply to any snippet, and are formally specified in Appendix C.2 B9.

A side effect of enforcing constraints and dependencies is that certain patterns of snippets may become more prominent in test programs, especially if automated or algorithmic test generators are employed. These patterns occur because certain parameter value assignments or selection of particular dependency snippets are needed in order to produce legally executable tests. The occurrence of such patterns and their effects are described in Section 4.5.8 of Chapter 4, in relation to our algorithmic test generator.

3.6.3 User influences during test creation

Biasing and user templates similar to those discussed in Section 2.2.1 Chapter 2 are employed by the user to explicitly manipulate the SALVEM test creation process. The test programs produced under the influence of biasing and templates will be directed toward testing user desired functionalities that are considered critical or interesting. Biasing controls the likelihood of various test creation choices. For example, if serial transfers are the focus of testing, then UART snippets will be biased for greater selection likelihood into the test programs. Similarly, in DMA snippets, the source or destination transferring devices will be assigned as the UART more often.

Templates outline the basic structure that a test program should follow. Templates can be facilitated by snippets and their parameters directly to specify certain types of critical hardware operations to be invoked regularly. Test program wide templates that apply to the entire test can also be employed. Such templates define certain snippets to be placed at certain points in the test program, or prevent selection of particular snippets. For example, if test programs are required to focus on CPU processing and functionalities only, then the template would only include CPU snippets throughout the test template.

Unlike constraints or dependencies, biasing and templates do not need to be followed strictly. In SALVEM, they are applied in conjunction with other randomised or algorithmic based test generation methods.

3.6.4 Creating executable test binary from test program for simulation

In order to simulate a test program as depicted in Figure 3.9, the graphical representation of the snippets test sequences must be converted into an executable software format. Our test cases are first transformed into ANSI-C software test programs. Figure 3.10 shows the resultant test program of the snippets sequenced test from Figure 3.9. The software test program is then compiled and linked with SoC device driver API and snippet functions API libraries to produce the object code. We then apply cross-compile conversion tools to transform this intermediate representation into the eventual test executable binary. This Verilog data (.dat file) binary format file is then loaded onto the SoC memory for simulation. Figure 3.11 shows the test creation and build procedural flow.

In Figure 3.10, the sequences of snippets that make up our SALVEM test is embedded within the main function of the overall ANSI-C software code. The main routine contains the same series of snippet function calls to the snippets library API (e.g., lines 13, 17, 21, etc) as the sequences of snippets represented as nodes in Figure 3.9. Each snippet API call contains information regarding the snippet parameters; in particular, what values to assign to the parameters of the snippet function. Outside the main routine, global variables to manage the overall test execution, and support snippets operations are declared (lines 3 to 8). For instance, array buffers to store serial data that is received and transmitted by the UART, snippets operation pass/fail status variables, or interrupt priority settings.

The TestEnd functions (e.g., lines 14, 18, etc) is a wrapper function that check and validate if the operations invoked by snippets were carried correctly, signalling failures to the testbench if needed. The final set of software code statements in the main routine is a loop (line 67) that waits for all operations invoked by snippets earlier to complete, before the entire test program and SoC simulation is terminated by the TestbenchFinish function.

During test program execution, a trace of the snippets executed and their success or failure is recorded in the simulation log file. Any snippet that fails automatically triggers overall test failure. For test failure debugging, the error scenario can be replicated by performing the same sequence of snippet operations up until the snippet that caused SoC malfunction. The sequence of snippets executed before the error snippet is essential for setting up the state of the SoC, so as to provide the necessary conditions for the error-invoking snippet to trigger the design failure again. If error snippet occurs early in the test, this can reduce test simulation time during debugging, as all subsequent snippets after the error snippet can be ignored. As an example, we refer to an actual design error uncovered in the Nios SoC by one of our SALVEM test programs described in Section 3.8 (and detailed further in Appendix D). The example demonstrates the use of the error-invoking snippets, analysis of test simulation log data, and the debugging processes conducted.

```
1         // Example SALVEM test program, manually created from the test case in Figure 3.9
2
3         // Pre snippet data declarations, e.g. UART receive and transmit buffers
4         char uart_rx_buf_1[250];
5         char uart_rx_check_buf_1[250] = {0xab, 0x1e, // other data bytes … etc … };
6         char uart_tx_buf_1[720] = {0xc8, 0xf2, // other data bytes … etc …
7
8         int ok;  // return status from snippet functions
9
10        int main () {
11
12            // ResetUart snippet
13            ok = ResetUart(na_Uart_1, 8, 2, 0);
14            TestEnd(1, ok);
15
16            // InitDMA snippet, between na_Ext_Flash to na_PIO_1
17            ok = InitDMA(na_DMA_1, id, 0x000cece4, (int) &na_PIO_1->np_piodata, 1686, 0x00, 1, 0, 1, 0);
18            TestEnd(2, ok);
19
20            // TxUart snippet
21            ok = TxUart(1, 720, 0, 0x7b, uart_tx_buf_1);
22            TestEnd(3, ok);
23
24            // ExecDMA snippet, blocking execution
25            ok = ExecDMA(na_DMA_1, id, 1, 0, 0, 1);
26            TestEnd(4, ok);
27
28            // TermDMA snippet
29            ok = TermDMA(na_DMA_1, id, 1);
30            TestEnd(5, ok);
31
32            // CheckDMA snippet
33            ok = CheckDMA(na_DMA_1, id);
34            TestEnd(6, ok);
35
36            // InitDMA snippet, between na_on_chip_RAM to na_Ext_Flash
37            ok = InitDMA(na_DMA_1, id, 0x009109f0, 0x0006b25a, 287, 0x00, 1, 0, 1, 1);
38            TestEnd(7, ok);
39
40            // ExecDMA snippet, non-blocking execution
41            ok = ExecDMA(na_DMA_1, id, 0, 0, 0, 1);
42            TestEnd(8, ok);
43            // Note that TermDMA and CheckDMA snippet for non-blocking DMA execution will be called
44            // automatically by the DMA interrupt routine when transfer is completed.
45
46            // SetupPIO snippet
47            ok = SetupPIO(0x78);
48            TestEnd(9, ok);
49
50            // RxUart snippet
51            ok = RxUart(1, 720, 0, 0x7b, uart_tx_buf_1);
52            TestEnd(10, ok);
53
54            // MatrixMultiply snippet
55            ok = MatrixMultiply(matA, matB, matC, 2, 2, 2);
56            TestEnd(11, ok);
57
58            // RestartTimer snippet
59            ok = RestartTimer(na_Timer, 0);
60            TestEnd(12, ok);
61
62            // WriteMemory snippet
63            ok = WriteMemory(na_Ext_Flash, 512, 0,128);
64            TestEnd(13, ok);
65
66            // Wait for all snippets to finish
67            while (SnippetsRunning());
68            TestbenchFinish();
69        }
```

Figure 3.10 ANSI-C code implementation of example test program from Figure 3.9

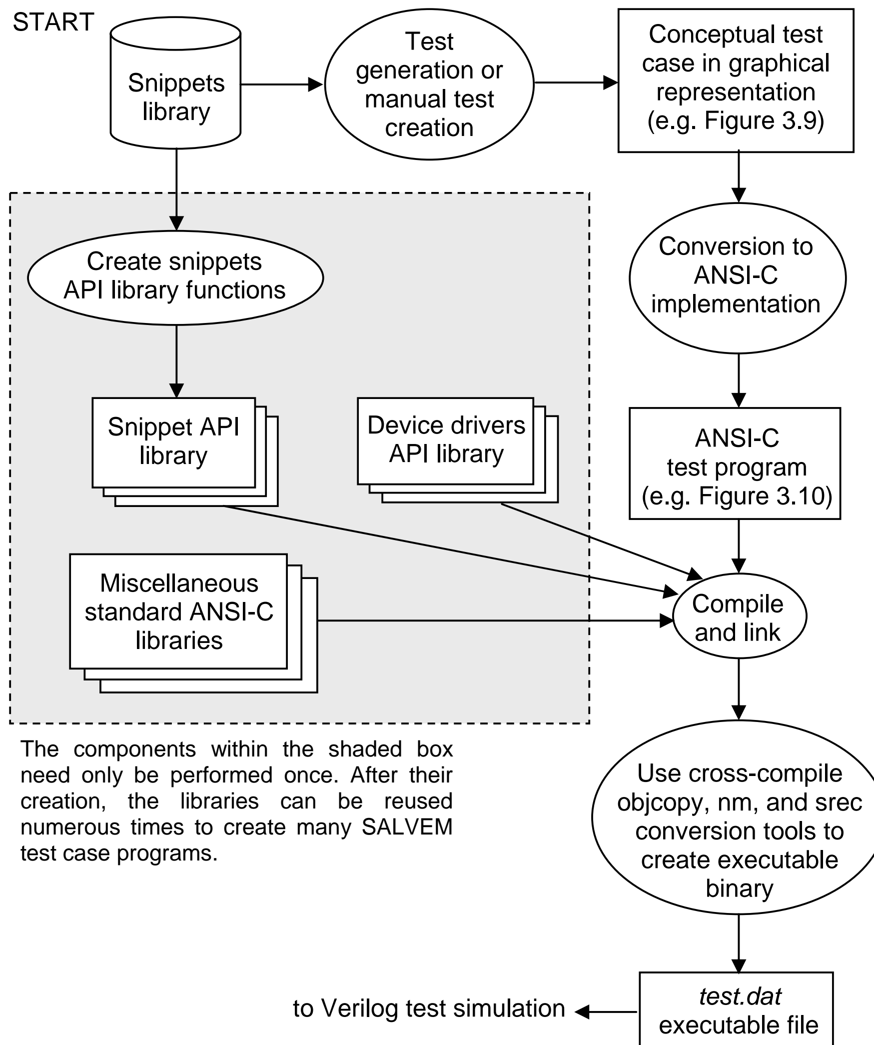


Figure 3.11 SALVEM test creation and build process

Appendix C.1 compares assembler instructions based microprocessor testing against our software application test creation methods, specifically in the context of the snippets approach in this section.

3.6.5 Practical test creation considerations

This section discusses the practical issues that must be taken into consideration when creating SALVEM test programs using snippets. Theses considerations take into account the SoC under verification, the simulation platform, and the test creation strategy. The effectiveness of SALVEM verification depends on how such issues are addressed.

First, the size of a test program and the overall test suite is examined. When creating a test, the number of snippets chosen into the test determines the length of the snippets sequences and in turn, the size of

the test. Long sequences of snippets are considered beneficial for verification purposes because it exercises the SoC for longer periods, with greater possibility for the design to eventually reach complex corner-case state behaviours. Such behaviours would be not exercised frequently otherwise. Intuitively, even though larger test programs are more favourable, there are a number of provisos with this approach. The size of a test is ultimately restricted by the available SoC executable memory that can hold the test program. Also, a larger test program implies longer test simulation times, which becomes an important factor if simulation computing resources are insufficient.

A shorter sequence of snippets and smaller test program on the other hand avoids these issues, but the test may not be as effective for verifying the SoC unless the snippets chained together are chosen carefully using some form of algorithmic strategy. Smaller tests also imply a larger test suite is needed. The number of tests required to cover the equivalent SoC design space is greater, compared to using larger individual tests but smaller test suites which use fewer resources and may be easier to manage.

Therefore, when selecting the size of a test for test program creation, various advantages and disadvantages trade-offs must be considered. The issue of long versus short tests has been debated previously for other verification and test generation platforms [HUZ99]. The test generation strategy employed is important for selecting the most appropriate and optimal test sizes when creating test programs. For the SALVEM snippets based test creation process, the individual test size was not given much consideration by our initial test generator tool in Section 3.8 (and Appendix D). However, in Chapters 4 and 6, our test generators are refined with genetic evolutionary algorithms and multi-objective optimisation that monitors the performance of previously created tests, and evolves effective test programs into optimal sizes.

Second, the quality and completeness of the snippets library determine the effectiveness of our test programs (and SALVEM verification), regardless of the algorithmic or optimisation enhancements to the test generation process. Snippets are the fundamental test building blocks that invoke various functions from within test programs. If snippets are not developed appropriately to invoke adequate application functions to begin with, then the SoC will not be sufficiently covered to expose bugs. Therefore, the snippets library is considered the most critical element of our SALVEM verification strategy.

Third, the capability to produce SALVEM software test programs requires the use of a cross-compiler for the target SoC under verification. The cross-compiler is needed to bring together the SALVEM test program with the device driver and snippets API libraries to form an executable binary that can be executed by the SoC. The cross-compiler tool-chain is usually provided by the SoC's software

development team, otherwise it must be developed as part of SALVEM verification. The tool-chain can also provide debugging facilities that are useful for snippet functions creations and test program development. More importantly, the tool-chain is responsible for performing code optimisations and will determine the final size of the test program execution binary.

Finally, once the snippet sequences are created into a test case, the test can be applied for other types and levels of verification as long as conversion of the test case to a format suitable for use on the target test platform is possible. For example, the current test creation flow from snippet sequences to test binary not only applies for simulations, but for hardware post silicon testing, because post silicon test platforms are able to use these test binary executables as well.

3.7 Experimentation with SALVEM

We implemented the SALVEM verification system and conducted preliminary experimentation to ensure our approach can be applied for SoC hardware testing. The target SoC under verification (used throughout this thesis) is the Nios SoC from Altera [Alt03]. The specifications and details of this SoC are provided in Appendix A. In order to analyse the verification methodology devised in this chapter, our experimental goals are to set up the verification system according to SALVEM, configure the SoC hardware environment, create a software test program, simulate application functions on the SALVEM platform, followed by analysis of the SALVEM verification.

3.7.1 Basic experimental analysis and discussions

The snippets library utilised was that from Section 3.5. Synopsis VCS was the simulator tool chosen. The simulator and verification platform was developed entirely on a Linux RedHat 7 operating system powered by an Intel Pentium 4 3GHz CPU and 2GB RAM. SALVEM hardware configuration for the Nios SoC included UART and PIO data receive and stimulus modules, SRAM and Flash external off-chip memories, and various checking function units (e.g., to validate data transfer between memories and DMA, or I/O signals are transmitted correctly). As more pseudo applications and snippets are devised into the verification system, further hardware configurations can be carried out.

According to the software test creation process in Figure 3.11, a test program was created to execute on the configured SoC. For this preliminary experiment, the test program was manually created. The

sequence of snippets composed for the test program was hand-picked to test that various I/O and CPU operations, or memory transfers were executed throughout the system. The resultant test program created was in fact the example test described in Figure 3.9 and Figure 3.10. The test program contained 13 snippets in total, with an instance of at least one snippet for each of the device on the Nios SoC; and the snippet parameters were chosen randomly adhering to constraints and dependency rules.

We create a script that uses the cross-compile tool-chain to combine this test program with the device drivers, snippets API, and miscellaneous ANSI-C libraries to produce the binary executable file automatically. Compared to test execution time, the time taken to build the software test program is negligible, requiring 5.1 CPU seconds.

Figure 3.12 shows the components that make up the final software executable binary, and their sizes when using the on-chip RAM and ROM memories to hold a test case for simulation. In terms of memory usage, the allocation taken up by various device drivers, and snippets and ANSI-C libraries will be the same for each test, unless a new SoC device or new sets of snippets are added. Therefore, 197Kbytes of memory will always be made available for the test program to hold the sequences of snippets that invoke SoC operations.

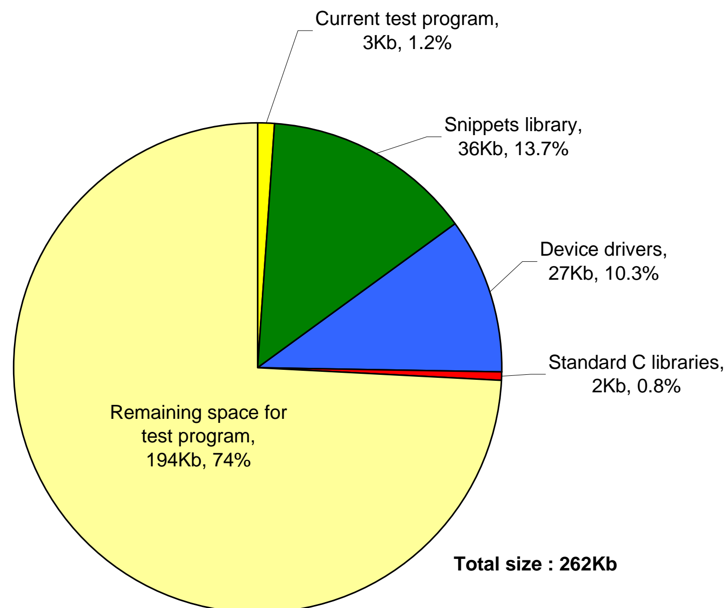


Figure 3.12 Break down of software execution binary

Given that the average size of a snippet is 72 lines of ANSI-C code, which translates to approximately 1,147 bytes, this implies the test program can hold up to 173 snippets if the snippets were embedded inline within the test program. However, in our setup, we use standard ANSI-C function calls to the

snippets instead (depicted in Figure 3.10). Therefore, after taking into consideration miscellaneous wrapper code statements around these function calls, a typical snippet within the main routine of the test program would take up 200 bytes on average, which allows for 990 snippets. This provides significantly more snippets deployment compared to 173, and is more than sufficient to compose various permutations of long snippet sequences into the test. If required, further additional test program space can be reallocated to accommodate new snippets into the library but still allow for adequate snippets into our tests, which is another important benefit.

Under the current memory allocations in Figure 3.12, we executed our test program on the Nios SoC. The simulation took 472 CPU seconds which was 8 minutes of actual real-time duration. Overall, 65.9% line and 63.8% conditional coverage was attained on the SoC. Compared to an actual application program provided by the developers of the Nios SoC which we also simulated, 26,614 CPU seconds and 7.5 hours real-time was required, and 64.8% line and only 49.1% conditional coverage was achieved.

Table 3.3 shows the coverage results from our SALVEM test program and the supplied application program, broken down for each SoC device. Line and conditional coverage were chosen for this preliminary experiment because they are the easiest and hardest metrics respectively to attain full coverage for. Using these conventional coverage measures, an objective assessment of how effective the SALVEM test program covered the SoC design can be attained.

Table 3.3 Coverage results from preliminary SALVEM experimentation

SALVEM test coverage %	**Overall SoC**	**DMA**	**UART**	**CPU**	**Memories**	**Timer**	**PIO**	**Other devices**
Line	65.9	99.2	95.0	65.3	24.1	79.2	75.0	83.2
Conditional	63.8	86.4	71.7	63.7	0	28.9	16.7	79.5
Application test coverage %	**Overall SoC**	**DMA**	**UART**	**CPU**	**Memories**	**Timer**	**PIO**	**Other devices**
Line	64.8	88.6	90.5	64.8	23.9	79.2	75.7	81.7
Conditional	49.1	34.9	53.8	49.4	0	25	16.7	57.8

The DMA line coverage easily attained high percentages using the DMA snippets from our test. This is not unexpected given that any simple DMA transfer operation would exercise many lines of the DMA design code. However, for conditional coverage, many more diverse types of transfers are needed to cover the extensive combinations of branch paths throughout the design code. Similarly, for other devices such as the UART, CPU, timer, and PIO, many more SoC application scenarios must be

executed by snippets to gain higher conditional coverage; despite some of these devices easily attaining favourable line coverage already. In any case, using just a single test composed of some typical snippets, we have successfully invoked a number of application based functions to cover more than half the SoC.

3.7.2 Bug insertion and detection by SALVEM – a case study

In order to demonstrate SALVEM's capability for bug detection, we inserted a number of intentional design errors into the SoC to check if they can be uncovered by our snippet functions. The types of errors include missing or mismatching signal to port connections, missing or incorrect module instantiations, erroneous variable assignments, or errors in the logical, conditional or arithmetic design code of the SoC design.

For example, the following three errors were inserted into the Nios SoC design.

1. Missing port connection for the DMA interrupt signal when instantiating the DMA device, e.g., port connection accidentally commented out or removed.

   Verilog design file: soc.v

```
6478        DMA_1 the_DMA_1
6479          (
6480           .clk                (clk),
6481           .dma_ctl_address      (DMA_1_control_port_slave_address),
6482           .dma_ctl_chipselect   (DMA_1_control_port_slave_chipselect),
6483      //     .dma_ctl_irq          (DMA_1_control_port_slave_irq),
6484           .dma_ctl_readdata     (DMA_1_control_port_slave_readdata),
6485           .dma_ctl_readyfordata (DMA_1_control_port_slave_readyfordata),
6486           .dma_ctl_write_n      (DMA_1_control_port_slave_write_n),
6487           .dma_ctl_writedata    (DMA_1_control_port_slave_writedata),
6488           .read_address         (DMA_1_read_master_address),
6489           .read_chipselect      (DMA_1_read_master_chipselect),
6490           .read_endofpacket     (DMA_1_read_master_endofpacket),
6491           .read_flush           (DMA_1_read_master_flush),
6492           .read_read_n          (DMA_1_read_master_read_n),
6493           .read_readdata        (DMA_1_read_master_readdata),
6494           .read_readdatavalid   (DMA_1_read_master_readdatavalid),
6495           .read_waitrequest     (DMA_1_read_master_waitrequest),
6496           .reset_n              (DMA_1_control_port_slave_reset_n),
6497           .write_address        (DMA_1_write_master_address),
6498           .write_byteenable     (DMA_1_write_master_byteenable),
6499           .write_chipselect     (DMA_1_write_master_chipselect),
6500           .write_endofpacket    (DMA_1_write_master_endofpacket),
6501           .write_waitrequest    (DMA_1_write_master_waitrequest),
6502           .write_write_n        (DMA_1_write_master_write_n),
6503           .write_writedata      (DMA_1_write_master_writedata)
6504          );
```

Error insertion : commenting out port connection

2. Missing instantiation of the UART receive handling module, e.g., the instantiation is accidentally commented out or removed.

   Verilog design file: Uart_1.v

```
1107    // Uart_1_rx the_Uart_1_rx
1108    //   (
1109    //     .baud_divisor     (baud_divisor),
1110    //     .begintransfer    (begintransfer),
1111    //     .break_detect     (break_detect),
1112    //     .clk              (clk),
1113    //     .clk_en           (clk_en),
1114    //     .framing_error    (framing_error),
1115    //     .parity_error     (parity_error),
1116    //     .reset_n          (reset_n),
1117    //     .rx_char_ready    (rx_char_ready),
1118    //     .rx_data          (rx_data),
1119    //     .rx_overrun       (rx_overrun),
1120    //     .rx_rd_strobe     (rx_rd_strobe),
1121    //     .rxd              (rxd),
1122    //     .status_wr_strobe (status_wr_strobe)
1123    //   );
```

Error insertion : commenting out UART receive module instantiation

3. Incorrect update of the number of data units transferred by the DMA during each transaction, e.g., the number of data units to send is incremented rather than decremented.

   Verilog design file: DMA_1.v

```
726    assign p1_length_eq_0 = inc_read && (!length_eq_0) && ((length  + {1'b0,
727      1'b0,
728      word,
729      hw,
730      byte}) == 0);
```

Error insertion : incrementation rather than decrement

These three design errors were all detected by various snippets in our test programs. We ran a number of individually hand-crafted snippets test programs and at least one of the above errors caused a snippet to flag test failure each time. For example, the first error will cause DMA termination snippets to fail after the designated DMA transfers has completed. Because the interrupt signal is not connected and cannot be asserted, when the transfer ends, the interrupt routine is not invoked to perform clean up operations and the CheckDMA snippet is not called. This failure and subsequent usages of the DMA for transfer operations by other snippets will cause further flow-on errors to be triggered.

For the second error, UART receive snippets fail each time because the data received is not handled by the UART receive module – which is essentially missing from the perspective of the SoC design and software execution because the module was not instantiated into the SoC at all. The third error results in DMA transfers that can proceed indefinitely or causes the CheckDMA snippet to fail, as the amount of data transferred will be incorrect.

Whilst these errors are intentional and in no way represent the entire range of design errors possible, they demonstrate that SALVEM can use snippets to detect and isolate sources of design bugs with application based test functions. Indeed, during one of the SALVEM verification runs using our

automated test generator, an actual design bug was discovered; even though the Nios SoC had been released for customer use by Altera, and presumed to be error free. We describe this real-life design bug in Section 3.8 (and Appendix D.4).

3.7.3 Other experimental analysis and observations

Test execution examination

Our experimentation in this chapter was also valuable for evaluating certain test application preferences and trialling various modifications, in order to enhance SALVEM verification for further experimentation and verification research in the thesis (such as the test generation and coverage research). First, the flow of the test compilation build process was examined. This involved inspection of the test program object code to ensure all device driver routines, snippet functions and data variables were linked and accessible by our test program at correctly specified memory locations. Additionally, transfer of SoC operational control to the main function routine of our test program must be guaranteed. Otherwise, after the SoC simulation completes the boot-up sequence, our test program will not be executed and the simulation will hang.

The memory and relative address at which the main function routine is located is crucial for commencement of SALVEM testing. This location is determined by the linker and the linker file provided by the user. The linker file is a configuration file that outlines where different segments of code from our entire test case binary should be placed. This includes initialisation boot-up code, our SALVEM test program code, library functions, and various data variables such as static, constants or temporary storage elements. The assignment of these various code segments across different memories can impact the efficiency of test execution and the types of SoC operations conducted. We experimented thoroughly using different code segment settings to create an optimal linker configuration file, which we use for the remainder of experiments in this thesis to ensure efficient test executions.

Test program memory management

During preliminary experiments, we were able to experiment with different configurations of code segment placements across different memory devices. Our target SoC consists of on-chip RAM and ROM, and external SRAM and Flash memories. Assigning our software code segments to different

memory devices provides different advantages and disadvantages. Using the RAM and ROM, code execution and test simulation of SoC functions is faster as the instruction fetch delay is minimised. However, being integrated on-chip, the size of RAM and ROM memories are small restricting the number of snippets and permitting only smaller test programs to be executed.

Using the SRAM and Flash memories on the other hand, a much larger test program and many more SoC application based functions can be simulated. The downside is that simulation is much slower. Instruction fetches from these memories suffer from significant control and data bus latencies. Additionally, the code execution delay is exacerbated because the SRAM splits up its data into separate rows amongst individual SRAM modules, and the Flash memory offers low data bandwidth from its 8-bit I/O ports.

From our analysis, it was decided that running software test programs on-chip from RAM and ROM would be the best viable option. Whilst the on-chip memory would not accommodate as many snippets to execute application functions, the bottleneck with off-chip test simulation is a greater drawback. Our experiments show that using off-chip memory causes a slow-down of up to 5 times in test simulations. In any case, when we ran on-chip test simulations for this chapter, Figure 3.12 showed there would still be adequate memory space to load sufficient snippets to exercise complex application scenarios.

As long as we can execute large numbers of tests at appropriate speeds, the SoC would still be covered adequately under SALVEM verification. If much larger tests and more test executable memory are needed, a system could be established whereby the main test program code is stored onto larger external memories, but only certain portions of the code is transferred to on-chip memory for execution when needed. For example, a caching technique using the RAM as a buffer could be adopted. In this case, a large test is placed in Flash or SRAM, and smaller sub-portions of the test are repeatedly fetched to the RAM for faster execution as needed; whilst the previous sub-portion of the test is executing.

In summary, we configured our linker file further to place executable software code on the ROM, with sub portions of the RAM allocated for data variable storage during test simulation. The ROM was used for the main software test program and interrupt routines to prevent accidental overwriting of execution code. This SoC memory configuration is used for all experiments of SALVEM verification research in the remainder of this thesis.

Test program control flow management

Another observation from our preliminary experiments was the need for some form of task management during execution of our test programs. The snippets in our test program are composed according to constraint and dependency restrictions to ensure an executable test program is produced. Despite this, the range of application functions that can be performed by snippets and the dynamically changing conditions on the SoC during runtime mean that these statically enforced restrictions may not be adequate. For example, resource sharing conflicts can still occur.

In order to properly manage the flow of operations performed by snippets and prevent conflicts, a simple OS that can manage SoC resources with these snippet operations would be ideal. However, our aim in SALVEM is to maintain a lightweight software package on the SoC under verification, and only load software necessary for executing application based functions from snippets. In the absence of an OS, we implemented a simplistic task scheduling system to manage execution of test program snippets and SoC resources. An important part of this system is to make use of various mutually exclusive (mutex) variables assigned for each SoC device to prevent conflicting or duplicate usages. These mutex variables are declared in the header sections of each snippet function.

3.8 Randomisation for creation of SALVEM tests – a summary

In previous experimental sections, whilst creating a SALVEM test program manually is not difficult, having to produce extensive combination of snippet sequences for many test programs by hand is challenging and tedious. Therefore, we extend the work in this chapter and enhance SALVEM with automatic test generation methods using randomisation. The remainder of this section summarises the SALVEM randomisation test generation case study, whose full details are described in Appendix D.

For automated test generation, we employ randomisation to inject diversification into the types of tests created and the range of SoC application functions tested. Besides the ability to generate many tests quickly, randomised test generation can create many variations of tests that are unexpected. Such test cases are especially useful because they exercise complex sequences of design functions and represent SoC scenarios that are unlikely to have been devised by engineers manually.

To facilitate automatic SALVEM test generation, randomisation is applied to the types of snippets chosen into the test program to form many different snippet sequences, and the parameters chosen for each snippet are also randomised. The randomisation strategy facilitates different permutations of long

snippets sequences in tests, so as to invoke the SoC into complex and difficult to reach operating states. Under certain randomisation conditions, test generation restrictions, and user bias influences, we developed a randomising test generator tool to automate SALVEM test creation; and produce many effective test cases for SoC verification. The approach enables large test suites to be applied for SoC testing as is the case in many industrial design verification projects.

The randomised test generation method was employed to create test suites for verification of the Nios SoC. During randomised testing of the Nios SoC, a number of our randomised tests failed. Upon closer analysis, we discovered a design error in the SoC. The design error occurred in the UART module, and is an incorrect configuration register declaration and incorrect register assignment of its control values. This error caused incorrect and incomplete data transfers using the UART via interrupt mode. Despite being released for customer usage in the engineering community by Altera Inc., our SALVEM method was still able to detect this critical error and a number of other issues in the Nios SoC. This demonstrates SALVEM's capabilities for uncovering real-life design bugs. Appendix D.4 describes fully the design error uncovered by our SALVEM work as part of this case study.

Besides uncovering actual design bugs, our coverage results also proved SALVEM is feasible and effective for system verification of SoCs. The major drawback with employing randomisation is that it can be loosely defined and applied ad-hoc within the test generation process. Therefore, subsequent set of tests created can be mis-directed away from verifying certain important corner cases.

To improve coverage results further and enhance randomised SALVEM test generation, user directed biasing techniques was used to focus on specific SoC devices and corner cases. The outcome of user directed feedback and biasing is a manually implemented coverage driven feedback verification procedure for SALVEM. As detailed in Appendix D.5, our manual coverage driven procedures is able to progressively improve coverage and enhance SALVEM verifications; paving the way forward for further investigations into algorithmic coverage guided testing research in the next chapter.

3.9 Conclusions

The key to our verification methodology is to identify and analyse application based software and SoC usages, and extract modular application based hardware operations that collectively invoke SoC functionalities to carry out these application use cases. Based on these modular hardware operations, a library of software (and hardware) test building blocks is created. The library of test building blocks is used to create many different compositions of test programs and hardware testbench configurations to verify the SoC.

Applying test programs composed of many different snippets of application based test building blocks ensures the eventual real-life SoC design functionalities are tested more thoroughly. The creation of test cases from such building blocks is highly beneficial because test programs of appropriate types and sizes can be generated for simulation based verifications at the pre-silicon level. The range of test programs composed from various snippets of test building blocks allows for large range of SoC design functions to be verified; compared to a static application program that tests similar behaviours each time.

Our preliminary experiments showed that the combinations of snippet sequences chosen in a test program play a major role in the different types of application functions tested. Furthermore, the useability and benefits of a manually created example SALVEM test program was demonstrated by its capability test an SoC design with greater coverage than a typical application software program.

We extended SALVEM test creation to employ randomisation for automatic test generations, and to facilitate a manually coverage driven feedback procedure. This work was conducted as a case study and is fully detailed in Appendix D. Besides further proving feasibility of SALVEM and providing favourable coverage results, the highlight of the case study was SALVEM’s ability to uncover actual design errors in the Nios SoC. Our investigations into manual coverage directed testing also enabled further research into creating automated coverage guided tests via algorithmic genetic evolutionary methods in the next chapter.

In fact, this chapter provides the foundations for further research chapters of this thesis, which describe algorithmic and multi-objective generation of tests, and coverage measuring based on the SALVEM approach.

CHAPTER 4. GENETIC EVOLUTIONARY TEST GENERATION

This chapter presents a test generation methodology using genetic evolutionary algorithms in the software application environment. The goal of the genetic evolutionary algorithm is to compose SoC snippet sequences in a directed manner, rather than randomisation. The genetic evolutionary test generation is posed as an optimisation process whose objective is to maximise coverage subject to verification constraints.

4.1 Introduction

Our motivation in proposing genetic evolutionary algorithm (GEA) for SALVEM test generation is to overcome deficiencies with randomisation methods. Even though randomised test generators can generate verification test suites quickly, the range of randomised tests often does not account for all the essential functional system-on-chip (SoC) behaviours that must be tested. Instead, manually derived test cases are required to complement randomised testing to cover missing test scenarios. In many ways, the need for manual intervention negates the goal of randomisation, which is to automate the test creation process. Furthermore, the test coverage attained by random snippets tests may not be efficient. The same level of coverage could be achieved using shorter sequences of snippets that are composed faster and earlier during the test generation process.

We use GEA methods to tackle these shortcomings. Whilst not all scenarios are guaranteed to be covered, a greater number of corner cases are tackled and less manual intervention will be required. This is because the composition of snippet sequences will not only be random but will be guided by supplemental information from prior GEA test creations.

Another motivation for GEA in SALVEM is to facilitate coverage driven verification. Coverage is a primary measure of test quality in many hardware verification strategies. To realise efficient creation of tests that continually target critical, missing, and desired SoC functional test regions, test generation should be guided by some form of coverage related information from previous test runs. Careful selection and exploration of the design test space to verify is essential in design verification. Our approach to realise coverage driven verification is to employ a guided test generation process such as GEA.

Genetic algorithms and evolutionary methods have been proposed as solutions to generate tests. A survey of GEA test methods was presented in Chapter 2. Briefly, Samarah et al. [SHTK06] devised a scheme whereby SystemC [OSCI] tests are transformed into a cell structure and optimised using genetic algorithms. Corno et al. [CSRS04b] also adopted a GEA approach in assembler instruction test generators to verify microprocessor designs at the machine code level. However, the use of GEA in highly complex SoC design verification and at the software application test level, to the best of our knowledge, has not been investigated.

We use genetic algorithms and evolutionary techniques for automated test generation because it allows for coverage directed verification. GEA is suitable to create effective test suites because (1) it also employs randomisation for exploring large extensive span of the test space, and (2) it continually incorporates feedback information from previously generated test cases that were successful at stimulating important functional test space or uncovering design bugs.

The remainder of this chapter is as follows. Section 4.2 outlines our concept of the genetic evolutionary test creation process for SALVEM; from which a genetic evolutionary test generation process and tool was specially devised. The test generation process and tool is described in Sections 4.3 to 4.10. Experimental results and analysis of our test creation technique against other methods follows in Sections 4.11 and 4.12. Two SoCs are employed for genetic evolutionary test generation, the Nios SoC (Appendix A) and a digital signal processing (DSP) SoC case study. Section 4.13 concludes the chapter.

4.2 Concept of SALVEM genetic evolutionary test generation

Genetic and evolutionary algorithms [Mic96] are based on survival of the fittest principles in biological evolution systems. Figure 4.1 shows the conceptual flow of a test generation process incorporating GEA. The application of GEA for SALVEM test generation is tightly coupled to high level procedural flows of genetic algorithms and evolutionary strategies.

In Figure 4.1, the GEA test generation process begins by creating an initial population of tests. In phase (A), the test population representation for the GEA process is designed once. After it is established, the same representation is employed throughout the test generation. Next, variation in phase (B) is applied to create further new tests into the population by modifying and intermixing characteristics of existing tests in the existing population. The effectiveness of tests are then evaluated and quantified during the fitness evaluation stage in phase (C). Based on the results of test evaluation,

in phase (D) the low performing tests that could not achieve high coverage are discarded. Only tests that attained highest SoC coverage (deemed as high fitness tests) are retained in the population. The final phase (E) checks if the GEA process should terminate, by assessing whether the test population has acquired satisfactory coverage. Otherwise, the evolution cycle repeats using the recently varied and fitness filtered test population.

The repetitive cycle of test variation, fitness evaluation, selection of new population, and termination checking, preserves existing and prior tests that achieve high coverage, and at the same time, creates new tests that further enhance additional SoC coverage based on previous generations of tests. Our GEA test creation strategy allows for previous coverage and test information to be captured and applied for future test generations. This brings about the derivation of a set of tests that is optimal with regards to higher coverage of comprehensive SoC test space.

Appendix E.1 describes the generic pseudo code of a GEA method such as that in Figure 4.1. The GEA characteristics and phases of Figure 4.1 are described in greater detail in Sections 4.4 to 4.8. The design and adaptations of these phases for our test generation procedure (and any GEA process in general) affects the effectiveness and efficiency of SALVEM verification and coverage attainment significantly.

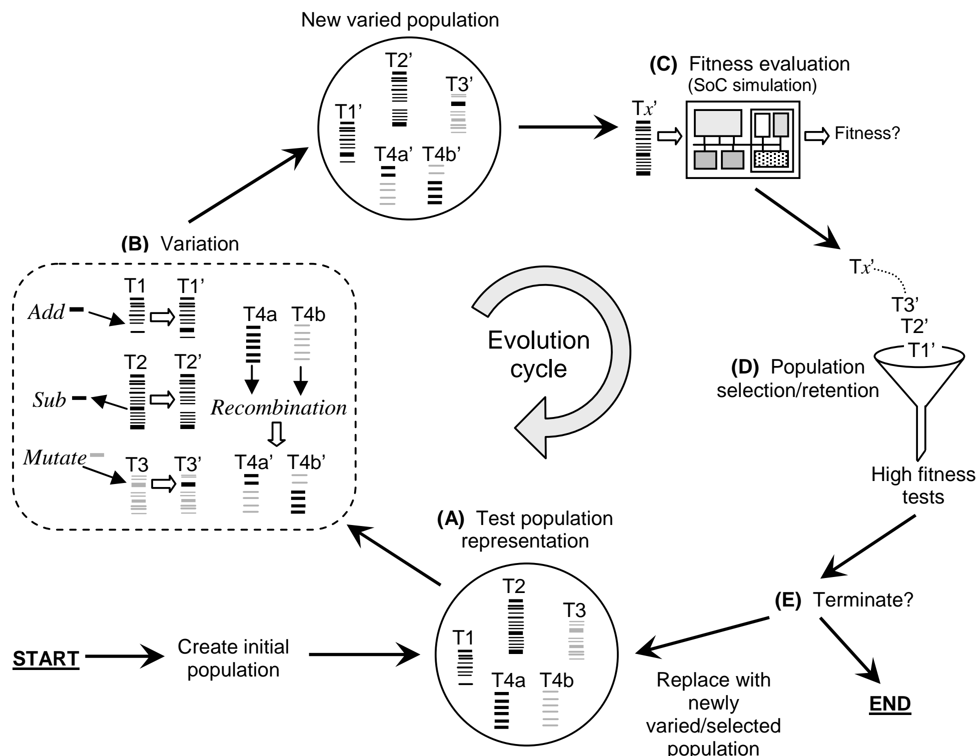


Figure 4.1 High level conceptual genetic evolutionary test generation flow

4.3 Software Application Level Verification Methodology Genetic Evolutionary Test Generator (SAGETEG)

Given our conceptual GEA test creation strategy, we design a genetic evolutionary test generator tool to generate test programs within the SALVEM platform. We refer to this test generator as the Software Application Level Verification Methodology Genetic Evolutionary Test Generator (abbreviated as SAGETEG). SAGETEG implements a GEA test generation process to automatically create software application tests as part of the SALVEM platform. It fits into the SALVEM platform as a module that uses the snippets library to create tests. The generated tests are then supplied to the remainder of the verification environment for SoC simulations and coverage results gathering. Figure 4.2 shows the high level flow of the SALVEM based verification system, and how SAGETEG is integrated into the SALVEM platform.

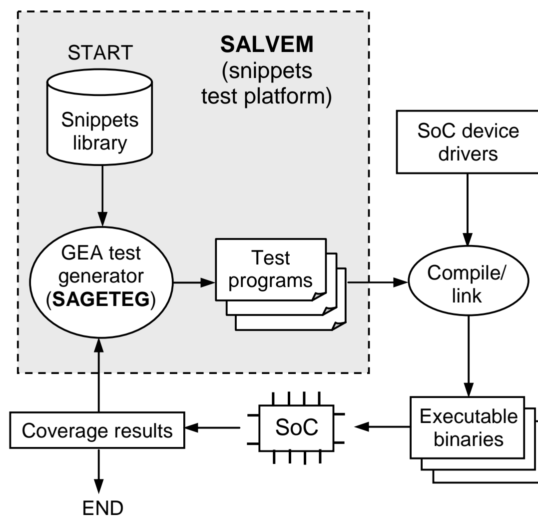


Figure 4.2 SAGETEG integration into SALVEM platform

The SAGETEG test generator generates test programs using SALVEM originated snippets from the snippets library. The test programs are then compiled and linked with SoC device drivers to form the executable test binary for SoC simulation. Coverage data gathered during simulation is then fed back as fitness results to control the test generator during future test evolutions.

The GEA test generation process conceptualised in Section 4.2 is facilitated and performed by SAGETEG. The objective of this GEA process is to maximise coverage of the SoC under test, subject to the limitations of test size capacity, resources and time. This will be realised by composing sequences of snippets automatically, based on information learned from previous test generations.

The initial philosophy for SAGETEG and how it was eventually architected and integrated into the SALVEM platform, was inspired by our feasibility study into GEA test generation of application test programs for SoCs. Before fully committing down the GEA driven research path, the usefulness and potential of GEA test generation had to be demonstrated. Our proposed GEA test generation concept for SALVEM was initially prototyped with the μGP tool [CSRS04b]. Because μGP operated at the microprocessor level, a number of modifications were needed, and integration issues had to be resolved in order to facilitate its usage within the SALVEM platform. Despite mismatch between μGP for SALVEM, our preliminary experimental results showed GEA based test generation for SALVEM was viable. Two thirds of the types of coverage evaluation reported results that were better than a randomised test creation approach. Additionally, the GEA process was also slightly more efficient, requiring less snippets, tests, and verification times. Appendix E.2 describes the feasibility study fully.

Whilst sufficient favourable results were shown to demonstrate GEA test generation under the SALVEM approach as feasible, the preliminary study also reveal numerous deficiencies with using μGP as the GEA test generation mechanism within SALVEM platform. μGP was never intended for software system-level test creations and SoC verifications, it was chosen simply because it was available for feasibility study of GEA test creations only. Therefore, SAGETEG was devised specifically for SALVEM.

SAGETEG builds on existing GEA strategies to tackle current issues with GEA test generations, such as the application of effective test variation, and avoiding overheads associated with current methods such as μGP. We demonstrate SAGETEG as the dedicated GEA test generation that achieves superior coverage over both existing GEA and non-GEA based methods.

Sections 4.4 to 4.8 explains fully, the components and phases of the GEA process employed by SAGETEG. We detail how GEA operates in a system level test creation environment to facilitate SAGETEG, including formalisations of operations conducted for the GEA test generation process. We also describe the modifications and enhancements to conventional GEA processes to adapt SALVEM test creation in order to achieve more comprehensive verification. Following these sections, an overview of the entire SAGETEG test generation is given in Section 4.9. Next, we begin by describing representation of GEA components for SALVEM test generation.

4.4 Genetic evolutionary representation

This section deals with the most essential element of any GEA process in general [Mic06] – how the genetic evolutionary building blocks (i.e., chromosome individuals and their genome constituents) are represented in our application problem domain of test generation. The correct mapping of SALVEM test generation components into appropriate GEA ingredients is an exercise that must be carefully considered at the beginning of any GEA test generation design. Otherwise, if operating with an unsuitable representation, the remaining GEA phases will not be able to achieve the objective goal of coverage maximisation.

The mapping of test generation into the GEA domain is illustrated diagrammatically in Figure 4.3. The GEA population of individual chromosomes is represented by the suite of SALVEM test programs. A chromosome individual is equivalent to a single test program, and snippets act as genes making up each chromosome as per the genetic coding governed by the SALVEM sequences of snippets methodology. The set of available genome is provided by the snippets library for the SoC under test.

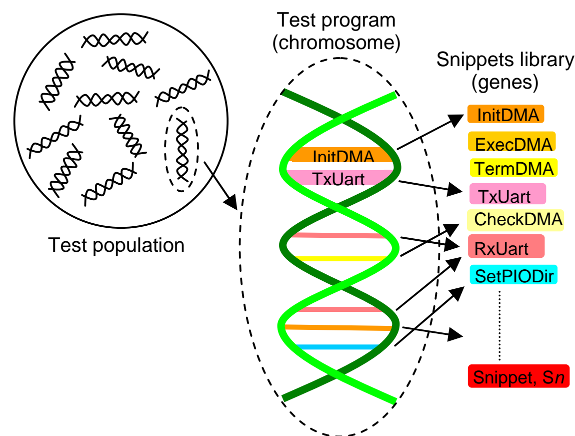


Figure 4.3 Genetic evolutionary algorithm test generation representation

Formally, we define the representation of the population, chromosome individuals, and genome for test generation as follows in Definition 4.1.

Definition 4.1 : SAGETEG genetic evolutionary representation

(i) The genetic evolutionary representation of the test suite population P is defined as a set,

$$P = \{\ t \mid t = \langle s_1, s_2, \ldots, s_n \rangle, s_i \in S \text{ for } i = 1, 2, \ldots, n\ \}$$

where S is the set of snippets which is the snippets library, n is the number of snippets in a test individual that varies for different tests, and t represents a test individual in the population which is defined in (ii).

(ii) The genetic evolutionary representation of the test individual t is defined as a n-ordered tuple $\langle \ldots \rangle$ delimited by angled brackets containing a sequence of snippets genes from S.

(iii) The genetic evolutionary representation of each snippet genome s is associated with a three member tuple $\langle V, L, M \rangle$, where V is the set of parameter variables that controls how the implemented software function of the snippet will operate to trigger SoC behaviours from the test program, L is the set of constraints, and M is the set of dependencies of the snippet governing how the snippet can be composed and ordered with other sequences of snippets in the test program.

Each of V, L, and M captures a characteristic of the snippet.

□

The number of snippets n that are composed to create a test program t grows during the evolutionary test generation process, and is limited by the test execution memory capacity of the test platform. From Section 4.11, for experimentation with SoCs, taking into account average sizes of snippets and the SALVEM platform, n is between 0 and 150 typically.

Snippet constraints L and dependencies M are predetermined and their influence is static throughout test generation. On the other hand, the snippet parameters V for each snippet influence a test program significantly. Different combinations of parameter values for a snippet yields different functional scenarios to be invoked each time the snippet is employed in a test program. For this reason, in Section 4.5.5 the primary variation operation, mutation, varies snippet parameters directly to explore test regions with each snippet.

Based on Definition 4.1, the test populations in Definition 4.2 are employed by SAGETEG.

Definition 4.2 : Genetic evolutionary test populations in SAGETEG

Let μ be the number of test individuals in the parent test population, and λ is the number of tests in the children test population.

(i) The parent test suite population $P_{\mu}(z)$ at evolution z is defined as the set of tests which contains test individuals from which variation is conducted to create new offspring tests.

(ii) The children test suite population $P_{\lambda}(z)$ at evolution z is defined as the set of tests which contains offspring test individuals that are newly created from the variation phase.

(iii) The combined parent and children test suite population $P_{\mu+\lambda}(z)$ at evolution z is defined as the set of tests which contains both the existing parent and new children test individuals, from which the population selection phase will select the next population of tests for survival into the next $(z + 1)$ evolution.

□

The populations in Definition 4.2 are subset of one another satisfying $P_{\mu}(z) \subseteq P_{\mu+\lambda}(z)$ and $P_{\lambda}(z) \subseteq P_{\mu+\lambda}(z)$. Note that z is a discrete natural valued integer specifying the current number of evolutions conducted thus far during the GEA process.

During the evolution, the test generator manages these test populations by inserting new test individuals and removing low fitness tests until the optimal test suite is attained. Our GEA representation for SALVEM test generation is effective because each of the test creation components maps naturally to their corresponding GEA elements. The test suite contains tests that collectively accumulate test coverage during test simulation, in the same way that a GEA population is evolved to tackle the application domain problem. The test program and snippets act as suitable chromosome individuals and genes respectively, because each individual test contributes differently to our objective coverage goal. Different ordering of snippets and different variants of snippets themselves can alter individuals to be (i) highly divergent or (ii) similar to one another; whenever either characteristic between individuals are required during the GEA process. At any evolution, the GEA process can guide the tests to differ greatly for exploring large test regions, or become relatively alike so as to bring the GEA process into a converged state when the test suite is close to optimal.

4.5 Genetic evolutionary variation

4.5.1 Overview

The goal of any verification is to explore the test space extensively in order to maximise coverage. To achieve this, a wide variety of test programs should be created using GEA variation. To invoke diverse range of SoC behaviours, variation must be able to produce many combinations of snippet sequences and various snippet parameter characteristics. From Definition 4.1, given our snippets library is the set

S, this implies a high percentage of the power set of S must be realised. According to set theory, the power set of S represent all possible snippet compositions, and thus provide a benchmark against which to measure capability of our variation operators.

To aim for the power set of S, we define five variation operators to create new tests. The operators are addition, subtraction, mutation, replacement, and recombination. Each variation aims to create new test *children* individuals by conducting various modifications to existing *parent* tests from the current population. The five variation operations are defined in the following sub sections.

4.5.2 Addition variation

The snippet addition operator randomly chooses a snippet from the snippet library and inserts this new snippet into the test program at a random point. The aim of this operation is to add new snippets that can invoke interesting combinations of SoC functionalities to attain higher coverage. The add operation is defined below.

Definition 4.3 : Addition variation, *Add*

The function $Add : P \times S \times \mathbf{Z} \rightarrow P$ is the addition variation that creates a new test t_{new} as follows.

$$t_{new} = Add(t, s^{add}, i) = \langle s_1, \ldots, s_i^{add}, \ldots, s_n, s_{n+1} \rangle \text{ subject to } g(t_{new}, s^{add}, i) = \text{true} \wedge n+1 \leq C$$

where t is the test that undergoes addition variation,

n is the number of snippets in t,

$s^{add} \in S$ is a randomly chosen snippet from the snippets library to add into the test,

$i \in \{1, \ldots, n\}$ is a randomly chosen position in the snippet sequence for snippet addition,

$g : P \times S \times \mathbf{Z} \rightarrow \{\text{true, false}\}$ is a function that checks if the new snippets sequence is legal and can be applied to the SoC, and

C is the maximum number of snippets that can be inserted into a test individual before the test exceeds SoC executable memory limits.

□

The function g checks the legality of the new test against constraints and dependency rules. Constraints and dependencies ensure the tests created by variation are executable on the SoC and test platform. If the added snippet's dependencies are not satisfied, the required snippets are added into the test. After all dependencies are resolved, if constraints are still not satisfied, another snippet for addition is selected. Constraints and dependency checks are described next.

Variation constraints

Constraints are categorised into explicit or implicit constraints. Explicit constraints express requirements specific to a snippet. For example, an InitDMA snippet chosen to perform transfers between byte sized I/O ports (e.g. with a UART device) must select the DMA transaction size to be byte sized; otherwise the snippet cannot be employed for testing.

Implicit constraints are defined for each snippet parameter to control the type and range of values chosen for the parameter. Each parameter can be restricted to a domain of allowable values, from which parameters should only hold. These parameter domain values are designed such that they will trigger the snippet to test interesting or critical SoC behaviours. The constraint is violated whenever the parameter contains a value not within its domain. The checking of explicit and implicit constraints are defined in Appendix E.4.

Variation dependencies

In addition to constraints, the function g conducts four types of snippet dependency checks, denoted as pre-strict, post-strict, pre-non-strict, and post-non-strict dependencies. A pre-dependency requires the dependant snippet to be inserted in the snippet sequence before the snippet under *examination*, which is the added snippet in this case. A post-dependency requires that the dependent snippet be inserted later on in the sequence after the snippet under examination has executed. A strict dependency implies the dependant snippet must be inserted immediately before or after the snippet under examination. A non-strict dependency simply requires the dependent snippet to be included in the test sequence at any point before or after the snippet under examination. Dependencies are essential for certain SoC operations to be conducted in the correct order. The four dependency checks are defined as follows.

Definition 4.4 : Pre-strict dependency

Let $g_{pre\text{-}strict} : P \times S \times \mathbf{Z} \rightarrow \{\text{true, false}\}$ be the function that checks for pre-strict dependency whereby the domain $P \times S \times \mathbf{Z}$ specifies the varied test, snippet to check dependency for, and the position at which this snippet is within the test. The pre-strict dependency is defined as

$$g_{pre\text{-}strict}(t, s, i) = \text{true} \quad \text{if } s^{pre\text{-}strict} = s_{i\text{-}1} \text{, false otherwise,}$$

where $s^{pre\text{-}strict}$ is the pre-strict dependency snippet that must be immediately before the snippet s under examination.

□

Definition 4.5 : Post-strict dependency

Let $g_{post\text{-}strict} : P \times S \times \mathbf{Z} \rightarrow \{\text{true, false}\}$ be the function that checks for post-strict dependency whereby the domain $P \times S \times \mathbf{Z}$ specifies the varied test, snippet to check dependency for, and the position at which this snippet is within the test. The post-strict dependency is defined as

$$g_{post\text{-}strict}(t, s, i) = \text{true} \quad \text{if } s^{post\text{-}strict} = s_{i+1} \text{ , false otherwise,}$$

where $s^{post\text{-}strict}$ is the post-strict dependency snippet that must be immediately after the snippet s under examination.

□

Definition 4.6 : Pre-non-strict dependency

Let $g_{pre\text{-}non\text{-}strict} : P \times S \times \mathbf{Z} \rightarrow \{\text{true, false}\}$ be the function that checks for pre-non-strict dependency whereby the domain $P \times S \times \mathbf{Z}$ specifies the varied test, snippet to check dependency for, and the position at which the snippet is within the test. The pre-non-strict dependency is defined as

$$g_{pre\text{-}non\text{-}strict}(t, s, i) = \text{true} \quad \text{if } \forall\, s^{pre\text{-}non\text{-}strict} \in S^{pre\text{-}non\text{-}strict}, \exists\, s_j \text{ in } t \mid s_j = s^{pre\text{-}non\text{-}strict} \text{ for } j = 1, \ldots, i-1 \text{ , false otherwise,}$$

where $S^{pre\text{-}non\text{-}strict}$ is the set of pre-non-strict dependency snippets that must be before the snippet s under examination, and is a subset of the snippets library such that $S^{pre\ non\text{-}strict} \subseteq S$.

□

Definition 4.7 : Post-non-strict dependency

Let $g_{post\text{-}non\text{-}strict} : P \times S \times \mathbf{Z} \rightarrow \{\text{true, false}\}$ be the function that checks for post-non-strict dependency whereby the domain $P \times S \times \mathbf{Z}$ specifies the varied test, snippet to check dependency for, and the position at which the snippet is within the test. The post-non-strict dependency is defined as

$$g_{post\text{-}non\text{-}strict}(t, s, i) = \text{true} \quad \text{if } \forall\, s^{post\text{-}non\text{-}strict} \in S^{post\text{-}non\text{-}strict}, \exists\, s_j \text{ in } t \mid s_j = s^{post\text{-}non\text{-}strict} \text{ for } j = i+1, i+2, \ldots, n \text{ , false otherwise,}$$

where $S^{post\text{-}non\text{-}strict}$ is the set of post-non-strict dependency snippets that must be after the snippet s under examination and is a subset of the snippets library so that $S^{post\text{-}non\text{-}strict} \subseteq S$, and n is the number of snippet in the test t.

□

The dependency checks are carried out one after another by the legality check function g. If any of the above dependency checks finds the added snippet has outstanding dependencies that are not presently

satisfied by the test, then the dependent snippets are added into the test. Given that the addition of dependent snippet could itself impose dependencies that are not satisfied, then the dependency check and addition of further dependent snippets must be conducted, and so forth. For this reason, in addition variation, the dependency check and insertion of dependency snippet is performed recursively, until all snippets that have been inserted into the test have no unresolved dependencies.

For each of the dependency checks that returns false, the function $d : P \times S \times \mathbf{Z} \rightarrow P$ is called to add dependency snippets. It takes in an input domain of the addition varied test, the added snippet, and the position of this inserted snippet, and returns a modified test that had undergone dependency snippet insertion. For example, $t_{new} = d(t_{new}, s^{add}, i)$ is invoked whereby t_{new}, s^{add}, and i are from Definition 4.3. The pseudo code implementation to perform dependency insertions are provided in Appendix E.5.

The recursive procedure of adding pre-strict, post-strict, pre-non-strict and post-non-strict dependency snippets are each executed one after another. The resultant test after the addition of one type of dependency snippet is passed on to add further snippets of other dependency types. The resultant test from each type of dependency snippet addition is summarised as below.

Given that the snippet s^{add} was added into the test at position i, the insertion of the dependent snippets will alter the test t_{new} as follows.

Pre-strict dependency snippet addition :

$t_{new} = \langle s_1, \ldots, s^{pre\text{-}strict}, s_i^{add}, \ldots, s_n, s_{n+1}, s_{n+2} \rangle$, where $s^{pre\text{-}strict}$ is inserted immediately before s^{add}.

Post-strict dependency snippet addition :

$t_{new} = \langle s_1, \ldots, s_i^{add}, s^{post\text{-}strict}, \ldots, s_n, s_{n+1}, s_{n+2} \rangle$, where $s^{post\text{-}strict}$ is inserted immediately after s^{add}.

Pre-non-strict dependency snippet addition :

$t_{new} = \langle s_1, \ldots, s^{pre\text{-}non\text{-}strict}, \ldots, s_i^{add}, \ldots, s_n, s_{n+1}, s_{n+2} \rangle$, where $s^{pre\text{-}non\text{-}strict}$ is inserted at an arbitrary position between the first snippet and s^{add}.

Post-non-strict dependency snippet addition :

$t_{new} = \langle s_1, \ldots, s_i^{add}, \ldots, s^{post\text{-}non\text{-}strict}, \ldots, s_n, s_{n+1}, s_{n+2} \rangle$, where $s^{post\text{-}non\text{-}strict}$ is inserted at an arbitrary position between the s^{add} and the last snippet.

Note that in each case above, the size of the test is increased by at least one snippet so that the total number of snippets in t_{new} is n+2. For non-strict dependencies, multiple additions of snippets from the

set of pre-non-strict or post-non-strict dependencies that are not satisfied in t_{new} can be performed multiple times.

Application of constraints and dependency checks

The function g – which ensures legality of the new test derived from addition variation – performs the constraints explicit and implicit checks g_{exp} and g_{imp}, and dependency checks $g_{pre\text{-}strict}$, $g_{post\text{-}strict}$, $g_{pre\text{-}non\text{-}strict}$ and $g_{post\text{-}non\text{-}strict}$ as a disjunctive OR,

$$g = g_{exp} \vee g_{imp} \vee g_{pre\text{-}strict} \vee g_{post\text{-}strict} \vee g_{pre\text{-}non\text{-}strict} \vee g_{post\text{-}non\text{-}strict}$$

where g_{exp}, $g_{imp} \in L$ are from the set of constraints, and $g_{pre\text{-}strict}$, $g_{post\text{-}strict}$, $g_{pre\text{-}non\text{-}strict}$, $g_{post\text{-}non\text{-}strict} \in M$ are from the set of dependencies in Definition 4.1. The constraints checks must be true, and dependencies resolved for the added snippet and existing snippets in the test, in order for addition variation to be deem successful.

4.5.3 Subtraction variation

The snippet subtraction operator picks a random snippet from a test individual and removes it from the test. The intention is to remove potentially redundant snippets that do not provide additional coverage. Snippet subtraction is defined below.

Definition 4.8 : Subtraction variation, *Sub*

The function $Sub : P \times \mathbf{Z} \rightarrow P$ is the subtraction variation that creates a new test t_{new} as follows.

$$t_{new} = Sub(t, i) = \langle s_1, \ldots, s_{i-1}, s_{i+1}, \ldots, s_{n-1} \rangle \text{ subject to } g(t_{new}, s^{sub}, i) = \text{true}$$

where n is the number of snippets in t,
$i \in \{1, \ldots, n\}$ is the randomly chosen position of the snippet to be removed,
$s^{sub} \in S$ is the chosen snippet at position i removed from the test, and
$g : P \times S \times \mathbf{Z} \rightarrow \{\text{true, false}\}$ is a function that checks if the new snippets sequence is legal and can be applied to the SoC.

□

Like snippet addition, the post-subtracted snippet sequence is checked by the function g to ensure it is legal and adheres to constraints. Otherwise, a new removal snippet is chosen and subtraction is attempted again. For snippet dependencies, similar dependency check functions defined in Definition 4.4 to Definition 4.7 are also conducted to check if any of the existing snippets in the test depend on s^{sub}. For example, is the snippet s_{i+1} pre-strict dependent on s^{sub} or is snippet s_{i-1} post-strict dependent on s^{sub}. For non-strict dependencies, are there any snippets before or after i that are post or pre dependent on s^{sub}. If any of these dependency checks are true, then the snippet s^{sub} cannot be removed.

4.5.4 Replace variation

The snippet replace operator creates different snippet sequences using both addition and subtraction operators one after the other. A random snippet is removed from the sequence first and then replaced with another using the add operator. The intention is for less effective snippets to be replaced by new snippet, so the test can be *recycled* whilst preserving existing characteristics. Even if the snippet removed is replaced with a snippet of the same type from the snippet library, the addition of the new snippet would still employ different snippet parameters. Snippet replace is defined in Definition 4.9 as a composition of the two previous variation operators.

Definition 4.9 : Replace variation, *Rep*

The function $Rep : P \times S \times \mathbf{Z} \rightarrow P$ is the replacement variation that creates a new test t_{new} as follows.

$$t_{new} = Rep(t, s^{add}, i) = Add(Sub(t, i), s^{add}, i) \quad \text{subject to} \quad g(Sub(t, i), s^{sub}, i) = \text{true} \wedge g(t_{new}, s^{add}, i) = \text{true}$$

where $i \in \{1, \ldots, n\}$ is the randomly chosen position of the snippet to replace,
$s^{sub} \in S$ is the chosen snippet at position i to be removed from the test,
$s^{add} \in S$ is a chosen snippet from the snippets library to add, and
$g : P \times S \times \mathbf{Z} \rightarrow \{\text{true, false}\}$ is a function that checks if the new snippets sequence is legal and can be applied to the SoC.

□

Snippet replace employs the same constraints and dependency checks of addition and subtraction in its test legality check function g. For dependencies, if the dependencies for subtraction is satisfied, but dependencies for the addition phase is not, then snippet replace also calls the dependency resolve function d described in Section 4.5.2.

4.5.5 Mutation variation

The snippet mutation operator alters a test individual by modifying the characteristics of a randomly chosen snippet. Specifically, mutation is performed by modifying the parameter values of that snippet. The aim of mutation is to seek out new test functions that have not been stimulated by current test programs.

Definition 4.10 : Mutation variation, *Mut*

(i) The function $Mut : P \times \mathbf{Z} \rightarrow P$ is the mutation variation that creates a new test t_{new} as follows.

$$t_{new} = Mut(t, i) = \langle s_1, \ldots, u(s_i), \ldots, s_n \rangle = \langle s_1, \ldots, s^{mut}, \ldots, s_n \rangle$$

where n is the number of snippets in t,
$i \in \{1, \ldots, n\}$ is the randomly chosen position of the snippet to mutate, and
u is a function that selects new parameter values for the chosen snippet and is defined in (ii) below.

(ii) Let $u : S \rightarrow S$ be the function to select new parameter values for a snippet s_i to return the mutated snippet, $s^{mut} = u(s_i)$, such that the new set of snippet parameters is given by,

$$\{ x_j^* \mid (x_j^* = rand(D_j) \wedge x_j^* \neq x_j) \text{ for each } v_j \in V,\ j = 1, .., |V| \}$$

where V is the set of parameters for snippet s_i,
v_j is the j-th parameter from V,
x_j is the value of the j-th parameter v_j,
x_j^* is the newly chosen mutated value of the j-th parameter v_j such that the chosen value is different from the previous value of the parameter x_j,
D_j is the domain set of values that can be chosen for parameter v_j to satisfy constraints, and
$rand$ is a function that randomly selects values for a parameter from its domain.

□

4.5.6 Recombination variation

Recombination produces two offspring test programs by selecting two existing parent tests from the current population and combining their snippet sequences. The operation is intended to mimic genetic mating and reproduction processes that occur in evolutionary nature. The goal of recombination variation is to preserve useful sequences of snippets and snippets characteristics from existing test

population into future evolutions; so that various critical SoC behaviours continue to be effectively tested in conjunction with other newly uncovered test space.

To select parent tests, a tournament selection function $TourSel_k : P^k \rightarrow P$ is employed, where k is the number of participants in the tournament. $TourSel_k$ is defined as a function that conducts a tournament between k tests, whereby the winning test selected provides highest coverage fitness f out of all participant tests in the tournament. The winning tests act as parents for recombination.

To every conduct recombination variation, the parent tests acquired from tournament selection and their crossover points are chosen beforehand as follows.

$$t_A = TourSel_k(t_1, \ldots, t_k) = \langle s_{A1}, s_{A2}, \ldots, s_{Ax}, s_{Ax+1}, \ldots, s_{An} \rangle$$
$$t_B = TourSel_k(t_1, \ldots, t_k) = \langle s_{B1}, s_{B2}, \ldots, s_{Bx}, s_{Bx+1}, \ldots, s_{Bn} \rangle$$

where t_A is the first parent test chosen to participate in the recombination,
t_B is the second parent test chosen to participate in the recombination,
k is the number of participant tests selected randomly for the tournament selection,
Ax is a randomly chosen crossover point in t_A,
Bx is a randomly chosen crossover point in t_B
An is the last snippet index in t_A,
Bn is the last snippet index in t_B.

A crossover point is defined as any randomly selected snippet within the parent test, such that the snippet sequence before or after the chosen crossover snippet will be replaced with a snippet sequence from another parent test. The crossover points Ax and Bx act as a separator creating two pairs of snippet sub-sequences each within t_A and t_B. The first child test program t_{AB1} is produced by linking the snippet sub-sequence of the first parent up to its crossover snippet, with the snippet sub-sequence of the second parent from its corresponding crossover point onwards. The second child test t_{AB2} is created similarly in an inverse manner. The definition for recombination variation is presented next.

Definition 4.11 : Recombination variation, *Recomb*

The function $Recomb : P \times P \times \mathbf{Z} \times \mathbf{Z} \rightarrow P \times P$ is the recombination variation that creates new tests t_{AB1} and t_{AB2} as follows.

$$(t_{AB1}, t_{AB2}) = Recomb(t_A, t_B, Ax, Bx) = (\langle s_{A1}, s_{A2}, \ldots, s_{Ax-1}, s_{Ax}, s_{Bx+1}, \ldots, s_{Bn} \rangle, \langle s_{B1}, s_{B2}, \ldots, s_{Bx-1}, s_{Bx}, s_{Ax+1}, \ldots, s_{An} \rangle)$$

subject to $g(t_{AB1}, s_{Bx}, Bx) = \text{true} \wedge g(t_{AB2}, s_{Ax}, Ax) = \text{true}$

where $g : P \times S \times \mathbf{Z} \rightarrow \{\text{true, false}\}$ is a function that checks if the new snippets sequences are legal and can be applied to the SoC.

□

Both children snippet sequences in t_{AB1} and t_{AB2} are required to undergo legality checks by g. Otherwise, new parents and crossover points must be selected again for recombination to succeed. Besides legality checks, the length of the new children tests must not exceed SoC memory limits as well. The legality checks by g are similar as per previous variation operators, in particular for the constraints checks.

In terms of dependencies, the dependency check functions from Definition 4.4 to Definition 4.7 are re-used for checks between different snippets. For both t_{AB1} and t_{AB2}, the pre and post strict dependencies between s_{Ax} and s_{Bx+1}, and also s_{Bx} and $s_{Ax\text{-}1}$ are checked, because each of their adjacent snippets has been exchanged from the other parent test at the crossover point. For pre-non-strict dependencies, each of the snippet from the sub-sequence after the crossover point must be checked to ensure any of its pre-non-strict dependent snippets are still satisfied, given that the snippet sub-sequence prior to the crossover point has been acquired from the other parent test. Similarly, for post-non-strict dependencies, each of the snippet from the sub-sequence before the crossover point must be checked to ensure its post-non-strict dependent snippets are still satisfied by the new post-crossover-point sub-sequence of snippets originally from the other parent. Both the non-strict checks must be repeated for both t_{AB1} and t_{AB2} with respect to their snippets sub-sequences at Ax and Bx.

Like subtraction variation, if dependencies cannot be met by the newly created tests, recombination variation will not succeed. For example, during early stages of evolution, the snippets sequences from parents may be too short to provide any suitable crossover points. Tests must undergo sufficient addition variation over a number of evolutions before they can be deem *mature* enough and sufficiently large to act as recombination parents.

4.5.7 Variation usage

Given the different variation methods, the test generator selects a test individual from the current parent population and applies one or more of the variation operators. These variation operations manipulate snippets genome and their sequences to create new tests using the existing test population. For each new test, because multiple variation operations can be applied, this allows for many small

fine-grain adjustments of test characteristics in new tests. This is so SoC coverage can be gradually attained without overlooking any important test space region.

Whilst some of these operators are typical of GEA variation, unlike traditional GEA, there are special mechanisms that must be put in place for test generation. These mechanisms are implemented with the use of constraints and dependencies. For instance, the size of a test program is limited by the amount of SoC executable memory. Therefore, the number of snippets that can be inserted into a test program is constrained by this hardware limitation. Given that the size of snippets varies, the aim of the GEA process is to seek out the most effective sequence of snippets that attains best coverage fitness whilst reducing the amount of left-over memory wastage unoccupied by snippets. Dependencies also affect test generation significantly and are discussed in the next section.

4.5.8 Effects of snippet variation dependency on test evolution

This section examines how snippet dependencies influence variation operations to create certain patterns of snippets orderings and characteristics in tests. We begin with a general discussion of dependency considerations for all variation operators.

One common restriction imposed by dependencies is variation *deadlocks*, whereby a test is prevented from undergoing variation due to the introduction of unresolved dependencies in the test if variation was to proceed. Variation deadlocks apply specifically to subtraction, replace, and recombination; because removal or exchanging of snippets sequences can cause the withdrawal of dependent snippets and break dependencies in the modified test. Addition and mutation variations do not suffer from this problem because addition automatically inserts dependent snippets to resolve unsatisfied dependencies, and mutation is not affected by snippet dependencies at all.

As an example, consider the sequence of snippets in Figure 4.4. The snippet sequence on the left is prior to any variation and contains numerous dependencies. For the first attempt at subtraction variation, below this snippet sequence, the possibility of applying subtraction on each snippet is listed. Due to dependencies, only snippet D can be removed, which shows the limited variation options for such short sequences. The resultant snippet sequence after subtraction is shown on the right. Any further attempts at subtraction result in deadlocks. This example is demonstrated for subtraction only, but applies for replace and recombination variation, which causes snippet removal as well.

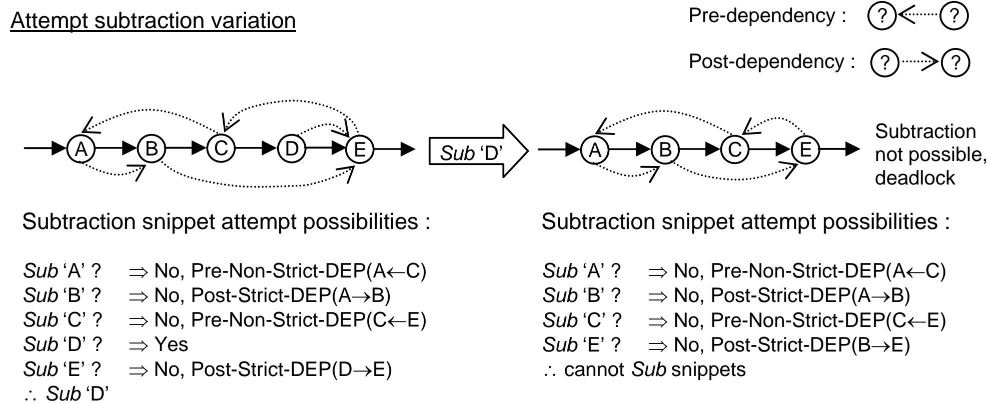


Figure 4.4 Subtraction variation deadlock caused by dependencies

When variation is repeatedly blocked by deadlocks, this implies (1) many dependencies exists between many snippets sequences and, (2) there is insufficient diversity of snippets in the test. Only when the snippet sequence of a test has grown such that dependencies can be satisfied by multiple snippets, then subtraction, replace, and recombination can be safely applied. This is because dependencies are more likely to be satisfied by other additional snippets despite the removal of a dependent snippet by the variation operation.

The effect of deadlocks is especially prevalent during the initial few evolution cycles of the GEA process when tests are small. In general, snippet removal due to subtraction, replace, or recombination variation is problematic and is usually futile until the test grows larger with the aid of addition variation.

The impact of deadlocks can be lessened by adding more snippets to alleviate and spread dependency responsibilities amongst multiple snippets. Therefore, in the beginning of our GEA test evolution, both additions and mutate variations is preferred over other variations. Another reason for promoting addition and mutate variation initially is because applying subtraction, replace, or recombination on small range of snippet sequences would yield little benefit even if dependencies are satisfied. Longer and wider range of snippet sequences not only reduces dependency deadlocks but also allow for greater choices in how the variation can be conducted. For example, in Figure 4.4, with a short snippet sequence initially, only one snippet could be chosen for subtraction. The remainder of this section describes dependency effects specific to each variation operation.

Addition variation dependency effects

The effect of dependencies on addition variation lies predominately with the growth rates of tests. During addition variation, the number of new snippets inserted into the test can be greater than one, depending on the dependencies of the snippet chosen for addition. If the addition snippet requires additional dependent snippets to be inserted into the test, the number of snippets inserted from one addition variation can be much greater because addition is conducted in a recursive manner as described in Section 4.5.2.

The effect of such recursive snippet addition is most pronounced at the start of evolution. Early tests contain little or no dependency satisfying snippets for new snippet addition variation. This effect is further compounded given addition variation is weighted for frequent application in the beginning of test evolution. Under such scenario, the addition variation of non-dependency satisfied snippet, and further non-dependency satisfied snippet insertion, and so forth, results in many snippets in the test from one variation. The overall effect is a rapid increase in test size at the beginning of test generation.

After some period of evolutions, sufficient snippets exist in the tests so that addition variation of new snippet can depend on these existing snippets. At approximately the same stage, other subtraction, replace or recombination variation will also be applied more often without deadlocks. Hence, after a period of high growth, the growth rate of tests will stabilise and could even decrease.

Figure 4.5 shows an example. The addition of a single D snippet to the initial snippet sequence on the left results in five additional snippets inserted into the test overall. This is because the initial snippet sequence is small, and contains only a few snippets that may not necessarily satisfy dependencies for new snippets added from the snippets library.

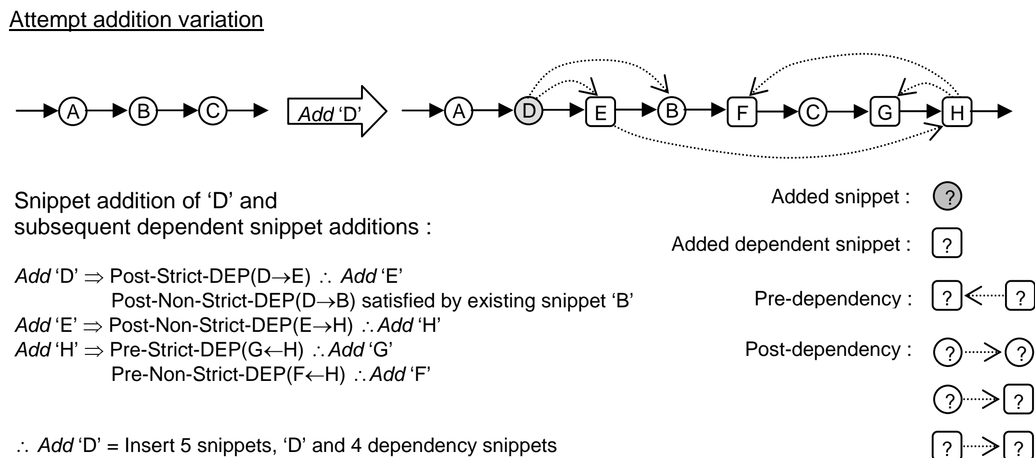


Figure 4.5 Insertion of addition snippet and dependent snippets by addition variation

Subtraction variation dependency effects

In subtraction variation, removal of snippets according to dependency restrictions can be beneficial. By adhering to dependencies, subtraction can (1) remove snippets deemed unnecessary or independent of other snippets, and (2) remove multiple instances of snippets that are the same where each satisfies the same dependencies of other snippets.

Removal of the above kinds of snippets produces tests that contain sub-sequences of inter-dependent snippets only. These inter-dependent snippet sequences usually interact with each other. They are responsible for multi-device interactions and concurrent operations tested on the SoC that acquire significant test coverage, and should be preserved. Also, independent snippets only contribute to coverage a limited number of times, hence removing multiple instances of such identical snippets do not cause reduction in coverage. Leaving dependent snippets sequences in-place and removing only independent snippets frees up snippets slots for further addition of other snippets. This allows for additional dependent sub-sequences to form in the future, enhancing coverage.

A common occurrence of dependency influenced subtraction is the removal of independent snippets near the start and end of the test. Such snippets are often placed or shifted (due to variation side-effects) into these locations during initial evolutions, because they do not depend on other snippets in the test. Dependent snippets on the other hand, are located strategically away from the test boundaries because snippets earlier or later in the test rely on them. Removal of independent snippets near test boundaries trims the test size. It reduces the possibility of repetitive snippets, relying on other identical independent snippets elsewhere in the test to provide the same test functionality and coverage. The truncation of snippets will enable new dependency sub-sequences to gradually form from the test boundaries as new snippets are progressively added later.

For example, the above behaviour can be observed with initialisation snippets that operate independently on I/O devices. During initial test generations, repetitive and sometimes consecutive PIO or ResetUart snippets are often located near the start and end of tests. The PIO and ResetUart snippets were either added directly to such locations or shifted there due to addition of other snippets. Clearly, executing multiple instances of these snippets does not provide additional coverage. This coverage would be already acquired by single executions of any one previous snippet of the same type. Instead, these initialisation snippets, along with other independent snippets should be distributed across middle sections of the test, between dependent snippet sub-sequences. By doing so, different combinations of independent and dependent snippet sequences can be configured to drive other SoC operations. Removal of multiple redundant initialisation snippets will allow addition of further new

snippets; this creates other sequences of snippet functionalities which require these initialisation snippets to be placed more effectively elsewhere.

Employing a diagrammatic example again, in Figure 4.6, let the P and R snippets represent PIO and ResetUart snippets respectively. If subtraction variation of either P or R is conducted, Figure 4.6 shows removal of which snippets are possible according to dependency restrictions. In Figure 4.6, two snippet sub-sequences (a) and (b) can be identified by their inter-dependencies amongst snippets. The P and R snippets within (a) cannot be removed because other snippets depend on them within this sub-sequence. Only the independent P and R snippets at either ends of the test can be subtracted. After their removal, the trimmed down snippet sequence will be dominated by snippets interactions and SoC operations from the two dependent sub-sequences only. If further snippet subtraction is required, then only the R snippet between (a) and (b) is removed.

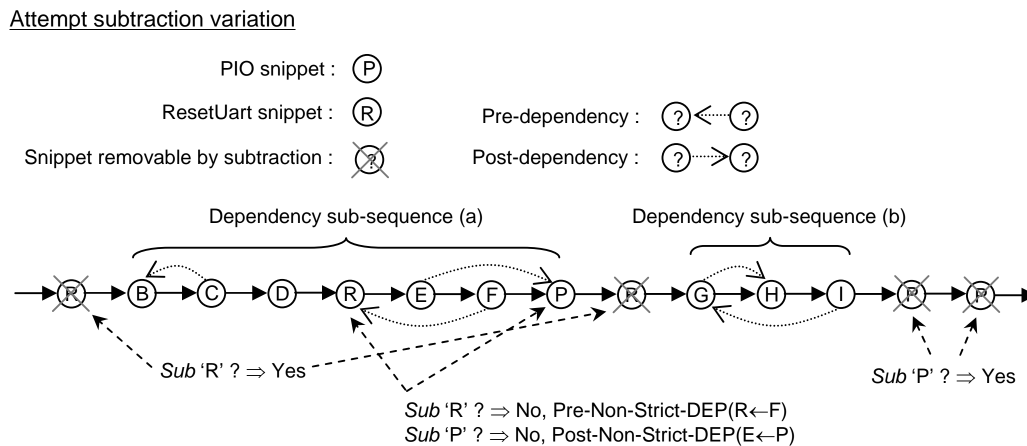


Figure 4.6 Dependency effects on subtraction variation

Replacement and mutation variation dependency effects

The replacement variation employs both addition and subtraction variation, and shares the same dependency effects from both variation operators. Mutation variation modifies internal snippet characteristics only, and is not affected by dependencies at all.

Recombination variation dependency effects

For recombination, the size of parent tests must be sufficiently large so that exchange of snippet sub-sequences between parents will produce diverse and effective new children tests. This prerequisite is required regardless of dependency restrictions, although dependencies can exacerbate the lack of

diversity by reducing the number of crossover points possible. With inadequate size or too many dependency restrictions, the offspring tests would be too alike to their parents. Similar sequences of snippets will be re-used for further evolutions rendering recombination effectively useless.

An example is shown in Figure 4.7, whereby recombination is attempted between two very short snippets sequences on the left. These short sequences are usually common at the start of evolutions. The possible recombinations and resultant snippets sequences are shown on the right of Figure 4.7. After recombination, both resultant sets of children tests do not add any value to the test population at all; because they are essentially the same as their parent tests. The parent tests were too short to allow for any different snippet sub-sequences in the children tests.

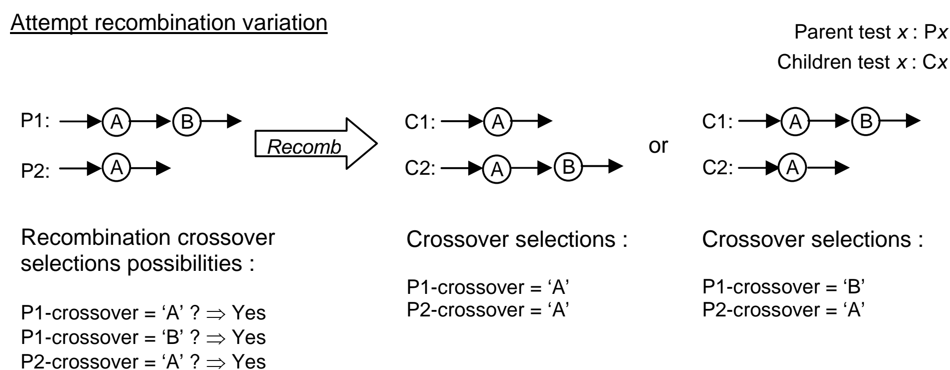


Figure 4.7 Recombination variation between very short parent tests

Besides small parent tests, recombination is usually prevented by dependency restrictions and deadlocks as well. Like subtraction variation in Figure 4.4, after recombination, existing dependencies amongst snippets are unlikely to be preserved across multiple sub-sequences that originate from different parent tests. Our strategy to tackle this issue is to control variation usage weights so that recombination is conducted only after initial evolutions when tests have become sufficiently large.

After initial evolutions, when recombination does become feasible, dependencies can still influence recombination to select crossover points near the start and end of parent tests. The reason for such selections is that these locations are usually the only permissible crossover points that can be chosen to maintain legality of dependencies in new offspring tests. Such selection of crossover points is most common immediately after initial evolutions, when recombinations have just qualified for usage. At this stage, even though parent tests are sufficiently large to overcome dependency restrictions, the parent tests available are not diverse enough to provide any range of crossover points to be chosen. To overcome this, recombination should be delayed even further, to allow the test population to continue

growing and become more diverse. Greater availability of crossover points throughout parents that produce balanced recombination in children tests is then possible.

If using crossover points close to test boundaries, the outcome is children tests whereby most of the snippets from parents' snippet sub-sequences are contained within one child test, whilst the other child test is populated with very short or empty snippet sequences.

This scenario is illustrated in Figure 4.8. The snippet sequences on the left are parent tests. Beneath the parent tests, the selection of crossover snippet points is restricted by dependencies. The possible children tests arising from recombination are shown on the right, using only crossover point snippets that do not break any dependencies in their resultant children tests. The first set of children tests produces identical snippet sequences to its parents, whilst the second set of children tests is unbalanced, with most of the snippets in one child. The reason for such children tests are because dependencies and short snippets parents do not provide adequate range of choices for crossover points to be selected. Hence, only crossover points near the start and end of parents can be used.

Despite the mechanisms put in place to allow suitable crossover point selections, selection of crossover points in some cases could default back toward the boundaries of parent tests. As evolutions continue to vary tests, it is possible that some tests could end up with many tightly coupled dependency sub-sequences of snippets. In this case, even in a large test, these multiple dependencies would prevent many snippets nested within middle sections of the test to act as crossover points; because they are too reliant by other snippets as dependencies. However, this scenario is considered extremely rare.

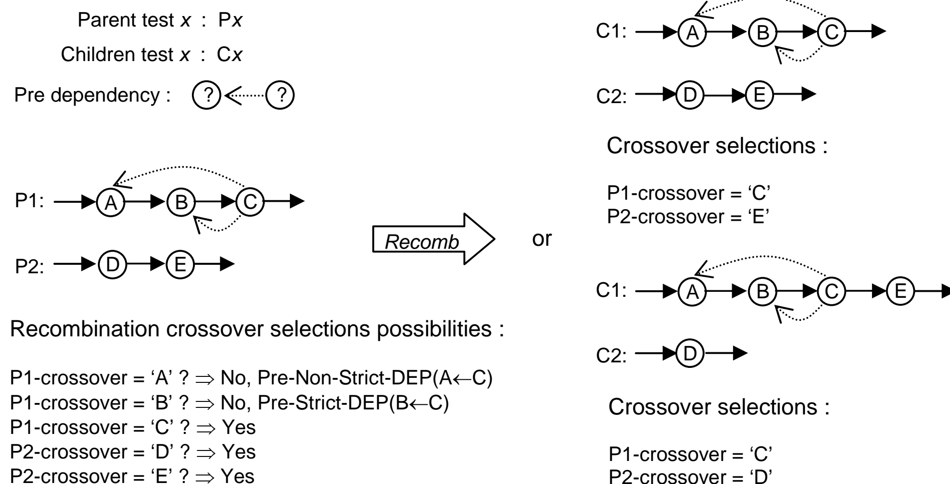


Figure 4.8 Dependency effects on recombination variation

Variation usage strategy and variation weights

Given the effects snippet dependencies impose on variation, we propose the following strategy for their usage. During early evolutions of test generations, addition and mutation variation are applied frequently. Subtraction, replace and recombination variation are applied after initial test phases when tests are of sufficient size and diversity are available for these variation operators to be applied effectively.

The variation usage is based on the following. As described at the beginning of Section 4.5.8, variation must be applied so as to avoid deadlocks. For this reason, our original approach proposed greater addition and mutation variation. Despite dependency and deadlock considerations however, our usage of addition and mutation during early GEA must not be excessive as well.

The addition and mutate variation is tuned toward uncovering new and extensive test space. However, they do not examine test regions thoroughly, unlike other variation. Addition and mutate variation must be complemented with other variation as soon as it is practically possible; in order for newly uncovered test region to be properly examined and avoid overlooking important related test scenarios. The goal in using addition and mutate variation with other remaining forms of variation is to methodically seek out comprehensive coverage of particular test regions in the test space.

With these factors in mind, we revised the usage strategy such that only during the initial evolutions phase, addition and mutation variations are applied rigorously. This ensures subtraction, replace, and recombination are not applied in vain in the beginning, but only when they can succeed without deadlocks. Furthermore, after this initial stage, the subtraction, replace, and recombination variation can also examine closely the local test regions discovered by add and mutate variation previously.

The management of variation usage is controlled by weighting values given to each variation operator, which controls the likelihood of their application to create new tests. The higher the value assigned to a variation's weight variable, the higher probability the variation will be chosen to create a new test. Therefore, in our variation usage strategy, addition and mutation are given higher weights than subtraction, replace, or recombination initially. These weightings are user assigned in the beginning, but are then controlled by an automatic self-adaptation process during evolution. This adaptation process is described in Section 4.5.9 next.

4.5.9 Variation self-adaptation

GEA test generation is highly sensitive to the type of variations and the frequency at which they are used. Variation must be applied at appropriate ratios with respect to each other and the GEA process. Even though different variation usage weights are given different initial values, during evolutions, each variation weight must be continually monitored and adjusted in order to provide best test generation conditions. The mechanism by which we facilitate automatic adjustment of variation weights is to implement variation self-adaptation.

In self-adaptation, variation weights are monitored and evaluated with respect to test generation and coverage results. Weight values are adjusted depending on test fitness values, and can also take into account dependency and constraint considerations, and other test generation issues to maximise best use of variation. The monitoring and evaluation of variation usage weights can be carried out by various means, one of the common methods is to employ Rechenberg's rule [Mic96, Rec73].

Rechenberg's rule

Rechenberg's rule states the ratio of successful variation to non-successful variation should be 1/5. A variation operation is deem successful if it created a new individual that yielded higher fitness than the original individual on which variation was applied upon.

If the ratio of a variation operator is greater than 1/5, the variation weight is increased; otherwise if the ratio is less than 1/5, variation weight is reduced. The probability change factor by which variation weights are adjusted each time is a constant value, and is a percentage of the pre-adjusted weight. Appendix E.6.1 describes fully the weight adjustment by Rechenberg's rule.

In the context of SoC test generation, we explain the usage of Rechenberg's rule with the aid of a test space diagram in Figure 4.9. Each point represents a test functionality scenario to test, and test scenarios can be grouped together within regions of circular boundaries based on common test functions or devices which they test. From the test space perspective, the test generator must make use of variation to search this test space extensively, seeking out as many of these functional test regions as possible. Next, each of the test regions must be examined thoroughly to verify local test functional scenarios fully. The addition and mutate variations are to search large span of the test space to uncover new test regions. The remaining variation can then look into local test scenarios to create appropriate tests for verification. The self-adaptation controls individual variation with the goal of tackling coverage of the test space in this manner.

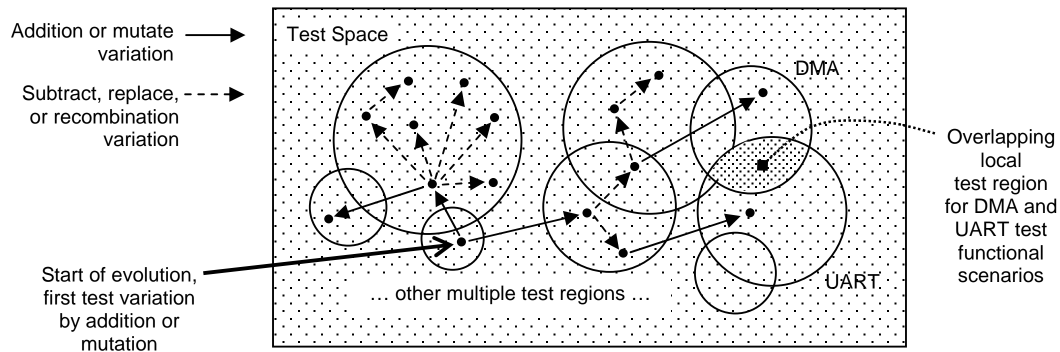


Figure 4.9 SoC test space diagram for GEA test generation

Employing a success ratio of 1/5 throughout evolutions implies variations will provide at least one out of five tests that enhance coverage fitness, and thereby uncover a new test region or cater for another locally captured functional test scenario. This manipulates variation usage relative to one another, bringing about a balance between (1) extensive exploration of new and previously un-encountered test space using strong add or mutate variation (solid arrows in Figure 4.9), and (2) continual examination of test scenarios within locally bounded test regions ensuring all test regions are completely covered using remaining variations (dashed arrows in Figure 4.9). Without any variation guidance, locally uncovered test regions may not be fully examined before the GEA process continue on to some other newly discovered test region; or too much emphasis will be placed on current test regions such that exploration of other portions of the test space do not occur.

Pitfalls of Rechenberg's rule

Whilst the 1/5 ratio goal for self-adaptation is effective, Rechenberg's procedure to achieve this goal ratio suffers from two major pitfalls when applied for SALVEM. They are: excessive adjustment of variation weights, and too early application of self-adaptation. The fixed size adjustment of weights immediately at the start of evolutions can force weights to extremely high or low values too quickly, even before middle stages of GEA. During initial evolutions, the tests are too small containing only short snippet sequences. The initial test population is not given opportunity to properly consolidate and examine the new uncovered test regions in the beginning, before adjustment of variation weights modifies the types of variations applied and shifts test focus to other test space.

Once variation weights reach excessive values, too many evolutions would be needed to recover them from such runaway conditions back to practical values; whereby variation can be managed again in some useful manner. With excessive weights, the GEA is forced to certain particular variation

constantly whilst other variation operations are never triggered. In most cases, weights will not be recovered from their extreme values before the end of test generation. These problems are also exacerbated by the fixed size increment or decrement of weights, whereby the constant change factor is often too large. Only a small number of evolutions need pass before weights become extreme.

The above issues were firstly tackled by way of minor refinements to the existing self-adaptation. Variation weight limits were enforced, and self-adaptation was delayed to apply only after early evolutions when the test population has matured to account for initial test regions. The change factor was also reduced so that weights were altered by smaller amounts. Whilst these refinements improved self-adaptation, other issues persisted.

The increment and decrement of weights when the success ratio is above and below 1/5, may not always provide the desired effect to steer subsequent success ratio toward 1/5. If the ratio is above 1/5, further incrementation of weights may drive GEA to other test regions, rather than thoroughly examine current test regions which are being focused upon. Moving on too soon to other test regions will raise the ratio further and critical test functions from previous test regions will be overlooked. When the ratio is below 1/5, lowering weights to focus thoroughly within local test regions can lead to further reduction in the success ratio. This occurs if local test regions have been thoroughly examined already. In this case, GEA should be steered to uncover other new test regions using stronger variation weights to increase the success ratio.

The impact from these problems is made worse whenever the ratio is already near 1/5. Blindly increasing or decreasing variation weights by fixed amounts in subsequent evolutions will simply produce opposing desired effect to drive the ratio further away from 1/5. Linearly adjusting weights by the same factor over many successive evolutions causes large changes to the GEA process, and the flow-on effect is a vastly different test population and undesired ratio. Rather than blindly adjust weights by fixed amounts each time, the amount of change should take into account the current ratio with respect to the 1/5 goal. The process should be fine-tunning rather than alter weights significantly.

We conducted extensive preliminary experimental runs to observe the above issues, including employing both SAGETEG and re-using μGP. Appendix E.6 details our full description and analysis of Rechenberg’s rule self-adaptation, and the observations and drawbacks of their application for SALVEM test generation. It is clear self-adaptation in its original form was inadequate, especially for the different interactions and complexities of snippets and test programs from SALVEM test generation. Therefore, we propose a revised self-adaptation strategy.

4.5.10 Revised self-adaptation strategy

Our revised self-adaptation method is unique in two aspects. First, our method compares variation success ratio against a target ratio which is expressed as a range. Second, the method adjusts variation weights depending on the distance and relative *closeness* of current evolution variation success ratio to the desired target ratio. These concepts are shown in Figure 4.10, and the self-adaptation policy is defined in Definition 4.12.

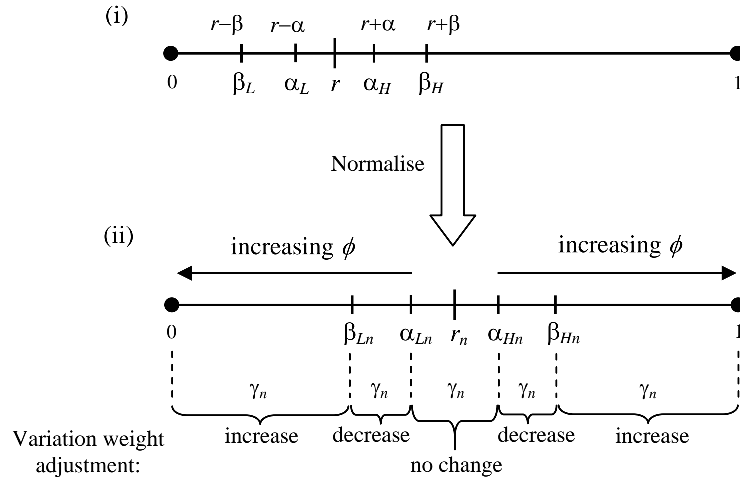


Figure 4.10 Variation success ratio range for revised self-adaptation in Definition 4.12

Definition 4.12 : Revised self-adaptation

The revised self-adaptation policy is as follows.

Let γ be the variation success ratio of the population, and r be the target variation success 1/5 ratio (r = 0.2).

The target ratio variance value α, is chosen to be 0.05, subject to $|r - \alpha| < \min\left(\frac{r}{2}, \frac{1-r}{2}\right)$, and 0 otherwise. The change-over point variance value relative to r, is denoted β, and is assigned to be, $\beta = \min\left(\frac{r}{2}, \frac{1-r}{2}\right) = 0.1$.

Let α_L be the lower target ratio range value, $\alpha_L = r - \alpha$,
α_H be the higher target ratio range value, $\alpha_H = r + \alpha$,
β_L be the lower change-over point, $\beta_L = r - \beta$,
β_H be the higher change-over point, $\beta_H = r + \beta$.

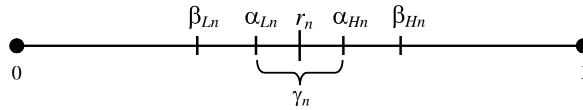
The ratio comparison is conducted against normalised target ratio and interval values. The scaling of r = 0.2 to the normalised value r_n = 0.5 is so the ratio comparison range has its midpoint as r_n.

To normalise the other interval range variables, the normalisation factor $n = r_n/r$ is multiplied to each of α, α_L, α_H, β, β_L, β_H, and γ variables to produce the corresponding normalised variables α_n, α_{Ln}, α_{Hn}, β_n, β_{Ln}, β_{Hn}, and γ_n.

The adjustment of a variation weight $\omega(z)$ at the current evolution index z is as follows for the following three cases, where σ = 0.9 is the variation weight change factor chosen from classical self-adaptation which is close to but within a limit of one, and ϕ is the weight adjustment factor which determines the amount to alter variation weights each time.

Case A : Population success ratio lie within target ratio range, then variation weight remains unchanged.

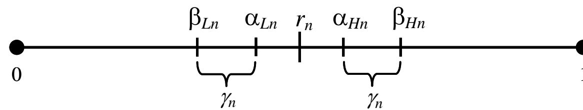
$$\omega(z) = \omega(z-1) \qquad \text{if } (\alpha_{Ln} \le \gamma_n \le \alpha_{Hn})$$



Case B : Population success ratio lie within lower or upper change-over point but not within target ratio range, then variation weight is decreased by factor proportionate to the distance between population success ratio and target ratio.

$$\omega(z) = \omega(z-1) \times \sigma \times \phi \qquad \text{if } (\beta_{Ln} \le \gamma_n < \alpha_{Ln}) \vee (\alpha_{Hn} < \gamma_n \le \beta_{Hn})$$

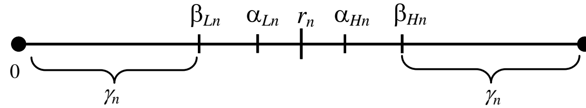
$$\text{where} \quad \phi = \begin{cases} (r_n - \gamma_n) \times (1/(r_n - \beta_{Ln})) & \text{if } (\beta_{Ln} \le \gamma_n < \alpha_{Ln}) \\ (\gamma_n - r_n) \times (1/(\beta_{Hn} - r_n)) & \text{if } (\alpha_{Hn} < \gamma_n \le \beta_{Hn}) \end{cases}$$



Case C : Population success ratio is less than lower change-over point or greater than higher change-over point, then variation weight is increased by factor proportionate to the distance between population success ratio and target ratio.

$$\omega(z) = \omega(z-1) \times (1/\sigma) \times (1+\phi) \qquad \text{if } (\gamma_n < \beta_{Ln}) \vee (\gamma_n > \beta_{Hn})$$

$$\text{where} \quad \phi = \begin{cases} (\beta_{Ln} - \gamma_n) \times (1/\beta_{Ln}) & \text{if } (\gamma_n < \beta_{Ln}) \\ (\gamma_n - \beta_{Hn}) \times (1/(1-\beta_{Hn})) & \text{if } (\gamma_n > \beta_{Hn}) \end{cases}$$



Note the $(1 + \phi)$ term is employed to ensure the weight value is increased.

□

In Figure 4.10 (i), the notion of the range based target ratio is shown. The 1/5 target ratio value r forms the focal point of the target ratio range from α_L to α_H, where $\alpha_L = r - \alpha$ and $\alpha_H = r + \alpha$ are the lower and upper target ratio range value, and $\alpha = 0.05$ is the variance from r that forms the target range. Test population ratios falling within $r \pm \alpha$ is considered to have satisfied the target ratio.

Figure 4.10 (i) also shows the weight adjustment change-over points β_L and β_H, which is determined by the change-over point variance factor β relative from r. The change-over points designate the transition between incrementation and decrementation of variation weights. Within β_L and β_H, the test population is considered sufficiently close to achieving the target goal ratio, and will be within the α_L to α_H range in following evolutions. Hence, variation weights are decreased to continue fine-tunning toward the target range and prevent any major test variation. External to the β_L and β_H range, variation weights are increased to excite greater test population changes and bring about changes to the success ratio as well.

In Figure 4.10 (ii), the success ratio comparison range for weight adjustment from Figure 4.10 (i) is normalised to display more appropriate range and intervals, in order to conduct actual weight adjustments. The normalised target ratio r_n and normalised interval range values α_{Ln}, α_{Hn}, β_{Ln} and β_{Hn}, are compared against the normalised test population variation success ratio γ_n. Figure 4.10 (ii) shows whether weights are increased, decreased or left unchanged depending on which range γ_n falls within. The degree of weight adjustment depends on the weight adjustment factor ϕ, which is proportionate to the distance difference between current population ratio γ_n and the target ratio. The greater the difference, the greater weight adjustment will be applied.

In Definition 4.12, the weight adjustment factor ϕ, is evaluated with respect to intervals $|r_n - \beta_{Ln}|$, $|\beta_{Hn} - r_n|$, $|\beta_{Ln} - 0|$, and $|1 - \beta_{Hn}|$ depending on the value of the population ratio. This is so that appropriate

adjustment factors that are proportionate to different intervals are taken into account to change variation weights by properly sized values. For example, population ratios near change-over points β_{Ln} and β_{Hn} experience much less variation as the ratio advance closer to r_n. The closer to r_n the greater the need to preserve similar kinds of tests in the population and maintain current levels of variation successes. A goal of using ϕ is to apply much greater reduction to variation weights that were already set low from previous evolutions. Similarly, when increasing variation weights for population ratios external to change-over point intervals $r_n \pm \beta$, ϕ will raise weight values more significantly because the population ratio has already drifted away from the target ratio.

Derivation discussions and observations of revised self-adaptation

Compared to traditional self-adaptation, our revised self-adaptation adjusts variation weights based on different criteria and with varying degrees of change. Maintaining the 1/5 success ratio goal, our self-adaptation is more sensitive to the success ratio, specifically the relation between current test suite success ratio to the desired goal.

Armed with the knowledge and experience of self-adaptation from our preliminary experiments, the following observations were reasoned. Whenever large variation weights are employed, the test population undergoes greater variation to produce highly diverse tests. This causes high probability of change to the success ratio, and was observed by Rechenberg as well [Rec73]. However, the ratio could increase or decrease away from 1/5 goal. When the ratio is substantially different to 1/5, in order to provide ample opportunity to drive the ratio toward 1/5 quicker, variation weights should be adjusted by greater amounts. This creates more new and vastly different tests from other test regions so the test population ratio can take larger steps progressing toward the 1/5 goal.

For low variation weights, the test population experience minor modifications from previous evolutions. Resulting tests are slightly refined and the success ratio experience slight alteration only. Hence, for test populations whose ratio is close to 1/5 already, subsequent new tests should not be varied too much. Current success ratios close to 1/5 indicates GEA was driving the test population toward the 1/5 goal already. To reduce likelihood of drifting away from 1/5, lower variation should be maintained to produce similar tests. Therefore, when success ratios are closer toward 1/5, the variation weights are reduced by decreasing amounts, regardless whether the ratio is greater or less than 1/5.

From these observations, in our revised self-adaptation, the amount of variation weight adjustment must be proportional to the difference between current test population success ratio and the 1/5 goal.

As the test population ratio advances toward 1/5, to maintain current levels of successes, less variation is applied so lower variation weights and adjustments are assigned. Eventually, the ratio will be very close to achieving the 1/5 goal such that either none or very fine-grained minor adjustments will be required by variation. In contrast, classical Rechenberg's rule simply increases or decrease weights by the same amount each time, depending if current ratio is lower or greater than 1/5.

Another significant distinction in our revised self-adaptation is to use a range based target ratio rather than a strict 1/5 valued goal. Given our test individuals are created such that many finer-grained differences between snippet characteristics and sequences are common, the likelihood of attaining an exact 1/5 success ratio is extremely low. Falsely adjusting weights when the ratio is sufficiently close to 1/5 would only force the ratio to drift away from the goal and be detrimental to the GEA process.

Revised self-adaptation for other operations

Our usage of self-adaptation is not restricted to variation weights only. Self-adaptation can also be applied to influence the number of multiple variation operators applied per test, or in selecting which snippet to add, remove, mutate, replace, or act as recombination crossover points by the five variation operators. For example, each snippet is assigned a selection probability weight value that is self-adapted. The size used by tournament selection for recombination variation can also be adjusted in a self-adaptive manner based on the fitness success of children tests produced with respect to parents.

4.6 Genetic evolutionary fitness evaluation

For GEA test generation, fitness evaluation measures the SoC coverage attained from applying a test individual to the SoC. Coverage reports the percentage of test events exercised by a test against all the possible events. A higher coverage implies greater likelihood of uncovering design bugs, which is a fundamental goal of verification. A test individual is considered the fittest when it attains highest SoC coverage. The test coverage event and matrix are described as follows.

Definition 4.13 : Test coverage event

Let E be the set of all measurable coverage test events. The test coverage events that can be exercised by tests from GEA test generation is defined as a tuple a,

$$a = \langle e_1, e_2, \ldots, e_m \rangle,\ e_i \in E \text{ for } i = 1, 2, \ldots, m$$

where m is the total number of possible test coverage events.

□

Definition 4.14 : Test coverage matrix

Let h be a function that that determines the number of times a test coverage event e is exercised by a snippet s, such that $h : E \times S \rightarrow \mathbf{Z}$. If h returns 0, this implies the snippet does not cover the coverage event e.

Given the tuple of n snippets in test t (as defined in Definition 4.1) and tuple of coverage events in a, the test coverage matrix $A \in \mathbf{Z}^{m \times n}$ is defined by

$$A = \begin{bmatrix} h_{11}(e_1, s_1) & \cdots & h_{1n}(e_1, s_n) \\ \vdots & \ddots & \vdots \\ h_{m1}(e_m, s_1) & \cdots & h_{mn}(e_m, s_n) \end{bmatrix}_{m \times n} \downarrow \text{coverage events } e_1,\ldots, e_m$$

$$\xrightarrow{\text{snippets } s_1,\ldots, s_n}$$

where the rows of A represent the coverage events from a and the columns represents the snippets in t.

□

Each ij-th element of A consists of the function h, which measures the number of times a coverage event e_i was exercised by the snippet s_j. The matrix shows not only which coverage events were exercised by which snippets, but also how many times each event was exercised and by which snippets. Establishing such a matrix, the GEA test generation process can be guided more effectively because previous snippet genome that was useful can be easily identified. Based on Definition 4.13 and Definition 4.14, the coverage metric for fitness evaluation is obtained as follows.

Calculating the coverage metric for fitness evaluation

The evaluation of coverage fitness metric is defined by the following two steps (i) and (ii).

(i) To calculate the coverage metric percentage of covered events, the matrix A is reduced. Multiplying A by a single column matrix containing n rows of 1s, the reduced matrix B is,

$$B = A \times \begin{bmatrix} 1 \\ \vdots \\ 1 \end{bmatrix}_{n \times 1} = \begin{bmatrix} h_{11}(e_1, s_1) & \cdots & h_{1n}(e_1, s_n) \\ \vdots & \ddots & \vdots \\ h_{m1}(e_m, s_1) & \cdots & h_{mn}(e_m, s_n) \end{bmatrix}_{m \times n} \times \begin{bmatrix} 1 \\ \vdots \\ 1 \end{bmatrix}_{n \times 1} = \begin{bmatrix} h_{11}(e_1, s_1) + \ldots + h_{1n}(e_1, s_n) \\ \vdots \\ h_{m1}(e_m, s_1) + \ldots + h_{mn}(e_m, s_n) \end{bmatrix}_{m \times 1}$$

B is a single column matrix where row i is the number of times the corresponding coverage event e_i was exercised by the snippets in the test.

(ii) To calculate a quantitative value of the coverage metric fitness, let v be a function that transforms coverage events and snippets data from simulation of a test t, into the form of the reduced matrix B.

The coverage metric fitness result c of a test t is given by,

$$c = f(t) = \frac{m - w(v(t))}{m} = \frac{m - w(B)}{m}$$

where $f : P \to \mathbf{R}$ denotes the fitness function that evaluates c, and w is a function that counts the number of row elements in B containing 0.

The function w counts the number of coverage events that were not exercised by the test. For every GEA test simulated to test an SoC, the matrices A and B are created for fitness evaluation based on the coverage events in a and snippets from the snippets library S. By representing coverage events using matrices, the range of coverage events can be readily analysed, and matrix operations can be performed to provide a quick summary assessment of verification effectiveness. The coverage metric c quantifies the accumulated coverage attained from a test, ensuring duplicate coverage events are not counted toward the final coverage metric result. The value calculated for c is used directly in some variation operations (e.g. recombination) or in the population selection phase to retain only the fitter tests that exhibit high coverage. In fact, f is essentially the objective fitness goal function of our GEA test generation process.

Given the objective function f, a number of different types of test coverage events for e can be employed in our GEA test generator. Coverage events may be specially devised or conventional coverage metrics can be reused. For example, a test coverage event may involve exercising statements in the design code (line coverage), toggling bit values of the SoC storage elements (toggle coverage), or traversing control and branch paths through the design description (conditional coverage). The total number of coverage events m depends on which coverage measurement is chosen. For example, in line coverage, m is the total number of lines in the design code. Separate GEA test generation processes are conducted for different fitness evaluations of each type of coverage measurement. Attaining high fitness for these coverage events ensure a thoroughly verified and higher quality SoC design overall.

4.7 Genetic evolutionary population selection

The population selection phase establishes the next population of tests to continue further test generate evolutions. In each evolutionary cycle, the number of tests representing parent individuals in the current parent population is denoted as μ, whilst the number of newly created test children in the children population is λ. In population selection, based on coverage fitness results, only the best μ number of tests from the combined existing parent population and newly created children population are retained. These tests form the new parent population $P_\mu(z+1)$ for the next evolutionary cycle. This selection scheme is based on the $(\mu+\lambda)$ evolutionary strategy [Fog00, Mic96]. Whilst other selection methods had been considered, $(\mu+\lambda)$ selection was chosen because it maintains a population of higher coverage yielding tests for exploring the coverage space.

To mimic more realistic evolutionary processes, we improve the population selection strategy by enforcing limited lifespan on test individuals. This implies that a test individual cannot be selected if it has existed in the population for more than a pre-specified number of evolutions, regardless of its coverage fitness. Like biological individuals that eventually pass away, even if a test is so fit to survive throughout the entire test generate process, it will be eliminated from the test population when it exceeds the allowed lifespan.

The goal of the limited lifespan policy is to reduce likelihood of population stagnating. If the population is consistently filled with selections of the same tests over many evolutions, overall fitness and quality of the test suite will not gain any significant improvements. Variation will simply create similar tests based on previous set of same parents such that the same tests are selected for many subsequent evolutions again. The test generation process will be given a false impression that no further GEA enhancements are possible. With limited lifespan, new tests will be injected into the population whenever older tests have reached their lifespan. The new tests will hopefully identify new test regions and allow for variation to create more diverse tests to further attain higher coverage. From preliminary studies and experimentation (and the research conducted in Chapter 5), a lifespan of five is typically used for test generation. The detailed formalised description of population selection is given in Appendix E.7.

4.8 Genetic evolutionary termination

In SAGETEG, the evolutionary process is terminated by checking for two termination conditions. The termination process is described as follows.

Let $F(z)$ be the average fitness of test population $P_{\mu+\lambda}(z)$ at the current evolution z such that,

$F(z) = \frac{\sum_{x=1}^{\mu+\lambda} f(t_x)}{\mu+\lambda}$ where μ is the number of parent tests, λ is the number of children tests, $f(t)$ is the objective function that evaluates the coverage of a test t.

The test generate evolutionary process terminates when one of the termination conditions below is satisfied.

(i) $F(z) \geq G$, where G is the target coverage fitness of the test generation process;

(ii) $F(i) \leq F(i-1)$ for $i = z-K, \ldots, z-2, z-1, z$ and $(K < z)$ where K is the number of prior consecutive evolutions to check for population fitness improvement.

For the termination condition in (i), alternatively, the test generation can also end if any test exceeds the pre-specific coverage fitness goal. In this case, rather than an average fitness goal, a higher coverage goal is used because comparisons are against single individual tests. Average test suite fitness can sometimes be pulled down by some low-performing tests. Usually, the target coverage is set to a very high value above 95%.

In (ii), if insufficient coverage persists, the GEA process may terminate by timing out after a certain number of evolutions. The test generation ends when there has been no overall coverage improvement of the test suite for a pre-chosen number of consecutive generations. This condition implies SAGETEG has exhausted almost every possibility for enhancing the test suite using the current population. If the SoC is still not sufficiently tested, SAGETEG should start a new evolutionary process with a different initial population.

4.9 Genetic evolutionary test generation for optimisation

Recall from Definition 4.14 that $f : P \rightarrow \mathbf{R}$ is a function that evaluates coverage fitness c attained by a test. One of the objectives of the evolutionary method in test generation is to find t so as to maximise $f(t)$, subject to limited test simulation time and the fixed SoC memory size that can hold a test. The optimised test attained by maximising $f(t)$ shall provide the optimal test coverage.

This optimised test is beneficial for post-silicon bring-up procedures when a newly fabricated SoC needs to be tested immediately with a test program to ensure the chip is operational. Instead of apply numerous application programs to test various parts of the chip individually, the optimised test program derived from the evolution process can be used ensuring higher coverage of the overall chip with one single test.

The optimisation of *t* is achieved by the evolutionary test generation process shown in Figure 4.11. The test generation process combines the GEA characteristics defined in Sections 4.4 to 4.8. The process creates a random set of μ number of test individuals in the beginning. These tests contain very small number of snippets, and are evolved through many evolutionary cycles to cultivate them into larger and greater coverage yielding tests over time. In each evolution, the variation creates new λ number of tests, the fitness evaluation quantifies their coverage, and the population selection delivers only the μ fittest number of tests to the next evolution cycle. This evolutionary loop is repeated until the optimised test is attained when a termination condition is triggered.

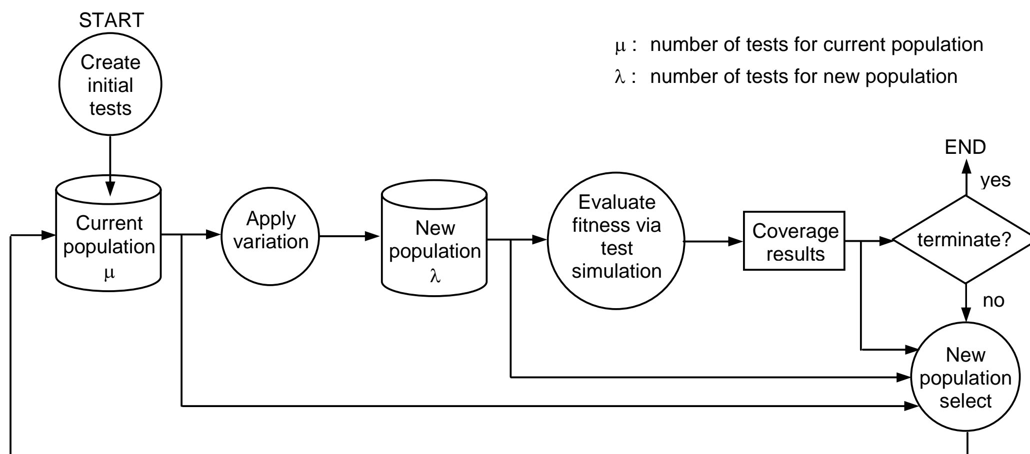


Figure 4.11 Genetic evolutionary test generation flow in SALVEM

The other outcome of GEA test generation is that the final population set of μ number of tests will be the set that attains the best accumulated test coverage from the GEA process. A higher overall coverage implies more comprehensive verification of the chip design. In our flow, different quantitative coverage metric is used independently as the fitness goal objective to drive separate evolutionary test processes.

The GEA test generation pseudo code implementing the flow in Figure 4.11, various test creation features of SAGETEG, and other implementation details are described in Appendix E.8. Integration of SAGETEG into the SALVEM platform was previously shown by the high level SoC architectural flow

diagram of Figure 4.2. Figure E.10 in Appendix E.8 shows the entire SALVEM verification platform incorporating SAGETEG in greater detail.

4.10 Benefits and shortcomings of SAGETEG

The key benefits of SAGETEG are threefold. First, unlike other test creation methods such as randomisation, our test generation is guided by information from earlier test creation and simulation runs. The coverage data from previous test suite population is used to select test characteristics to generate further new tests. This gives rise to a coverage driven method with the aim of uncovering new test regions and thoroughly examine existing ones.

Second, despite the single objective GEA process, a beneficial side effect is the minimal test sizes that eventuate from higher coverage yielding tests. Because our tests are initially empty or contain only few snippets, they are progressively cultivated over many evolutions to hold only the required snippets that contribute to the coverage they attain. Thus, the number of snippets and test sizes is kept low. For SAGETEG, this is unintended and considered an additional benefit, whereas Chapter 6 optimises test size with multi-objective GEA explicitly.

SAGETEG can also be easily expanded with new snippets into the genome set, to be chained together with other snippets as long as constraints and dependencies are resolved. Additional snippets can be created to target critical SoC behaviours or provide further dimensionality of SoC functions for testing. SAGETEG is also highly applicable to verification of other SoCs as long as appropriate snippets are available; which is generally possible because snippets are highly re-usable as discussed in Chapter 3.

In terms of shortcomings, SAGETEG could suffer from the following issues. SAGETEG is based on a guided search process, and is prone to take many evolutions and long simulation runs to achieve best coverage. Any technique based on searching the test space could develop into a runaway process, whereby the test population stagnates and further exploration of new test functions do not occur. Hence, various mechanisms described in earlier sections are applied to maintain diversity and injection of new tests during evolutions, until exhaustive search with the current process eventuates. Another drawback is that SAGETEG is fundamentally reliant on and limited by the snippets test building blocks. If snippets themselves do not provide adequate capability to exercise the SoC, employing them in SAGETEG composed tests would be futile. The strength of SAGETEG is to make use of snippets in the most effective and efficient manner by configuring snippets individually, and combining them appropriately to expose SoC functions and interactions that need verification.

With respect to other GEA test generators such as μGP, SAGETEG operates specifically at the software application ANSI-C level. It employs our specially crafted snippets test building blocks which can be configured for any different SoC functions, and additional snippets can be incorporated to expand the snippets library. μGP on the other hand employs assembler instructions as test building blocks. Whilst this does not require upfront effort to develop, the assembler instructions set are fixed and test programs is limited to the types of instructions that are defined to trigger hardware processor behaviours only.

SAGETEG employs a refined self-adaptation method as described in Sections 4.5.9 and 4.5.10. It is also applied to other GEA test generation parameters, not just the variation operators' weight values. Other GEA test generators generally use classical self-adaptation which may not be suitable for test generations (e.g. μGP uses traditional Rechenberg's rule with some inertial mechanism on variation weights only).

For GEA variation, SAGETEG provides greater control and flexibility in the types of variation that can be performed, thus allowing for more diverse test creation. For example, multiple forms of variation can be performed at one time to create a test. As for variation specific operations, the number of participants in tournament selection of parents is allowed to vary in recombination. Other GEA test creations strictly allow only one form of variation at a time, and all variation operators use the same application weight value.

SAGETEG makes use of multiple termination conditions to ensure proper detection of GEA test creation exhaustiveness. For example, termination can be triggered when consecutive best fitness values do not improve. Such condition checks prevent test generation from continuing needlessly without actually enhancing population fitness. Triggering of such termination conditions indicates the test population needs to be refreshed with new set of diverse tests again, and a new GEA process should be restarted. Other GEA test methods may not allow for such useful monitoring of the population.

Finally, whilst other GEA test generators may allow parallel processing of test evolutions, SAGETEG does not require such capabilities. Given sufficiently fast simulation speeds readily available today and with the minimal sizes of snippet test programs, extending SAGETEG for parallel execution may not provide any benefits. Parallelism would only be useful for multi-objective GEA test generations of multiple objectives such as that in Chapter 6. Appendix E.9 discusses more detailed and comprehensive analysis of SAGETEG with other GEA test generators such as μGP.

4.11 Experiments and results

4.11.1 Setup and experimental method

The aim of our experiments is to investigate the usability and effectiveness of employing GEA methods for test generation implemented by SAGETEG. Additionally, SAGETEG shall be compared against previous test generation methods employed for SALVEM verification. To achieve this, SAGETEG was applied to create evolving test suites to verify the Nios SoC (Appendix A). All experiments, including other test generations for comparison with SAGETEG, were conducted on a RedHat 9 Linux platform powered by a 3Ghz Intel Pentium 4 CPU and 2GB RAM.

The full set of snippets from the snippets library was used for GEA test program composition by SAGETEG, and is described and listed in Appendix E.10. To select test generation parameters for SAGETEG, a number of preliminary test generations were conducted. These test generations employed a range of parameter values which were appropriate from a GEA test creation perspective. The preliminary GEA processes serve to calibrate SAGETEG for proper test generations in SALVEM. The important GEA parameters for SAGETEG are tabulated in Table 4.1.

On the use of test population sizes of 30 for μ and 20 for λ, these are the most favourable sizes based on preliminary testing. Such population sizes provide a sufficiently large set of diverse individuals to mutate and mate with. Using not overly large parent and child population size also allows fitness evaluations to complete within practical time frames. The initial size of individual test programs contained a small number of snippets, less than three; so the evolution process can vary the snippets sequences cultivating them into larger test individuals over time. The small initial test size also enables the test programs to grow and be refined into an effective snippet sequence over progressive evolutions. Tests containing too many snippets initially would be too restrictive for GEA to insert other new or potentially useful snippets before the test size exceeds SoC program memory limitations.

Table 4.1 GEA test generation parameters

Parameter	Value
Parent population size μ	30
Children population size λ	20
Pre-selected number of evolutions	30
Lifespan of a test (number of evolutions)	5
Number of consecutive evolutions *K*, to check for test generation improvements; used for triggering appropriate GEA termination	5

Other remaining GEA parameters are selected as described in earlier Sections 4.4 to 4.8 for each of the relevant GEA phase descriptions. For example, initial variation weights are assigned according to dependency analysis described in Section 4.5.8. Greater addition and mutation variations are introduced at the beginning of GEA test generation, at twice the initial weight of other variation operators. Like other parameters, during the GEA process, various parameters will also be adjusted as necessary according to GEA operations such as self-adaptation, even though self-adaptation is only invoked after five evolutions.

With these parameters, GEA evolved tests were created by SALVEM to maximise line, toggle and conditional coverage of the Nios SoC. Individual GEA test generations were carried out and driven by each coverage metric. The number of evolutions carried out for each test generation runs vary, depending on the type of coverage metric. On average, 28 evolutions were performed by SAGETEG, culminating in 558 tests and 16,745 snippets.

For comparative evaluation purposes, we conducted equivalent randomised (described in Appendix D) and µGP based test generations on the Nios SoC. We also compare against a manual application test generation approach whereby application based tests under the SALVEM strategy were manually created by-hand. To ensure fair comparison results, the same snippets library was used for all randomised and GEA test generations. Unlike GEA, the random approach does not use coverage to drive test generation. Hence, a single test generation run was used to generate random-only tests whilst measuring all coverage metrics concurrently. For µGP, the test generation conducted is configured to be equivalent to our SAGETEG generations for evaluation purposes.

4.11.2 Coverage results and test snippets usage

Table 4.2 presents the coverage, and tests and snippets usage results for SAGETEG and three other comparative test generations. Tests generated by SAGETEG are more effective and efficient. They achieve higher coverage over other methods that would have required more snippets and tests. The higher coverage is a direct consequence of the evolutionary process continually seeking out new test functions based on coverage data from previous evolutions. Based on the snippet variation dependency analysis, and along with our self-adaptation method (Sections 4.5.8 and 4.5.9), various characteristics with snippets composed tests are addressed to provide enhanced SALVEM verification.

In Table 4.2, the cumulative coverage refers to coverage accumulated from all tests created by a particular test generation method. The raw coverage figures correspond to the best test from each

method considered to have achieved the highest coverage. Both coverage results are recorded at the end of test generations when further improvements are no longer evident. At the end of the test generations, the cumulated coverage is maximal from the test suite, and the raw coverage is that of the optimised test *t* that maximises *f*(*t*).

Table 4.2 Coverage, tests and snippets usage results

Line coverage	**SAGETEG**	**μGP**	**Randomised**	**Manual application**
Raw coverage %	98.5	97.2	87.7	66.5
Cumulative coverage %	98.9	97.5	89.9	66.6
Tests	602	2,373	950	37
Snippets	15,691	72,990	31,685	N/A
Toggle coverage	**SAGETEG**	**μGP**	**Randomised**	**Manual application**
Raw coverage %	93.0	86.6	77.1	48.8
Cumulative coverage %	93.7	87.1	79.2	48.8
Tests	652	2,292	750	37
Snippets	28,000	158,101	34,824	N/A
Conditional coverage	**SAGETEG**	**μGP**	**Randomised**	**Manual application**
Raw coverage %	81.0	69,8	67.9	62.1
Cumulative coverage %	83.0	72.4	69.3	66.6
Tests	420	1,089	420	37
Snippets	6,545	25,586	18,722	N/A

Note : Manual application tests are created entirely from scratch, they do not employ snippets.

For both cumulative and raw coverage, SAGETEG reports the best results demonstrating both superior overall test suite and the best optimised test. It achieves between 1.5% to 32.4%, 6.6% to 44.9%, and 10.5% to 17.4% improvement over other methods for line, toggle and conditional coverage respectively. These improvements must also be considered within the context of number of snippets and tests utilised. With many more tests, a test generation method should achieve higher coverage, but the number of tests should not be excessive. A truly effective test generation method is one that provides high coverage verification without using more tests than necessary.

In the test generations of this experimental setup, the total number of snippets and tests varies. This is because each test generation undergoes different variations with every GEA process. Different number (and types) of snippets will be added, removed, replaced, and recombined. Recombination produces at least two test off-springs, and population selections differ between GEA processes to give different

number of tests. Given random tests also contain different number of snippets, we report both tests and snippets usage in Table 4.2 for comparisons.

According to Table 4.2, 37 manual application tests were hand-created, which provided the lowest overall coverage as expected. But given this small number of tests, the coverage achieved can be considered satisfactory. The main barrier is the effort and time required to manually create these tests, directing them for all coverage goals. This is not practical for SoC verification. Manual application tests are to complement automated techniques only.

Randomised tests achieve higher coverage by way of automation and executing many tests. However, these tests do not receive any guidance, and the approach is to run as many random tests as possible to gain whatever coverage eventuates at the end. The number of tests and snippets are unpredictable without test generate control measures. For this reason, randomised tests are inferior to strategically guided test generations.

Whilst μGP is guided by GEA, it generates the greatest number of snippets and tests because its test generation and execution is assembler instructions based. Because snippets are hand-crafted to assembler instructions equivalent macros, it is smaller in size and is able to fit more snippets in tests. Hence, it can run more tests within the memory limit. μGP's coverage results are better than randomisation because of GEA. But we observed that its saturated coverage results are achieved well before the last μGP test is run, indicating lack of coverage progression. Despite being able to run many tests, this indicates the GEA test generation by μGP is also not suitable for SoCs or SALVEM verification. In spite of efforts to map snippets to assembler macros, there are major shortcomings in μGP's GEA process and the SoC functions that can be triggered.

The coverage achieved by SAGETEG is greater than all other test generation methods. This is because SAGETEG is customised specifically for SALVEM verification and SoC testing. It caters for snippet and variation dependency effects, and self-adaptation of GEA parameters are attuned for snippets and their characteristics during GEA testing. With the exception of manual application tests, SAGETEG employs the least number of tests and snippets as well.

In general, larger and longer tests do not always provide corresponding gains in coverage. The same level or even greater coverage can be achieved with shorter tests and less resources if optimised properly. SAGETEG's GEA process provides such optimality. This is demonstrated by SAGETEG's results over μGP and randomisation. Even though μGP is GEA based, it requires many more tests to attain higher coverage over randomisation, and still could not surpass SAGETEG's performance.

In terms of particular coverage metric, conditional coverage is observed to be most demanding. Conditional coverage is lowest overall because of the number of conditional paths to exercise and measure. This places greater complexity during test and coverage measuring, so that only shorter tests and test generation runs are possible. Given this complexity and lower initial coverage, unlike line or toggle coverage, conditional coverage presents test generation methods with greatest prospect of demonstrating coverage improvement. Again, the best conditional coverage improvement is from SAGETEG

From a GEA perspective, SAGETEG also demonstrates greatest gain in terms of the average coverage attained. The gain also takes into account the number of tests and evolutions conducted, under which the coverage improvement was realised. The coverage gain results show SAGETEG was able to best exploit the opportunities available for improving each coverage metric the most. These gain results are shown in Table 4.3 and underlines SAGETEG as the better GEA technique over μGP. SAGETEG is more suited to test generations for SALVEM. Randomised test experiences favourable gain results partially because its initial coverage from earlier tests was low to begin with.

Table 4.3 Average coverage gain results

	SAGETEG	μGP	Randomised
Coverage gain %	18.9	10.7	16.2
Coverage gain % per test	0.53	0.25	0.49

Note : The coverage gain results are averaged over the line, toggle and conditional coverage verification runs. Manual application created tests results are not applicable as these tests were not generated automatically.

For completeness, the coverage results for individual SoC devices attained by each test generation method are shown in Appendix E.11. In general, SAGETEG achieves higher coverage for almost every device. One exception is the PIO, of which SAGETEG coverage did not present improvements. However, this can be addressed by refining SAGETEG's GEA and examining the applicable PIO snippets for enhancements.

Given SAGETEG's superiority over other methods, we put SAGETEG's results into broader verification context. For instance, despite not achieving full coverage or only providing limited improvement for certain devices, SAGETEG's coverage results and gain are highly valuable. In SoC verification, when coverage levels exceed 80%, any improvement is considered extremely beneficial. The remaining 20% of coverage events are usually hidden deep within the logic of the SoC design and cannot be easily exercised. Whilst further improvements in SAGETEG is possible and forms the basis

for future work to enhance coverage, the coverage levels attained is critical to Nios SoC testing. Appendix E.12 presents in-depth discussion of remaining coverage attainment above 80% or more.

4.11.3 Coverage progress

Coverage progress was monitored during the entire test generation and verification runs for each test generation method (except manual application tests creation). The graphs for line, toggle and conditional coverage are plotted against number of tests in Figure 4.12, Figure 4.13 and Figure 4.14. On each plot, pairs of coverage graph lines are displayed. In each pair of coverage trend lines, one corresponds to the plot of raw coverage of individual tests, the other is cumulative coverage. The cumulative coverage is the coverage attained from all tests thus far at any particular stage of the test generation process; excluding duplicate coverage of same design test space (coverage events) by multiple tests. Raw coverage trend lines are always lower than their counterpart cumulative line because raw coverage represents only the coverage for each one particular test.

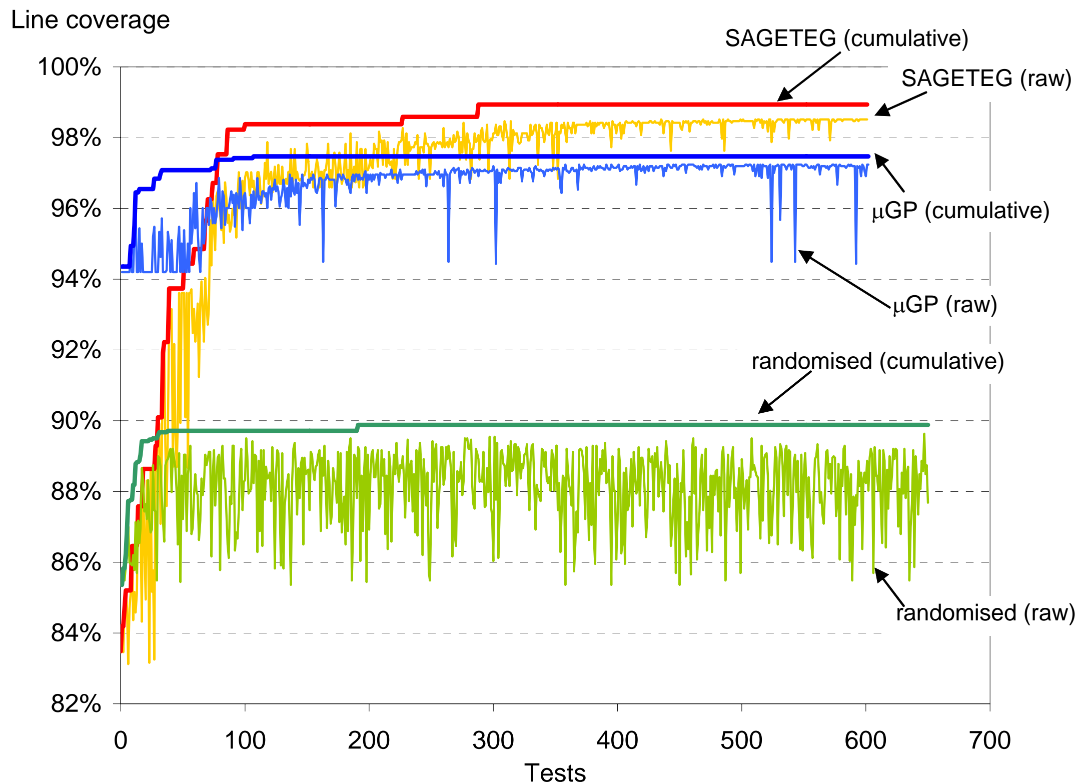


Figure 4.12 Line coverage process versus tests

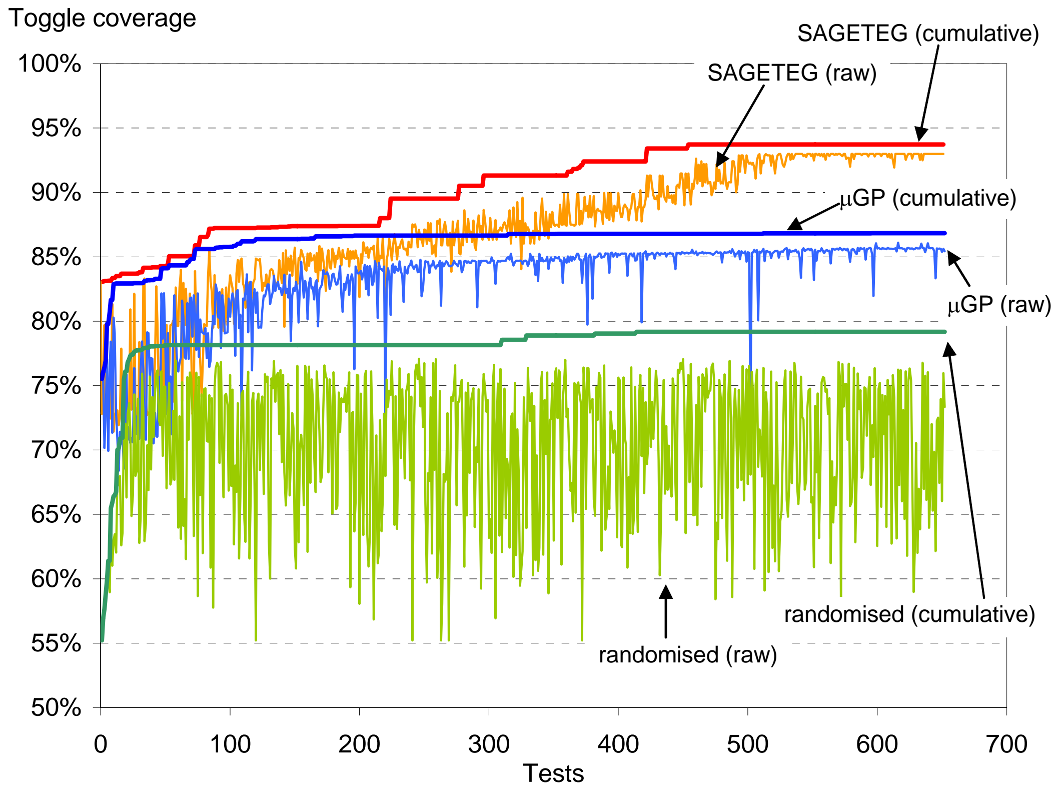


Figure 4.13 Toggle coverage process versus tests

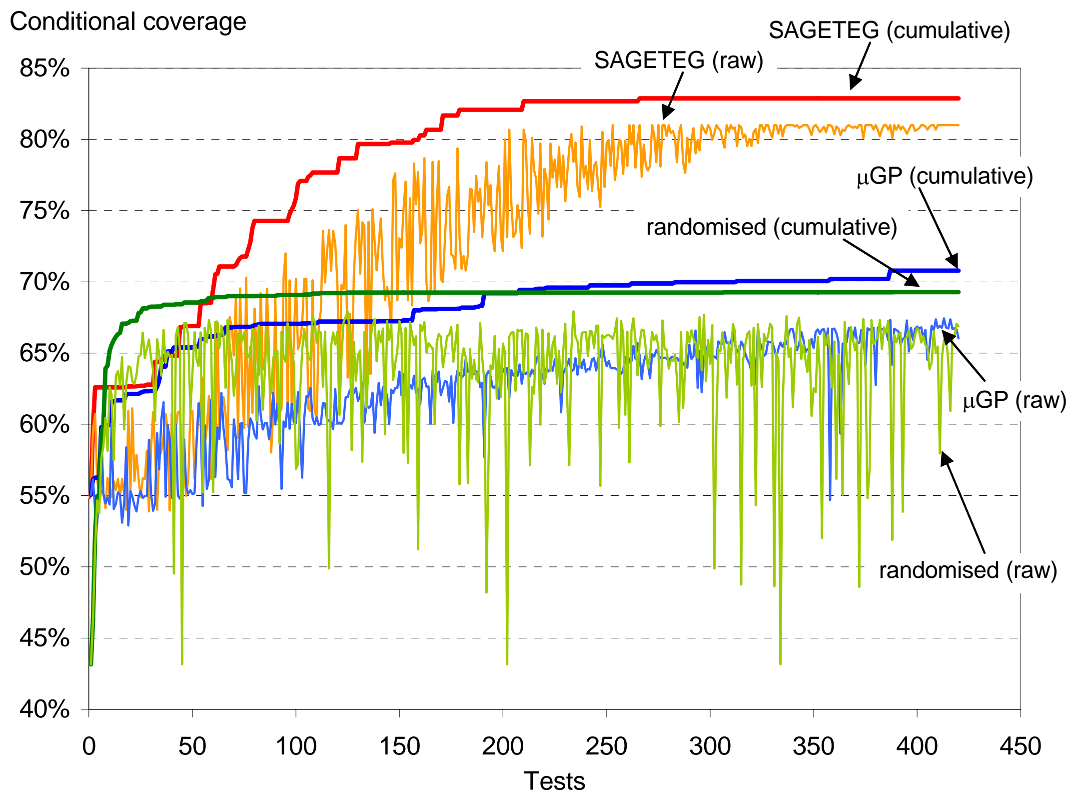


Figure 4.14 Conditional coverage process versus tests

Significant results from coverage progress graphs

The graphs reveal a number of key results. First, after initial tests, SAGETEG consistently outperforms other test methods by achieving a higher line, toggle and conditional coverage throughout the remainder of all test generations. This can be attributed to SAGETEG's superior GEA process and revised self-adaptation method.

Second, SAGETEG's GEA process is able to optimise and create a higher coverage attaining test suite than μGP. SAGETEG's GEA test generation technique is better suited for SALVEM, and this can be observed from the raw coverage data graph lines. Third, SAGETEG's coverage attainment rate is lower, even though it achieves much higher coverage eventually.

The evidence for these results arises from various observations of the coverage progress graphs. Specifically, the coverage attainment and progress of all tests for every coverage metric during the entire test generations of each method was examined closely. We studied the test coverage results of initial, middle, and final stages of test generations from cumulative coverage; and also, the GEA characteristics revealed by oscillating raw coverage progress graphs. The following subsections elaborate further our observations explaining the above key results extracted from these coverage progress graphs.

Overall coverage progression and initial test generations observations

Overall, SAGETEG's coverage graphs surpass those of other test generation methods to achieve higher final coverage percentages. Using a well-trained GEA process, SAGETEG's guided random search and coverage driven test generations produces coverage graph lines that attain higher coverage no later than 100 to 150 tests into the test generation process. Additionally, SAGETEG's coverage continues to grow beyond 150 tests unlike randomised or μGP test coverage, which are saturating already.

The differences in initial coverage between test methods are due to the different snippets employed in initial tests, and also the different test program infrastructural code that is present in an empty snippet test. Various infrastructural code are needed to support the snippets composed test programs differently depending on the test generation method employed. The GEA test generations employs careful selection of snippets in its tests, and invokes the evolutionary process immediately after initial

populations is created. Hence, coverage attained from GEA processes is equivalent or higher than random method, even at the beginning.

Despite equivalent (or slightly lower) initial coverage, SATEGEG is still able to exceed coverage of other methods eventually. Other test generation methods suffer from coverage saturation earlier on. Hence, they do not exploit the opportunities available to enhance coverage significantly. Even though the coverage growth rate of SAGETEG is slower, its GEA process is able to properly explore and cover the test space more effectively; resulting in more thoroughly verified SoC. After initial tests, SAGETEG's coverage is consistently higher than other methods. The coverage progress and results, and GEA characteristics after initial test generations are discussed next.

GEA characteristics and observations from mid to final test generations

The graph lines displayed in Figure 4.12 to Figure 4.14 are all characteristic of the randomised or GEA test generation processes. The randomised raw coverage lines suffer from extensive spikes, both upper and lower, throughout test generation because tests are always created randomly; they bear no relation with each other. For the GEA processes, rapidly oscillating spikes are also present, especially at beginning and up till middle stages of test generation. These graph line are typical of GEA. At the start of the evolutionary process, variation is extensively applied in order to explore larger and different regions of test space, to try to generate high coverage yielding tests. Such extensive exploration of the test space may occasionally lead to an ineffective or redundant test region causing lower coverage. This causes the large spikes of peaks and troughs in the raw coverage lines.

As the evolutionary process continues, variation is reduced to fine tune the test population and search within a smaller area of the local coverage space. Hence, the coverage spikes reduces. Furthermore, after every evolution, only the best tests are retained. Low coverage tests are slowly eliminated from the population and the graph line increases gradually. The practice of using larger variation in the beginning before eventual reduction is consistent with that of Rechenberg's rule and our self-adaptation strategy for GEA methods. In contrast, during random-only test generation, there exists no mechanism to reduce low coverage tests and maintain a population of high coverage tests. Any previous tests, regardless of their effectiveness are ignored and new random tests are continually generated. Lower coverage tests will likely continue to remain in the test population reducing the overall coverage.

Finally, near the end of evolution process, the GEA coverage line stabilises to signify the best possible coverage has been achieved. This indicates the GEA test population has fully evolved and self-adapted to the local coverage optimum. In order to enhance coverage further, new GEA processes with new initial populations are required.

GEA characteristics and performance between SAGETEG and μGP

Comparing the raw coverage GEA graphs of SAGETEG and μGP, we observe the following. For μGP, some large coverage variation spikes still occur at the end of its evolutionary test process, especially for conditional coverage in Figure 4.14. This suggests the GEA process has not achieved the optimal coverage goal and is still attempting (in vain) to explore other test space without gaining any coverage improvements. The outcome is low coverage spikes, much lower than current coverage levels. In contrast, SAGETEG's GEA process which employs refined self-adaptation show decreasing and hardly any coverage variation spikes by the end of test generation.

This demonstrates our self-adaptive method has been effective in fine-tuning, to attain the best possible coverage and stabilise the GEA process by the end of test generation. The majority of SAGETEG's raw coverage variation spikes are at the early to middle stages of test generation. Along with our self-adaptation method, this indicates our snippet variation dependency analysis and subsequent measures were effective in promoting population diversity. SAGETEG's test generation does not stagnate, some coverage variation are still evident at the half way stage indicating active exploration of other test space is ongoing. Unlike μGP, the source of SAGETEG's raw coverage spikes actually exposes previously uncovered test space, and thus some increase in raw and cumulative coverage is achieved. μGP's raw coverage variation does not enhance coverage at all. In fact, SAGETEG's decreasing raw coverage variation shows our self-adaptation is correctly maintaining desired one in five success ratio. This facilitates balance between test exploration and fine-tunning of existing test space uncovered, in order for higher likelihood of best optimised coverage at the end of testing.

By the end of GEA test generation, if significant raw coverage variations are still occurring, this implies either (i) longer test generation process is required and further test generate evolutions are needed to stabilise the coverage line and attain the coverage optimum, or (ii) the GEA process's self-adaptation is not effective for the application domain; as is the case in μGP being applied for SALVEM verification.

Observations and reasons for slower coverage progression by SAGETEG

Despite more effective self-adaptation, the slower coverage progression in SAGETEG can be attributed to operations and associated overheads conducted for self-adaptation. During initial stages of SAGETEG's GEA process, self-adaptation triggers more extensive exploration of test space. This exposes wider range of different SoC functions for testing. In subsequent evolutions, many variations of testing within the domain of these SoC functions can then be performed; as the GEA process fine-tunes exploration of SoC functions within these test space. The larger expanse of test space from initial evolutions steers the GEA process to eventually achieve best optimised coverage.

However, the greater variation to uncover this wide-ranging test space also increases likelihood of other test regions that only provide low coverage. Hence, there are greater coverage variation spikes which reduce the rate of coverage progress. Despite this, the trade-off is that more extensive SoC functions and test regions are uncovered upfront, which enable SAGETEG's coverage to surpass those of other methods in the long run. Unlike other test generations that saturates earlier, SAGETEG's coverage progression continues to improve up to and beyond middle stages of evolutions; albeit with decreasing rates of coverage improvements, but achieves coverage enhancements nonetheless.

Coverage progress measured against number of snippets

Note that despite certain disparity between tests and snippets from different test generation methods (described in previous section), the overall coverage process trends and key results are in fact similar when coverage is graphed against snippets. Appendix E.13 shows plots of these equivalent coverage graphs against snippets.

4.11.4 Experimental time and test memory usage

Time results

Our experiments also examined measurements of time for each test generation method. Table 4.4 tabulates the results of combined test generation and execution times of each method. At a glance, a mixture of varying time results are reported for the test generation methods. This is primarily due to various complexity and requirements of different coverage metric verification runs, and the tests and snippets employed by each test generation method. However, a number of observations can be made.

Table 4.4 Time results in CPU seconds

Line coverage	**SAGETEG**	**μGP**	**Randomised**	**Manual application**
Total	233,780 (2days 17hrs 56mins)	261,234 (3days 1hr 24mins)	199,235 (2days 7hrs 21mins)	4,317 (1hr 12mins)
per test	389	110	306	N/A
per snippet	14.9	3.6	6.3	N/A
Toggle coverage	**SAGETEG**	**μGP**	**Randomised**	**Manual application**
Total	324,667 (3days 19hrs 21mins)	454,897 (5days 6hrs 22mins)	239,003 (2days 18hrs 23mins)	6,226 (1hr 44mins)
per test	498	199	319	N/A
per snippet	11.6	2.9	6.9	N/A
Conditional coverage	**SAGETEG**	**μGP**	**Randomised**	**Manual application**
Total	201,918 (2days 8hrs 17mins)	249,611 (2days 21hrs 20mins)	196,389 (2days 6hrs 33mins)	13,757 (2hrs 49mins)
per test	480	229	468	N/A
per snippet	30.8	9.8	10.5	N/A

Note : For manual application created tests, total times refer to test execution only. Per test and per snippet times are not applicable because these tests are manually created without employing any snippets.

In general, the time needed for toggle and conditional coverage are longer compared with line coverage because of their higher measuring requirements and processing complexities. Conditional coverage should exhibit greater total time durations than toggle, but because conditional coverage is much more complex and requires overly long run-time durations, our test generations for conditional coverage were limited to much shorter run lengths.

As expected, regardless of coverage metric, manual application testing takes up the least time because we created and executed only a handful of tests compared to the automated techniques. However, note that a total of approximately 14 days of a single engineer's effort was required to create these 37 application tests manually; which underlines the impracticality of relying on manual application testing as the sole verification strategy.

Comparing automated test generations, across the three coverage metrics, μGP took the longest duration. This is because μGP generated and executed the most number of tests and snippets. Despite employing less tests and snippets, SAGETEG's overall test time was greater than that from randomisation. This can be attributed to the additional GEA operations that are conducted by SAGETEG; whereas randomised test generations involve much less complicated processes. The

effects of SAGETEG's GEA operations on generation and execution times are elaborated further below when discussing timing results with respect to the number of tests and snippets employed.

To put our timing results into proper context, one must consider the number of tests and snippets generated and executed for each test generation and verification run. In Table 4.4, beneath the total time results, we report the time taken per test and per snippet for all test generation methods and coverage metrics. By examining time results against number of tests and snippets, the resultant times are considered normalised to a baseline from which comparisons of test generation methods can be evaluated objectively. For instance, even though μGP took up the greatest total duration, its time per test and per snippet is in fact the lowest. This is because μGP's tests are assembler instructions based, and running the hand-crafted snippet mapped macros (Appendix E.2) is much faster. Therefore, it was possible to conduct many evolutions of test generations and executions even though its coverage was still lower and could not surpass SAGETEG's.

In contrast, the snippet functions and test programs used for SAGETEG and randomised test generation are properly encapsulated as ANSI-C programs, which required the cross-platform Nios compile tool-chain. Despite using optimisation switches, the Nios compiler's optimisations were still deficient. The Nios compile tool-chain package was not a fully certified release version but a trial beta version. Hence, the optimisation functions in the compiler do not always provide the desired code speed-up. Therefore, SAGETEG and randomised test program execution efficiency cannot match that of μGP.

Comparing the time per test and per snippets between SAGETEG and randomised testing, randomised tests use up less time. This is somewhat expected considering the more complex GEA process conducted by SAGETEG; whereas randomisation is not driven explicitly by prior tests or other influences. All random tests are independent of each other. For example, besides typical GEA variation or population selection operations, SAGETEG also performs more effective but more complex self-adaptation operations. Our self-adaptation examines greater test information, taking into consideration relative proximity of success ratios of each variation to the desired ratio; and adjusts the variation weights by differing amounts based on various criteria checked. Snippet variation dependencies effects are also managed during test generations (Section 4.5.8).

Note that SAGETEG's additional GEA operations are also slightly more complex than that of μGP. Hence, besides assembler instruction based tests that provides faster execution, the lower time results of μGP is also attributed to its simplistic (but less effective) GEA operations (such as self-adaptation) compared to SAGETEG. Appendix E.14 describes the effects of SAGETEG's GEA operations on its

test efficiency in greater detail. Regardless, the additional time requirement from SAGETEG's GEA process is an insignificant trade-off when considering the additional coverage gained.

Test generation times

Next, we consider test generation times independently to discount the effects from process variations during test executions of many tests or snippets. Compared to test execution times, actual test creation times from test generation methods are negligible. The main overhead of the verification runs were simulation times for test executions. Taking the average test generation and compilation times for all coverage metric (line, toggle and conditional) verification runs, the average times for SAGETEG, μGP, and randomisation are 4.5, 11.4, and 15.7 CPU seconds respectively to create a test. These times reflect the duration required to compose the snippets based test program and transform it into a test binary image suitable for immediate execution on the SoC.

SAGETEG test creation time is quickest over μGP because it does not suffer from the overhead of the snippets to assembler instructions macros mapping and generation process (Appendix E.2). SAGETEG on the other hand, was developed from scratch specifically for SALVEM verification. Its design and development was tuned to perform fastest on the SALVEM platform. It only relies on the Nios SoC compile tool-chain to create the test execution binary. Randomised test generation time was the slowest primarily because of its implementation technique. For the ease of implementation and prototyping, the randomised test generation was developed quickly using an interpretive programming script language. Its performance cannot match that of SAGETEG (or μGP), which is developed as a dedicated executable in ANSI-C/C++.

Test memory usage results

Experimental measurements were also recorded for test program sizes from each test generation method. Specifically, we examine the amount of memory and percentage of test program allocation taken up by test programs, out of the total SoC executable memory available. On average, across the three coverage driven verifications, the test program memory usages are 71Kbytes, 29Kbytes, and 63Kbytes for SAGETEG, μGP, and randomised testing respectively. The corresponding test program percentages of memory usage are 27%, 11%, and 24%.

SAGETEG uses up the largest amount of memory whilst μGP requires the least. The least memory usage by μGP can be attributed to its assembler instructions based testing, which require lower memory as instructions are hand-optimised to mimic snippets. There is no overhead from ANSI-C based program level functionalities (e.g. function calls and returns, and push or pop of stack parameters). Such overheads are only present in SAGETEG and randomised test programs because they are composed of proper ANSI-C function-style snippets and compiled as needed.

SAGETEG employs the greatest memory because it has additional overheads from managing and chaining snippets together in a GEA manner. It also requires start-up (prologue) and clean-up (epilogue) code within its test programs to cater for GEA snippets sequences. Also, other SAGETEG specific test program features such as the external snippet function based testing capabilities (Appendix E.8) take up further memory. But regardless of the time and memory usage size requirements, compared with other methods, SAGETEG still achieve superior coverage results – which is the primary goal of SALVEM verification and the work in this chapter. Higher coverage enhances greater likelihood of bug detection and contributes directly to the quality of a hardware design. Faster generation or run times, or shorter tests do not certify correctness of a design directly.

Effectiveness of SAGETEG

Appendix E.15 considers the effectiveness of the test generation methods using an effectiveness factor evaluated from coverage, time and test size results. The effectiveness factor supports the notion that SAGETEG performs most effectively to achieve best coverage given trade-offs in time requirements and memory size usage

4.11.5 Summary and concluding remarks of experimentations

These experiments demonstrated the ability of SAGETEG's GEA processing and generated test programs to explore larger untested test regions on the target SoC. This translates to higher test coverage, greater likelihood of uncovering design bugs, and higher quality of SoC verification. Randomised test generation is not properly guided in any way, and is simply an automated technique for creating many tests quickly. μGP was never intended to generate system level tests for SoCs, hence suffers from a number of shortcomings that inhibits its capability to verify a complex hardware design system.

Our GEA technique and results also emphasise a key point. Larger and longer tests do not always return corresponding gains in coverage. With feedback and GEA optimisation, we can optimise coverage gain using shorter tests and resources.

To this end, SAGETEG is the most effective and efficient test generation method. It facilitates proper coverage feedback verification for SALVEM. SAGETEG is a dedicated GEA test generator for SALVEM, with the sole purpose of creating snippets based SoC test programs. Unlike μGP, it overcomes many of the limitations of previous methods and avoids the unnecessary adaptations needed to map a test generation method for SALVEM. SAGETEG is more compatible with characteristics of our snippets test building blocks. It provides specialised GEA functions and options specifically catered for various types of snippets test programs.

As for the remaining coverage still to be achieved, these untested SoC functions are a direct consequence of deficiencies in the snippet library itself. SAGETEG can only make use of the snippets test building blocks made available to invoke SoC interacts in its most effective and efficient manner. This implies further opportunities for enhancements in our snippets library, and eventually refining SAGETEG for these snippets. However, this is beyond the scope of the intended research in this chapter.

4.12 Verifying the Tsinghua University digital signal processor system-on-chip (THUDSP2004) – A case study

4.12.1 Overview of verification case study

To provide further evidence of SAGETEG's effectiveness for SALVEM SoC verification, we applied SAGETEG and associated SALVEM techniques to verify a digital signal processing (DSP) SoC, the THUDSP2004 SoC. This verification work was part of the author's research exchange visit at the Tsinghua University's Institute of Microelectronics in Beijing. The THUDDSP2004 SoC is more akin to an industrial-like SoC, and would greatly support SAGETEG's effectiveness as a coverage driven test generator.

A DSP SoC can be designed with a variety of architectures and techniques, and employed for a range of applications and end-products. Hence, the eventual real-life usages, application design features, and

functions must be tested in-depth, which makes SALVEM and SAGETEG test generation an ideal verification strategy.

The THUDSP2004 was designed specifically for multi-media applications and portable hand-held products. It contains common DSP function blocks such as high performance mathematical and fast data transfer units, along with other specialised modules. Architecturally, the SoC consists of a very-long-instruction-word processor, memories, interrupt and memory controllers, and I/O modules like the DMA for transferring large signal data. Figure E.17 in Appendix E.16 shows the SoC layout. The THUDSP2004 is also highly configurable, employing dynamically changing clusters and function units, depending on the intended usage of the end-product it is embedded in. Therefore, a unique register file based inter-cluster bus system is implemented to facilitate communication and data transfers between differing configurations of function units. Other specialised arithmetic logic, multiply, address branch, loop control, and load/store units also exists to serve DSP operations.

These DSP SoC architectural, design, and application features are to be tested by SAGETEG derived SALVEM tests, to further enhance the design and verification quality of the DSP SoC. For this reason, new set of snippets and a different snippet library was devised specifically for the THUDSP2004 SoC. For example, snippets were created to trigger mathematical and arithmetic processing capabilities of the SoC, such as Fourier and cosine transforms, or other filter functions. For the inter-cluster register file system, our snippets test this communicational bus system with repetitive data transfers and numerous register address resolutions. Specifically, we use token ring transfer based snippets to pass control to many function units and invoke various transfer scenarios. Other hardware design based snippets were also implemented such as interrupt testing snippets or snippets to verify DSP signal data transfers with DMAs. For the DMA, the snippets were derived directly from the Nios SoC snippets library; which fulfils one of the key principles in SALVEM verification – to re-use as many snippets from snippet libraries for a range of SoC verifications.

The GEA process employed by SAGETEG for the THUDSP2004 is no different to that for the Nios SoC in previous sections. In fact, SAGETEG and the SALVEM platform were re-used as is again. The differences are simply the SoC under test and the snippets library employed to compose test programs.

Appendix E.16 and our paper in [CLS+08] provide full details of our work to verify the THUDSP2004 SoC using SAGETEG, including additional background and motivation driving this verification. In addition, specification of the DSP SoC, and further descriptions of the snippets library developed for the DSP are described. For this case study, to support SAGETEG's viability as a test generator, we focus on the THUDSP2004's verification experiments and results next.

4.12.2 Case study experiments and results

The goal of experimentation is to demonstrate feasibility and effectiveness of SALVEM GEA driven test generation on the THUASDSP2004 DSP SoC. Like the Nios SoC, we conducted simulation and coverage measurement of test programs using Synopsys VCS on a register transfer level (RTL) Verilog description of the DSP SoC design. All experiments were run on a Linux operating system powered by a 3.2 GHz AMD CPU and 4GB RAM.

For preliminary experimentation purposes, SAGETEG used the basic DSP snippets library that includes snippets from Appendix E.16. SAGETEG was configured with μ and λ population sizes of 10 and 20 respectively. Preliminary experiments were conducted again for the DSP SoC to select appropriate parameters. Specifically, given the different SoC under test and snippets library, new test generation parameters were required to ensure appropriate tests and generation process within SoC and test resource limits were realised. Our paper [CLS+08] provides full details of parameter selections.

After SAGETEG calibration, three separate test generation runs were conducted targeting line, toggle and conditional coverage as the fitness evaluator. The duration of each test run was about 50 evolutions. In total, 707, 751, and 785 tests were generated respectively for the line, toggle and conditional coverage test runs.

For comparison, in addition to GEA tests, a random test generation process was also conducted. Similar to Nios SoC experiments, under this scheme, all test creation decisions are random without any influence from previous coverage or other test information. The random approach created an equivalent number of tests as GEA for each of the line, toggle and conditional coverage test runs. By that stage, coverage levels were already maximised with further improvement unlikely. Note that comparisons against μGP were not conducted because of the additional effort and limited time (of the research exchange visit) to map snippets to μGP assembler instruction macros. The randomised test generation comparisons should suffice as extensive comparisons with SAGETEG were conducted for the Nios SoC previously.

The accumulated coverage results and total number of snippets executed from GEA and random tests are shown in Table 4.5. For GEA, the results represent the best coverage achieved when the test suite has evolved to an optimised state.

Table 4.5 THUASDSP2004 SoC coverage and snippets usage results

Test generation method	Coverage %			Number of snippets		
	Line	Toggle	Conditional	Line	Toggle	Conditional
SAGETEG	91.3	86.7	80.2	30,400	28,100	42,300
Randomised	83.0	78.1	70.4	56,800	60,100	62,800

In the random approach, the number of snippets in each test was random, as long as the test size did not exceed SoC memory limits. In contrast, GEA tests contain smaller number of snippets, relying on the evolutionary process to cultivate them into larger test individuals over time. This provides more efficient usage and lower test sizes overall.

For all coverage measures, the GEA approach attained better results compared to random tests. Despite more snippets, the random approach could not match the coverage from GEA. The snippets sequences evolved under the GEA method was more effective compared to randomly combining snippets together.

For comparison purposes in Table 4.5, we used number of snippets instead of tests because the test sizes between each GEA test process and the random approach differ. The total number of snippets (and tests) created by GEA varies depending on the variation conducted during the GEA process. In variation, various snippets will be added, removed or replaced. Therefore, our tests contain different number of snippets, and one GEA test is not equivalent to one random test. If recombination is used, multiple new tests with new sizes are created each time, and each evolution may end up with more than λ new tests.

Test generation times between the GEA and random approach were similar and negligible compared to test simulation times. Including other overhead, the GEA approach required approximately 5 days for the three test runs compared to 8 days for random tests. This was expected given the larger number of snippets executed under the random approach.

The application of SAGETEG test generation for the DSP required 3 months of a single engineer's effort. The main effort was the development of snippets. However, once implemented, the snippets can be used by the test generator to automatically create many tests to exercise various scenarios, reducing overall effort if test cases had to be created manually. As discussed previously, re-use benefits were already demonstrated by the re-application of DMA snippets from Nios SoC verification for

THUASDSP2004 testing. Therefore, there is high likelihood of re-use from the DSP SoC snippets library for other DSP SoC verifications.

Whilst we did not discover any new bugs, the goal was to demonstrate SAGETEG GEA test generation on a real-world SoC design. By doing so, the design quality and confidence in error-free SoC operation can be considered enhanced. Although full coverage was not attained, our preliminary experiments show that applying SAGETEG GEA technique is indeed feasible. By expanding our snippets library with more extensive snippets that perform other DSP filtering, transforms, or mathematical functions, and analysing for remaining dead code in the design, we are confident full coverage can be achieved with greater efficiency.

4.13 Conclusions

In this chapter, test program generation in SALVEM is enhanced using genetic evolutionary algorithms (GEA). The process of GEA test creation for the SALVEM platform was conceptualised and implemented as an automated test generation tool called SAGETEG.

The key to successful test generation by SAGETEG is our encoding of SALVEM test generation components and processes into suitable GEA representation, in order to conduct genetic evolutionary cycles. Additionally, effective verifications facilitated by our GEA derived tests can be attributed to in-depth snippet constraint and dependency GEA variation analysis, and the revised self-adaptation method. Our variation analysis along with recommendations to address their effects, apply not only to test generation snippets, but any GEA method that enforces constraints or dependency rules on the individual genome set. Our self-adaptation method is more effective because it was developed based on actual observations of SALVEM test generations. The method takes in greater test generation and execution data to better mange test variation operators more accurately as the GEA test creation process closes in toward desired test creation goals.

To demonstrate the viability and benefits of GEA test generations for SALVEM, SAGETEG was applied to verify the Nios SoC and a digital signal processing SoC as a case study. Compared to other test generation methods, SAGETEG's GEA test creation technique is superior in terms of coverage attained and test sizes required. Despite the effectiveness of SAGETEG, the GEA test creation procedure still relied on preliminary test generations and calibration runs to identify appropriate test generation parameters before actual SoC verifications. Such parameter selections can be detrimental and is inefficient given the empirical data upon which parameters are chosen. The next chapter describes an analytical approach to GEA test generation parameter selections.

CHAPTER 5. MARKOV MODELLING OF GENETIC EVOLUTIONARY TEST GENERATION FOR PARAMETER SELECTIONS

This chapter examines the selection of parameters for genetic evolutionary test generation. Making use of Markov chains to model sub-processes and components of the genetic evolutionary test generation process, appropriate values for test generation parameters can be analysed and deduced. Markov modelling various features of the test generation process for the purpose of parameter selection is unique. It reduces the preliminary test calibration and experimentation effort needed to select these parameters.

5.1 Introduction

The effectiveness of our genetic evolutionary algorithm (GEA) based test generation, and indeed any GEA process in general, is governed by the GEA parameters chosen for conducting evolutionary optimisation. Whilst the strength of GEA is its applicability for a wide range of problems, the downside is that a set of best parameters selected for one application area can seldom be reused for other GEA problem areas. For example, parameter selections for optimising non-linear mathematical functions using genetic algorithms have been thoroughly studied [Fog00, Fog94, Mic96], however these methods are not directly applicable to other GEA problem domains.

Previously in Chapter 4, our parameter selection process involved conducting preliminary test runs to acquire empirically derived parameter values, and iteratively refining these parameters to provide favourable test generation outcomes. Whilst this approach facilitates immediate verifications to be conducted, it does not necessary give the best parameters possible, and can be rather time consuming, wasting valuable verification resources.

Determining the best parameter values for different GEA processes can often be ad-hoc and ill-defined. No analytical approach exists for identifying and selecting appropriate parameter values. In practice, for many GEA applications such as test generation, parameters are usually chosen and refined based on empirical results after preliminary GEA process runs have been conducted [CCRS02b, CCRS02c, CCRS03a, CCRS03b].

The GEA parameter selection gap shall be addressed in this chapter. We describe an analytical based parameter selection approach. Modelling components of our GEA test generation process as Markov chains, and by analysing characteristics of specific interest in our GEA process based on their Markov behaviour, our parameter selection guides parameter values to be chosen more efficiently for specific needs in our test generation. The need to conduct extensive preliminary test generation runs to calibrate the test generator before actual verifications is also eliminated.

The use of Markov chains for modelling a GEA process is highly suitable because a GEA flow contains many sub-processes with independent states. Markov chains can be used to capture any of these combinations of sub-processes and their varying number of changing states. Markov models have been previously employed to model GEA processes, for simple analysis and extracting basic information [Fog00, MH94, NV92]. For example, Mao and Hu [MH94] modelled an evolution process used for physical design tools with Ergodic Markov chains [GS06] in order to study convergence properties. However, the use of Markov chain to model GEA process can be problematic because most application domain requires too large a Markov chain to be able to successfully model the entire GEA flow.

For example, in his textbook [Fog00], Fogel modelled a GEA process using Markov chains. His aim was to deduce the number and density of absorbing states for absorbing Markov chains of an artificial GEA application domain. However, the size and complexity of real-life GEA processes causes too many states in the Markov chain. For instance, a practical GEA application such as the test generation in Chapter 4 holds 30 possible snippet genes, test individuals sizes up to 125 snippets, and an average population size of up to 30. This would require $30^{(125+30)}$ number of states, which is impossible to model. In his example, Fogel could only implement his Markov chain for a simple GEA example; with a binary bit genome (i.e., each gene is 0 or 1), and both individual and population size of 2 only. This is equivalent to just $2^{(2+2)}$ states. Similarly, Nix and Vose [NV92] modelled a genetic algorithm but had to employ fixed length binary strings for containing the number of states to be able to make use of the Markov model.

For a GEA process, due to size issues with Markov chain modelling, the subsequent Markov analysis is often difficult and the range of useful information that can be extracted from the Markov chain is limited. In addition, employing Markov chains to analytically examine GEA parameters and resolve test generation parameter values, to the best of our knowledge, has never been attempted before. We elaborate on the test generation parameter selection problem next, before outlining our solutions to address this problem.

5.2 Problem formulation and description

System for examination

The GEA parameter selection strategy devised in this chapter can be applied to the test generation process in Chapter 4. The process is summarised in Figure 5.1. The GEA process is an iterative cycle whereby tests are successively created based on components of previous tests in the preceding cycle. The test generation is a refining process reliant on parameters that control and influence the characteristics and outcome of the test creations.

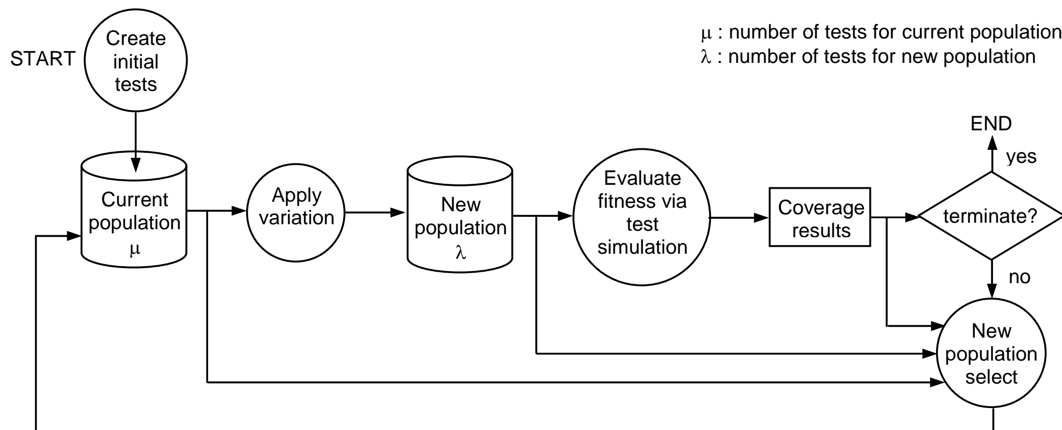


Figure 5.1 GEA test generation process.

Problem goal and requirements

Given this GEA process, the problem goal is to find suitable parameter values that will facilitate the most effective and efficient test generations. The effectiveness of the test generation is measured by performance characteristics such as test population diversity, and ultimately test coverage and size results.

The test generation parameters of interest are the number of evolutions, initial variation and snippet selection probabilities, probability change factor according to GEA self-adaptation, parent and children population sizes, and number of previous consecutive evolutions to examine if GEA termination should be triggered.

Values for these parameters must be identified and chosen using analytical methods which are based on Markov models. The analysis procedure must be repeatable and reusable for different test generations and a range of parameters. The technique must also be more efficient than empirical

methods. Parameter values uncovered by analysis should not take longer than conducting preliminary GEA runs to calibrate for the same parameters.

The parameter values from our analysis technique should be accurate within acceptable tolerance. The numerical values must be comparable and sufficiently close to those uncovered by equivalent empirical results and also, the eventual parameter values used for actual verifications. This requirement is subjective and certain error margins may be accepted. In most cases, the intention of selecting parameter values analytically using Markov models is to provide an initial assessment of the range of domain values deemed appropriate for invoking GEA test generations and verifications.

Appropriately derived parameter values for test generations imply design verifications that should provide test results greater than 80% coverage and non excessive simulation times. Without conducting any prior preliminary calibrations but using analytical parameter values, test generations for verifying an SoC can be conducted immediately and with greater confidence of more thoroughly verified design. With analytically derived parameters, the test generation will be performed according to desired test creation goals and demonstrate characteristics required for effective verifications. For example, maintaining higher variety of tests or applying appropriate types of GEA variation throughout the test generation.

Constraints and limitations

Analytical parameter selections are restricted by the extent of Markov chain analysis that can be undertaken. Specifically, the size of the Markov chain cannot be too large, otherwise practical analysis outcomes within acceptable timeframes is not possible.

In Markov chain parameter selections, there may be certain factors that could not be modelled or taken into consideration. These unaccounted factors could arise due to limitations for containing the Markov chain modelling sizes; otherwise the Markov chain model would be too large to reveal any useful information. Factors that may not be taken into account include external considerations that were unknown or likely to change after the parameter selection analysis stage. For example, available compute resource capabilities for conducting test generation and simulations can vary, and would affect the manner in which test generations are conducted. The number of test generate evolutions that can be run is a parameter that is affected by such un-modelled variations.

By discounting certain considerations from the Markov chain analysis, assumptions are enforced instead. Therefore, the analytical parameter selections need not be strictly applied. They act as a guide to the approximate range of values parameters should take up, depending on the soundness of various assumptions and their likelihood for change. After early test generations, it would not be unexpected if the Markov chain derived parameter selections were refined slightly if further test generation improvement was obvious. In this sense, the analytical Markov chain parameter selections and their subsequent refinements from actual test generations form a closed loop procedure; so that the analysis can be repeated to further improve parameter selections if needed.

The possibility for refinements of analytically derived parameters is the main limitation of the method. This is unavoidable whenever certain GEA or verification factors cannot be encoded into our analysis, and other assumptions are required to ensure Markov chain models are manageable. To address this shortcoming, the GEA analysis and parameter selections are broken down into smaller individual sub-problems; each of which are tackled by the same Markov analytical approach, but each focusing on different components of GEA test generations for their smaller group of parameters. The segregation of the parameter selection problem into sub-problems is elaborated further by our parameter selection analytical approach described in the next section.

5.3 Markov modelling analysis for parameter selections

We tackle the parameter selection problem by employing Markov chain modelling as the analytical technique. The inspiration for modelling GEA test generation with Markov chains arises from a key observation. The next state of the newly created children test population is solely dependent on the previous parent test population immediately prior to the current GEA cycle. This was demonstrated by Figure 5.1 where the new population of tests depends on GEA variation (and indirectly, prior test fitness and selections) applied to the test population from the previous evolutionary cycle. By satisfying such Markov *previous state dependency only* criteria, the GEA process and its sub-processes can be analysed directly by Markov chains and their solutions.

Even if the Markov criteria is satisfied, modelling the entire GEA test generation with a Markov chain is not plausible. Markov chain size containment is needed to allow for successful analysis. Rather than restrict one Markov chain for representing the entire GEA flow and discount certain GEA components, processes and states, multiple smaller Markov chains are used. We employ a divide-n-conquer strategy to identify smaller sub-components and processes of the overall GEA flow that display Markov

characteristics and are of interest to us, and we model these GEA sub-flows using individual and appropriately sized Markov chains instead.

The GEA sub-components and associated Markov chains that are of interest are those that are affected by specific GEA test generation parameters of our choosing. The GEA sub-processes modelled by our Markov chains are designed such that the transition probabilities of the Markov chain are encoded directly with variables representing test generation parameters. Based on the type of GEA sub-process and analysis of Markov chain characteristics, we can deduce appropriate parameter values such that desired outcomes and behaviours of test generation are achieved with greater likelihood during actual verifications. For example, in our Markov chains, certain parameters can affect the diversity and injection of new tests into test populations during evolutions. By analysing specially derived Markov chains, the transition probabilities and associated parameters to enhance diversity and new tests can be deduced.

Figure 5.2 illustrates our approach in employing Markov chains for parameter selections. Besides modelling GEA with Markov chains, the other key element in our analytical parameter selection strategy is to break down test generation parameter selections into sub-problems, and encode parameters or other test variables into the Markov states or state transition probabilities explicitly. The modelling of GEA sub elements and encoding of parameters are essential ingredients for the design of each Markov chain.

The flow in Figure 5.2 provides a consistent approach which is to be applied and reused for every Markov chain that models a sub GEA component process. Each of the sub-divided Markov chain focus on a specific subset of parameters only, and are analysed to attain values for those parameters. The information from one Markov chain analysis may also be reused for derivation and analysis of other subsequent Markov chains, and other parameter selections from those Markov chains. Whilst each Markov chain is individually designed, collectively, these sub-divided Markov chains come together to model important facets of the overall GEA test generation. The progression of individual but inter-dependant Markov chain analysis facilitates selections of greater set of test generation parameters, which influence test generations most significantly.

We present four Markov chains to aid parameter selection guidance. The four Markov chains to model and examine are, (i) GEA self-adaptation characteristics, (ii) GEA variation probability weights transitions, (iii) composition of different types of snippets in tests, and (iv) the progression of test population diversity and compositions.

Each Markov chain addresses a specific and different part of the GEA test generation flow. We analyse our GEA test generation using these Markov chains and determine how GEA parameters affect the evolutionary process. The analysis enables identification of suitable values for parameters associated from each of these four GEA sub-areas. These Markov chains act as sub-problems that make up the overall parameter selection problem goal.

Studying various Markov behaviours to identify how the associated test generation process can be steered toward desired outcomes, appropriate parameter values can be deduced that facilitates greater likelihood of achieving effective test generations. No restrictions are placed on the types and number of sub Markov chains derived and analysed. For GEA test generations in this chapter, the four Markov chains employed were sufficient for modelling relevant GEA parameters deemed essential for analytical selections. The following sections describe these Markov chains and the process of parameter selections in detail.

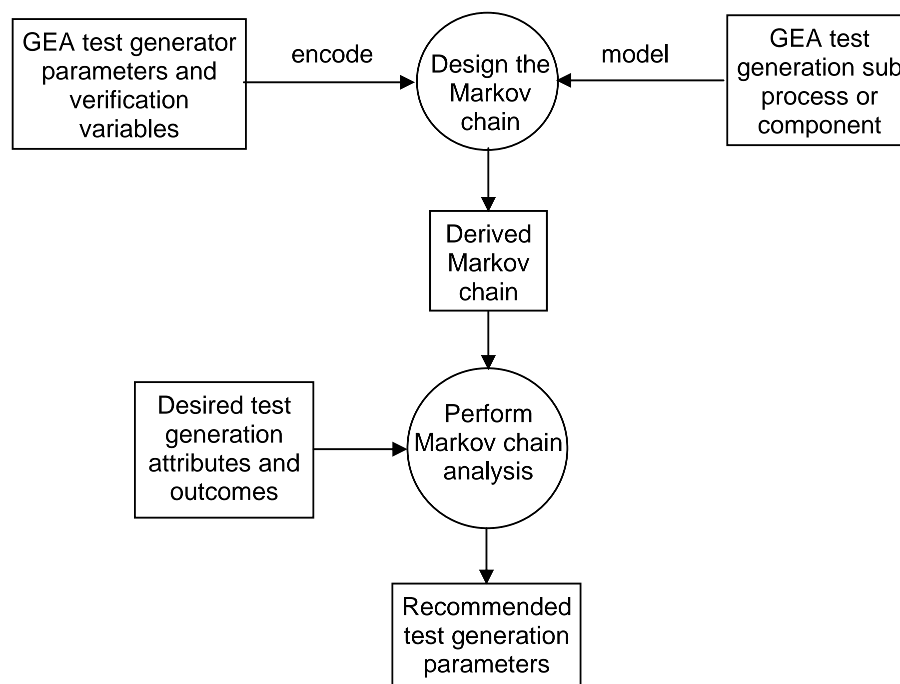


Figure 5.2 Test generation parameter selections based on Markov chain analysis

5.4 Modelling self-adaptation characteristics

The goal of GEA self-adaptation Markov modelling is to analyse how our GEA self-adaptation method adjusts GEA variation weight variables. Specifically, we examine the types and number of changes, and how variation variables are modified according to self-adaptation during the entire test generation

when modelled by the Markov chain. Based upon this Markov chain, the test generation parameter under examination for selection is the number of GEA test generate evolutions cycle.

A high-level overview of the self-adaptation method modelled by the Markov chain is summarised by Figure 5.3. The flow diagram components in Figure 5.3 is a sub-process of the components in Figure 5.1 which describes the overall GEA test generation flow.

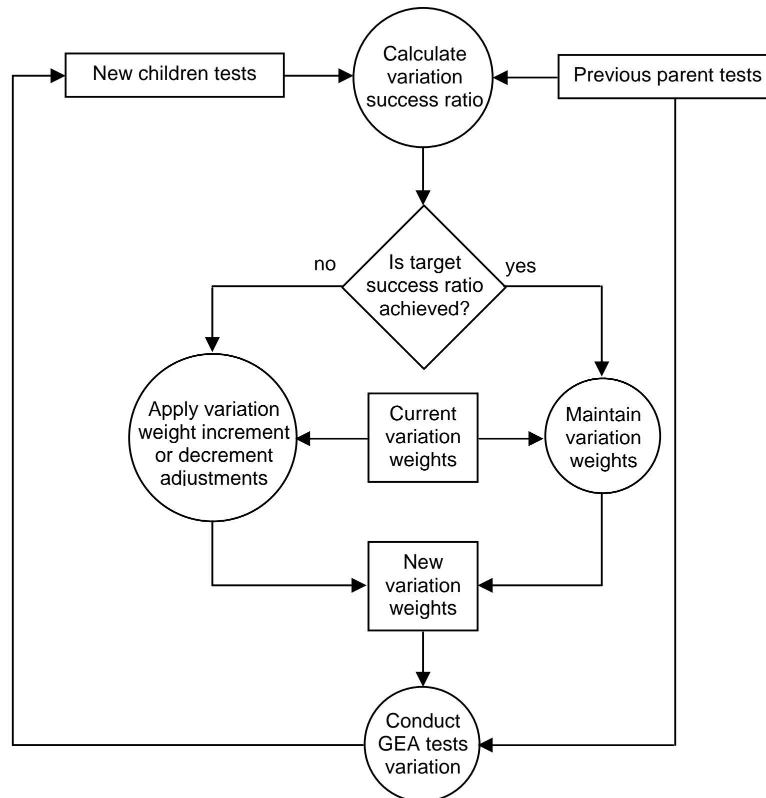


Figure 5.3 Self-adaptation flow for Markov chain modelling

The GEA self-adaptation adjusts GEA variation variables of usage after every evolution cycle. Variation operators create new tests differently based on previous tests. The variation weight variables control the usage of each of the variation operators. Variation weights are adjusted based upon their successes in creating tests. Success is measured by coverage fitness of the newly varied test compared to previous test. The goal of GEA self-adaptation is to adjust variation usage weights so that a target variation success ratio can be maintained. Unless the target success ratio has been achieved within acceptable error margins, increment or decrement adjustments will be applied to variation weights. Section 4.5.9 Chapter 4 described self-adaptation fully. Typically, a target success ratio of one fifth is desired. For Markov analysis, we are interested in how variation usage weights will be adjusted; specifically the types and progression of weight changes applied. Therefore, Markov states are arranged to correspond to the 'Apply variation weight increment or decrement adjustments' or 'Maintain variation weights' components in Figure 5.3.

5.4.1 The Markov model

The Markov chain is designed as follows. We define the set of states of the Markov chain to be {*A*, *D*, *S*}, where *A* represents the state when a GEA variation usage is increased, *D* is the state where variation usage decreased, and *S* infers desired variation target success ratio has been achieved. The variation success ratio is a measure of the amount of GEA variation being applied to create tests that are successful (i.e., achieved greater test coverage) compared to non-successfully varied tests. When the variation target success ratio is acquired, this implies the types of tests generated should be most effective (e.g. in terms of test coverage), and so the desired level of variations employed has been attained and should be maintained.

Given the variation change states *A*, *D* and *S*, for any GEA variation, we define $0 \leq \omega < 1$ to represent a variation weight variable that can increase, decrease, or does not change. We also observe an increasing or decreasing variation state of application is more likely to continue onto subsequent evolutions of the GEA process. Change of states between variation increase or decrease is less frequent, they only occur when the GEA process encounters conditions such as test saturation of certain snippet types. Additionally, when a variation attains its target variation success ratio, the GEA test generation has achieved optimality for the current GEA process.

To describe attainment of target success ratio and variation weight stability state, the Markov state is said to be absorbing if the probability of leaving that state is zero. The state *S* implies a state of stability whereby application of variation remains constant because variation success goal has been attained. *S* represents an absorbing state because the self-adaptation method is to maintain current levels of variation usage so that GEA variation usage remains indefinitely in the *S* state. The variation weight variable value is fixed.

We observe during the GEA process that it is difficult to attain the target test success ratio. This is usually achieved toward the end of GEA test generation. Also, even though we assume that a state of *S* with no adjustments to variation usage corresponds to target variation success ratio, in reality, a state of *S* could also infer that no variation adjustments could be applied because variation weights has reached minimum or maximum limits, and no adjustments are in fact allowed. This is discussed later in Section 5.5. For the remainder of this section, we assume ideally that *S* corresponds to success ratio goal attainment. We derive the Markov state diagram and state its transition probability matrix next.

The Markov state diagram and transition probability matrix

The Markov state diagram is shown in Figure 5.4.

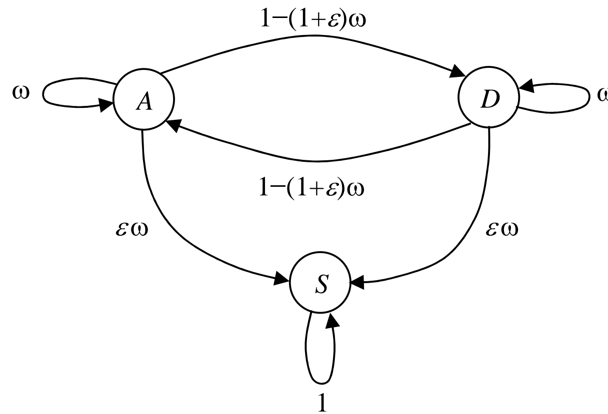


Figure 5.4 Self-adaptation characteristic Markov chain state diagram

The transition probabilities are deduced as follows. For transitioning between the same states A or D, the tendency for variation usage to continue increasing or decreasing is essentially the variation weight value. Hence, we assign these same-state transition probabilities to be ω.

The probability of entering the steady state S from previous increase or decrease state is always lower than the probability of continual increment and decrement, and we set this to be represented by $\varepsilon\omega$. The variable ε is intended to signify this transition probability to be a percentage fraction of ω; whereby $0 < \varepsilon < 0.5$ and is examined under further practical test parameter selection considerations in Section 5.4.3 later. In order for transitions exiting either A or D state (and the rows of the transition matrix (5.4.1) below) to sum up to a probability of one, the transitions between increasing and decreasing, and vice versa, are both $1-(1+\varepsilon)\omega$. Finally, the S state is absorptive so the transition probability to itself is 1, and other states 0.

The transition matrix P corresponding to the Markov chain in Figure 5.4 is:

$$P = \begin{matrix} \\ A \\ D \\ S \end{matrix} \begin{matrix} \begin{matrix} A & \qquad D & \qquad S \end{matrix} \\ \begin{pmatrix} \omega & 1-(1+\varepsilon)\omega & \varepsilon\omega \\ 1-(1+\varepsilon)\omega & \omega & \varepsilon\omega \\ 0 & 0 & 1 \end{pmatrix} \end{matrix} \tag{5.4.1}$$

The elements p_{ij} of matrix P describe the probability of transitioning from a current state s_i in the existing evolution to another state s_j in the next evolution, where $s \in \{A, D, S\}$.

The Markov chain in Figure 5.4 and its transition matrix P is time independent, the Markov model thus satisfies the homogenous Markov criteria. Modelling the GEA's self-adapting process with a Markov chain is appropriate because the next state of GEA variation usage changes depends only on the present state of test success-to-failure ratio from the population of the current evolution. The variation change applied is independent of earlier evolutionary states, and only examines existing tests (both parent and children tests) from the current evolution. Our Markov chain models these variation usage changes via states A, D and S, and the likelihood of transition from one state to another.

From this Markov model and transition probabilities, the effects from a range of self-adaptation changes and variation outcomes that may occur during GEA are captured. With given probabilities values, we can then deduce and select appropriate GEA parameters.

5.4.2 Markov chain analysis

In this section, we analyse the steady state long term behaviour of the Markov chain, and extract information regarding the number of evolutions parameter for the GEA test generation process.

Partitioning and mapping P to the canonical form of an absorbing Markov chain transition matrix [GS06], the transition matrix P is,

$$P = \begin{array}{c} \\ TR \\ ABS \end{array} \begin{array}{c} \begin{array}{cc} TR & ABS \end{array} \\ \begin{pmatrix} Q & R \\ 0 & \mathrm{I} \end{pmatrix} \end{array},$$

where TR and ABS denote the transient and absorbing states. The matrices,

$$Q = \begin{pmatrix} \omega & 1-(1+\varepsilon)\omega \\ 1-(1+\varepsilon)\omega & \omega \end{pmatrix},$$

$$R = \begin{pmatrix} \varepsilon\,\omega \\ \varepsilon\,\omega \end{pmatrix},$$

are derived, and I is the identify matrix. The matrix Q corresponds to the probabilities of transitioning within transient states, and R is the probabilities of transition from a transient state to an absorptive state.

Let n be the number of state transitions of the Markov chain during the GEA process. For steady state behaviour, we evaluate P^n as $n \rightarrow \infty$, and employ the Markov chain formulation from [GS06] stated as,

$$P^n = \begin{matrix} & TR & ABS \\ TR & Q^n & N_n R \\ ABS & 0 & \mathrm{I} \end{matrix}$$

where Q^n indicates the probabilities of transitioning within the transient self-adapting increase or decrease states after n transitions and before entering the absorbing state, and $N_n = I + Q + Q^2 + \ldots + Q^{n-1}$.

In an absorptive Markov chain such as (5.4.1), if $n \rightarrow \infty$, then $Q^n \rightarrow 0$ and N_n tends to $N = (\mathrm{I} - Q)^{-1}$ [GS06]. Thus, N is evaluated from,

$$N = \frac{1}{\omega(2\varepsilon + (1-(1+\varepsilon)^2)\omega)} \begin{pmatrix} 1-\omega & 1-(1+\varepsilon)\omega \\ 1-(1+\varepsilon)\omega & 1-\omega \end{pmatrix} \begin{matrix} A \\ D \end{matrix} \quad \text{(columns } A, D\text{)} \tag{5.4.2}$$

The fundamental matrix N reports the total expected number of times the Markov chain (and the GEA process) will operate in each transient A or D state. We interpret N as follows.

In (5.4.2), the first row and column of N corresponds to state A and the second row and column corresponds to D. For any state $s \in \{A, D, S\}$, the n_{ij} element of N is the expected number of times the GEA process is in state s_j if the process started in state s_i. For instance, n_{12} corresponds to the number of times the process is in D assuming its initial state was A.

Using the fundamental matrix, the time to absorption $t = \begin{pmatrix} t_1 \\ t_2 \end{pmatrix}$ is evaluated by summing the row elements of N. This is equivalent to multiplying N by a single column vector matrix of 1s, $t = N\begin{pmatrix} 1 \\ 1 \end{pmatrix}$.

5.4.3 Parameter selections with the Markov model

Based upon the Markov chain and results derived above, we analyse further taking into account practical considerations of GEA test generations. The GEA self-adaptation and variation characteristics shall be examined in order to identify suitable values for GEA test parameters that are affected by this Markov chain, and the GEA behaviours it models. Specifically, we seek to select the

number of GEA test generation evolutions parameter based on the Markov model under actual test considerations and observations.

Recall from Figure 5.4 that the probability transition of entering the steady state S from previous increase or decrease state (i.e., $A \rightarrow S$ or $D \rightarrow S$) is $\varepsilon\omega$, where $\varepsilon < 0.5$. Depending on ε, the likelihood of achieving S state can be optimistically half that of remaining within A or D state, but in reality, from our observations, it can be as low as one-tenth.

For practical analysis, we employ ε to be conservatively one-tenth. This is based on our observations that target variation success ratio is difficult to achieve. Given that a possible change to variation weight is only conducted every evolution, in the best case, it would require at least ten evolutions of increasing or decreasing weight adjustment before achieving the target stability ratio. This implies at worst, a tenth of the variation application for the likelihood of realising stability. Therefore, $\varepsilon = 0.1$ and the transition probability of entering absorptive state S is one-tenth of ω. Also, depending on the probability value of ω, the GEA process operates within transient states A or D ranging between a maximum and minimum number of evolutionary cycles.

To determine this, we assume two possible values for ω are sampled for use in our analysis. The values we use for ω are determined to be 0.48 to 0.9. We assign 0.9 to be the near maximum probability value of the variation weights that bring about changes between transient states. Weight values greater than 0.9 are considered to be impractical for the GEA process (explained in Sections 4.5.9 and 4.5.10). In our Markov chain, given that the probability of changing between increase or decrease states is not greater than the probability of maintaining the same self-adaptive variation, we have $\omega \geq 1-(1+\varepsilon)\omega$. Substituting for $\varepsilon = 0.1$, then $\omega \geq 0.48$. With $\varepsilon = 0.1$, we find N to be,

$$N = \begin{array}{c} \\ A \\ D \end{array}\!\!\begin{array}{c} \begin{array}{cc} A & D \end{array} \\ \begin{pmatrix} 13.1 & 9.9 \\ 9.9 & 13.1 \end{pmatrix} \end{array}\Bigg|_{\omega=0.48} \quad \text{and} \quad N = \begin{array}{c} \\ A \\ D \end{array}\!\!\begin{array}{c} \begin{array}{cc} A & D \end{array} \\ \begin{pmatrix} 10.1 & 1.0 \\ 1.0 & 10.1 \end{pmatrix} \end{array}\Bigg|_{\omega=0.9} \qquad (5.4.3)$$

The numerical results indicate that variation can (i) remain in an increasing A or decreasing D state for 10.1 to 13.1 evolutionary cycles, and (ii) transition between A or D states from 1 to 9.9 cycles, before the Markov chain is finally absorbed into S. Additionally, the overall time to absorption t can be calculated by summing the row elements of N. Multiplying N by a column matrix of 1s,

$$t = \begin{pmatrix} 23 \\ 23 \end{pmatrix}\Bigg|_{\omega=0.48,\varepsilon=0.1} \quad \text{and} \quad t = \begin{pmatrix} 11.1 \\ 11.1 \end{pmatrix}\Bigg|_{\omega=0.9,\varepsilon=0.1} \qquad (5.4.4)$$

We interpret the above results as follows. Considering both limiting cases of ω, from (5.4.4), the minimum and maximum number of self-adaptive changes applied to variation range between 11 and 23 respectively, regardless of which state (*A* or *D*) the variation starts within.

From a parameter selection viewpoint, we are particularly interested in the number of transient states the variation undergoes. From (5.4.4), in order for the GEA process to achieve absorption, under worst conditions, at least 23 evolutions must be conducted for the case $\omega = 0.48$. If ideal conditions are used, then 11 evolutions are needed. However, one must allow for at least 23 evolutions for the GEA test generation.

The above analysis thus far assumes each evolutionary cycle performs variation weight and usage adjustment. This may not always be the case. Therefore, extra evolutionary cycles must be considered for execution in order to compensate. The question then arises, how many extra evolutions should be performed, and what would be the upper limit to set for the number of evolutions GEA parameter? To determine this, we consider the number of variation weight increments and decrements during GEA.

Let σ be the variation change factor – i.e., the amount σ to decrease the variation, or amount $1/\sigma$ to increase variation. σ is chosen to be at least 0.9 to ensure changes applied to variation usages are small. Given there are five main variation operators, the initial likelihood of applying each variation is evenly spread at 0.2. Next, we calculate the number of variation increases *k*, that is possible before the maximum probability hard limit of 1 (imposed in Section 4.5 Chapter 4) is reached, i.e., $0.2 \times (1/\sigma)^k < 1$. Subsequently, *k* is approximately 15 at most. Note that the number of variation decrements is not considered because variation decrease simply reduces closer toward 0 but never attains such probability. Calculating for number of decrements would return a large value greater than 15. Hence, we use $k = 15$ only.

This value of *k* alone implies the additional number of evolutionary cycles should be 15, such that the maximum number of evolutions is 38 (i.e., 23 + 15). The prior number of evolutions evaluated (23) assumed ideal conditions of reaching target ratio absorption quickly, whereby variation is adjusted every evolution. This is generally not the case. More evolutions are needed and actually achieving target variation success ratio is not guaranteed. Hence, in the worst case, taking into account an additional 15 evolutions could be conducted before maximum variation weight limit is reached, the number of evolutions that should and can be conducted is between 23 to 38. For selecting the number of evolutions parameter, we choose the median between the minimum and maximum, which is 31 evolutions.

Note that in this Chapter, we explicitly refer to variation usages and weights in terms of probability values between 0 to 1. In Section 4.5 of Chapter 4, equivalent variation weights used larger values multiplied by a factor of 100 for test generator implementation reasons. For Markov chains analysis whereby transition probabilities are manipulated, using probability equivalent values for variations is more appropriate. Also, unlike the self-adaptation in Sections 4.5.9 and 4.5.10 Chapter 4, the adjustment of variation usage applied for our analysis above does not consider relative attainment to target success ratio. The self-adaptation is examined from the perspective of increment or decrement by fixed amounts. For our parameter selection purposes, assuming a variation ratio distance factor ϕ of 1 is adequate. This was intentional so as to simplify the analysis here. Otherwise, additional factors would have to be incorporated into the Markov chain making the Markov model larger and complicating the overall parameter selection procedure.

5.5 Modelling variation weight characteristics

The Markov chain in this section models the characteristics of a variation operation and its usage during the GEA test generation lifecycle, specifically the variation weights characteristics. Recall that variation weights are variables employed by the GEA process to manage the types and amount of variation operations in order to create tests. The Markov model chain is used to analyse the long term behaviour of variation usage. Specifically, we are interested in the probability of achieving the target success ratio for variations.

The variation usage and its weight adjustments are carried out the same as the flow in Figure 5.3. For modelling variation weight values directly, the states for the Markov chain are designed to correspond to the ‘Current variation weights’ and ‘New variation weights’ components in Figure 5.3. The Markov chain will reveal the range of variation usage weight values that are employed by variation during the GEA process.

We use the Markov model to analyse what GEA behaviours and parameters affects our goal of acquiring the target success ratio. If desired variation usage weight values are not achieved to attain the goal, then the GEA variation could enter into a *saturated* state, whereby variation weights are either at undesired maximal or minimal application values.

In comparison, the previous Markov chain of Section 5.4 examines what are the possible types of adjustments (or non-modification) that can be applied to variation usage; including the sequences and

rate of changes applied to variation according to self-adaptation. The information analysed from Section 5.4 and the Markov chain analysis in this section reveals important GEA variation behaviours, and will be employed to build up further Markov chains to characterise the snippet compositions of tests in the next section. Whilst no parameters are selected from the Markov chain in this section, the variation usage weights information extracted will be employed for selection of parameters later in Sections 5.6 and 5.7.

5.5.1 The Markov states

We identify three states in which usage of variation could end up at GEA termination:

- Target variation success ratio state *G*, which represents the level of variation at which the desired goal of one in five successfully varied test is attained.
- Upper boundary state *U*, which represents the unwanted condition whereby the maximum variation weight value causing saturation defined from Section 4.5 of Chapter 4 has occurred.
- Lower boundary state *L*, which represents the unwanted condition whereby the minimum variation weight value causing saturation has occurred.

These states are all absorptive. Any one of the above states can be the final state at the end of the GEA process. The aim of Markov chain modelling and analysis in this section is to determine the likelihood of variation falling into these absorptive states.

In addition to the *G*, *U*, and *L* absorptive states, we define two other transient states to represent the condition of variation before it becomes absorbed. They are:

- State *E*, which represents the state in which variation weight holds the intermediary values above the minimum variation weight represented by *L* and the eventual goal variation value at which *G* occurs, and
- State *H*, which is the state corresponding to values between *G* and *U*.

5.5.2 Design of Markov chain and transition probabilities

To determine transition probabilities between states, we employ the fundamental matrix of the previous Markov chain in Section 5.4. Let c be the number of times a variation operation's usage is continually increasing or decreasing (i.e., increment state A and decrement D from Section 5.4), and s be the number of times a variation switches between increase and decrease. c is proportional to the likelihood of the variation under continual change from one evolution to the next. Similarly, s is proportional to the likelihood of the variation switching. The number of times a variation undergoes changes can be mapped to the fundamental matrix of the Markov chain in Figure 5.4. Using (5.4.2), the mapping is as follows.

$$\begin{pmatrix} c & s \\ s & c \end{pmatrix} = \frac{1}{\omega(2\varepsilon + (1-(1+\varepsilon)^2)\omega)} \begin{pmatrix} 1-\omega & 1-(1+\varepsilon)\omega \\ 1-(1+\varepsilon)\omega & 1-\omega \end{pmatrix} \tag{5.5.1}$$

Based on the absorptive and transient states defined in Section 5.5.1, and the variation increment and decrement variables, we design the Markov chain to model the variation weights lifecycle characteristics in Figure 5.5.

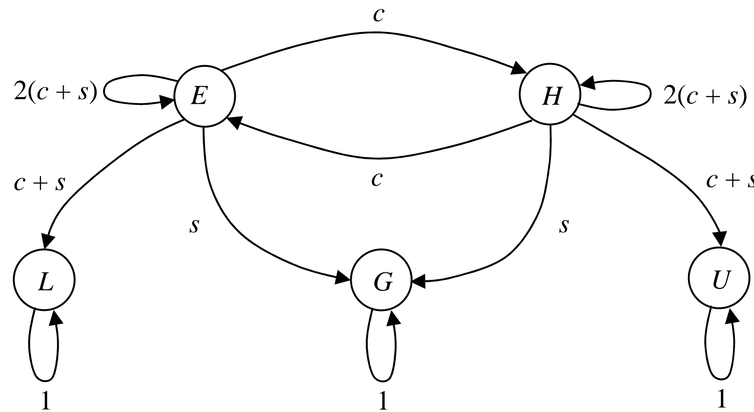


Figure 5.5 Variation characteristics Markov chain state diagram

The transition probabilities in Figure 5.5 are determined in a similar approach to that of the Markov chain in Section 5.4, and are based on certain assumptions and observations of the GEA variation, self-adaptation and test generation process. These transition probabilities depend on the state transitions between A and D in the previous Markov model from Section 5.4. It follows that c is the transition probability for E traversal to H, and also for $H \rightarrow E$ transition. For the transition E to G to occur, it requires a change opposite to what was previously applied in order to avoid overshooting past the goal state G. Thus, the transition probability is s. The same applies for the transition $H \rightarrow G$.

The transition $E \rightarrow L$ is where the variation usage undergoes decrement. It corresponds to the transitions $D \rightarrow D$ and $A \rightarrow D$, of the Markov chain in Section 5.4. The transition probability is $(c + s)$. Similarly, the transition probability for $H \rightarrow U$ is $(c + s)$. For the transitions of E to itself and H to itself, they are the usage of increment or decrement. Here the variation weights are adjusted but remain in the same intermediate state. Such changes corresponds to any of transition $A \rightarrow A$, $A \rightarrow D$, $D \rightarrow A$, and $D \rightarrow D$. The transition probability for $E \rightarrow E$ is $2(c + s)$, and is the same for $H \rightarrow H$. The states G, L and U are absorptive, hence the transition probability to itself is 1. Appendix F.1 describes the derivation of these transition probabilities in greater detail.

5.5.3 Markov chain analysis

Using the Markov chain in Figure 5.5, similar absorptive Markov chain analysis carried out in the Section 5.4 is performed. Note that the transition probabilities for the Markov chain are normalised with a factor $4(c + s)$ to ensure rows of the transition probability matrix sum up to one. By restructuring the transition matrix of this Markov chain into the absorptive Markov chain canonical form $\begin{pmatrix} Q^n & N_n R \\ 0 & \mathrm{I} \end{pmatrix}$ with $N = (\mathrm{I} - Q)^{-1}$, we acquire the fundamental matrix N and the matrix R as follows.

$$N = \frac{16(c+s)^2}{4(c+s)^2 - 4c^2} \begin{array}{c} \begin{matrix} \quad E & \quad\quad H \end{matrix} \\ \begin{pmatrix} \frac{1}{2} & \frac{c}{4(c+s)} \\ \frac{c}{4(c+s)} & \frac{1}{2} \end{pmatrix} \end{array} \begin{matrix} E \\ H \end{matrix} \tag{5.5.2}$$

$$R = \begin{pmatrix} \frac{1}{4} & \frac{s}{4(c+s)} & 0 \\ 0 & \frac{s}{4(c+s)} & \frac{1}{4} \end{pmatrix} \tag{5.5.3}$$

where the first row and column of the fundamental matrix N corresponds to the transient state E, and the second row and column corresponds to H.

Using N from (5.5.2) and R from (5.5.3), the long term probabilities of the GEA process being absorbed into the target success ratio state G or boundary L or U states are determined by considering infinite number of evolutions (i.e., transitions) in the Markov chain.

For steady state as $n \to \infty$,

$$B = NR = \begin{array}{c} \\ E \\ H \end{array}\!\!\begin{array}{c} \begin{array}{ccc} L & G & U \end{array} \\ \begin{pmatrix} 0.54 & 0.31 & 0.15 \\ 0.15 & 0.31 & 0.54 \end{pmatrix} \end{array}\Bigg|_{\omega=0.48} \tag{5.5.4}$$

where c and s are calculated for $\omega = 0.48$ and $\varepsilon = 0.1$ using (5.5.1), which represents the case that is of interest for our analysis as described in Section 5.4.3 previously.

The matrix B is interpreted as follows. If the test generation began in the intermediate transient state E, there is a 0.54 probability of absorption into the boundary L state, or 0.15 probability of absorption into the other boundary state U when the GEA process terminates. If the variation weight begins in the lower intermediate state E between low boundary state L and target goal state G, then it is more likely to be absorbed into L than U as it was initially closer to the L state. An equivalent argument can be put forth for a commencement state of H and greater likelihood of absorption into U.

From (5.5.4), regardless of which state variation starts from, there is a 0.31 probability of attaining the target variation success ratio state G (i.e., the variation adjustment stability state). Avoiding absorption into boundary U or L states is important because once a variation attains the maximum or minimum boundary values, further variation applied will continue to be either excessive (U) or insignificant (L). This results in runaway conditions for the GEA process which is difficult to recover from, even if variation weights can be scaled back towards intermediate values. In general, attaining state G is the desired goal, but avoiding U and L states are just as critical.

Besides revealing the possible long term behaviour and value ranges of variation usage weight adjustments, the variation weight characteristic Markov chain in this section will be used to build up the other Markov models from Section 5.6 onwards.

5.6 Modelling composition of test programs

The goal of modelling and analysing the composition of test programs is to examine what kinds of snippets are composed into tests during the GEA process, and how the inclusion of snippets are affected by test parameters and other GEA factors.

Figure 5.6 shows a simplistic diagram of the possible new test compositions after certain snippets are included into the test via GEA variation and snippet selections (Section 4.5 Chapter 4). As per the

GEA evolutionary cycle, the new test composition of snippets will act as current test for varying further new test and their snippet compositions in the next iterative cycle. Depending on the variation and snippet selections conducted, the new test will be composed of new type of snippets if variation inserted snippets; or be dominated by other type of snippets if previous snippets were excluded in the new test due to constraint or dependency rules for certain variations. The new composition of snippets in tests can be distinguished by snippets of certain types. For example, snippets that caters for specific SoC devices or snippets that perform different types of operations on the SoC. Depending on the Markov chain complexity and size considerations, different customisation of these new test snippet types will be designed to act as states of the Markov model as described next in Section 5.6.1.

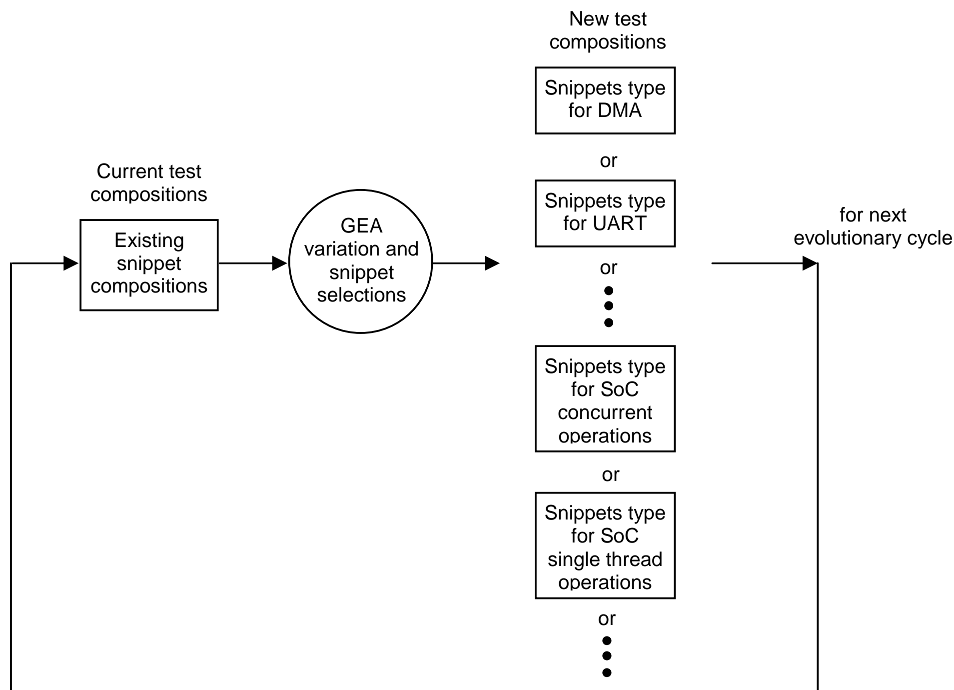


Figure 5.6 Types of snippet test compositions due to variation and snippet selections

In this section, we are interested in parameter settings that will ensure the best set of snippets are realised within tests and test populations throughout GEA test generation. The parameters that will be selected from analysis of this Markov model are the initial snippet selection probability weighting values, and the number of consecutive evolutions to check for GEA termination variables.

5.6.1 The Markov model

The Markov states

In order to characterise snippet selections, we define states in our Markov chain to represent the types of snippets that can be selected into a test. These states correspond to snippets selected each time a variation operation is conducted in the next evolution. The snippet selection represents the next state of the Markov chain, which is influenced by variation and other parameters in the current state. The kinds of snippet selection includes a *no-change* state in which the test composition of snippets remains fixed without influence from any new or existing snippets. For example, if mutation or removal variation is selected, or a snippet cannot be added due to unresolved dependencies.

The goal of the Markov model is to capture the kinds of snippets selected in the next state based on current snippet inclusions within existing composition of the tests. To this end, it is natural to define the states of the Markov model in terms of individual snippets from the snippets library. But such an approach poses immediate problems when managing the large size of such Markov chains to perform analysis. The Markov model analysis would also become increasingly complex when snippets are added into the snippets library in the future. Expansion of the snippet library cannot be catered for.

Therefore, we reduce the size and complexity of our Markov chain by grouping together snippets that share common and important characteristics. For example, snippets that cater to exercising an SoC device, or performing particular types of testing on the SoC. Even though various forms of snippet groupings can be employed, these groupings must allow for scalability of the snippets library whilst containing the number of Markov states.

Our first attempt at reducing the Markov chain and grouping together snippets involved states that capture snippets common to each type of SoC device. This reduced the number of states to seven including the state whereby test composition did not change with any new snippet. However, to minimise complexity and analysis further, the Markov chain can be reduced again based on larger groupings of more snippets. Appendix F.2 discusses in more detail our attempts at reducing the Markov model size and grouping of snippets.

In our eventual final Markov model, we employ grouping of snippets that execute either concurrent or non-concurrent types of SoC test operations to act as Markov states. Specifically, the Markov states are, (i) fixed snippet compositions with no changes (F), (ii) snippets performing multiple and concurrent operations (C), and (iii) snippets that execute one single threaded operation only (O).

Two types of concurrent snippets belong to *C*, the DMA and UART snippets. For *O*, this state captures the remaining snippets in the snippets library, these snippets being the PIO, timer, memory and CPU snippets.

The Markov state diagram and transition probability matrix

The Markov state diagram of our Markov model is shown in Figure 5.7. The transition probabilities for this Markov chain depend on the variation operations performed, and also the snippet selection probabilities under the same influences of self-adaptation. Therefore, the analysis results from previous Markov chains of prior Sections 5.4 and 5.5 are reused within the transition probabilities in Figure 5.7.

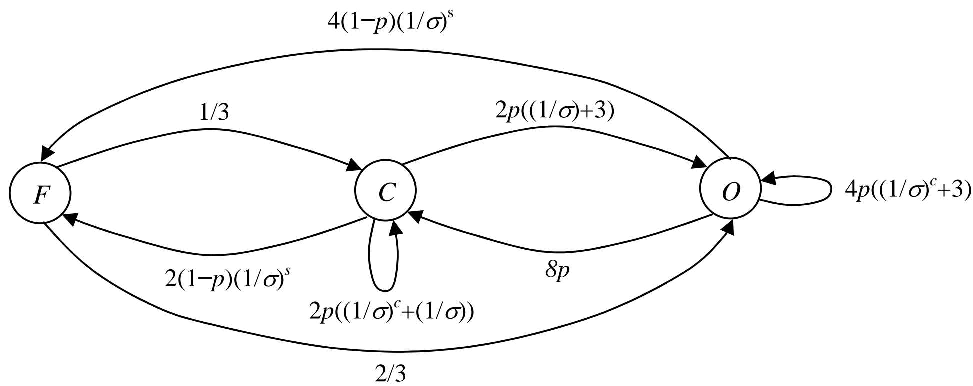


Figure 5.7 Composition of test program Markov chain state diagram

We summarise the derivation of the transition probabilities in Figure 5.7 as follows. Originally, there were six types of snippets, one type for each SoC device. Two types of these snippets belong to *C* and four belong to the *O* state. We assert it is improbable that there would be two consecutive acts of non snippet selection; in fact, the test generator forbids this. Hence the $F \rightarrow F$ transition probability is 0. Therefore, the remaining transition probabilities from *F* state are 1/3 and 2/3, respectively, for $F \rightarrow C$ and $F \rightarrow O$.

For the remaining transition probabilities, we need to consider the snippet selection probability. Let p be the snippet selection probability that is influenced under self-adaptation during GEA. We assume that all snippets hold the same probability of selection to begin with. Within the *C* state, the probability of (i) adding the same DMA or UART snippet consecutively is greater compared with the probability of (ii) adding a DMA followed by a UART snippet (and vice-versa). Therefore, the transition probability for $C \rightarrow C$ is composed of $2p(1/\sigma)^c$ for (i), and $2p(1/\sigma)$ for (ii), where c comes from (5.5.1)

of the Markov chain in Section 5.5, and σ is the self-adaptation change factor. The probabilities are each multiplied by two as there are two snippets types, DMA and UART in C. There is a $(1/\sigma)$ factor for case (ii) when a DMA snippet is selected followed by a UART snippet (and vice versa) because dependencies exist between these snippets. The total transition probability for $C \rightarrow C$ is $2p((1/\sigma)^c+(1/\sigma))$.

For the transition $C \rightarrow O$, whenever a DMA or UART snippet is selected, there is a greater likelihood that the PIO snippet will be selected due to dependencies between these snippets. Hence, the collective probability from both DMA or UART to PIO snippet is $2p(1/\sigma)$. For the remaining transition probabilities between a DMA or UART snippet followed by selection of a timer, memory or CPU snippet, the probability is $6p$, for all of the six possible transitions. Hence, the overall transition probability for $C \rightarrow O$ is $2p(1/\sigma)+6p$, which simplifies to $2p((1/\sigma)+3)$.

For transitions from C or O states to the fixed snippet composition state F, the probability of transition is $(1-p)(1/\sigma)^s$. The $(1/\sigma)^s$ factor is included to account for the value of the snippet selection variable switching between increasing to decreasing adjustment rate (and vice-versa), whilst the $(1-p)$ factor is the probability of a snippet not selected. Given that there are two snippet types in C and four snippet types in O, the transition probabilities for $C \rightarrow F$ and $O \rightarrow F$ are $2(1-p)(1/\sigma)^s$ and $4(1-p)(1/\sigma)^s$, respectively.

The remaining transition probabilities are determined in the same manner as above − by breaking down the types of snippets within each state, identifying the relevant probability factor between each combination of transition from one snippet type to another, and combining these probability factors together to produce the overall state transition probabilities. Appendix F.2 details the derivation of transition probabilities in Figure 5.7, including how the transition probabilities are broken down and extracted from (i) a transition matrix of larger set of individual snippets from the snippets library and then (ii), groups of snippets for each SoC devices. Both transition matrices of (i) and (ii) were originally proposed to act as states in the Markov chain, and their influence are captured in Figure 5.7.

The Markov chain transition probability matrix corresponding to Figure 5.7 is shown in (5.6.1). A normalisation factor is employed to ensure the probability transition rows of the matrix sum up to one during actual analysis and evaluation of the Markov chain.

$$P = \begin{pmatrix} 0 & 1/3 & 2/3 \\ 2(1-p)(1/\sigma)^s & 2p((1/\sigma)^c + (1/\sigma)) & 2p((1/\sigma)+3) \\ 4(1-p)(1/\sigma)^s & 8p & 4p((1/\sigma)^c + 3) \end{pmatrix} \quad (5.6.1)$$

(columns F, C, O; rows F, C, O)

5.6.2 Markov chain analysis and parameter selections

The Markov chain produces a regular and Ergodic transition matrix. A regular and Ergodic Markov chain has a transition probability matrix P such that any power of P contain positive elements, and it is possible to go from any Markov state to any other state in one or more Markov transition steps. According to Ergodic Markov chain evaluation [GS06], our analysis involves finding a vector w within $W = \lim_{n\to\infty} P^n$ such that $wP = w$. The vector w has elements that correspond to the states of the Markov chain, i.e., $w = [w_F, w_C, w_O]$.

Each vector element of w reflects the proportion of time the Markov chain (modelled GEA process) was in each state during the entire test generation (i.e., Markov chain steady state as $n \to \infty$), regardless of which initial state the Markov chain began from. In spite of any snippet selections made initially, w indicates what proportion of the different kinds of snippets (or no snippet for F state) that will be selected during the GEA process. For our test generation process, high values of w_F and w_O, and low value of w_F is desired to ensure new and different composition of snippets are introduced for inclusion into the tests. This enables new test functions to be tested during the GEA process more readily.

Additionally, during the GEA process, the mean recurrence time back to snippet selection states C and O from a static snippet non-changing state F must be minimised. In particular, when the GEA process enters F, the time spent in F must be reduced and the GEA process must revert back to C or O as soon as possible. The mean recurrence time for C and O is equivalent to $1/w_F$ and $1/w_O$ respectively, supporting the need for maximising w_C and w_O.

Initial snippet probability parameter selections with practical considerations

Based on the from Sections 5.4 and 5.5, where $\sigma = 0.9$ and $\omega = 0.48$ for $c = 13.1$ and $s = 9.9$, we can calculate w for different values of the initial snippet selection probability parameter p. Table 5.1 shows our evaluation results for relevant subset of sample values for p. The results indicate that the greater

the value of p, the higher the values for w_C and w_O. However, like the variation variable probabilities in Section 5.5, the snippet selection probability is influenced by self-adaptation as well. Hence, the maximum value of p is restricted by the probability boundary states U and L; which must not be absorbed into under similar self-adaptation Markov chain analysis.

Table 5.1 Test composition results for different snippet selection probability *p*

p	w_F	w_C	w_O
0.05	0.46	0.12	0.24
0.1	0.43	0.19	0.38
0.2	0.36	0.21	0.43
0.5	0.19	0.27	0.54

Therefore, we choose $p = 0.17$ to ensure that the sum of w_C and w_O (i.e., $w_C + w_O = 0.21 + 0.42 = 0.63$) are still greater than $w_F = 0.37$. The maximum value of p that can be chosen is at most 0.17 because there are six types of snippets originally (one for each device on the SoC) in the initial Markov chain which we designed; and the probability of not selecting a snippet initially is zero. Even though our Markov model is a reduction of the larger seven state element Markov chain, the transition probabilities were originally derived from this larger Markov chain (described in Appendix F.2). Hence, p must be selected based on considerations from the larger Markov chain as well. Our selection of p allows for each of the initial snippet selection probability for all six types of snippets to sum up to one.

Mean first passage time analysis and termination parameter selection

Another analysis that can be conducted with regular Ergodic Markov chains is to determine the mean first passage time matrix M. The mean first passage time matrix has the same number of rows and columns as P in (5.6.1); and similar to P, each sequence of row and column elements correspond to F, C, and O states. The ij-th element of M shows the time (i.e., number of evolutionary cycles) required to go from the state i to j state for the very first time, where $i, j \in \{F, C, O\}$. Naturally, the diagonal entries of M are zero.

In order to attain M, the fundamental matrix Z of the regular Ergodic Markov chain must first be determined via $Z = (\mathrm{I} - P + W)^{-1}$. Then, each ij-th element of M is evaluated from $m_{ij} = \dfrac{z_{jj} - z_{ij}}{w_j}$

where z_{jj} and z_{ij} are the jj-th and ij-th elements of Z, and w_j is the j-th element in the w vector. For an Ergoodic Markov chain, this is equivalent to any j-th column element in the W matrix because all rows of W are the same, with each row being the w vector.

Using the same values $\sigma = 0.9$, $\omega = 0.48$, $c = 13.1$, $s = 9.9$, and $p = 0.17$ from earlier sub sections, the mean first passage time matrix is evaluated to be,

$$M = \begin{array}{c} \\ F \\ C \\ O \end{array} \begin{array}{c} \begin{array}{ccc} F & C & O \end{array} \\ \begin{pmatrix} m_{FF} & m_{FC} & m_{FO} \\ m_{CF} & m_{CC} & m_{CO} \\ m_{OF} & m_{OC} & m_{OO} \end{pmatrix} \end{array} = \begin{pmatrix} 0 & 4.1 & 2.3 \\ 2.9 & 0 & 1.7 \\ 3.1 & 2.6 & 0 \end{pmatrix} \tag{5.6.2}$$

Using (5.6.2), we determine the GEA termination parameter governing the number of evolutions to check for non-improvements in coverage. Recall from Section 4.8 Chapter 4 that GEA test termination can be triggered if the previous K consecutive number of evolutions did not yield any enhanced tests. We assign the parameter K to be $(m_{FC} + m_{FO}) = 6$ evolutions because $(m_{FC} + m_{FO})$ represent the time it takes to start selecting new snippets into the tests when the GEA process begins from the F static test composition state. This number of evolutions is considered the maximum time for which no new snippet changes or test enhancements will be realised. It usually occurs at initial stages of the GEA process because later during test generation, it is assumed that variation and snippet selection probability variables would have invoked various snippets and test modifications to enhance tests already.

We assume that only at the start of GEA and in the worst case, it is acceptable for up to K consecutive number of generations to proceed without improvements. Hence, we set the number of evolutions termination parameter to be 6 based upon (5.6.2). During GEA, if there is a period greater than K number of evolutions for which there is no improvements, then the initial phase whereby no snippets of concurrent (C state) or single thread (O state) types are included when initially in a static test composition F state, no longer applies. The GEA process has stagnated and has attained the best set of enhanced tests it can, therefore test generation should be terminated. If the above $(m_{FC} + m_{FO})$ mean time for selection of snippets into tests is not taken into account, then halting test generation less than $(m_{FC} + m_{FO})$ evolutions without improvements may cause pre-mature termination conditions.

Finally, note that the analysis conducted in this section was specifically for the snippets library of the Nios SoC, which is used widely as the SoC under verification for experiments in this thesis. The analysis may not apply to other SoCs, for example, the Tsinghua digital signal processing (DSP) SoC

case study in Chapter 4. However, the technique of snippet selection and test composition analysis in this section can be adopted for equivalent analysis of other SoCs and snippets library.

5.7 Modelling composition of test populations

The final Markov chain in this chapter characterises the composition of tests in the test population during the GEA test generation process. Figure 5.8 shows the flow at every cycle of the GEA process which controls creation of new tests, and selection of existing parent and new children tests for the next evolution population.

Our goal is to examine the ratio of existing and newly created tests that are selected to make up the next evolution population. By doing so, we deduce appropriate test population size parameters. The basis of our parameter selection approach relies on the observation that at any stage during the GEA process, the test population should contain higher percentage of newly varied tests from the current children population. Greater selection of offspring tests and lower retention of parent tests from previous evolutions imply the newly created tests have been evolved effectively.

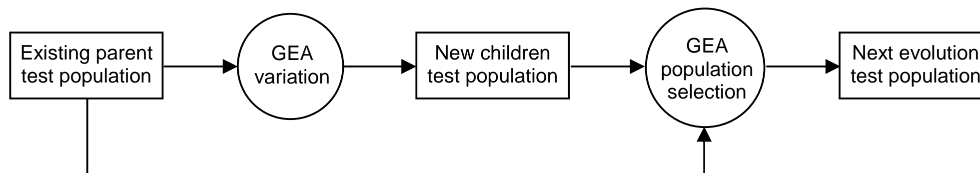


Figure 5.8 Test population selection flow

With this in mind, the population composition Markov chain analysis will determine test generation parameters to prolong the number of evolutions in which the test populations contain greater children tests; before the GEA process eventually stagnates with no further evolutions of enhanced tests. By this stage, the test population should contain entirely of parent tests. The parameters to select based on analysis of the Markov model in this section are the parent and children test population sizes.

5.7.1 The Markov states

To facilitate our analysis, we propose a number of desirable conditions on the test population. These conditions are to facilitate and promote test population diversity with greater selections of successful new children tests for the next evolution population. During the GEA process, at any evolutionary

stage, our goal is for at least half of the population of tests selected to contain a greater proportion of children tests than parent tests from prior evolutions. Ideally, the test population should maintain this condition on the test population for as many evolutions as possible. Whilst it may seem more beneficial for the entire test population to be made up of children tests, we propose at most half the test population to contain children to ensure some parent tests from earlier evolutions are at least retained. This is so that the GEA process does not converge prematurely. The test population will be allowed to evolve properly rather than GEA injecting rapid and wide-ranging changes into tests. Otherwise, this will restrict exploration of the search space and cause too-early GEA termination.

The states of the Markov chain are defined to represent the ratio of children to parent tests making up the test population during each evolutionary cycle. The states are as follows.

V : Half the population consists entirely of new children tests.

X : Half the population consists of greater number of children tests than parent tests.

Y : Half the population consists of less than or equal number of children tests compared to parent tests.

J : Half the population consists of mostly parent tests, with either none or a few children tests only.

The first three states, V, X and Y, are transient states during which the GEA process evolves the test population containing a mixture of children and parent tests. State J is an absorptive state that corresponds to the termination phase of the GEA process, when no new children tests are selected and the test population stagnates with no further enhancing tests to simulate the SoC.

5.7.2 Design of Markov chain and transition probabilities

The transition probabilities for the absorptive Markov chain are made up of parameter μ which represent the number of parent tests, and parameter λ which is the number of children tests. These parameters influence the combination of parent and children test population sizes that will ideally facilitate longer durations within the transient V and X states before absorption in J. The results and variables from Markov chain analysis of Sections 5.5 and 5.6 are incorporated into transition probabilities here. For instance, the test population composition Markov chain also depends on self-adaptive variation and snippet selections. The transition probabilities are defined in terms of variation upper boundary, variation lower boundary, and the variation usage target ratio goal probabilities from (5.5.4). From the matrix elements of (5.5.4), we assign these probabilities to be u = 0.54 for the upper

boundary, $l = 0.15$ for the lower boundary and $g = 0.31$ for the variation success ratio goal probability, under the assumption that the Markov model in Section 5.5 began in the E state for initial GEA variation. Additionally, the snippet selection data from Section 5.6, $w_F = 0.37$, $w_C = 0.21$, and $w_O = 0.42$ (for $p = 0.17$), are also reused for the population composition Markov chain in this section.

Figure 5.9 shows the population composition Markov chain.

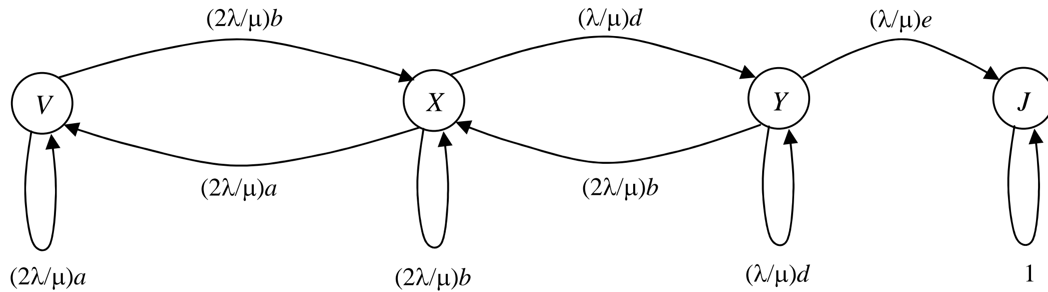


Figure 5.9 Test population composition Markov chain state diagram

We introduce variables a, b, d, and e to simplify the representation of the transition probabilities in the Markov chain. They are:

$$a = 0.9u(3w_F + 2w_C + 2w_O), \quad b = 0.5g(3w_F + w_C + w_O),$$
$$d = 0.5g(3w_F + 2w_O), \quad e = 0.1l(3w_F).$$

The transition probabilities of the Markov chain in Figure 5.9 are derived and assigned in a similar manner to that of previous Markov chains in prior sections. Appendix F.3 describes further the transition probabilities used in the population composition Markov chain.

5.7.3 Markov chain analysis and parameter selections

This absorptive population composition Markov chain can be mapped to an equivalent transition matrix P. Absorptive Markov chain analysis similar to that in Section 5.4, is then carried out. The transition matrix is firstly mapped to canonical form,

$$\begin{array}{c c} & \begin{array}{cc} TR & ABS \end{array} \\ \begin{array}{c} TR \\ ABS \end{array} & \begin{pmatrix} Q & R \\ 0 & \mathrm{I} \end{pmatrix} \end{array}$$

The transient (*TR*) and absorptive (*ABS*) transition state probabilities are grouped together into matrices *Q*, *R* and the identity matrix I. After substituting for u, l, g, w_F, w_C, and w_O with values given in Section 5.7.2, the fundamental matrix N is evaluated by $N = (I - Q)^{-1}$.

Recall that the fundamental matrix indicates the proportion of time that the Markov chain is within a destination state if it transitioned from a certain source state initially. Given that the rows of N correspond to the source states, and the columns represent the destination states, the matrix element $n_{11} = \left(\frac{0.56\mu - \lambda}{0.56\mu}\right)$ of N, which corresponds to the $V \rightarrow V$ transition, must be maximised to ensure that the duration of time spent in the V state is maximised. By maximising the V state duration, there will be a greater number of evolutions in which the test population selects children tests predominately. This indicates successful GEA variation and SoC testing, which is our primary goal.

By maximising the $V \rightarrow V$ transition probabilities factors, we can determine the best ratio of children and parent test sizes for GEA test generation. The analysis to achieve this goal is as follows.

$$\mathrm{Max}\left(\frac{0.56\mu - \lambda}{0.56\mu}\right) = \mathrm{Max}\left(1 - \frac{\lambda}{0.56\mu}\right) \Rightarrow \mathrm{Min}\left(\frac{\lambda}{0.56\mu}\right) \qquad \therefore \lambda < 0.56\mu \qquad (5.7.1)$$

For greater children tests to be selected, the children test population size should be as large as possible. To establish the desired ratio of children and parent test population sizes and attain an upper limit on the children population size, (5.7.1) suggests the ratio of children to parent population size should be chosen to be approximately 1:2.

To determine actual population sizes, we consider what the parent test population should be. The parent test population size is essentially the test population that is used at the beginning of each evolutionary cycle to seek out new tests. A number of factors govern selection of this population size. First, the population size must be sufficiently large to provide a diverse mixture of tests, in order for variation in each subsequent evolution to be effective. However, the population size must not be too large, otherwise fitness evaluation of tests, and storage and management of tests of various evolutionary life spans during the GEA process becomes a bottleneck. The overhead in switching between many test executions for each population also becomes significant if population size is too big.

Given our GEA process begins with very low number of snippet genes to make up a test individual, the test population size is set to the number of unique snippets that is available in the snippets library to build up an individual. This ensures the population size will provide sufficient diversity at the start of GEA by allowing for the equivalent of at least one different snippet to be included in a test. This

serves as a rudimentary guideline for selecting population sizes that is neither too small to restrict test diversity, nor too large to adversely affect the GEA process.

For GEA test generation, our snippet library holds approximately 30 unique snippets, hence the parent population size (μ) is set to 30. From (5.7.1), this implies that the children population size (λ) should be 15.

5.8 Evaluation and summary of method

Based on the Markov chain modelling and analysis of the GEA test generation process, the following test generation parameters and their values were assigned:

- Number of evolutions : 31
- Initial variation probability : 0.2
- Self-adaptation variation change factor, σ: 0.9
- Initial snippet selection probability variable : 0.17
- Number of last consecutive evolutions to check for test improvement (or otherwise trigger termination) K : 6
- Size of parent test population, μ : 30
- Size of children test population, λ : 15

These parameter selections were attained from Markov analysis of sub processes and components of the GEA test generation, and were conducted under certain constraints from practical considerations. In order to contain Markov chains to manageable size and complexities, not all elements of the GEA process and other verification factors could be incorporated into the modelling. The Markov chain analysis and subsequent parameter selections are based upon observations of the GEA test generation and specific assumptions. For example, we could not model each individual snippet and their characteristics for a particular SoC verification.

Therefore, it must be emphasised that parameter selections from our technique of GEA Markov chain modelling is most effective as a guideline to how parameters should be chosen at the beginning of an SoC test generation phase. Guided by these initial parameter selections, some minor fine-tunning of these parameters may then be performed during actual GEA test generation runs. The aim of Markov chain analysis for parameter selections is to provide sound recommendations and proven starting points from which parameter selections can be conducted. The advantage of this is to reduce the

unnecessary preliminary test generations and calibration effort to find suitable parameter values. Based on our suggested parameter selections, only some small adjustments are required, allowing for immediate and effective test generations of the SoC earlier. This saves valuable verification time and facilitates more efficient verification from the beginning.

Evaluation

Given that the Markov chain derived parameter selections are not strictly enforced, preliminary test generation fine tuning can lead to some slight adjustments of final parameter values employed. This accounts for the minor differences in parameter selections suggested in this chapter and those employed in Chapter 4 for GEA test generations.

For example, the number of consecutive evolutions to check for fitness improvement termination parameter K was adjusted to 5 instead of 6 (which was recommended by our analysis). It was discovered during proper GEA runs, shorter number of evolutions for these parameters was needed because the test generation could be prolonged unnecessarily. This was prevalent especially for conditional coverage testing. Depending on the different type of coverage fitness test generations, different number of GEA evolutions was conducted before termination. Our analysis in this chapter could not take into account the different types of fitness criteria, and could only recommend a common number of evolutions. For line and toggle coverage runs, our suggested number of evolutions was appropriate and did not require further adjustments. But given the higher processing resource needs of conditional coverage runs, we had to reduce recommended number of evolutions for conditional coverage.

Another refinement was the following. From Section 5.4, recall that the suggested number of evolutions was 31 whilst the minimum recommended was 23. However, for conditional coverage runs, we could only manage up to 20 evolutions as stated in Chapter 4. This is because we could not account for conditional coverage complexity and available simulation test resource capabilities, which varies from verifications and SoC projects. In any case, the suggested 31 evolutions gave us an immediate starting point from which we could identify the extent of additional complications and resource usage from conditional coverage verifications. Subsequently, we were also able to reduce the GEA process to the suggested 23 evolutions and quickly calibrated the test generator to 20 evolutions from one preliminary test run. A few other minor variations between recommended and actual parameter selections exists between the work conducted in this chapter and the experiments in Section 4.11 Chapter 4, but the other parameters used match those given by our analysis.

Overall, the recommended parameter values derived from this chapter are all within acceptable margins and tolerance of the eventual values used for actual verifications. This validates our Markov model parameter selection analytical method as a credible approach. The parameter values acquired from Markov chain analysis would have been essentially equivalent to those calibrated by numerous empirical preliminary experimental runs; except that valuable time and effort to conduct these calibrations are avoided.

Summary

Besides being sufficiently accurate and saving verification resources, the Markov modelling and parameter selection approach is also reusable. Once the analytical procedure has been established as was demonstrated in this chapter, similar Markov analytical process can be adopted and applied for other different parameters of other GEA test generations and SoCs.

Our technique is limited only by overly large GEA sub-processes which are difficult to handle. Hence, carefully selected variables for the Markov chain must be used to represent only the important components of the GEA process. Otherwise, various assumptions are applied. Regardless, we have demonstrated that by using Markov modelling and analysis of various sub-flows of our GEA test generation, valuable information can be deduced to reveal characteristics of the GEA process. This aids parameter selections with useful recommended values, otherwise there are no sound parameter values one can be confident of initially employing in the GEA domain of test generation.

5.9 Conclusions

This chapter presented an analytical method in which Markov chains are employed to model sub components and processes of a GEA test generation process. Based on the derivation of these Markov chains and their subsequent analysis, information about the GEA test generation can be acquired from the analysis before performing any actual verification runs. From these Markov model analysis, test generation parameters values can then be chosen to realise desired characteristics and behaviours of the test generation, and to facilitate effective SoC verifications. Parameter selections are conducted in a structured approach instead of relying on ad-hoc preliminary test runs and empirical results.

The key to our analytical parameter selection method is to employ a divide and conquer strategy for modelling sub components and processes of the GEA test generation using multiple Markov chains. Each Markov chain can be employed to select values for certain parameters, or to build up other

Markov chains that model more complicated GEA sub processes and select other parameters. In addition, GEA test generation parameters can be encoded into the Markov chains explicitly via Markov states or state transition probabilities. This allows for parameter value selections to be attained directly from the Markov chain analysis.

CHAPTER 6. MULTI-OBJECTIVE GENETIC EVOLUTIONARY TEST GENERATION

This chapter describes multi-objective optimising strategies for test generation. Besides individual test coverage objectives, concurrent coverage metrics and other objectives such as test sizes are incorporated into the verification process. We design multi-objective genetic evolutionary methods and combine the concepts of aggregation and Pareto optimality for creating tests.

6.1 Introduction

In system-on-chip (SoC) hardware verification, the progress and comprehensiveness of testing is often measured in terms of coverage from the design under test. To enhance verification effectiveness, several coverage measures are usually employed to examine how the SoC is tested from different perspectives. Different types of coverage directed verification exercises greater variety of operating scenarios in the design. For example, our software application level verification methodology (SALVEM) can employ line, toggle and conditional coverage independently as measures of testing. High coverage results from multiple coverage metrics implies more thoroughly verified SoCs overall.

Coverage facilitates effective fitness objectives in SALVEM genetic evolutionary algorithm (GEA) test generations. Previously, our GEA test generation was limited to a single objective only. Different coverage metrics were used to drive GEA test generation in separate evolutionary processes. In this chapter, we extend SALVEM with multi-objective GEA test generation. The GEA test generation shall be driven by several different criteria, each with different and specific verification goals. This exploits the parallelism available from multi-objective optimisation; by amalgamating multiple GEA processes into a single flow. SALVEM test generation can then be directed by more than one coverage measure and other test objectives simultaneously.

For the design and demonstration of our multi-objective test generation, we adopt the following approach. In SALVEM multi-objective GEA test creations, each objective shall be mapped to a coverage metric to form a collection of multiple objectives. Besides coverage objectives, our multi-

objective GEA also incorporates test size minimisation as an additional goal during test generation. A smaller test size is important as it reduces the time and resources needed to execute the test, thus providing more efficient verification.

The goal of multi-objective GEA test generation is then to create tests that provide maximal line, toggle and conditional coverage whilst using minimal number of tests with low test sizes. With this in mind, consideration must be given to the verification requirements and system platform of the multi-objective test generator during its design.

Difficulty with multi-objective GEA

The success of multi-objective GEA depends on how objectives are incorporated into the test generation flow and verification platform. For instance, when multiple objectives are simultaneously targeted, a number of these objectives may be conflicting. In our approach, coverage and test size objectives compete against one another during the GEA process. This is because higher coverage requires excitation of more design functions, which results in greater test size. And vice-versa, a smaller test size usually restricts the extent of testing on the SoC giving lower coverage.

Conflicting objectives is a challenge common in multi-objective optimisation. We show and justify our techniques for confronting this difficulty. A strength of our multi-objective GEA is its ability to find a set of trade-off solutions which are optimised to give best overall results for all objectives, in spite of conflicts.

As an example, if the generated tests were to be reused for hardware emulation testing, the size of tests being applied becomes an important consideration. For such test platforms, the memory available to hold tests is limited. Multi-objective test generation can provide a set of tests minimised by size that is able to run on the system, and still provide the best coverage given its restricted test size. Using such coverage-versus-size optimised tests, the test engineer can identify how much coverage to trade off to enable suitably sized tests to run on the test platform. Regardless of the test chosen, the multi-objective optimised test will attain the best coverage possible for the given test size.

Such scenarios show multi-objective GEA generated test suites are highly beneficial for verification and testing of SoCs across multiple design levels; from pre-silicon simulation to post-silicon hardware testing. At each level of verification, different test requirements are needed, which can be managed by customising various objectives.

Chapter outline

The next section introduces our concept of multi-objective GEA optimisation, in particular, our use of aggregation and Pareto optimality. Following this, a detailed treatment of the multi-objective GEA test generation technique is provided. Experiments and analysis are then presented to prove feasibility and benefits of our multi-objective test generator.

6.2 Multi-objective optimisation – preliminary concepts and approach

Compared with single objective GEA optimisation, the challenge in multi-objective GEA is how to encode multiple objectives into the GEA method to conduct fitness evaluation, population selection, variation, and termination. In multi-objective GEA, no single solution exists that can simultaneously enhance the performance of each objective, especially when objectives are conflicting. The goal of the GEA process is to seek out a set of solutions that exhibit the best possible trade off performance for all the objectives.

This family of solutions are considered optimised in the sense that they perform well for certain objectives without imposing too much penalty on other objectives. The successful creation of an optimised solution set depends largely on the handling of multiple objectives to perform fitness evaluation and population selection, and termination.

For fitness evaluation and population selection, the manner in which multiple objectives are managed distinguishes the type of multi-objective GEA process from another. We devise and provide an overview of multi-objective GEA optimisation focusing on two main categories, aggregation strategy and Pareto based strategy.

We introduce and review concepts and definitions for these methods next, before describing how we combine and incorporate them uniquely into our test generation domain in Sections 6.3 to 6.7. Section 2.2.3 in Chapter 2 provided a summary survey of previous multi-objective aggregation, Pareto, and both non-aggregation and non-Pareto methods.

6.2.1 Aggregation strategy

Aggregation strategy is an extension of traditional single-objective GEA. Given that traditional GEA employs only a single fitness value, to operate within a multi-objective domain, the most straightforward approach is simply to combine multiple objective fitness values into an equivalent single fitness representation. Once the fitness aggregation process returns the single fitness value, the remaining GEA operations can be conducted as before. We generalise the fitness aggregation process as follows.

Definition 6.1 : Fitness aggregation strategy

Let x be a solution in the solution space X, where the solution points in X are applied to maximise (or minimise) the set of m multiple objective functions $f = (f_1, \ldots, f_m)$.

The aggregate fitness $f_a(x)$ from these multiple objectives is,

$$f_a(x) = g(f_1(x), \ldots, f_m(x))$$

where g is a user defined aggregation function that combines the individual objectives and transforms all the multiple fitness values into a single fitness.

□

The effectiveness of fitness aggregation depends solely on g. Besides its simplistic implementation, the other advantage with aggregate fitness strategies is that the optimisation considers all the objectives collectively as a single entity. Aggregate fitness strategies do not treat objectives separately, meaning that GEA variation and selection operations are conducted based on all objectives. No objectives are neglected, leading to more evenly optimised set of solutions for all objectives.

The downside with an aggregate approach is that one aggregation method may not be suitable for another GEA application problem. The success of the method is dependant on the types of objectives and how they can be manipulated. For example, different objectives can be ranked differently according to the needs of different GEA processes, or their application domain.

Fine-tunning the various options in an aggregate function also requires many re-runs of the GEA process, and greater input from the user rather than facilitating automation during the optimisation. Also, the aggregate method assumes all objectives can be combined using a single aggregate fitness to uncover better performing solutions for each objective. This may not always be the case, as some objectives may be highly conflicting. These kinds of objectives must be considered individually. We alleviate these shortcomings by combining aggregation with Pareto optimisation described next.

6.2.2 Pareto optimising strategy

Pareto optimality was first proposed by Vilfredo Pareto [Par96] in the late 19th century. It can be used to establish the criteria in which solutions driven by multiple goals are considered optimal. We devise a Pareto optimal approach and design Pareto methods into our test generation process to attain multi-objective GEA optimisation.

In our Pareto based GEA, the goal is to evolve solutions into a Pareto optimal test population. A Pareto optimal population contains the set of solutions which are unable to simultaneously enhance any of the multiple objectives further. These solutions have been fine-tuned such that additional improvement in any one objective requires trade off in performance of other objectives. Hence, Pareto optimal solutions are considered most effective because these solutions provide the best performance for all objectives overall.

We denote the set of Pareto optimal solution as non-*dominated* solutions. Solutions are termed non-dominated if they are not inferior to any other solutions with respect to the objectives targeted; including solutions within the same Pareto optimal set. All Pareto optimal solutions exhibit better performance in at least one objective, and attain at least the same performance for all other objectives. We define Pareto optimality and non-dominance in the context of multi-objective optimisation below.

Definition 6.2 : Pareto optimality

Let $x_1, \ldots, x_n$ be n solution points in the solution space X, where the solution points in X is applied to maximise (or minimise) a set of m multiple objective functions $f = (f_1, \ldots, f_m)$

(i) A solution $x_1 \in X$ is said to dominate another solution $x_2 \in X$, denoted as $x_1 \succ x_2$, if and only if

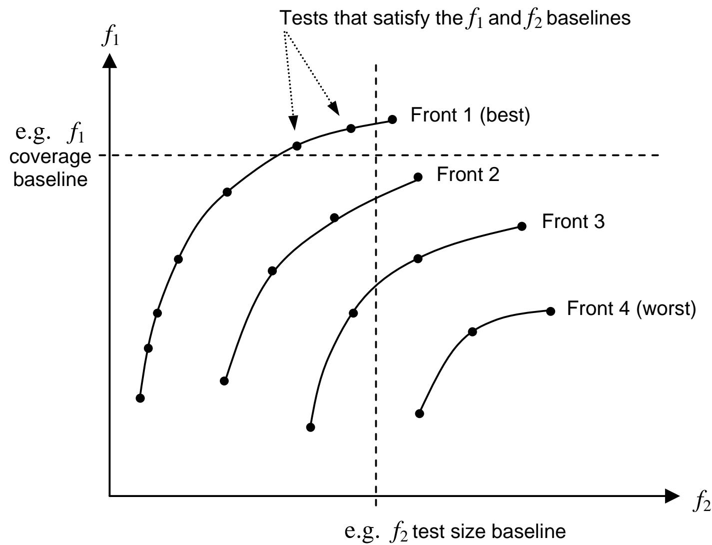
$$\forall\, i \in \{1, \ldots, m\} : f_i(x_1) \geq f_i(x_2) \quad \wedge \quad \exists\, j \in \{1, \ldots, m\} : f_j(x_1) > f_j(x_2)$$

(ii) A solution $x \in X$ is said to be Pareto optimal (or non-dominated) if there does not exist any other solution $x_k \in X$ such that x_k dominates x. That is,

$$x \succ x_k \quad \forall\, k \in \{1, \ldots, n\} \wedge x \neq x_k$$

□

The Pareto optimal set of solutions governed by Definition 6.2 can be considered graphically as shown in Figure 6.1, for a two objective optimisation process. In Figure 6.1, the goal is to maximise f_1 and minimise f_2. Hence, the set of Pareto optimal solutions are located toward the top left region of the graph. In fact, by joining the Pareto optimal solution points together with a line, the concept of a wave-like front is formed (i.e., refer to Front 1 in Figure 6.1). This Pareto optimal front *covers* (i.e., dominates) all the other solutions to its lower right.



(Each solid circle represents an input solution point, which is a test in this example)

Figure 6.1 Pareto optimal plot

Given the objectives of f_1 and f_2, the goal is to steer the Pareto optimal front toward the top left portion of the graph as much as possible. The best compromised solutions amongst the two objectives are displayed along the line forming the best Pareto front 1. The user can then examine the performance trade-offs for each objective that is attained by each solution, and choose the most appropriate solution for the application problem.

For example, in Figure 6.1, f_1 could be to maximise test coverage whilst f_2 is to minimise test size. Using the Pareto-optimal set of solutions, one can identify the most efficient test (i.e., smallest test) given a baseline coverage level. Alternatively, if the test application platform restricts the size of tests that can be executed, the highest coverage yielding test for a given maximum test size baseline can also be identified easily from the Pareto optimal front.

Multiple hierarchical Pareto fronts

Besides one Pareto optimal front, multiple Pareto optimal fronts in hierarchical order can also be plotted (i.e., Fronts 2 to 4 in Figure 6.1). By excluding the current best Pareto optimal set of solutions (in Front 1), the remaining subset of solutions can be re-examined to identify another new Pareto optimal set (Front 2) as per Definition 6.2. This then acts as the next lower hierarchical Pareto optimal front. Further Pareto optimal set of solutions and fronts can be identified by repeatedly excluding the previous best Pareto optimal solutions. This provides a series of Pareto optimal fronts in Figure 6.1, whereby the first front represents the best Pareto optimal set, and the second, third, etc fronts represent successively lower Pareto optimal solutions. The realisation of multiple Pareto optimal fronts is described in Definition 6.3.

Definition 6.3 : Multiple hierarchical Pareto optimal fronts

Let $p : P(X) \rightarrow P(X)$ be a function that identifies the set of Pareto optimal solutions to form a Pareto front according to Definition 6.2, where X is the input solution space. The input set of solutions given to p to identify non-dominated solutions, and the output set of Pareto optimal solutions produced by p, are from the power set of X denoted by $P(X)$.

(i) Let $A \in P(X)$ and $B \in P(X)$ be subsets containing solutions from X. The function p is defined as follows,

$$B = p(A) \qquad \text{such that } \forall\, x \in B,\ y \succ x \text{ is false } \forall\, y \in A\backslash B.$$

That is, the function p identifies and populates B with non-dominated Pareto optimal solutions from A, such that for all solutions in B, there is no solution in A (which is not already in B) that dominates all those solutions in B.

(ii) Let $\langle H_1, H_2, \ldots, H_l \rangle$ be the ordered tuple of l hierarchical Pareto optimal fronts that can be formed from the set of solutions in X according to the multiple objectives function f from Definition 6.2.

The hierarchical set of Pareto fronts are defined as $\langle H_1, H_2, \ldots, H_l \rangle$ such that

$$H_1 \cap H_2 \cap \ldots\ H_l = \varnothing \quad \wedge \quad H_1 = p(X) \quad \wedge \quad H_i = p(X\backslash H_{i-1}) \text{ for } i = 2, .., l$$

$H_1, H_2, \ldots, H_l$ are the Pareto optimal fronts that contain disjoint set of solutions from X. The first Pareto optimal front H_1 contains the best solutions that are not dominated by any other solutions in X. The second Pareto optimal front H_2 contains solutions which are not dominated by any other solutions not in H_1 and H_2, and so forth. H_l is the last Pareto optimal front that contains remaining solutions that do not dominate any other solution, and are dominated by other solutions in (l-1) earlier Pareto fronts.

□

Identifying the series of hierarchical Pareto optimised fronts facilitates a sorting process for the population. This Pareto optimal sorting is used for multi-objective GEA population selection as described in Section 6.5 later.

6.2.3 Pareto optimal front characteristics

Figure 6.2 shows the desired characteristic when plotting the first (best) Pareto optimal front for each evolution of test creation during multi-objective GEA. Retaining the objectives f_1 and f_2 from Figure 6.1 for the x-axis and y-axis, the three desired characteristics are as follows.

(a) The Pareto front for each evolution should expand towards the upper left region as more evolutions are conducted to optimise the population. The further the Pareto front approaches the upper left region, the more optimised the solutions are.

(b) The larger the expansion achieved by the Pareto front, the more diverse the population will be. Diversity is important because it provides the user with a greater sample of solutions to choose from, when prioritising which objectives to trade off, and also to create further new effective tests.

(c) The degree of convexity of the Pareto front curvature should be high. A convex Pareto front implies the front is more likely curved towards the target region (i.e., the upper left corner in Figure 6.2). The peak of the Pareto front is aimed towards the target region. In contrast, the curvature and peak of a concave front is directed away from the target region.

The solid line Pareto fronts in Figure 6.1 and Figure 6.2 are convex, the dotted Pareto front in Figure 6.2 is concave and is shown for example purposes only. The dashed (and solid) lines in Figure 6.2 shows the desired characteristics (a), (b), and (c) of the Pareto front during evolution as increasing number of evolutionary optimisation cycles are conducted. For multi-objective GEA, the closer a solution is to the peak of the convex Pareto front along the best Pareto front line, the more optimised it is for each of the objectives.

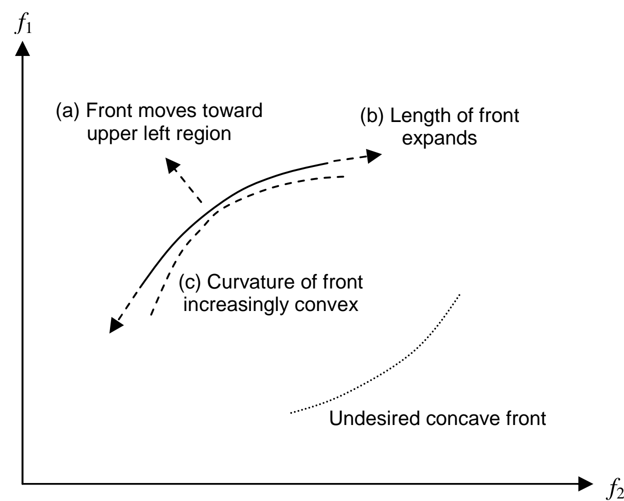


Figure 6.2 Plot of first Pareto optimal front characteristics

Pareto optimisation advantages and disadvantages

The main benefit with Pareto optimising strategies is that it attains the best compromised solutions amongst all objectives. Unlike aggregation methods, Pareto optimal methods treat each objective separately during the optimisation operation but the overall optimisation process is still performed in parallel. Solutions that perform best for some objectives will continue to be created via GEA recombination variation and maintained in the GEA process during selection; even if they are not beneficial for other objectives. This is possible because each objective can be treated independently by Pareto methods as noted in [Sch84, Sch85].

Compared with aggregation methods, treating each of the multiple objectives separately during the fitness evaluation and population selection phase enable competing objectives to be better optimised. If the fitness of multiple objectives were treated collectively, any likely improvement in one objective will lead to degradation of a competing objective, which lowers the overall fitness and optimisation of the original objective. This leads to a final solution set that may not provide the best compromise performance amongst all objectives. Pareto optimal solutions alleviates the impact of conflicting objectives by ensuring solutions that perform well for an objective can be retained for future evolutions based on an individual's fitness for that specific objective.

This is particularly useful for SALVEM test generation because our goal of attaining highest SoC coverage using minimal test size is in fact conflicting against one another. Usually, a larger test that exercises greater SoC design functionalities is needed to achieve high coverage.

The downside with Pareto strategies is that favouritism of objectives may occur. That is, the optimisation process is steered towards enhancing certain objectives whilst neglecting others. Because objectives are not considered collectively, the optimisation may be overly influenced by some objectives whose fitness is much better than other objectives. The low performing objectives could become isolated during the optimisation process.

The remaining category of non-aggregate and non-Pareto strategy was previewed in Section 2.2.3 Chapter 2, along with reasons as to why this strategy was deem unsuitable for SALVEM test generation. We preview our multi-objective GEA optimising strategy next.

6.2.4 SALVEM multi-objective GEA optimising strategy

Our SALVEM multi-objective GEA approach is based on a mixed Pareto and aggregation strategy. It leverages the benefits of both Pareto and aggregation methods whilst compensating for each other's weaknesses. In our method, multi-objectives are both considered individually and collectively during the fitness evaluation and population selection process. This novel approach is performed in three phases whereby test individuals are first inserted into appropriate Pareto optimal frontal sets, then aggregation is used to rank individuals within each Pareto optimal fronts, and finally the tests are selected into the next evolution population.

In addition, multiple objectives are divided up into subsets of objectives that conflict explicitly with one another. Non-conflicting objectives are separated from each other. This allows the aggregated fitness from each set of conflicting objectives to be considered directly. It also reduces the likelihood of any one objective dominating the other in the optimisation process. Greater opportunities are presented for objectives to be optimised from a Pareto optimal viewpoint. The outcome is a more evenly optimised solution set amongst all the objectives.

We also reduce the risk of low population diversity by conducting recombination variation between tests from different sets of conflicting objectives only. And finally, genetic drift – optimisation of only certain objectives due to stochastic test selection errors – is reduced by providing greater guidance during the test selection process from Pareto sorted and aggregated fitness values. The next section details our multi-objective GEA test generation.

6.3 Multi-objective genetic evolutionary test generation

6.3.1 Problem statement

The goal of SALVEM multi-objective GEA test generation are to create tests that simultaneously maximise the line, toggle and conditional coverage of the SoC design, whilst ensuring the size of these tests is kept as small as possible. Formally, the objectives of SALVEM GEA verification are as follows.

Definition 6.4 : SALVEM GEA test generation multi-objectives

Let x be a test from the set X which represents the input test space of SALVEM test generation. Let $f_l(x)$, $f_t(x)$, and $f_c(x)$ be fitness functions that measure the line, toggle and conditional coverage of the SoC when exercised by the test x, and $f_s(x)$ is the function that evaluates the size of the test.

For test generation, the objectives of the optimisation problem are,

Objective 1: Maximise $f_l(x)$
Objective 2: Maximise $f_t(x)$
Objective 3: Maximise $f_c(x)$
Objective 4: Minimise $f_s(x)$,

subject to $f_s(x) \leq$ M, where M is the maximum size available to hold a test for execution. (M is usually determined by the size of the executable memory in the simulator or hardware tester.)

□

The challenge with the optimisation problem in Definition 6.4 arises from the presence of conflicting objectives. A larger test that exercises greater amount of SoC functions provides higher coverage. However, the larger a test is, the more time and greater resources are needed to execute the test. The test size should be minimal so it can be reused on other test platforms with lower memory capacity. The underlying test generation goal is to create efficient tests that exercise many SoC functions to expose bugs, whilst using least number of SoC operations to reduce test sizes.

We tackle this challenge using multi-objective GEA aggregation and Pareto optimality. From a multi-objective aggregation and Pareto optimal perspective, the aim is to achieve the Pareto frontal characteristic shown in Figure 6.2, whereby the f_1 axis represents coverage and f_2 axis is test size. The multi-objective GEA flow is outlined next.

6.3.2 Overview of multi-objective GEA test generation flow

The SALVEM multi-objective GEA test generation is an extension of our single objective GEA test generator in Chapter 4. Whereas the single objective test generator was driven by one SoC coverage metric independently, the goal of multi-objective test generation is to simultaneously optimise all the coverage and test size objectives from Definition 6.4.

The prerequisite for employing GEA in any application problem is how to encode the problem within the GEA domain. For SALVEM test generation, the GEA encoding is the same for both single and

multiple objectives. To summarise, the GEA population is represented by the SALVEM test suite. Each individual chromosomal solution is equivalent to a SALVEM test program; and the genome encodings that make up each individual chromosome is given by the library of snippets. With this GEA representation, Figure 6.3 provides an overview of the multi-objective GEA test generation process.

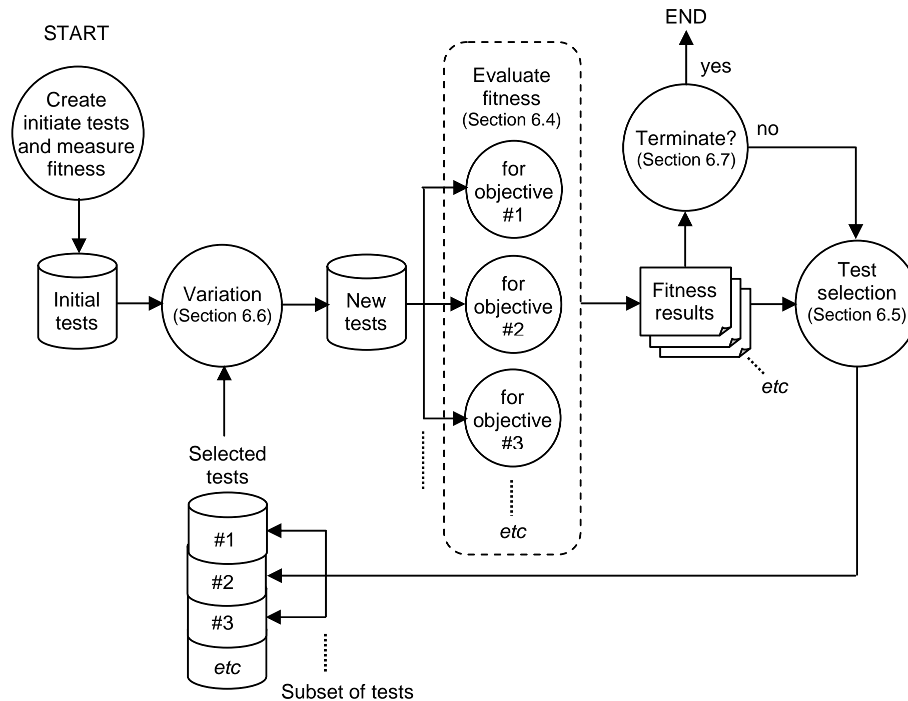


Figure 6.3 Multi-objective GEA test generation flow

The GEA process begins by creating an initial population of tests and measuring their initial fitness performance for each objective. These initial tests are created without any preference to any of the objectives.

Next, using these initial tests, variation is conducted. The variation operations carried out on individual tests, such as addition, subtraction, mutation, or replacement, is the same for both single and multi-objective GEA. Recombination variation differs for multi-objective GEA in that the tests chosen for recombination are selected by assessing its performance for certain objectives. Our multi-objective GEA promotes test diversity by recombining tests that exhibit high fitness for a variety of different objectives. Section 6.6 outlines the recombination process in greater detail.

Once variation has created the next population of tests, their fitness is evaluated in the same way as the initial test population. Fitness evaluation is conducted concurrently for all the objectives to ensure

efficiency and prevent bottlenecks in the GEA process. It is essential that any potential fitness evaluation overhead arising from measuring fitness of multiple objectives is avoided or reduced.

Using fitness results, population selection chooses which tests to retain for the next evolution cycle. Population selection differs most from single objective GEA because it must interpret multiple objectives, and establish the selection criteria by which tests are retained to optimise all objectives simultaneously. Preserving a diverse population is essential for the GEA process to continue evolving tests that caters for all objectives in subsequent evolutions. Otherwise, if tests are incorrectly chosen, the GEA process will optimise certain objectives only.

We devise a unique selection scheme that classifies tests into different subsets according to conflicting objectives criteria. The population selection ensures tests are selected to optimise every objective. Population selection is a major challenge in multi-objective GEA. Section 6.5 describes our multi-objective GEA selection policy fully.

The termination of a multi-objective GEA process is also more complex than single objective GEA. In single objective GEA, the GEA process is terminated by monitoring fitness progress of the single objective. With multiple objectives, it is not as clear when the GEA process can be deemed to have achieved optimisation. For example, the GEA process may have optimised certain objectives but has not or is unable to achieve any better performance for other objectives. Our GEA termination caters for multiple objectives and assesses whether fitness improvements for any objectives are still possible, otherwise the GEA process should not continue needlessly. Furthermore, if multiple objectives are competing, then achieving better performance for one objective may degrade other objectives. In this case, the GEA termination must employ suitable trade-off criteria to halt the GEA process only when the best achievable fitness has been attained amongst all objectives. Section 6.7 expands on these multi-objective termination difficulties and our GEA termination solutions. The following sections describe each GEA phase in more detail.

6.4 Fitness evaluation

Our fitness evaluation of multiple objectives is conducted concurrently. This is to reduce any additional evaluation time that would be required if fitness of each objective were measured sequentially. For the objectives in GEA test generation, the line, toggle and conditional coverage are all measured at the same time from a single test execution on the SALVEM platform. The test size fitness value is simple to measure and can be carried out independently after coverage evaluation.

Under a multi-computer environment, the multi-objective GEA fitness evaluation can be enhanced by performing fitness evaluation of the entire test population in parallel on each computer. Whilst fitness values are measured concurrently, these fitness values are managed individually so they can be Pareto sorted and aggregated. We describe this in the population selection and variation sections next.

6.5 Population selection

Population selection is the most important phase in multi-objective GEA because it must select tests that further enhance all multiple objectives fitness in subsequent evolutions. The goal of population selection is to identify a variety of tests that caters for different objectives, and retains them for the next evolution to undergo variation. Using such diverse selection of tests, the variation phase can then create new populations that optimise greater number of objectives concurrently. Eventually, the GEA process should produce a population of tests that is optimised for all objectives.

The multi-objective GEA population selection phase is more complex than single objective GEA because it must deal with many objectives at once. In order to simplify the test selection process, we separate multiple objectives into different subsets. This allows the selection process to operate more efficiently on less number of objectives first, instead of all objectives at the same time. Pareto sorting and aggregation based ranking are then performed on each of these subsets only, rather than having to operate on the entire set of objectives at once. The process of segregating objectives apart into appropriate groups is termed *specialisation*.

6.5.1 Objectives specialisation

The partitioning of objectives into subsets is governed by their relationship with other objectives. Objectives are examined to determine if they compete against one another. An objective is considered to compete against another objective if an increase in optimality for one objective leads to a decrease in optimality of the other objectives. Such competing objectives are considered to be in conflict with one another. The definition of conflicting objective is as follows.

Definition 6.5 : Conflicting objectives

Let f_1 and f_2 be any two objective functions, and let x_1 and x_2 be arbitrary solutions points in the input solutions space, whereby the solution points in X is applied to maximise (or minimise) f_1 and f_2.

The objectives f_1 and f_2 are said to be conflicting if,

$$f_1(x_1) > f_1(x_2) \rightarrow f_2(x_1) < f_2(x_2) \quad \wedge \quad f_2(x_1) > f_2(x_2) \rightarrow f_1(x_1) < f_1(x_2)$$

That is, an improvement in fitness for objective f_1 leads to reduction in fitness for objective f_2, and vice versa. (If the objectives are to be minimised, the comparison operators above are replaced with their inverse operators.)

□

Given the definition for conflicting objectives, the definition for specialisation to form objective subsets is as follows.

Definition 6.6 : Specialisation of objectives

Let g and h be any two different objective functions in the set of multiple objectives F of the application problem, a specialised subset of objectives O is defined as,

$$O = \{\ g \in F\ : \forall\ h \in O,\ g \neq h \wedge g \text{ is in conflict with } h \text{ according to Definition 6.5 }\}$$

That is, all objectives in the objective subset are conflicting against one another.

□

Based on Definition 6.5 and Definition 6.6, all conflicting objectives are grouped into one unique subset. Note that an objective may be conflicting against multiple objectives. In this case, such an objective would belong to more than one objective subset. If an objective do not conflict with any other objective, and cannot be allocated to any conflicting objective subset, then these objectives form a subset each, containing their own objective only.

By grouping together and only considering conflicting objectives within each subset, the population selection can optimise for these competing objectives more effectively, without having to consider all the other objectives at the same time. Essentially, the GEA process gives higher priority toward optimising these critical groupings of conflicting objectives first. Once these subsets of conflicting objectives are processed, the population selection phase can then proceed with selecting tests that perform best for each subset.

For the SALVEM multi-objective GEA process defined in Definition 6.4, objectives are specialised into three conflicting objective subsets. These subsets are identified as follows.

Definition 6.7 : Objective subsets for SALVEM multi-objective GEA test generation

Let L, T, and C be the objective subsets for SALVEM multi-objective GEA test generation such that,

$L = \{ f_l(x) , f_s(x) \}$
$T = \{ f_t(x) , f_s(x) \}$
$C = \{ f_c(x) , f_s(x) \}$
where $f_l(x)$, $f_t(x)$, and $f_c(x)$ are the objective functions defined for test generation in Definition 6.4.

□

In practical terms, Definition 6.7 implies that the three coverage metric types do not conflict with one another, but each is in conflict with the test size objective; thus optimising one particular type of coverage does not directly impinge on the fitness of other coverage objectives.

Optimising conflicting objectives such as coverage versus test size is beneficial for verification. It aims for maximum bug detection via high coverage and minimum testing time and resources through small test sizes. In fact, other objective metric such as test resources or test complexity can be included in the GEA test generation to enhance the effectiveness of verification further. But this was beyond the scope of the research in this chapter.

The intention of forming objective subsets is to facilitate a divide-conquer approach for GEA optimisation of the overall objectives set; in particular, for use during population selection. By intermixing the test population with tests that perform well for each of these objective subsets, the test population retained will be able to achieve better trade-off fitness for all the conflicting (and non-conflicting) objectives overall. This is elaborated further in our recombination variation in Section 6.6.

6.5.2 Three phase test population sort and selection

Once objectives are specialised, a three phase test population sort and the test selection using these objective subsets is conducted. The three phases are: (a) Pareto optimal sorting, (b) aggregate ranking, and (c) round-robin test selection.

The first phase performs Pareto sorting and creates multiple Pareto optimal fronts according to each conflicting objectives subset. The second phase combines the fitness of conflicting objectives within each subset using aggregation. The aggregated fitness is used to further sort tests within each Pareto optimal front created during the first phase. Once the current test population is sorted by the first two phases, tests are selected for the next evolution population in a round-robin manner. Tests are chosen from the Pareto and aggregated sorted fronts of each objective subset. Figure 6.4 shows the overall flow of multi-objective GEA test selection, followed by descriptions of the three phases.

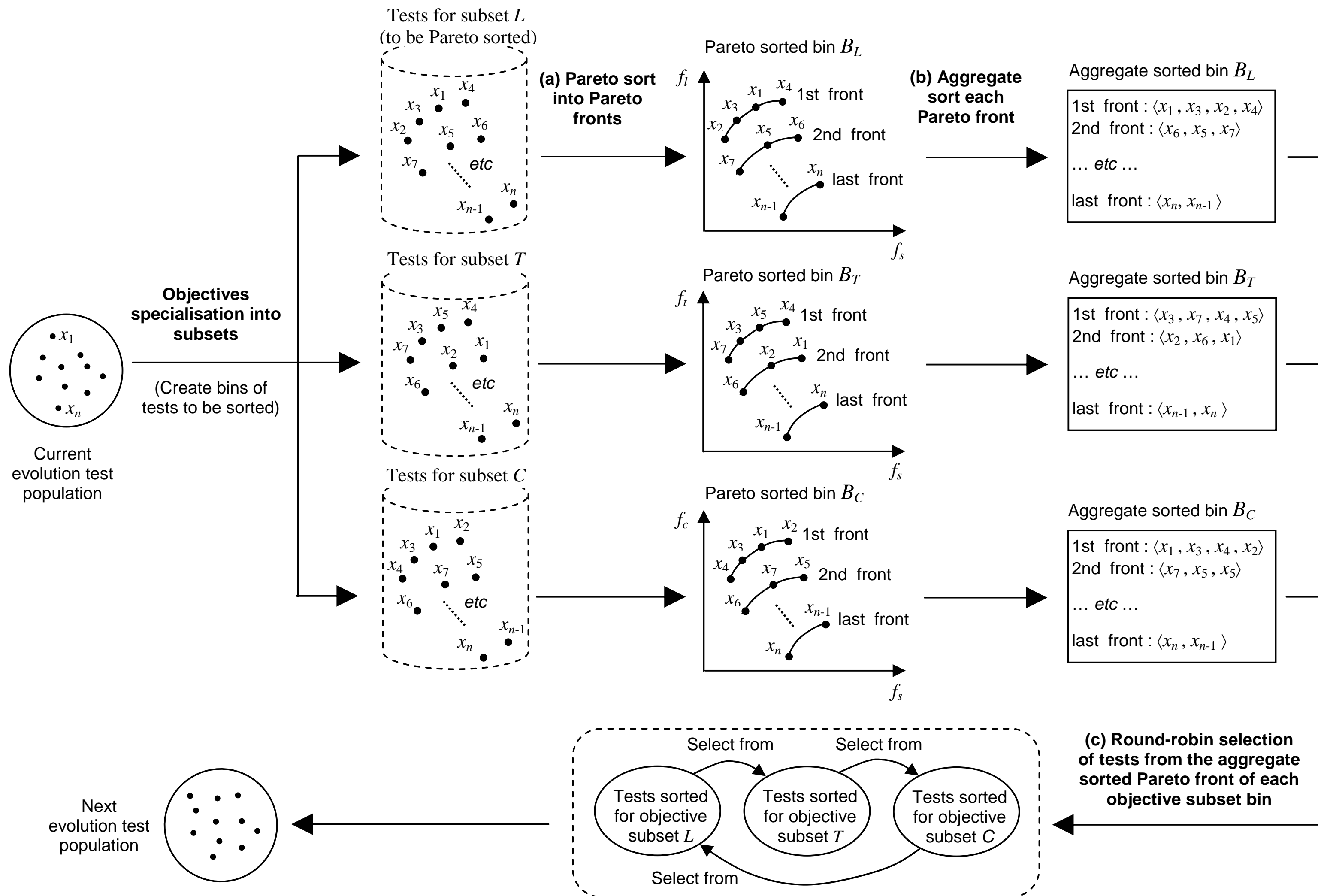


Figure 6.4 Three phase multi-objective GEA test population sort and selection

(a) Phase 1 : Pareto optimal sorting

Using the current test population, a hierarchy of Pareto optimal fronts are created for each objective subset based on the process in Section 6.2.2 (Definition 6.2 and Definition 6.3). The outcome is multiple hierarchical sets of Pareto fronts and multiple orderings of tests from the same test population – one hierarchical set of Pareto optimal fronts per objective subset. We denote each set of Pareto optimal fronts as a Pareto sorted *bin* of tests. Each Pareto bin provides a different view of the test population, and reveals the progress of the optimisation for each set of conflicting objectives.

The Pareto optimising phase is shown in Figure 6.4 after objectives specialisation. From Definition 6.7, each objective subset is used to create a set of Pareto fronts denoted as B_L, B_T and B_C; which corresponds to L, T and C objective subsets respectively. In Figure 6.4, the test population is ordered to produce different Pareto fronts depending on which objectives were used to perform Pareto sorting. A test may perform well for a set of conflicting objectives but not effectively for another set of objectives. For example, tests x_1, x_2, x_3, and x_4 perform well for the L and C objective subsets, and are placed within the first front of B_L and B_C. But x_1 and x_2 do not optimise the T objectives as effectively, and are placed in the second Pareto front of B_T. Also, from these Pareto plots, it is clear that x_3 is the best performing test because it occupies the first Pareto fronts for all objective subsets (and by aggregate ranking, it ranks first or second within each front as well). The test selection makes use of the Pareto fronts from each Pareto sorted bin to ensure best tests that cater for each objective subset are retained for further optimisation in future evolutions.

Note that each objective subset facilitates a low level GEA Pareto optimising process for the objectives within that subset. The output from each of these sub-GEA Pareto processes will be combined later in the test selection phase to continue overall multi-objective optimisation in subsequent evolutions.

Also, we point out that Pareto sort optimisation orders the test population into a hierarchical structure. Each front within a bin captures a set of trade-off tests, separated into different levels. The first front contains tests that simultaneously optimise all objectives most effectively, whilst the last front is considered the worst optimised. Each hierarchy frontal set of tests achieve different levels of compromised fitness for conflicting objectives. This allows the test selection process to choose tests from different levels, thus promoting population diversity.

After Pareto sorting, the tests within each front may be considered equivalent from a Pareto optimality perspective because tests from the same front do not dominate each other (Definition 6.2). But, in order to ensure the best tests from each Pareto front are selected, aggregation is employed to sort these frontal tests further. This is described next.

(b) Phase 2 : Aggregate sort ranking

Aggregate ranking combines the fitness functions of multiple objectives together into one function. By doing so, tests can be sorted according to a common goal rather than any single objective (which could be in conflict with other objectives). Aggregation is employed to further sort tests within each Pareto optimal front. Specifically, the conflicting objectives fitness of each objective subset is combined to sort the tests. For multi-objective test generation, the three sets of conflicting objectives, line coverage and test size (*L*), toggle coverage and test size (*T*), and conditional coverage and test size (*C*), are each combined together to form a metric for ranking tests.

The aggregation of the line coverage and test size subset *L* is defined below.

Definition 6.8 : Aggregation for line coverage and test size objectives subset

For multi-objective GEA test generation, let $f_a : X \rightarrow \mathbf{R}$ be the function that takes in a test and calculates the aggregated fitness value of the combined line coverage and test size objectives.

For a test individual $x \in X$, the aggregated fitness value is given by,

$$f_a(x) = \frac{\left(f_l(x) + \frac{\mathrm{M} - f_s(x)}{\mathrm{M}} \right)}{2}$$

where f_l and f_s are the line coverage and test size objective fitness defined in Definition 6.4, M is the maximum test size that can hold a test in the given test platform, and *X* is the input domain space of possible tests that can be created by multi-objective GEA.

□

For other objective subsets involving toggle or conditional coverage, the aggregation function is given by the same function f_a in Definition 6.8, but the f_l fitness function is replaced by f_t for the *T* subset and f_c for the *C* subset.

The aggregate ranking is shown diagrammatically in Figure 6.5 for line coverage and test size subset *L*. Similar aggregation diagram and ranking applies to other objective subsets as shown in Figure 6.4. In Figure 6.5, every hierarchical Pareto front is sorted so that each test is given a ranking according to their aggregated fitness objective value from Definition 6.8. Tests with a higher aggregate ranking will have higher likelihood of selection in the test selection phase described next.

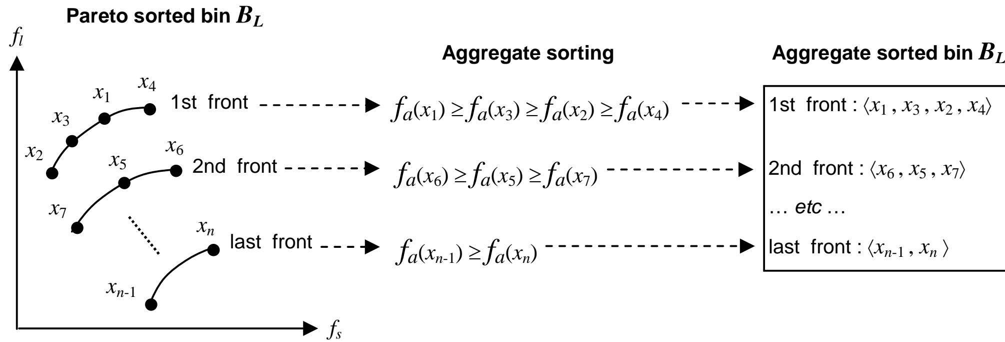


Figure 6.5 Aggregate sort ranking for line coverage and test size subset *L*

(c) Phase 3 : Round-robin test selection into next evolution population

The test selection process makes use of the Pareto and aggregate sorted bins of tests to select tests for the next evolution population. Figure 6.6 shows the flow of the test selection process, which corresponds to the round-robin based test selection at (c) in Figure 6.4.

In Figure 6.6 at step (5), if the selected test was the last remaining test in the Pareto front of a bin, the Pareto front is removed. Selection of tests from this bin will resume from the next highest Pareto front the next time tests are chosen from this bin again (i.e., at step (7)).

For step (8), once the current selection bin is assigned to select tests from, if the remaining number of tests to select into the population is greater than the number of processing bins (i.e., three bins), the test selection process repeats at step (2). Otherwise, further tests are chosen according to step (9). Step (9) selects remaining tests for the case when the number of remaining tests to be selected is less than the number of bins, and the number of remaining tests is not a multiple of the number of bins.

In step (9), unlike steps (2) to (7), the remaining tests are no longer selected in a round-robin manner from each of the bins. This is because tests cannot be selected equally from each of the bins any further. Instead, the *impartial remainder test selection policy* is used.

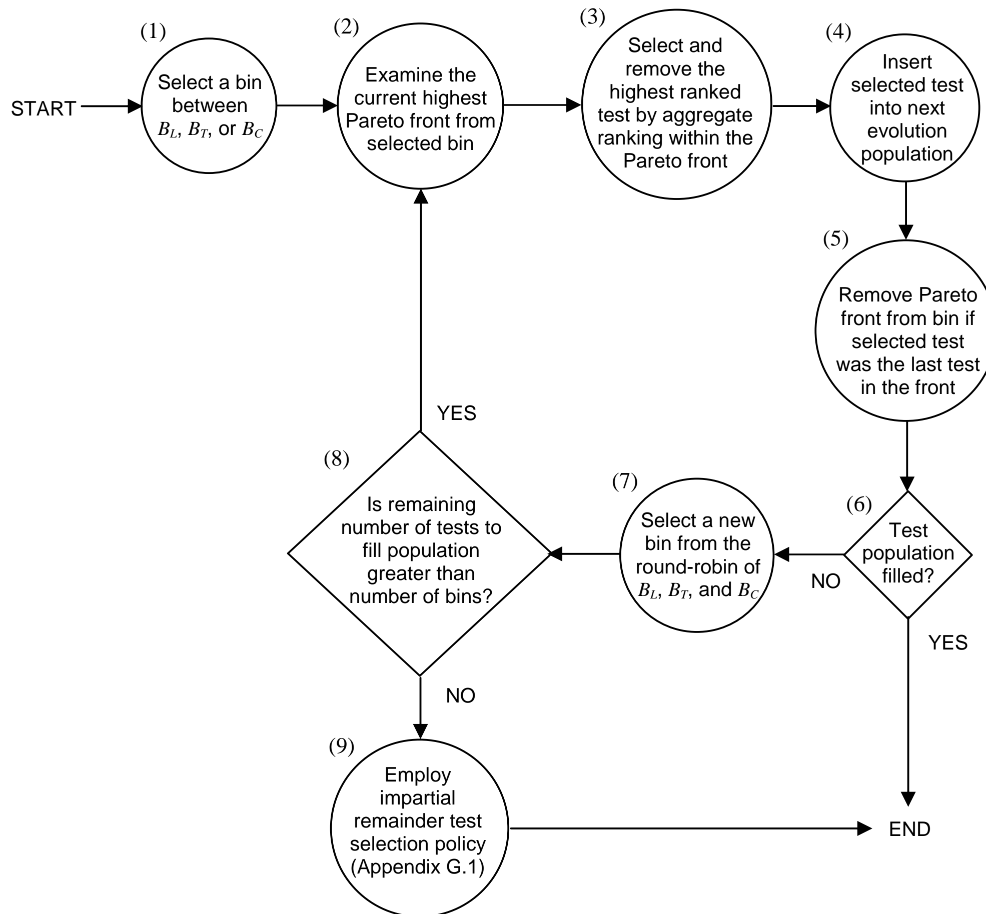


Figure 6.6 Test selection procedure

Impartial test selection policy

The impartial test selection policy ensures tests continue to be selected fairly when round-robin selection amongst all bins is no longer possible. The impartial selection policy is described fully in Appendix G.1, and also outlines the motivations and intended goals of this supplemental selection scheme. Briefly the process is as follows. For each test to be selected, tournament selection is conducted using a merged fitness metric according to Definition G.1 in Appendix G.1. The participants of the tournament are chosen fairly from each bin. That is, for each bin, the next test that would have been selected according to the round-robin method in steps (2) to (7) of Figure 6.6 is chosen for participation in the tournament. This then produces three tests that participate in each of these tournaments, with winning tests of the tournaments being inserted into the next population.

Observations for round-robin test selection phase

Based on the test selection flow in Figure 6.6, and using the test selection example in Figure 6.4, the sequence of tests selected will be, x_1, x_3, x_1, x_3, x_7, x_3, x_2, x_4, x_4, x_5, x_2, x_6, etc.

Using this test selection strategy, the highest aggregate ranked tests from the highest Pareto optimal frontal set is selected preferentially from each bin. Once a test is selected, it is removed from the Pareto front and bin. In this way, the best trade-off tests as ordered by the sequence of Pareto optimal fronts are chosen. Also, given the aggregate rankings within each front, the best performing tests for all conflicting objectives are also chosen first. Low performing tests for any objectives will not be selected from any bin, and will be released from the GEA process.

By dividing objectives into different subsets and sorting tests within each bin according to various objectives, diversity in the test population is also enhanced. The tests selected will vary greatly from an objectives optimisation perspective. The resultant test population will be effective for optimising entire set of objectives

The selection method also promotes fairness. A test is selected from each bin every iteration using round-robin sharing amongst these bins. Tests are given equal selection priority from each of the objective subset bins. Toward the end of test selection, if tests cannot be chosen evenly from each bin, a test selection policy that is impartial to all objective test bins is employed. Next, we discuss the presence and effects of duplicate test selections arising from our selection method.

6.5.3 Selection of duplicate tests

Selecting tests from multiple bins in a round-robin manner produces the side effect of introducing duplicate tests into subsequent evolution test populations. Using Pareto and aggregation sorting, each objective subset bin sorts the same test population according to different objectives. This causes the test selection process to select from different sorted ordering of identical set of tests.

During the selection process, a test can be selected multiple times from different objective subset bins if it achieved high optimised fitness for these different objectives. In general, the test selection examines a much larger test set with multiple ordering of same tests, but produces a smaller test population of pre-specified size for the next evolution. This is described by Definition 6.9 below.

Definition 6.9 : Test selection input size

Let μ be the number of tests in the current population such that μ is also the number of tests to be selected for the next evolution population. Let λ be the number of new tests that is created from the current population using variation.

If there are *b* objective subset bins of tests, the test selection process selects μ tests for the next population from $b \times (\mu+\lambda)$ number of tests.

That is, there shall be *b* set of the same tests for test selection to choose tests from. The tests chosen will depend on how they are sorted within each objective subset bins.

□

Because selection is taken from *b* sets of the same tests, this implies a test may be selected and duplicated up to *b* multiple times in the next test population, depending on how well it optimises each objective subset. For example, if a test significantly enhances many different subsets of objectives, then it is likely the same test will be selected from the bin of each objective subsets. This indicates the test is highly effective and should be preserved in the test population ahead of other tests that enhanced less objective subsets.

If a test is duplicated in the new test population, it will be chosen more often to undergo variation in the next evolution. This is desirable because the GEA process should continue to create new variations of these tests. This optimises further the multiple subsets of objectives that the original tests were effective for.

However, if there is high degree of duplication, this implies the duplicated tests achieved similar levels of optimisation for all objective subsets. Multiple objective subsets can be regarded as being reduced to a single objective subset bin, whereby tests are all sorted in a similar manner. In this case, optimisation of all objectives can be considered equivalent and the divide and conquer approach of using objective subsets may be unnecessary.

On the other hand, if the occurrence of duplicates is not excessive, the objective subset classification and round-robin test selection will ensure the entire range of multiple objectives is optimised. A range of different tests that optimises for different objective subsets will be chosen for the next test population. All objectives will be given equal priority to be optimised by the GEA process.

Managing other test duplication effects and genetic drift

In general, duplication in the test population is not undesirable. However, the test population must be monitored to properly manage test duplications, otherwise certain adverse characteristics becomes prevalent.

The downside with existence of duplicate tests is that the same best performing tests may be selected too many times. This reduces the genetic snippet diversity in the test population. Variation is unable to evolve new tests that are distinctly different, hence new areas of the objective search space cannot be uncovered easily. Eventually, the test population will become too similar and stagnated, focusing solely within the same objective search space. Optimisation of objectives will not improve and the GEA process will terminate earlier.

Duplication also affects genetic drift in the test population. Genetic drift is the lost and extinction of certain genetic characteristics in the test population due to various stochastic factors. Genetic drift causes undesired favouring of certain genome and objectives such that the population becomes overly dominated by certain types of individuals over time. The consequence is uneven optimisation of objectives and an ineffective multi-objective GEA process. Duplication of tests may accelerate genetic drift such that any potentially useful snippet sequence genome in tests will be lost, thereby preventing certain objectives from improving further.

We address diversity issues and genetic drift by lowering and restricting the amount of duplication possible in the test population. During initial stages of the GEA process – for example during the first quarter of GEA evolutions – each test may be duplicated at most once. This prevents the test population from saturating with the same tests, resulting in pre-mature convergence of GEA optimisation. Duplications are specifically restricted during early evolutions because these initial tests would not have undergone sufficient variation to produce diverse test populations yet. Hence, during early GEA, the test population can be highly susceptible to over duplication.

In addition, throughout the remainder of the GEA process, duplication is constrained. The number of times a test may be duplicated is limited by the number of objective subset bins that is defined. And finally, the GEA process imposes a check at every evolution that ensures the number of duplicate tests is less than half the size of the population.

Besides these duplicate control mechanisms, other GEA features also reduce excessive duplication indirectly. For instance, our GEA mimics real-life biological evolutions by enforcing a limited lifespan on tests. That is, a test can only live for a finite number of evolutions, and hence, can only be

duplicated a limited number of times throughout the GEA process. Therefore, duplicated tests have limited lifespan as well.

Also, recall that our test selection procedure employs aggregate ranking of tests within each Pareto front of the objective subset bins. A unique test from each of these fronts is chosen each time, from the highest to the lowest ranked tests within the front. Other GEA Pareto optimal based test selection employ a random approach, randomly selecting a test from the front instead. A random selection may result in the same test from the front to be chosen many multiple times; thereby further increasing the occurrence of duplicates not just from objective subset bins, but from different fronts as well.

6.5.4 Test selection summary

The goal of our Pareto optimising and aggregation phase is to rank tests so that test selection can be conducted according to different subsets and priority of certain objectives. It addresses a common issue (described in [TKK96]) whereby ranking skews tests to focus on certain conflicting objectives only, but ignores remaining objectives. Our approach alleviates this problem by identifying subsets of conflicting objectives and managing these smaller groups of objectives individually first. In this way, the likelihood of one objective dominating another is lower, and all objectives are guaranteed for optimisation.

Additionally, by employing round-robin selection to select tests from the Pareto fronts of each objective subset, diversity of tests that cater for both conflicting and non-conflicting objectives is maintained. Appendix G.2 elaborates further our motivations for promoting diversity in multi-objective GEA test generation.

In contrast to our selection technique, other methods that manipulate the test selection process focus on enhancing objective fitness using prioritisation through elimination. Objectives that do not exhibit favourable fitness after optimisation are simply discarded, which defeats the original purpose of multi-objective optimisation in some ways. The other problem with this strategy is that it requires user intervention whereas our method promotes automation.

6.6 Variation

In multi-objective GEA variation, operators such as addition, subtraction, mutation, and replacement can be conducted as per single objective GEA. Recombination however, must take into account multiple objectives. The goal of recombination is to promote test diversity and retain test snippet genome sequences that target all objectives.

To achieve such tests, recombination combines existing tests, each of which contain the desired variety of snippet genome for all objectives overall. Recombination must select candidate parent tests that enhance different objectives. Then, by recombining these parent tests, the resultant children tests will cater for a greater variety of objectives based on the collective set of objectives snippet genome from its parents.

Given these requirements, the strength of multi-objective GEA recombination lies with the parent selection scheme. The parents should not longer be tournament selected randomly as was the case in single objective GEA. Instead, the parent selection scheme must ensure that parent tests do not cover the same set of objectives.

To facilitate this, recombination combines tests from different objective subset bins, each of which was sorted for different conflicting objectives, as described in Definition 6.7. The L, T, and C objective subset bins of tests were previously partitioned during the population selection phase. The tests within each bin covers different objectives, hence is ideal for recombination. Recombination simply selects parents from each of these bins to participate in the inter-breeding test creation process. Each test children will then contain some facet of their parents to cater for different objective subsets, thereby optimising a wider range of objectives. The recombination parent test selection and children test creation are as follows.

Definition 6.10 : Multi-objective GEA recombination

Let $B_1, B_2, \ldots, B_k$ be k objective subset bins that group the tests differently depending on the objectives assigned for each subset, such that

$\forall B_i : B_i \subseteq X$, for $i = 1, 2, .., k$ where X is the set of tests in the current population.

Let $x_1, x_2, \ldots, x_k$ be k parent tests, each of which are selected from the corresponding objective subset bins, $B_1, B_2, \ldots, B_k$ respectively. That is,

$\forall x_i : x_i \in B_i$, for $i = 1, 2, .., k$

Let $y_1, y_2, \ldots, y_k$ be k children tests that are produced by performing recombination on the k parent tests, such that

$$(y_1, y_2, \ldots, y_k) = Recomb(x_1, x_2, \ldots, x_k)$$

where $Recomb : X^k \rightarrow X^k$ is defined as the recombination operator based on same definition for single objective GEA from Definition 4.11 Chapter 4.

□

For example, in a two point recombination process, two parent tests are selected from any two different objective subset bins (i.e., $k = 2$). Next, two crossover points are identified within the parent tests, one crossover point from each parent. The first new children test is created by combining the genetic snippet sequence of the first parent test from the first snippet to its crossover point, with the snippet sequence of the second parent test from its crossover point onwards to its last snippet. The second children test is produced vice versa.

For our multi-objective GEA test generation, the objective subset bins correspond to objective subsets L, T, and C so that $k = 3$. The recombination operation is performed using three crossover points, one chosen within each test. The three test children are each created by intermixing any pre or post crossover point genetic snippet sequences of a parent, with the pre or post crossover point snippet sequence of another parent. Note that the recombination operation is not restricted to k crossover point recombination only; but recombination with more than k crossover points complicates the variation, without much significant improvement in fitness.

The recombination strategy of employing parent tests from different objective subset bins promotes a form of speciation. Speciation inter-breeds a range of individuals to form new types of species in biological processes. Similarly, the speciation in our recombination is to recombine new species of tests from existing tests that optimises different objectives.

The relationship between parent tests is important for the success of recombination. Tamaki [TKK96] was the first to study these relationships and concluded that random selection of parent tests is not effective. Instead, parents should differ in terms of their test composition as much as possible. A simple approach was to select parent tests with the greatest difference in fitness. This is still a weak strategy because it is too simplistic and does not encode any information about how diverse the selected parent tests actually are.

Our strategy is to use parent tests that perform well for different objectives as ordered by our objective subset bins. This produces tests that cover all objectives overall. Our parent tests still exhibit

differences in terms of objectives fitness, but each parent also performs best for at least one of the objectives specifically. Each parent's genetic snippets are responsible for optimising certain objective further in the newly created children test.

In addition to Tamaki's observations, experiments later in Sections 6.9 and 6.10 also show our selection scheme is more effective than random selection of parent tests.

6.7 Termination

The presence of multiple objectives requires our GEA process to be terminated only when all objectives have been optimised as best as possible. More importantly, termination should only be triggered when no further improvements are possible for all the objectives. To identify the termination conditions, information from Pareto optimal fronts are examined. Using Pareto optimality information, the criteria for GEA termination are as follows.

(i) Drifting and levelling off of the Pareto front.

(ii) Reduction in gap between the best Pareto front and worst Pareto front.

Other criteria and methods can be used, such as the amount of test duplication during the test selection stage, and adaptations of single objective GEA terminations; but this is beyond the scope of this chapter.

6.7.1 Drifting and levelling of Pareto fronts

In multi-objective GEA, the Pareto fronts that are formed at every evolution are useful for analysing the status of GEA test generation. The Pareto fronts show how effective objectives have been optimised. By examining the Pareto fronts over time, the progress of the GEA optimisation process can be assessed. When Pareto fronts indicate no further objectives fitness improvement, the GEA process is terminated.

Pareto front behaviour

In order to establish termination conditions from Pareto fronts, the expected behavioural characteristics of Pareto fronts must be examined. Initially, at the start of GEA, the Pareto front will expand rapidly outwards towards the desired optimal region where all objectives are best optimised. In the beginning, individuals can gain higher performance for objectives easily because much of the objective search space is still unexplored. Over time, attaining objective fitness improvements becomes difficult because many easily uncovered objectives test space have already been found. Conflicting objectives also become harder to cater for. Subsequently, the expansion of the Pareto front will decrease.

Eventually, the Pareto front will no longer expand. Instead, it flattens out and drifts away from this best optimal region for all objectives. When this occurs, the GEA process can be considered stagnated. The test generation is unable to optimise all objectives' fitness functions further. Improvements in objectives is difficult to achieve without degradation in other objectives, hence the Pareto front cannot expand outward. Consequently, the Pareto front will shift in the directions of fitness improvement for some objectives, but away from optimality for other objectives. In multi-objective GEA test generation, termination will be triggered when the Pareto front continues to drift away in the opposite direction from the desired optimal region of the Pareto plot.

Figure 6.7 demonstrates the lifecycle of Pareto fronts during a GEA process. The lifecycle applies for a two objective GEA using the same objective functions from Definition 6.4 and the example diagrams (Figure 6.1 and Figure 6.2) in Section 6.2. For GEA test generation, recall that the goal is to maximise line (f_l), toggle (f_t), or conditional (f_c) coverage objectives, and minimise the test size objective (f_s). Hence, the Pareto front should expand towards the upper left corner of the plot area as evolutions are conducted. This area is the desired region for the Pareto fronts. The front *expansion* characteristics are demonstrated by the solid Pareto front lines in Figure 6.7.

During the GEA process, the convexity curvature of the Pareto front will also gradually decrease as fitness improvements for both objectives become harder to achieve. Eventually, both conflicting objectives cannot be optimised concurrently, and the Pareto front will start to flatten out and drift away from the desired upper left region. This is represented by the dashed lines in Figure 6.7.

The Pareto front prior to when it starts drifting away from the desired region is considered the best Pareto front. This best Pareto front is shown as a thicker solid line in Figure 6.7. This Pareto front corresponds to the best optimised test population, and from this evolution onwards, it becomes difficult

to achieve any further significant fitness improvements. When Pareto front begins to drift, this signals possibility of GEA termination.

The reason why the Pareto front continues to drift is because the GEA process is unable to seek out new test individuals that can optimise both objectives. Nevertheless, the variation process will continue to create new tests that are larger in size but cannot achieve higher coverage. Given the lack of coverage improvement and continual drifting, the eventual flattened Pareto front of the last evolution is shown as the dotted line in Figure 6.7. Appendix G.3 examines in more detail the reasons for Pareto front drifting.

In the GEA test generation, Pareto fronts are formed and monitored for each of the line coverage versus test size, toggle coverage versus size, and conditional coverage size objective subsets. The GEA process detects when contraction of the Pareto front occurs, and when the Pareto front flattens out. If Pareto fronts are no longer expanding, termination can be triggered.

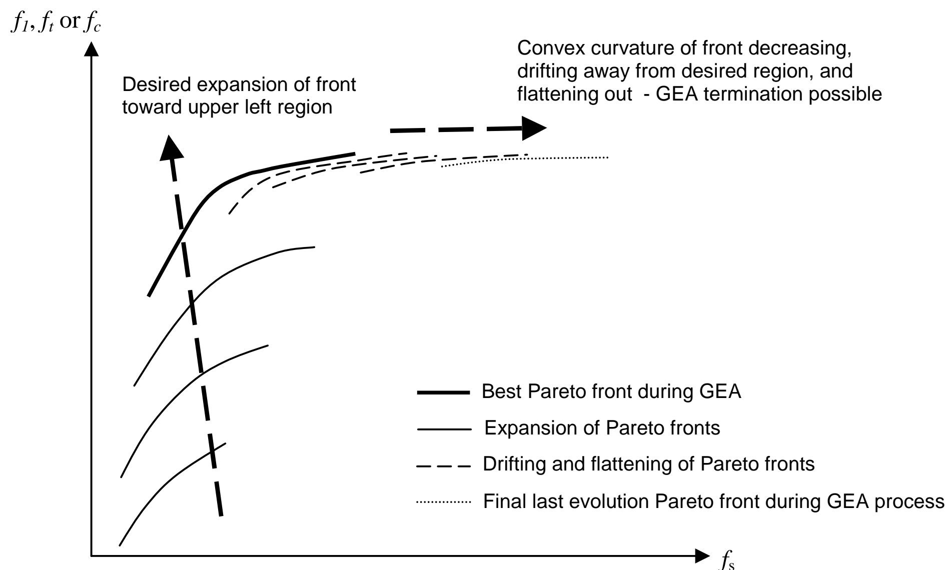


Figure 6.7 Pareto front characteristics during GEA lifecycle

GEA termination by Pareto front drift and levelling

To detect Pareto front drifting, levelling and convexity curvature reduction, the slope of the best Pareto front is analysed at the end of each evolution. By calculating the slope of consecutive pairs of test data points that form the Pareto front, the curvature and flatness of the Pareto front can be determined. If the slopes for all consecutive tests on the Pareto front are below a threshold value, this means that the

Pareto front has entered a stage in which further optimisation improvement of objectives is unlikely. The Pareto front slope measurement concept is shown in Figure 6.8 and defined in Definition 6.11.

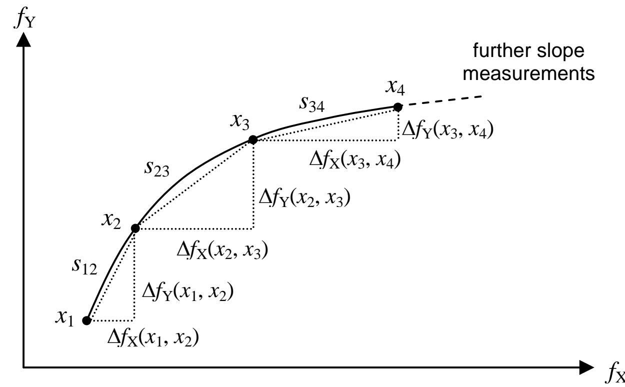


Figure 6.8 Slope measurements of consecutive tests data points along Pareto front

Definition 6.11 : GEA termination by Pareto front drift and levelling

(i) Let the tuple $\langle x_1, x_2, \ldots, x_n \rangle$ be the n number of test data points that form the best Pareto front in a given evolution. Let s_{ab} be the slope between two consecutive test data points x_a and x_b of the Pareto front. The slope is defined as,

$$s_{ab} = \frac{\Delta f_Y(x_a, x_b)}{\Delta f_X(x_a, x_b)} = \frac{f_Y(x_b) - f_Y(x_a)}{f_X(x_b) - f_X(x_a)},$$

where f_Y and f_X represent the objective fitness function of the y-axis and x-axis of the Pareto front plot respectively, and $b = a + 1$ for $a = 1, 2, .., n - 1$.

For GEA test generation, f_X is the test size objective fitness function f_s, and f_Y can be the line, toggle, or conditional coverage objective fitness functions f_l, f_t , or f_c, depending on which objective subset the Pareto frontal plot is for.

(ii) Let G be the threshold value which is compared to the slopes of consecutive test data points on a Pareto front defined in (i). The Pareto front is considered to have levelled off with low curvature if,

$$s_{ab} < G \qquad \forall\ s_{ab} \text{ such that } a = 1, 2, .., n - 1 \text{ and } b = a + 1$$

That is, the Pareto front has drifted and flattened off if the slopes of all consecutive pairs of tests data points are below G. G is usually assigned a very small value (e.g. below 0.2) to check that the Pareto front has contracted close to a flattened level.

□

Based on Definition 6.11, the termination scheme is as follows. When all Pareto fronts from each of the objective subsets *L*, *T* and *C* have levelled off and drifted according to Definition 6.11 (ii), the GEA process is considered to have deteriorated without improving any objectives. Therefore, if the slopes of the Pareto fronts continue to lie below the threshold for a number of consecutive evolutions, then GEA termination is triggered. The number of consecutive evolutions to check is typically set to 5 evolutions for multi-objective GEA test generation.

6.7.2 Reducing Pareto front gap

Another GEA termination strategy that involves Pareto fronts is to examine the gap between the best and the worst hierarchical Pareto fronts at the end of each evolution. Recall from Section 6.5.2 that the best Pareto front captures tests that dominate all other tests in the population, whilst the worst Pareto front is the set of remaining tests that hold lowest fitness for objectives in the population. Given that the intermediate tests between the best and worst fronts represents the range of objective fitness attained by the test population, the test population diversity corresponds to the gap distance between the best and worst fronts.

The wider gap the best and worst fronts are, the more diverse the tests are. A more diverse test population enables the GEA variation to choose greater variety of tests to seek out other regions of the objective test space. This indicates the GEA process can continue to improve objective fitness. If the best and worst Pareto front gap decreases, this implies the test population is beginning to converge. When the gap becomes small, the low variety of tests signifies the GEA process is unable to select and sufficiently vary different types of tests to improve objectives fitness further; and GEA termination should be invoked.

A wider gap also shows the degree of improvement for concurrent optimisation of objectives during the current evolution. If the gap is small, not much improvement between the best and worst tests was achieved in the current population. In this case, the GEA process should be terminated as well.

Figure 6.9 shows the concept of measuring the gap distance between the best and worst Pareto fronts. During evolution, the Pareto front gap distance will decrease as fitness of objectives from tests in the population converges to the best possible values the GEA process can attain. If the Pareto front gap distance for all the objective subset dips below a threshold value, then the GEA process is terminated. Figure 6.9 shows the gap distance is calculated between the average fitness points on both the best and

worst Pareto fronts. Note that the average fitness points may not lie on their respective Pareto fronts, they serve to provide approximate proximity locations as to where their fronts lie.

Assuming equivalent objective Pareto frontal plots (Figure 6.1 and Figure 6.2) from Section 6.2 for each of the *L*, *T*, and *C* objective subsets, the derivation of the average fitness points, measurement of the Pareto front gap distance, and the GEA termination conditions are described in Definition 6.12.

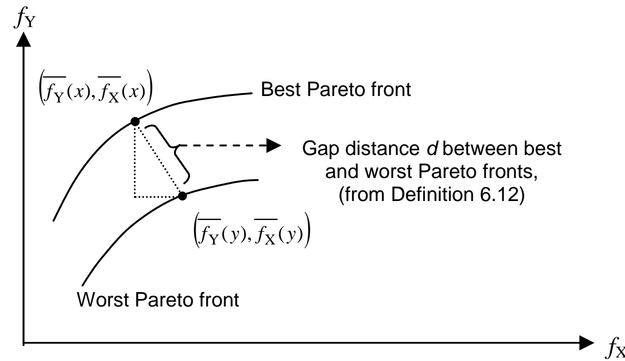


Figure 6.9 Gap distance between best and worst Pareto fronts for GEA termination

Definition 6.12 : GEA termination by best and worst Pareto front gap

(i) Let the tuple $\langle x_1, x_2, \ldots, x_n \rangle$ be the *n* number of test data points that form the best Pareto front in a given evolution. Let the tuple $\langle y_1, y_2, \ldots, y_k \rangle$ be the *k* number of test data points that form the worst Pareto front in the same evolution.

Let $\overline{f_Y}(x)$ and $\overline{f_X}(x)$ be the average fitness of the f_Y and f_X objectives of tests in the best Pareto front, and $\overline{f_Y}(y)$ and $\overline{f_X}(y)$ be the average fitness of the same objectives of tests for the worst Pareto front, and are defined as follows:

$$\overline{f_Y}(x) = \frac{\sum_{i=1}^{n} f_Y(x_i)}{n}, \quad \overline{f_X}(x) = \frac{\sum_{i=1}^{n} f_X(x_i)}{n}, \quad \overline{f_Y}(y) = \frac{\sum_{i=1}^{k} f_Y(y_i)}{k}, \text{ and } \overline{f_X}(y) = \frac{\sum_{i=1}^{k} f_X(y_i)}{k}$$

where f_Y and f_X represent the objective fitness function of the y-axis and x-axis of the Pareto front plot respectively.

□

Using Definition 6.12, let the $\left(\overline{f_Y}(x), \overline{f_X}(x)\right)$ and $\left(\overline{f_Y}(y), \overline{f_X}(y)\right)$ coordinates be the averaged objective fitness test data point on the Pareto front plot for the best and worst Pareto fronts respectively.

Measuring the distance between these two points, we attain the best and worst Pareto fronts gap distance result, $d = \sqrt{\left(\overline{f_{Y}}(x) - \overline{f_{Y}}(y)\right)^2 + \left(\overline{f_{X}}(x) - \overline{f_{X}}(y)\right)^2}$.

Using *d*, the termination scheme is as follows. At the end of every evolution, when the gap distance between the best and worst Pareto fronts of each of the objective subset falls below a threshold distance value *D*, the GEA process can be terminated.

Recall for GEA test generation that f_X is the test size objective fitness function f_s, and f_Y can be the line, toggle or conditional coverage objective fitness functions f_l, f_t or f_x respectively, depending on which objective subset the Pareto frontal plot is for. The threshold value *D* is set to a low value, typically no more than 0.5.

6.7.3 Multi-objective GEA termination – summary

Besides Pareto front based termination schemes, a number of traditional single objective GEA termination methods can also be employed for multi-objective GEA. For example, termination can be triggered after a fixed number of evolutions, or when an objective does not gain fitness improvement for a given number of consecutive evolutions. For multi-objective GEA test generation, the above multi-objective Pareto front termination schemes (Sections 6.7.1 and 6.7.2), along with termination by a fixed number of evolutions to limit the duration of the GEA process is employed.

Our GEA termination indicates when objectives fitness improvement is unlikely, and the test population has matured to achieve the best fitness results possible. If additional objectives improvement is sought, a new test population and GEA process must be conducted.

In Appendix G.4, we summarise our multi-objective GEA termination scheme based on test selection duplications. This termination method was not examined further and remains as future work.

6.8 Experimental setup and procedure

The multi-objective GEA test generation method was implemented as a test generation tool. It is integrated into the SALVEM verification platform as a module to create GEA derived tests for simulation on the Nios SoC. Figure 6.10 shows the integration of the test generation tool within SALVEM, making use of snippets genome from the snippets library to generate test programs. The

majority of modifications to the verification system was limited to and encapsulated within the test generation tool itself. The only significant change required in the SALVEM platform is for the fitness evaluation of multiple objectives. Specifically, the Synopsys VCS simulator used to simulate execution of test programs must be configured to measure the line, toggle, conditional coverage and test size simultaneously. The entire SALVEM verification platform was implemented on Linux Redhat 7 powered by 2.66 GHz Intel Core 2 CPU and 2GB memory. The same Linux platform was also employed to conduct test generation experiments.

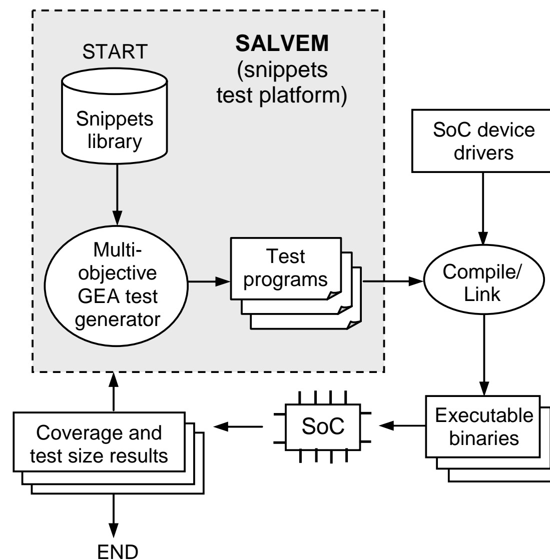


Figure 6.10 Multi-objective GEA test generation tool within SALVEM platform

Our test generator and experiments are configured in a similar manner to that of single objective GEA test generation in Section 4.11. This is because many GEA parameters and assumptions used for both single objective and multi-objective GEA are generally equivalent. In multi-objective test generation, the parameter selections can be conducted in the same way for each of the multiple objectives, and produces similar GEA parameter selections. Parameters can be chosen with empirical results from preliminary test runs, or using the analytical Markov chain methods in Chapter 5. Table 6.1 shows the GEA parameters used for multi-objective GEA test generation runs.

The multi-objective GEA test generation experiments are conducted in two phases. The first experimental phase presented in Section 6.9 examines characteristics of the multi-objective GEA test generation process. The goal is to gain a better understanding of how various attributes affect test generation, and to check the actual multi-objective optimisation behaviours against expected outcomes. Specifically, we use the Pareto fronts to analyse test population diversity, test duplication, and the GEA selection and termination policies employed.

Table 6.1 Multi-objective GEA test generation parameters

Maximum number of evolutions	30
Parent population size (μ)	30
Children population size (λ)	15
Initial number of snippet genome per test	0
Maximum lifespan of test individuals (number of evolutions)	5
Number of simultaneous objectives fitness to evaluate	4
Pareto front threshold slope value (*G*)	0.2
Best and worst Pareto front threshold gap distance value (*D*)	0.5

The second phase of experiments presented in Section 6.10 is to assess the performance of the multi-objective GEA test generation against other verification schemes. In particular, we compare against single objective GEA test generations, random-biased method, and application specific manual testing strategies.

6.9 Multi-objective GEA experiments and analysis

6.9.1 Multi-objective Pareto fronts – general observations

Figure 6.11 to Figure 6.16 show the Pareto front plots from the multi-objective GEA test generation process. In Figure 6.11 to Figure 6.13, the best Pareto fronts from each evolution are plotted for line, toggle and conditional coverage, achieved against test sizes. These best Pareto front plots show the fitness trade offs possible between coverage and conflicting test size objectives.

In Figure 6.14 to Figure 6.16, the entire set of Pareto fronts per evolution is displayed for selected evolutions. This reveals the change in Pareto front characteristics during the test process. The Pareto fronts are plotted at evolutions 1, 5, 10, 15, 20, 25 and 30 for each of the line, toggle and conditional coverage against test size objective subsets.

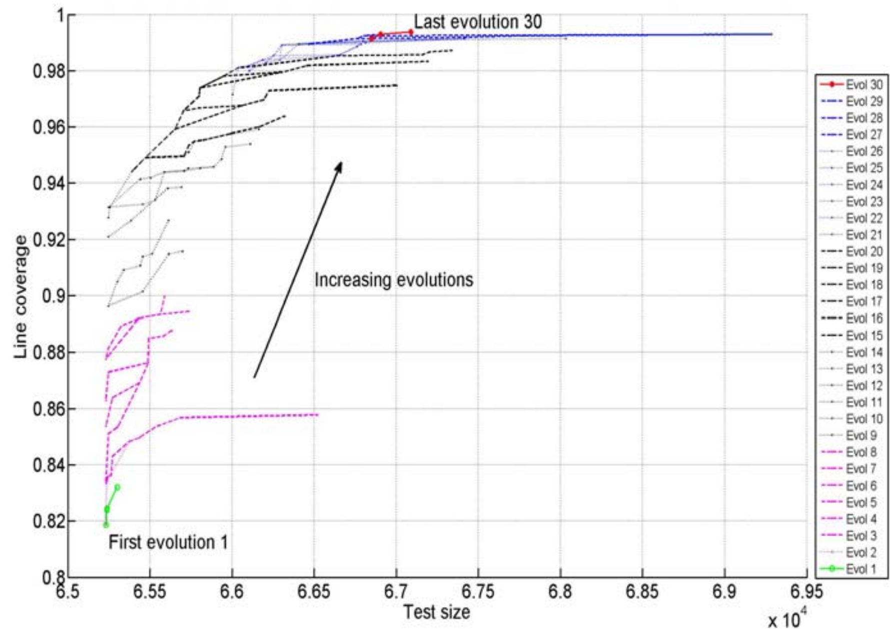


Figure 6.11 Best Pareto front from every evolution (line coverage vs. test size)

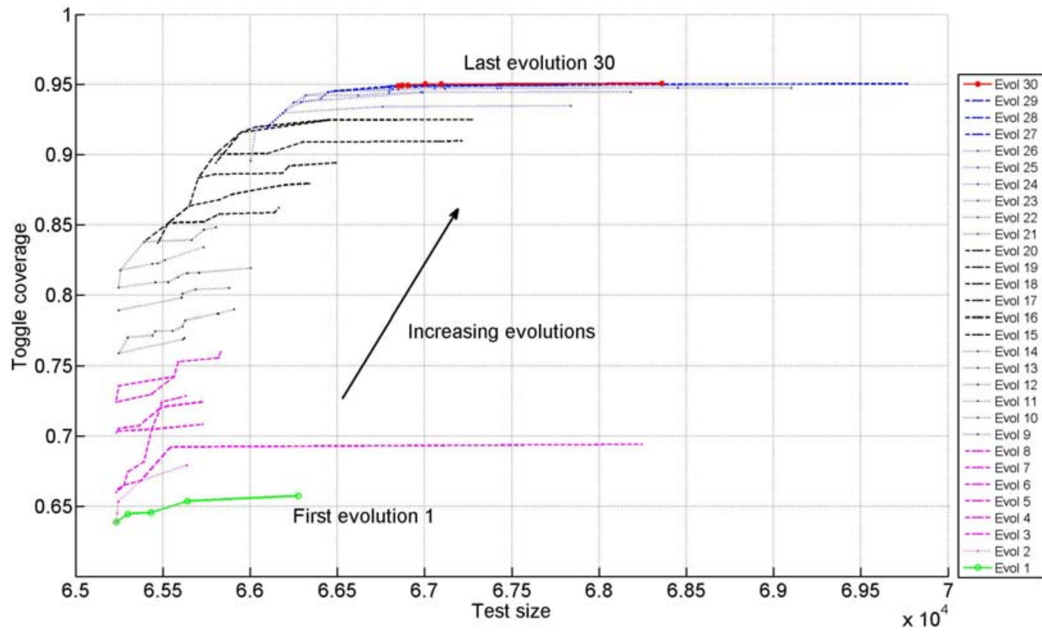


Figure 6.12 Best Pareto front from every evolution (toggle coverage vs. test size)

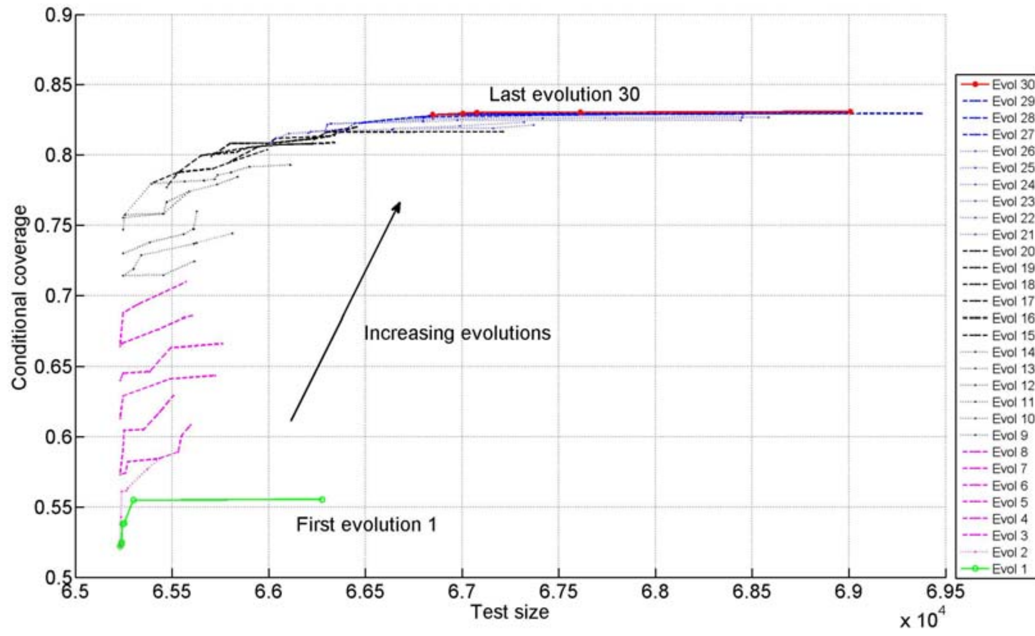


Figure 6.13 Best Pareto front from every evolution (conditional coverage vs. test size)

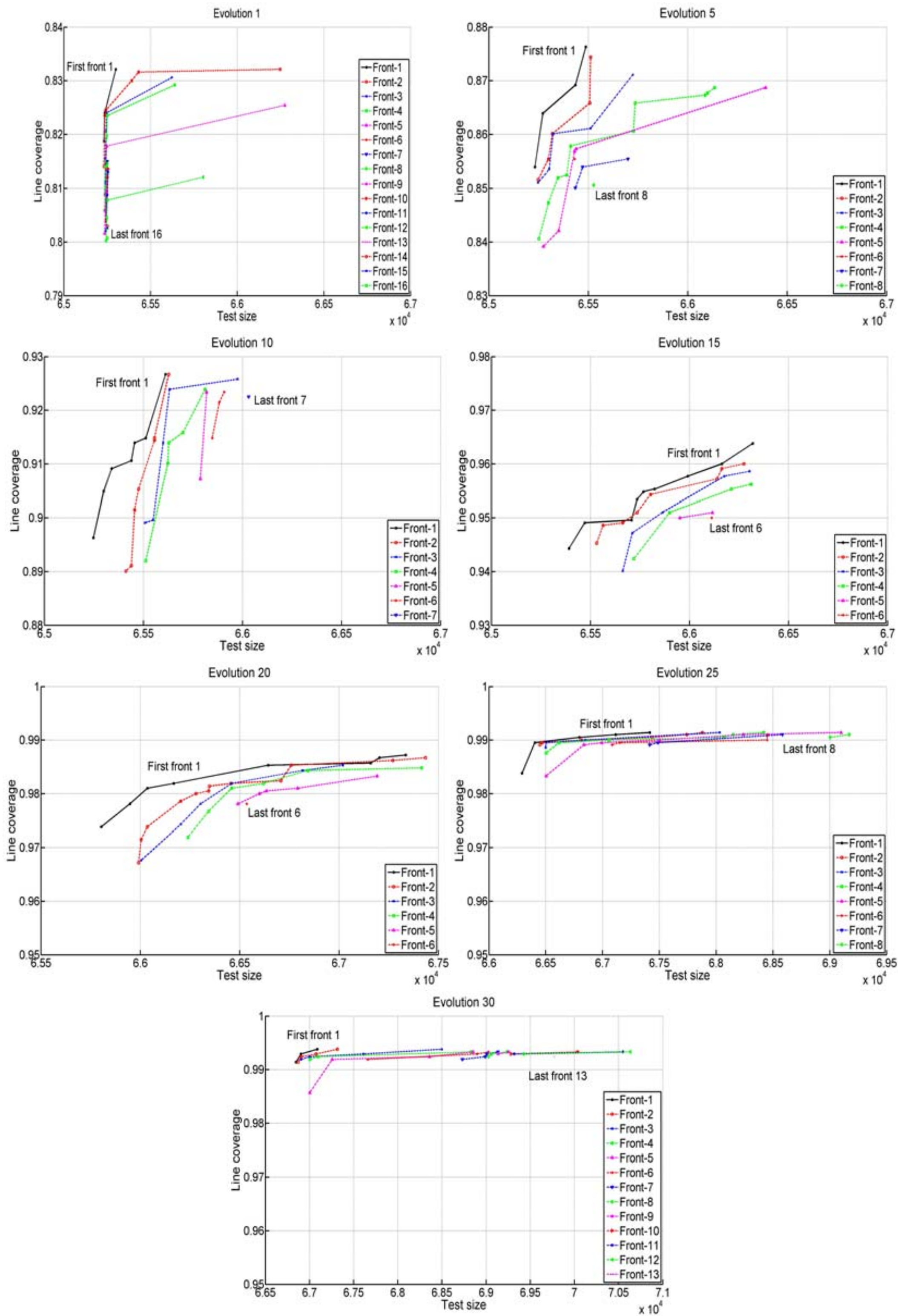


Figure 6.14 Pareto fronts at selected evolutions during GEA (line coverage vs. test size)

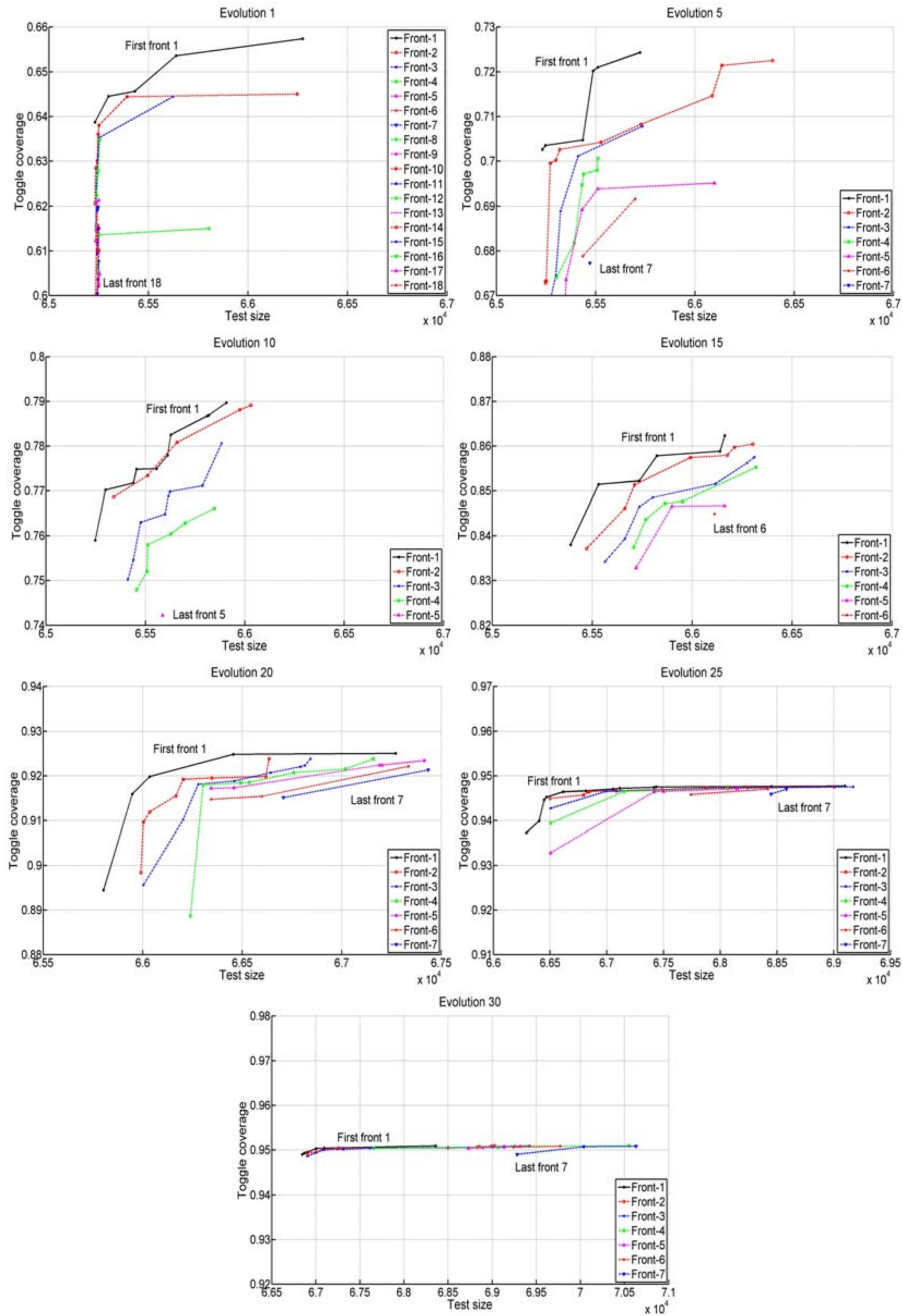


Figure 6.15 Pareto fronts at selected evolutions during GEA (toggle coverage vs. test size)

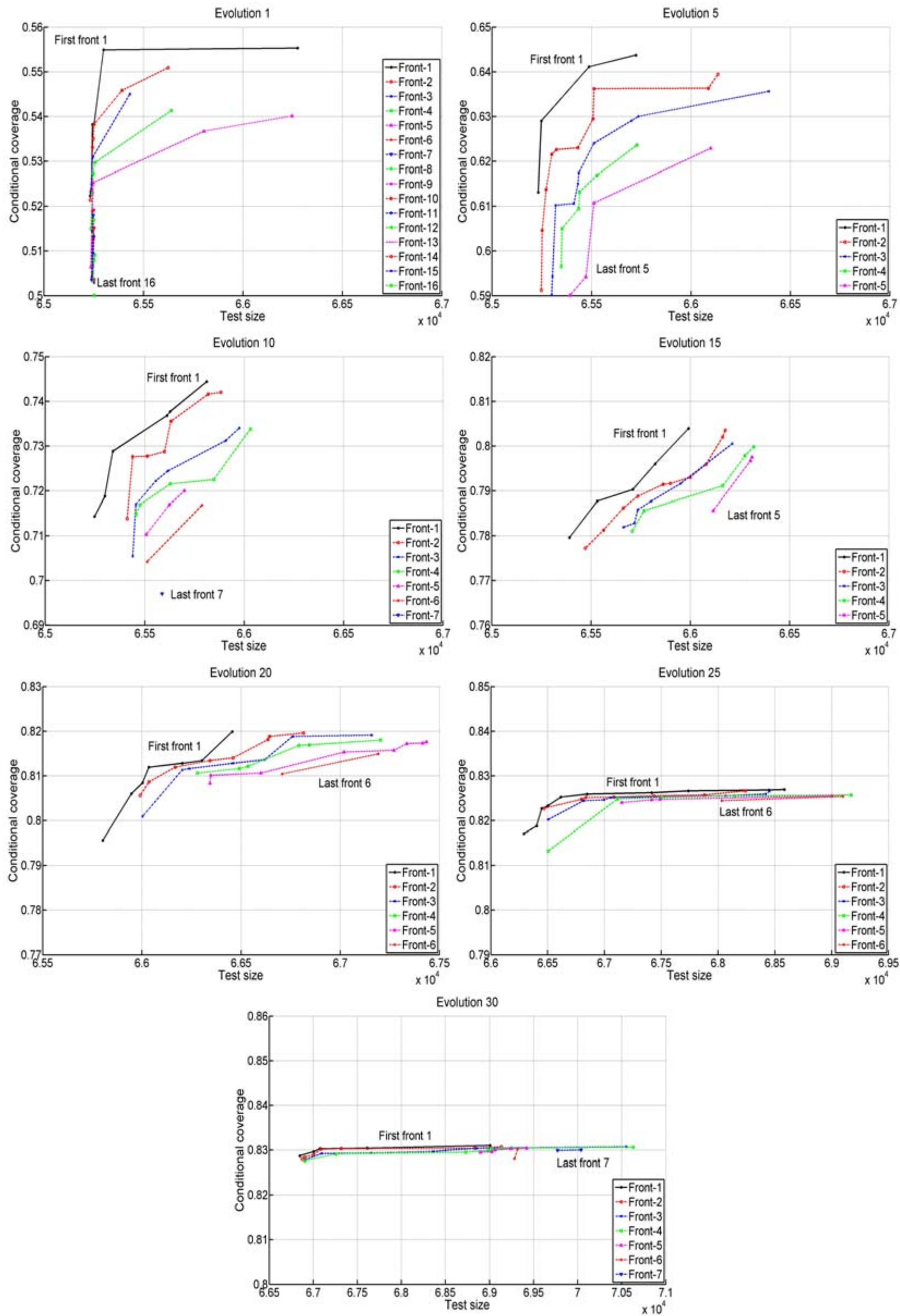


Figure 6.16 Pareto fronts at selected evolutions during GEA (conditional coverage vs. test size)

Coverage and test size objectives observations

The improvement in objectives fitness during evolutions can be identified from both types of Pareto front plots in Figure 6.11 to Figure 6.13 and Figure 6.14 to Figure 6.16. The coverage fitness improvement between the first and last evolutions is easily identified from Figure 6.11 to Figure 6.13. Also, the tests achieve coverage enhancements whilst trying to contain the test size. Toward later evolutions, optimisations to attain further coverage improvements whilst containing test size become increasingly difficult.

During early to mid evolutions, the best Pareto fronts rise upwards along the y-axis showing coverage increases, but smaller drifts to the right along the x-axis indicates test sizes did not increase rapidly. This pattern continues up till middle stages of the evolution process, because opportunities for optimisation and uncovering of new test space are higher with low test size penalty.

In later evolutions, the gap between consecutive Pareto fronts becomes smaller. Hence, the rate of coverage improvements drops but the test size continues to increase. The continual increment in test size is due to addition of new test functions via GEA variation. But despite new test functions, similar gains in coverage enhancement do not eventuate because the uncovered or hidden test space is not easy to identify; compared to earlier evolutions when much of the test space was still unexercised. Eventually, the GEA process reaches a plateau whereby no further coverage enhancement is possible using the current snippet genome and test population. This characteristic is reflected by the flattened Pareto front lines from the last group of evolutions. These Pareto fronts stretch right along x-axis toward increasing test size without any gain in coverage.

From a test generation performance perspective, Figure 6.11 to Figure 6.13 (and Figure 6.14 to Figure 6.16) show greatest coverage improvement gain of approximately 30% for toggle and conditional coverage. This is because both coverage metrics present the best opportunity for enhancements. Their initial coverage levels were lower and they required more elaborate and larger tests to be created to achieve higher coverage. In contrast, line coverage is an easier coverage to target and can be achieved quickly with less tests that are not as complex. The initial line coverage is slightly higher such that 16% gain is achieved.

Note that in any one line, toggle or conditional coverage objective subset Pareto plot, there were consecutive evolutions whereby the best Pareto front for an objective subset increased substantially compared to Pareto fronts of other objective subset plots. The reason for this behaviour is due to the selection policy for establishing the test population for every next evolution. To ensure all objectives

are given fair optimisation priority, the round-robin selection scheme only selects tests that achieve high fitness for an objective subset one after another.

Ideally, a test that is selected for high fitness attainment for one particular objective subset should also provide high fitness for other objectives as well. However, due to various stochastic processes employed in GEA, it is possible (though uncommon) that a selected test may not perform well for other non-conflicting objectives. If such a scenario occurs, the coverage gained by an objective subset Pareto front may not be matched by Pareto fronts of other objective subsets.

These coverage gains and objective subset fitness improvements are affected by the length, slope, and curvature of Pareto fronts, which we examine next with respect to test diversity.

6.9.2 Test diversity analysis

Size, slope, and curvature characteristics of Pareto fronts

Another facet of our experiments is to examine the test population diversity attained during GEA test generation. The Pareto front length, slope, and curvature characteristics at every evolution are all indicators of the types and range of tests that make up the test population.

In Figure 6.11 to Figure 6.13, Pareto fronts are generally large throughout the GEA process. The larger the length and slope of the Pareto fronts, the more diverse the tests are. The Pareto fronts toward the end of testing however, cannot gain further coverage improvements because the GEA process has stagnated. The lengthening of these fronts at later stages of GEA is caused by addition of snippets that do not successfully achieve further coverage. Test sizes were simply increased.

The Pareto front curvature characteristics reveal how effective the GEA process optimises objectives. A Pareto front with low slope and curvature suggests the GEA process has not identified any particular high performing tests. This occurs when most tests have been optimised to achieve similar fitness, and not many tests contain unique sequences of snippet test genome to enhance fitness above current levels.

On the other hand, a high curvature implies the GEA process is optimising all objectives to find the best trade-off between coverage and test size fitness. Ideally, if the GEA process is to optimise multiple objectives effectively, the Pareto front curvature should also be convex; with the peak of the front toward the top left region of the plot. The Pareto front whose curvature is obvious and its peak

aimed at this region indicates tests exists along the Pareto front for which the highest coverage and lowest test size are being actively evolved.

The above characteristics can be observed in a number of Pareto fronts in Figure 6.11 to Figure 6.16. In contrast, Appendix G.5 describes an equivalent multi-objective GEA test generation process without incorporating our test selection strategy or test duplication management mechanisms. In Figure 6.11 to Figure 6.16, during early evolutions, whilst desirable Pareto front size, slope and curvature characteristics are maintained (with the frontal peak toward upper left region), the coverage fitness continues to improve without giving away too much penalty in test size. The Pareto front plots in Figures G.1 to G.6 in Appendix G.5 show shorter Pareto fronts with lower curvatures. Their coverage fitness increases are smaller and requires much larger test sizes.

Also, the best fitness attained in Appendix G.5 for each of the corresponding line, toggle and conditional coverage is lower compared to results in Figure 6.11 to Figure 6.16. The multi-objective GEA is inferior, the coverage gained originates from the addition of snippet functionalities held by larger tests. Rather than concurrent coverage and test size optimisation, test size has been sacrificed to gain coverage. Hence, the best coverage fitness attained required much greater test sizes, and the Pareto fronts sway away significantly from the desired upper left region of the plot.

Analysing diversity from the best Pareto fronts

To measure population diversity, we also examined the spread and expansion of the Pareto fronts during each evolution. Figure 6.11 to Figure 6.13 showed the best Pareto front from each evolution during the GEA process. During early stages of evolution, the best Pareto front at each evolution rises rapidly towards the desired upper left region of the plot. In the beginning, it is easy to uncover unexercised SoC functionalities. Hence, the Pareto fronts exhibit large coverage gains for small increases in test size.

As coverage is gathered, the best Pareto front expands increasing in both coverage and test size. This stretches the Pareto front to the right and upwards. At the same time, previous tests along the Pareto front that are smaller in size but still attained sufficiently high coverage are retained. This extends the Pareto front to remain within the left-ward region. The overall outcome is a widening of the Pareto front line, caused by the smallest test that gives lower coverage and the largest test that attains best coverage. The expanding and lengthening of Pareto fronts are controlled at the ends of the Pareto front where these two contrasting tests are located.

The larger expansion of the best Pareto front at each evolution, the more diverse the tests are with regards to the different types and amount of SoC functions exercised. This is indicated by their coverage levels and test sizes along the Pareto front as well. Diversity in tests is extremely beneficial for following evolutions to interbreed more assorted range of higher coverage and lower sized tests.

When the best Pareto front begins to flatten and reduce in length, this implies lower test diversity. Such a condition usually occurs during middle stages of the GEA process, when test generation has exhausted the easily identified uncovered design functions. To achieve further coverage, more complex SoC functions must be exercised requiring larger test sizes. The rate at which Pareto fronts rises in Figure 6.11 to Figure 6.13 starts to drop from around evolution 15 onwards due to lower coverage gains. In addition, the slope of the Pareto front also reduces and lengthens to the right as tests are now larger.

From evolution 15 onwards, as test diversity continues to fall, the resultant test suite is unable to achieve the same rate of optimisation for all coverage and test size multi-objectives. Eventually, the best Pareto fronts for later evolutions will be near horizontal indicating no further coverage gains. By this stage, the length of the Pareto front will continue to diminish, whilst the position of the front may shift right along the test size axis. This is due to the GEA process unsuccessfully attempting to uncover more SoC functions by adding more snippets in the test, which increases test size but without any more coverage improvement.

The expansion characteristics of the best Pareto fronts during GEA shows test diversity is important and goes hand-in-hand with achieving best optimisation performance for multiple objectives. A more diverse test population provides greater opportunities to intermix a range of tests to explore further unexercised design space and enhance coverage. Whilst at the same time, test sizes are contained by selecting from this diverse test set, the smallest snippet genome sequence to achieve equivalent coverage gains. A goal of the multi-objective GEA is simply to maintain large Pareto fronts for as many evolutions as possible, in order to exploit diversity.

Analysing diversity from the span of Pareto fronts

The diversity attained by the test population is also evident from the span of the best and worst Pareto fronts at specific evolutions in Figure 6.14 to Figure 6.16. The span of all Pareto fronts (between best and worst fronts) at each evolution indicates the diversity of the test population created at that evolution.

A more diverse test population is indicated by greater separations between the best and worst Pareto fronts. Pareto fronts that are spread out over a greater plot area correspond to tests that attained a greater range of coverage and test sizes. The wider the best and worst Pareto fronts, the more spread out the intermediate Pareto fronts will be. Hence, the composition of the test population, which corresponds to all these Pareto fronts, must be of greater variety to have achieved such range of coverage and test sizes.

In each of Figure 6.14 to Figure 6.16, we examine the change in span of Pareto fronts as the number of evolutions conducted during GEA increases. During early and middle stages of the evolution process, the span between the best and worst Pareto fronts grows and reaches maximal separation between evolutions 10 and 15. Then, toward later stages of the evolution process, the worst Pareto front closes in on the best Pareto front. It is during these evolutions that the rate of optimisation drops. The optimised tests from the best Pareto front do not achieve the same level of improvements in objective fitness as before.

In effect, the only remaining tests in the population that can still be enhanced are those from the lower and worst Pareto fronts. By optimising these remaining low performing tests, the lower and worst Pareto fronts are drawn closer to the best Pareto front. This trend is clearly shown in Figure 6.14 to Figure 6.16, where Pareto fronts for evolutions 20 to 30 are much closer than those before evolution 20.

The closer together Pareto fronts are, the less diverse the tests at those evolutions. Lower diversity tests population imply further evolutions cannot optimise as effectively. This in turn leads to Pareto fronts coming closer together and reducing test diversity even more.

Ideally, the Pareto fronts at every evolution should be kept apart as much as possible to maintain effective GEA. When Pareto fronts are too close together, this signals a termination of the GEA process as described in Section 6.7.

Test duplication and diversity

Given the importance of diversity, we examine test duplications, which can occur (as discussed in Section 6.5.3) and must be managed so as not to reduce diversity. During test generations, a test duplication check mechanism ensures that the amount of test duplication does not exceed the threshold 75% mark; which was set from calibration of the test generator by empirical results. Figure 6.17 shows the test duplication results given by the percentage of duplicated tests in the population throughout the

evolution process. The duplication rate is centred around 50%. This is an acceptable range that ensures better performing tests that were effective for certain objectives are given higher selection priority; but at the same time, prevents over duplication in test populations so the GEA process does not stagnate.

In Figure 6.17, the duplication reaches at most close to 60% several times, but this is contained immediately toward 50% in subsequent evolutions. To ensure effective tests are given selection priority, these tests would have been chosen multiple times with a minimum duplication rate of 30-40%, which is evident at evolutions 4 and 25 for example. If the duplication rate were to stray away too much from 50%, then this indicates the GEA process has deviated greatly from the multi-objective optimising goals.

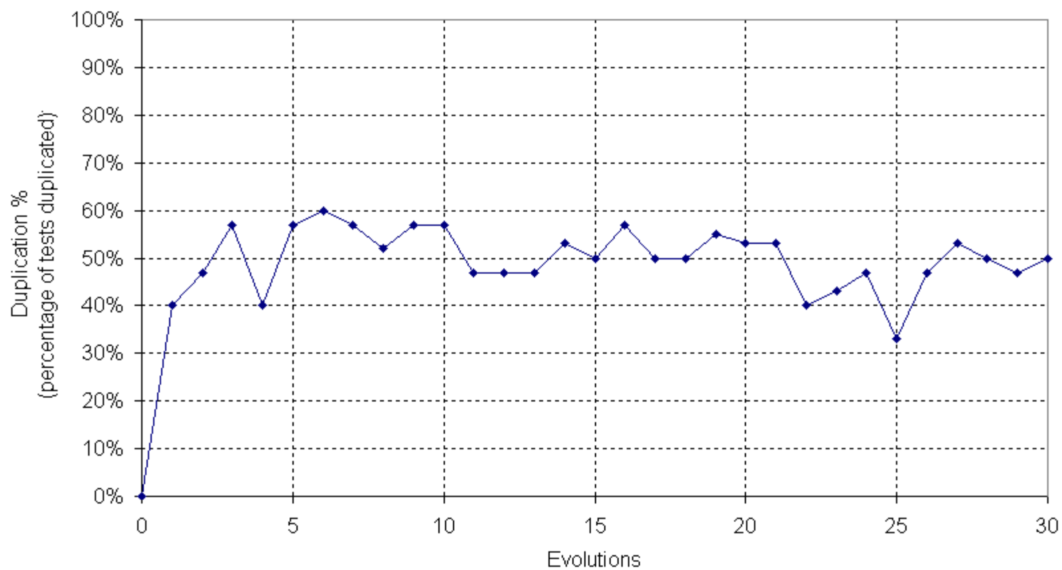


Figure 6.17 Test duplication rate during multi-objective GEA test generation

6.9.3 Multi-objective tri-axial graphs

The measure of GEA performance (and test diversity) can also be examined by plotting test coverage and size data of all tests on a three dimensional axial graph. Figure 6.18 shows such a plot of the coverage and test size objective results attained by all tests during multi-objective GEA. These objective fitness results plotted are from the GEA test generations conducted for Figure 6.11 to Figure 6.16. The three dimensional space of the graph in Figure 6.18 is representative of the objectives space that tests can be evolved within, in order to achieve objective fitness goals.

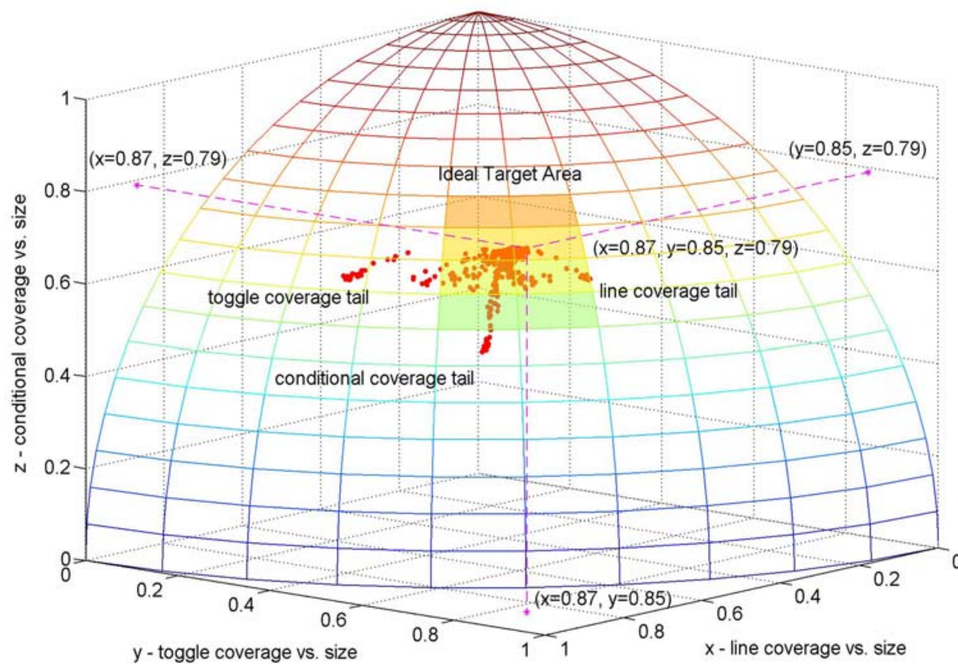


Figure 6.18 Multiple objective GEA tri-axial graph

Each of the axes in Figure 6.18 indicates the performance of a test by the line, toggle, or conditional coverage it attains for its test size. Therefore, the x, y, and z axis values for a test is evaluated by summing the line, toggle, or conditional coverage with the remaining size available from memory that can hold the test, and normalising the combined value by divisor of two (for two objectives in each objective subset).

A test with high line coverage and low test size acquires a greater value along the *line coverage vs. test size* x-axis. Similarly, a high toggle (or conditional) coverage test that is small will be plotted further along the y-axis (or z-axis). The goal is to evolve tests so they achieve objectives fitness to be plotted at 45 degrees to all three axes, and directly outwards from the origin.

The best possible test desired, as measured by this three dimensional tri-axial graph should hold a value of one for all three axis (i.e., x, y, z axial coordinates of (1,1,1) is the goal). We refer to this (1,1,1) axial coordinate as the *ideal test target point* goal. For multi-objective GEA, the highest axial values for the best test is shown corresponding to x, y, z axis coordinates of (0.87, 0.85, 0.79). This best test is located at the intersection of the three dashed lines in Figure 6.18.

In Figure 6.18, the graph shows a circular quadrant surface which provides a guide as to the diversity of tests and the range of objective fitness results attained. As evolutions are conducted, plotting of tests

should expand outwards from the origin with this surface and toward the direction of the ideal test target point. The region around the ideal test target point is denoted as the *ideal target area*, and is shaded in Figure 6.18. The goal is to create tests that are covered by this target area. As long as tests are within proximity of the ideal target test point, higher fitness for all objectives is attained.

If the multi-objective GEA process was unsuccessful, then the created tests would be plotted away from this target area. This implies tests were only successful at optimising certain objectives only, not all the objectives could be optimised concurrently.

Figure 6.18 shows that our GEA test generation process is effective, with tests scattered across the proximity of the ideal target area. This also demonstrates our test generation method promotes test diversity, and manages duplication appropriately. Appendix G.6 elaborates further our analysis of diversity and duplication analysis from the tri-axial plot.

Other tri-axial graph characteristics

In Figure 6.18, three distinct plots of tests originate from different regions, and eventually converge toward the ideal target test point. These three distinct *tails* of tests are labelled as line, toggle and conditional coverage tails. Each of these test tails correspond to the optimisation of tests for one of these coverage metric versus test size objective subsets. Every test tail begins at a region of low fitness for their corresponding objective subset, and is evolved toward increasing objective fitness values.

For example, the conditional coverage test tail plot begins at its origin where fitness for the conditional and test size objectives subset is low. The GEA process then creates tests that optimise this objective subset specifically, leading to improvements in their fitness and lengthening of the tail along the z axis. At the same time, these tests are intermixed during GEA variation with tests from the other tails, which are responsible for optimising other objective subsets. Eventually, the test tails come together with each other, all increasing within the ideal test target area and toward the ideal test target point.

The tests from each tail not only gain fitness improvement for their objective subset, but evolve with snippet genome from other test tails. This provides overall fitness for all objectives concurrently. The presence of test tails for each objective subset and their convergence demonstrates our divide and conquer approach in multi-objective GEA, whereby each subset of objectives is ensured for optimisation. We discuss further the effectiveness and characteristics of multi-objective GEA test generation with respect to the tri-axial graph in Appendix G.6.2. To demonstrate the benefits of our

multi-objective GEA process and usefulness of the tri-axial graph, we compare against other test generation variants and their tri-axial graphs in Appendix G.6.3.

6.9.4 Multi-objective GEA Pareto front termination

This section examines two Pareto front GEA termination schemes. The first is the slope of the best Pareto fronts from one evolution to the next (Section 6.7.1). The second is the span of all Pareto fronts at each evolution (Section 6.7.2).

Slope based Pareto front GEA termination

The plot of the best Pareto fronts in Figure 6.11 to Figure 6.13 shows that well before the last evolution, the best Pareto fronts were already flattened with slope close to zero. At this stage, the GEA process can in fact terminate, as it is obvious no coverage objective fitness improvement had recently occurred or will occur. Coverage does not increase and test size fitness is simply degrading toward larger sizes.

If conducting the GEA test generation with the Pareto front slope based termination mode, the GEA process would have terminated after evolution 25, which is the first instance in which the best Pareto front slopes of line, toggle and conditional coverage versus test size falls below the slope termination threshold.

The termination occurs when the GEA process reaches optimal maturity, and with the test population stagnated. If further evolutions are to be conducted, the Pareto fronts simply flattens out more, and lengthens to the right without gaining coverage improvements. The GEA process should be terminated earlier as facilitated by the Pareto front slope criteria.

In order to examine the application and usefulness of our termination schemes, the test generations described previously were conducted for fixed number of evolutions. This allows examination of the Pareto front characteristics long after the termination conditions would have been triggered. Figure 6.19 plots the combined slope of the best Pareto fronts against progressive numbers of evolutions. Referring back to Figure 6.11 to Figure 6.13, the correlation between objective improvements and slope values above the threshold can be deduced. When the slope drops below the threshold at evolution 25, objectives optimisations no longer takes place and the GEA process should be brought to an end. The identification of the threshold is based on empirical data and is discussed in Appendix G.7.1.

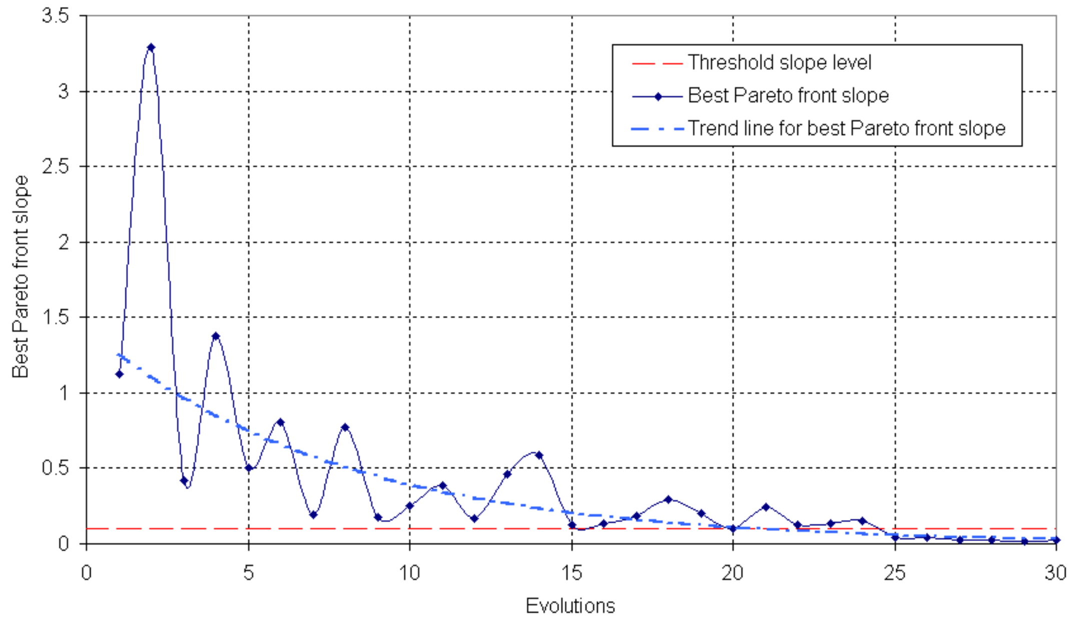


Figure 6.19 Slope of best Pareto fronts vs. evolutions

Span distance between Pareto fronts GEA termination

The second Pareto front termination scheme is to examine the proximity of and span between the best and worst Pareto fronts at the end of every evolution cycle. A wide gap between these fronts indicates the test population is still diverse and further objective enhancements are possible. A narrow gap points to lower likelihood for objectives optimisation, as the snippet genome in tests are more alike than before. This presents less opportunities for uncovering new test search regions. When the gap distance falls below a threshold, the GEA process should be terminated. Appendix G.7.1 describes the selection of Pareto front gap distance threshold.

Figure 6.14 to Figure 6.16 plot all the Pareto fronts at selected evolutions during test generation. From the start and up to middle stages of GEA test generation (between evolutions 1 and 20), the gap between the best and worst Pareto fronts is larger as the GEA process still has ample possibilities to enhance objective fitness. During the first half of GEA evolutions, the large gap is maintained because the test population is most diverse after many generations of variation. From the middle stages of GEA test evolutions onwards, the gap starts to reduce as it becomes harder to seek new test space, and the rate of objectives optimisations is not as high as before. Eventually towards the end of the GEA process, from between evolutions 15 to 20 and beyond, the gap falls below the termination threshold and the test generation process should be halted.

Figure 6.20 shows the plot of the best and worst Pareto front gap against number of evolutions. After initial evolutions, the gap distance starts to reduce. Between evolutions 15 to 20, the gap regularly drops below termination threshold. Referring back to Figure 6.14 to Figure 6.16, objective fitness improvements drops between these evolutions. Further optimisations have stalled and hence termination should be triggered by the Pareto front span criteria. If so, unnecessary and wasteful test evolutions well before evolution 30 would have been prevented.

The narrowing of Pareto fronts toward GEA termination reveal much about internal GEA operations, which requires further discussion and is presented in Appendix G.7.2.

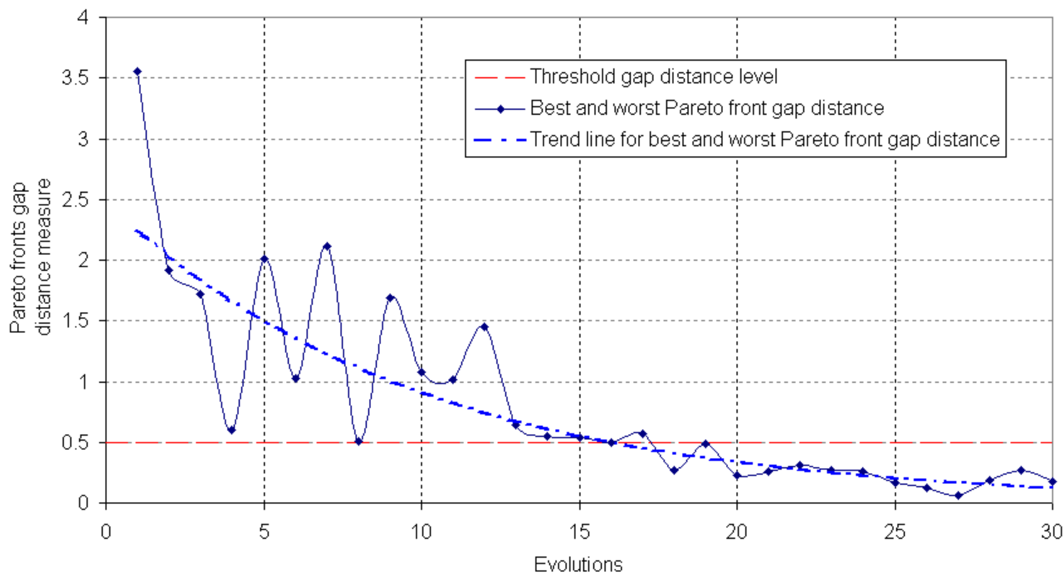


Figure 6.20 Gap distance between best and worst Pareto fronts vs. evolution

6.9.5 Test size characteristics for multi-objective GEA test generation

In Figure 6.21, the test size data is plotted for each individual test during the GEA process. The left y-axis shows the percentage usage by tests from the available memory in the simulation platform. The right y-axis is the absolute test size. At the beginning of test generation, in order to restrict test sizes, the GEA process will exhaustively use existing snippet genome of early evolutionary test populations to achieve as much test coverage as possible. Hence, test sizes do not increase much.

Only when coverage attainment rates become low and new coverage is no longer uncovered readily, more elaborate forms of variation is then conducted. This includes adding new forms of snippets to the existing snippet genome pool. The outcome of these variation usage changes is to increase the test sizes, which is reflected in Figure 6.21 from test 250 onwards.

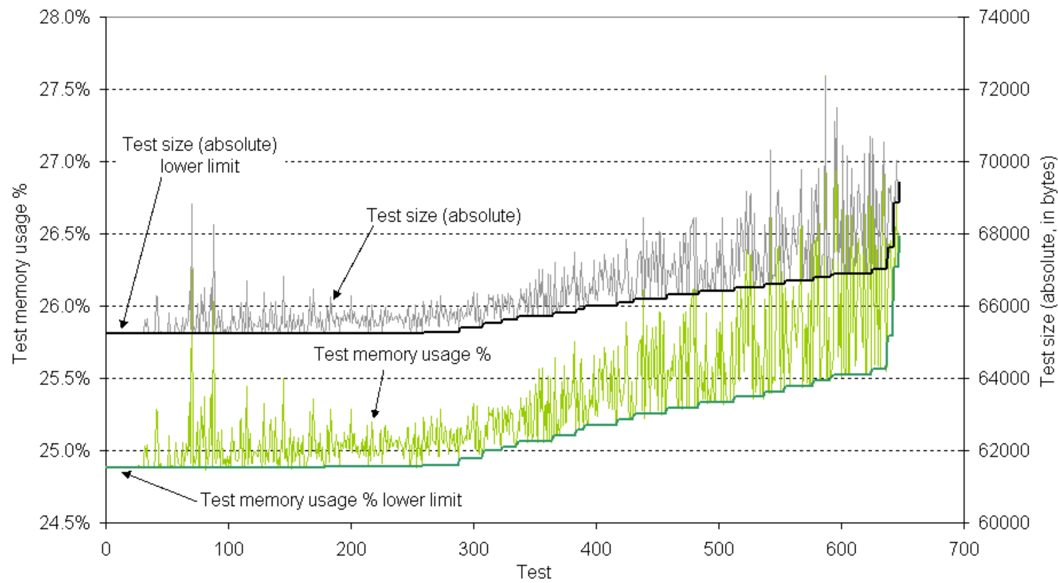


Figure 6.21 Multi-objective GEA test sizes for each test

Note that a distinct lower edge is formed at regular points of the graph line for the lowest test sizes throughout the GEA test generation. This lower limit test size line demonstrates attempts by the GEA process to curb test sizes as low as possible, whilst achieving similar levels of coverage as other larger sized tests at the same evolutions.

The two large test sized spikes near the start of the GEA process are due to spurious addition of the same snippets multiple times in those tests. There is inherently some form of randomisation employed by GEA variation, and from time to time, this can cause such unnatural variation anomalies.

In summary, despite tests taking up only one quarter of the available memory in the simulation platform, one cannot ignore the benefits of containing test sizes, especially if there were stricter memory limitations. In our case, test sizes only increase by about 2%.

6.9.6 Test execution times during multi-objective GEA test generation

Appendix G.8 examines the test execution times for each test during the GEA process. Three distinct phases are apparent. They reveal when test times are (i) maintained low, (ii) increase, and (iii) eventually stabilises. These time characteristics have strong correlation to the Pareto front objectives fitness, and test size results in the previous section, and is detailed in the appendix.

6.9.7 Summary of multi-objective GEA test generation experimental analysis

Test selection and population diversity are critical elements of multi-objective optimisation. The detailed study conducted in this research support the observations made by Tamaki et al. [TKK96]. Selection and recombination variation of a wide variety of tests that addresses different objectives are essential, and brings benefits from test population diversity to tackle genetic drift. Our selection method follows this philosophy by providing even priority to all objectives, and selecting each objective's best performing tests in a round-robin manner. With this selection approach, managing duplications and the range of conflicting objective trade-off tests become essential factors which we take into account as well. Appendix G.9 outlines further the multi-objective GEA characteristics and review test generation performance from experiments in this Section 6.9. We also discuss the role and impact of test elitism on test generation characteristics in the appendix.

6.10 Performance comparison of multi-objective GEA test generation

Performance of the multi-objective GEA test generation is evaluated against four test creation techniques. The reasons for their choice are as follows:

- The SALVEM Genetic Evolutionary Test Generator (SAGETEG) of Chapter 4, it employs similar GEA test generation technique, but on a single objective.
- µGP test generator [CCS], an assembler instruction based test generator that also employs single objective genetic evolutionary method.
- Random constraint-biased test generations described in Chapter 3 and Appendix D, it is an automated technique used widely in industry.
- Applications driven test creations that is conducted manually, it is a standard technique often used to complement and compensate for gaps in automated methods.

Each of the above test generation methods was configured to perform similar verification on the Nios SoC, using the same snippets library and similar test setup where applicable.

6.10.1 Coverage results comparisons

Shown in Table 6.2 are the coverage results. For line coverage, the coverage improvement of multi-objective GEA is 0.6% to 32.9%. Toggle coverage also performs better achieving 95.7%, which is 2% to 46.9% improvement. Conditional coverage from multi-objective test generation also shows improvement of 0.3% to 17.7%. These improvements are attributed to the bigger range of SoC design state storage values that arise from exercising other types of functionalities; which are also driven by the concurrent attainment of other coverage objectives such as conditional coverage during GEA.

Conditional coverage is the most difficult metric to maximise. Any gain in coverage above 80% is always considered valuable. To exercise greater number and variety of conditional paths through the design, other more complex interactions and concurrency amongst snippets are needed. By incorporating more snippets, multi-objective GEA should be able to improve on this result without any change to our existing multi-objective GEA test generator.

Table 6.2 Summary coverage results

	Multi-objective GEA	Single objective GEA (SAGETEG)	µGP	Random test generation	Manual application based tests
Line coverage (%)	99.5	98.9	97.5	89.9	66.6
Toggle coverage (%)	95.7	93.7	87.1	79.2	48.8
Conditional coverage (%)	83.3	83.0	72.4	69.3	65.6

Coverage gain comparison

Another useful measure for comparison is to examine the gain in coverage of the test generation methods. The coverage gain measures the difference between the highest attained coverage and initial coverage for a particular test generation technique. Incorporating coverage gain is to account for the different initial coverage of each test generation method. This is due to different test program administrative set up that must be performed to facilitate various chaining together of snippet genome by each test generator. For example, initialisation of the number of snippets, and data inputs used by snippets during testing must be performed in initial sections of each test. These administrative functions facilitate snippet execution but also provide certain amount of default coverage within each test program. Hence, initial coverage levels at the start of different test generation runs vary. The coverage gain results are presented in Table 6.3.

Table 6.3 Coverage gain results

	Multi-objective GEA	Single objective GEA (SAGETEG)	μGP	Random test generation	Manual application based tests
Line coverage (%)	15.4	15.5	3.1	2.3	2.0
Toggle coverage (%)	29.6	10.7	11.6	24.0	13.7
Conditional coverage (%)	26.4	28.0	17.4	26.1	23.1

For line coverage, multi-objective testing achieves the same gain as SAGETEG, and higher gain than other test methods. Line coverage is relatively easy to achieve except for the final few remaining percentages, which represent extremely difficult to exercise lines of design code that are rarely used. Thus, it is not unexpected that the gain did not improve much further over the previous methods like SAGETEG.

The best gain of 29.6% by multi-objective testing was acquired for toggle coverage. The gain is significantly greater than other methods, and is due to snippet influences from concurrent optimisation of other coverage objectives. As for conditional coverage, the multi-objective gain achieved is slightly less than SAGETEG, but higher than other test generation methods. Again, for the same reasons as absolute conditional coverage comparisons, a more extensive snippets library is needed to provide higher conditional coverage gains over single objective GEA testing.

The coverage gain results shows multi-objective GEA testing has potential to become more effective. The fundamental limitation with our existing method is not the test generation algorithm itself, rather the snippets library. Snippets are the test building blocks to create tests and exercise the SoC. If the test generation algorithm has fully utilised the snippets, but full coverage is still not attained, this implies remaining uncovered test space cannot be exposed by the existing snippets library easily.

With further refinements to the multi-objective GEA optimisation process and a better snippets library, the coverage gain could be enhanced to achieve even greater coverage results. Currently, the line and toggle coverage is only 0.5% and 4.3% away from full coverage, whilst conditional coverage requires slightly more effort.

Appendix G.10 examines individual SoC device coverage for each of the test generation methods. Overall, multi-objective GEA test generations perform better for all devices, and provides much more comprehensive testing for certain devices. The appendix also accounts for coverage differences in individual devices amongst the test methods.

6.10.2 Testing time comparisons

Test execution time provides a measure of how fast tests are completed. In Table 6.4, the combined times of other test methods and individual test time of multi-objective GEA is shown. Table G.2 in Appendix G.11 tabulates the individual line, toggle and conditional coverage test run times of single objective test generation methods that make up their combined times in Table 6.4. The 'Test execution CPU time' in Table 6.4 represent the test simulation times only; whilst the 'Elapsed time' represents the time taken by entire verification process of each particular test generation method, including automated or manual test creation durations and other overheads.

Table 6.4 Time results

Total tests time	Multi-objective GEA	Single objective GEA (SAGETEG)	µGP	Random test generation	Manual application based tests
Test execution CPU time (s)	307,336	752,884#	751,023#	600,455#	22,554*
Elapsed time (days)	3.6	8.8#	9.4#	8.1#	4*†

– For fair comparisons with multi-objective GEA testing, the combined total time for these automated methods are summed together from their corresponding individual line, toggle and conditional coverage test runs (in Table G.2, Appendix G.11).
* – Note that the number of manual tests executed is much less than the other automated test methods.
† – Includes time to manually create the application test cases, which is equivalent to automated test generation time.

Unlike multi-objective GEA test generation, the other test generation methods only cater for single objectives of one coverage type each time. Hence, for experimental comparison purposes, each of these test generation schemes was repeated one after another for line, toggle and conditional coverage.

To provide fair comparisons with multi-objective GEA, certain results from each of the coverage test runs by the other four test generation methods are considered collectively and combined together. For example, when examining test execution time results in this section, the duration for each of the line, toggle or conditional coverage testings are combined to provide an overall value for every test generation method. This is used for comparison against the single test time result of multi-objective GEA, which performs all coverage and other objectives driven testing in one go.

Examining the combined times of previous test generation methods in Table 6.4, multi-objective GEA test generation can be considered efficient with regards to test execution and overall verification time. Whilst multi-objective GEA may require longer duration to execute compared to individual line, toggle or conditional coverage driven test generations (Table G.2, Appendix G.11), when evaluated within the context of combined total time for all coverage test runs, multi-objective GEA runs faster

than other methods. For example, instead of 8.9 days for SAGETEG, 9.4 days for µGP, and 8.1 days random testing, multi-objective GEA completes within 3.6 days; achieving superior or equivalent coverage as well.

For manual application testing, the elapsed testing time comparison is not as applicable because manually creating tests will naturally be longer compared to automation. To create equivalent number of tests and achieve similar coverage is impractical because such manual test creation methods are for testing of specific and critical functionalities. Manual application test times are shown for the sake of results completeness.

6.10.3 Test size comparisons

The results of the test program size are shown in Table 6.5. Similar to time results comparison, the size results of individual line, toggle and conditional coverage single objective test methods are combined together in Table 6.5. This represents appropriate overall values for comparison against multi-objective GEA. The last three rows of Table 6.5 show the individual test size results of each coverage test run making up the combined sizes of single objective tests.

Table 6.5 Test size results

Average test size (bytes)	Multi-objective GEA	Single objective GEA (SAGETEG)	µGP	Random test generation	Manual application based tests
Total or combined tests	66,435	71,137	29,239	63,591	N/A†
Line coverage tests	N/A‡	67,870	27,585	64,259	N/A†
Toggle coverage tests	N/A‡	74,735	32,269	63,345	N/A†
Conditional coverage tests	N/A‡	70,065	26,464	62,999	N/A†

† – Application based tests are manually created and optimised by-hand, without any snippets or algorithmic test composition technique. It is not applicable for comparison here.
‡ – Multi-objective GEA testing maximises line, toggle and conditional coverage in a single test run, not individually; so different coverage driven tests have the same total size.

Table 6.5 shows multi-objective GEA testing outperforms single objective GEA test generation of SAGETEG. Overall, multi-objective test size per averaged test is 1,435 bytes smaller for individual line coverage, 8,300 bytes smaller for toggle coverage, and 3,630 bytes less for conditional coverage driven test sizes from SAGETEG. This indicates multi-objective test generation achieves greater or equivalent coverage using smaller tests. Smaller tests imply faster test runs overall. And this supports the test duration results in the previous section. Multi-objective GEA requires shorter test time

compared to the combined test times of each individual line, toggle and conditional coverage objective test runs.

Other than SAGETEG, average multi-objective test size is in fact larger then individual line, toggle and conditional coverage test runs from µGP and random testing. This is not unexpected. µGP genome and test generations rely on assembler instruction programs, which are more space efficient than high level software based test programs of SAGETEG or multi-objective GEA. In the random approach, the snippets and test programs are created by stochastic means only, and do not increase in size significantly when it strives to attain higher coverage. On the other hand, GEA test programs evolve and grow incrementally using snippet genome in order to seek greater coverage.

The important outcome from Table 6.5 is that equivalent combined tests sizes (and test duration) from other test generation methods will always be greater than the multi-objective approach. For instance, whilst the average test size for µGP and random testing is smaller, it requires three individual test runs conducted one after another, resulting in greater overall test sizes. Even then, their achieved coverage does not match multi-objective GEA testing.

6.10.4 Number of tests and snippets results

The effectiveness of multi-objective GEA is measurable by the number of tests and snippets employed. These results are tabulated in Table 6.6. For both number of tests and snippets, multi-objective GEA test generation uses significantly less tests and snippets than other methods. This demonstrates the advantage of our multi-objective approach, especially when there are size or test resource constraints. In practical terms, if the best overall test is to be reused for later verification phases at the post-silicon hardware level, there could be memory restrictions on a hardware tester with insufficient capacity to hold large tests. In this case, multi-objective GEA would provide the best candidate test that is minimal in size, and can achieve the best overall coverage of all metrics concurrently.

Table 6.6 Number of tests and snippets results

	Multi-objective GEA	Single objective GEA (SAGETEG)	µGP	Random test generation	Manual application based tests
Number of tests	648	1,674	5,754	1,820	37
Number of snippets	16,547	50,236	256,671	85,231	N/A†

† – Application based tests are manually created without using snippets.

Appendix G.12 describe one final comparison between multi-objective GEA and other test creation methods based on the test generation effectiveness factor from Appendix E.15 for test generations in Chapter 4.

6.10.5 Coverage attainment rates, trends, and comparisons

In this section, we evaluate progress of coverage attainment versus test resources utilisation. Figure 6.22 to Figure 6.24 plot the line, toggle and conditional coverage progress against number of tests, whilst Figure 6.25 to Figure 6.27 plot coverage versus time. Manual application test creation is not included in these plots because it is not automated, hence does not provide any fair or sensible comparisons.

In Figure 6.22 to Figure 6.24, the solid thicker graph lines in each plot refer to the cumulative coverage from all the tests executed, hence it is higher than the coverage from any single one test. The raw coverage – which indicates individual coverage from each single test – is shown by thinner oscillating graph lines. In Figure 6.25 to Figure 6.27, the test time x-axis indicates the CPU utilisation time that has elapsed during test generations and simulations of tests.

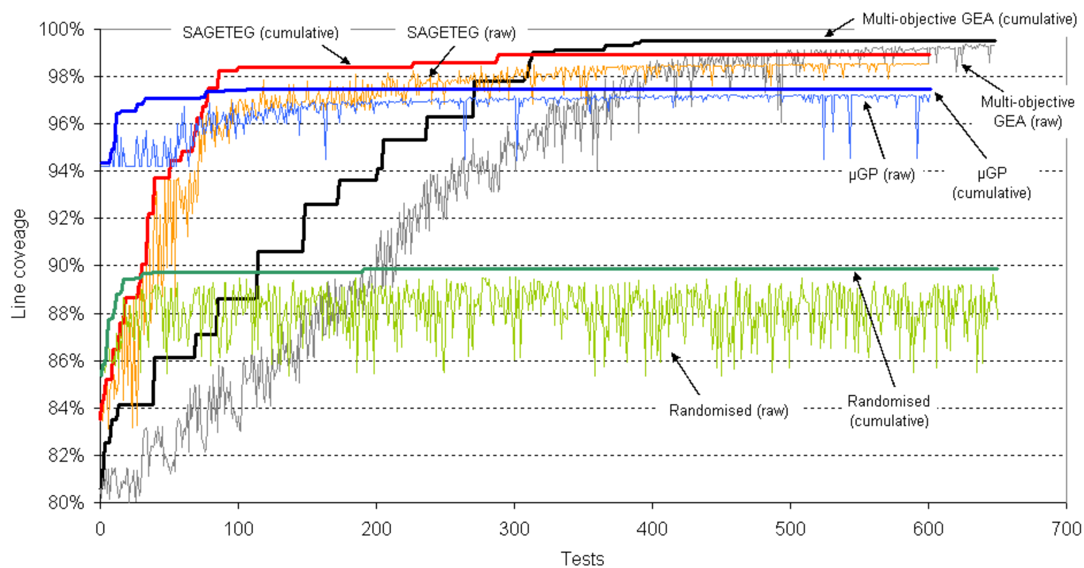


Figure 6.22 Line coverage vs. number of tests

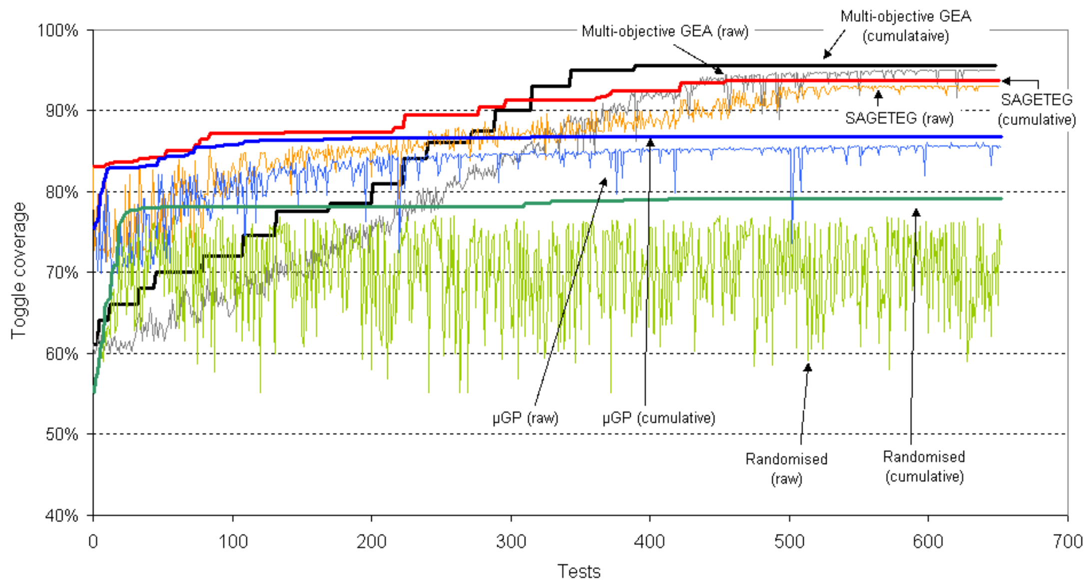


Figure 6.23 Toggle coverage vs. number of tests

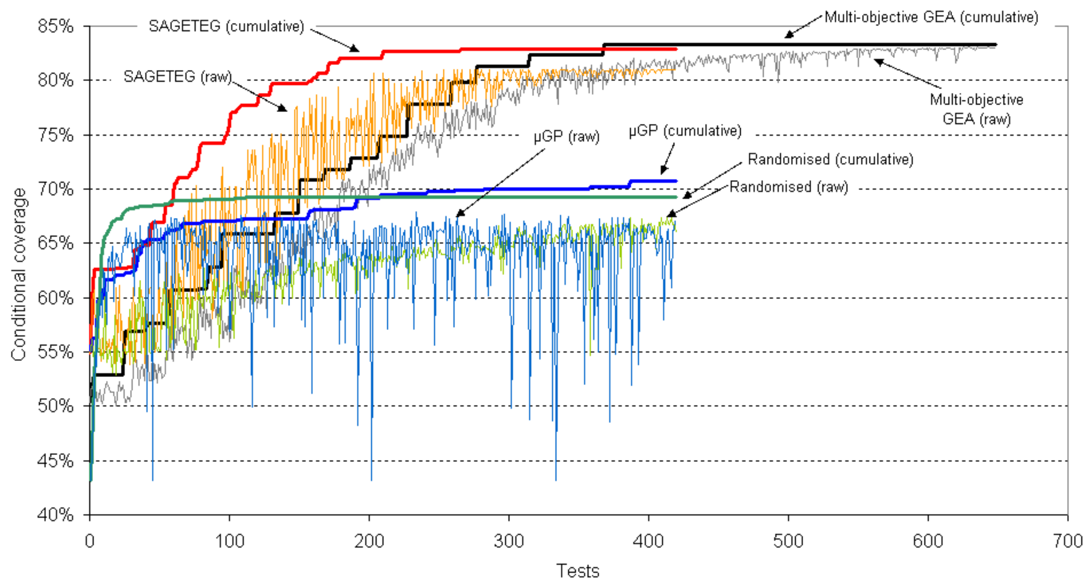


Figure 6.24 Conditional coverage vs. number of tests

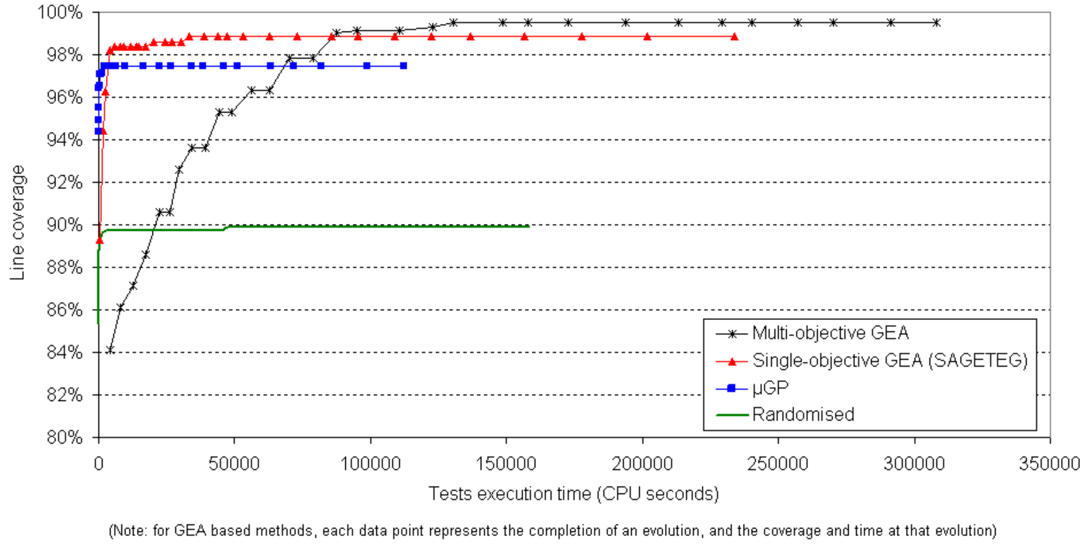


Figure 6.25 Line coverage vs. time

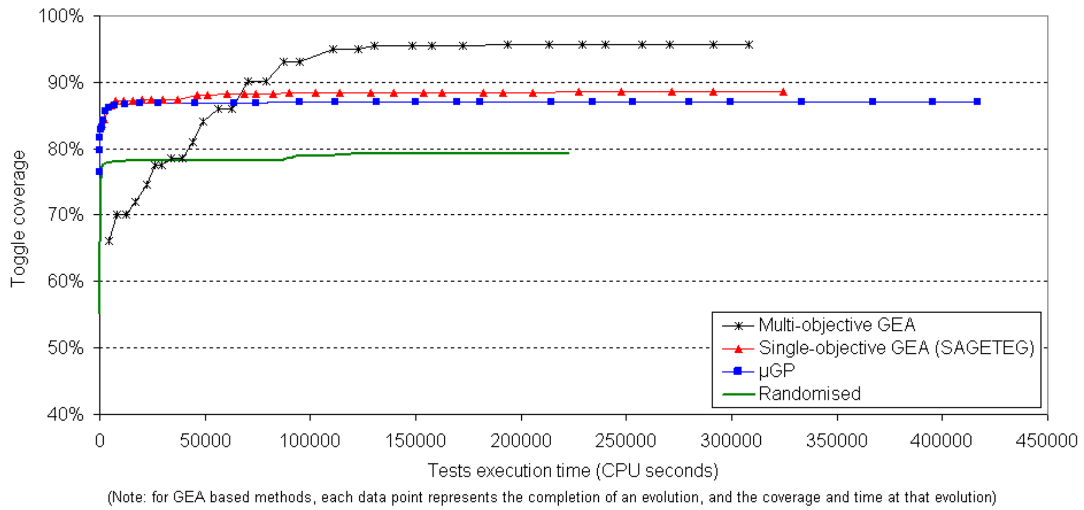


Figure 6.26 Toggle coverage vs. time

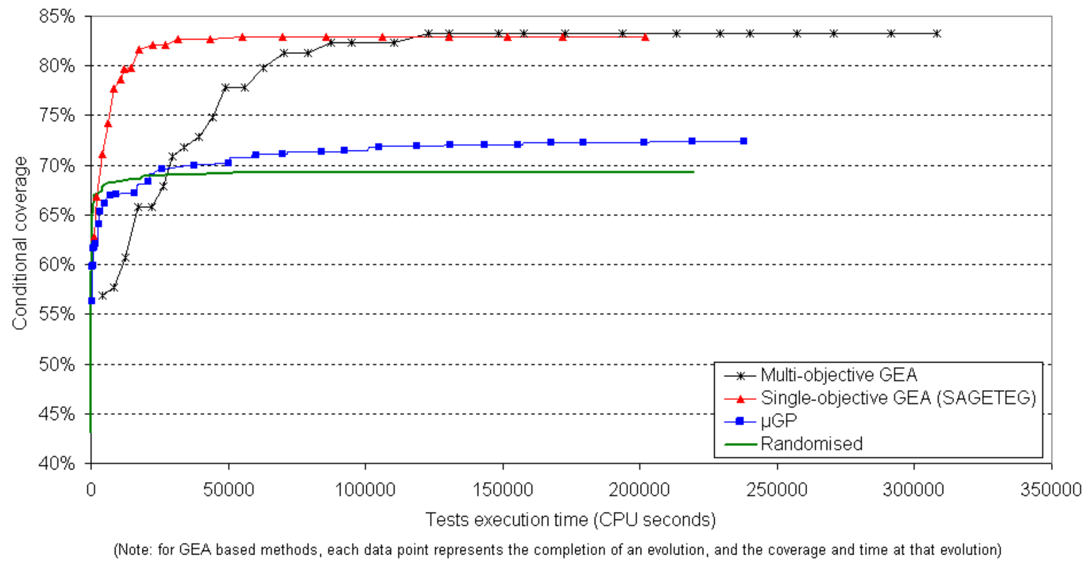


Figure 6.27 Conditional coverage vs. time

Coverage attainment rate graph comparisons with other methods

The most distinctive trend from the coverage attainment graphs is that the multi-objective GEA method attains coverage at a slower rate compared to previous test generation methods. Previous test generation methods achieve coverage faster but saturates at early to middle stages of the test verification process. Multi-objective GEA however, experiences a steadier and slower rate of growth in coverage. For all coverage metrics, it reaches maximum coverage after conducting approximately 20 multi-objective optimising evolutions, utilising 421 tests and 4,969 snippets. Line coverage reaches saturation slightly earlier as it is the easiest coverage metric objective to attain.

In contrast, other test methods do not require as complex test creation algorithms; and their coverage levels saturate quicker using less evolutions, tests, and snippets in general. With the exception of SAGETEG, the saturated coverage levels of these methods also fall well short of multi-objective GEA test generation. Even then, SAGETEG does not surpass coverage levels of multi-objective GEA.

Despite lower coverage rate, in all graph plots, the multi-objective GEA coverage eventually exceeds those of other methods, without requiring too much more time, evolutions, test, or snippets. Supported by the results in Sections 6.10.2 and 6.10.3, the time and test resource usage comparison is even less if the individual line, toggle and conditional coverage test runs of previous methods are combined together.

Whilst the lower coverage attainment rate may at first seem as a shortcoming, one has to bear in mind that the multi-objective method targets all coverage metric and test size at the same time. An equivalent outcome using previous test generation methods would require conducting the same test runs multiple times one after another for each of the line, toggle and conditional coverage. These test generation methods would not complete within the same total elapsed timeframe as multi-objective GEA. Overall, the multi-objective GEA method is still more efficient and achieves higher coverage for less test sizes.

Note that the multi-objective GEA graphs lines all display similar trends for all coverage metrics plotted against number of tests or time. Appendix G.13 also shows supplemental coverage progress graphs plotted against number of snippets and evolutions, which display the same trends.

Analysis of lower multi-objective GEA coverage rates

The reasons for lower coverage progress rates lie directly with multi-objective optimisation processes. By concurrently optimising multiple coverage and test size objectives, this slows down the improvement rate for individual objectives. Enhancements (or degradation) of an objective is no longer independent. Unlike previous test methods whose sole purpose was to gather coverage only, multi-objective GEA must also ensure other coverage metric are similarly enhanced, whilst restricting the size of tests to achieve such enhancements.

This adds further interdependencies and restrictions between multiple objectives, which must be taken into consideration. Specifically in our test generations, the lower coverage progression rate is due to the test size minimisation objective, which directly conflicts with the coverage maximisation goal.

The lower coverage process rate can be explained further by examining the multi-objective GEA process in detail. During test generation, in order to actively control the test size, the test generation process does not recklessly or readily add new snippets to exercise other design functions for uncovering new test space. Instead, it is tuned toward re-using existing pools of snippets genome of current populations; and performing recombination or mutation variation to test new functions and enhance coverage.

Only when the coverage levels are truly stagnated and has not improved for a certain period will the multi-objective GEA process resort to adding new snippets and allowing the test size to increase. The new snippets will hopefully exercise new functionalities and uncover new test space. By taking on this

approach, rather than simply adding new snippets in an uncontrolled manner, this reveals why the coverage attainment rate is slower than previous methods.

In addition to conflicting objectives, complementary coverage objectives may also affect one another. For example, in toggle coverage the goal is to trigger a range of data bit or registers values. To do so, many different input signal or storage values are supplied to the same portion and sequential flow of design code to toggle these bit data and registers with wide ranges of values. Whilst this enhances toggle coverage, the same lines of design statements and same conditional control paths are simply exercised repeatedly. This repetition adds to neither line nor conditional coverage.

Appendix G.13.1 discusses other observations and comparisons of the coverage progress graphs for multi-objective GEA and other comparative test methods; including reasons for the shortened conditional coverage test runs of single objective test generations.

6.11 Experimental results – a summary

The test generated by multi-objective GEA outperforms all other test generation methods using the least test size. The multi-objective GEA test is essentially the best test at the upper left most peak of the best Pareto fronts. In contrast, for other test methods, to attain equivalent coverage, individual coverage driven tests would need to be combined thus expanding the overall test size and test execution time required.

Despite the benefits of multi-objective GEA, the lower coverage attainment results show that further potential for improvements are still possible to increase the coverage rate. Specifically, additional analysis and research must be conducted for the GEA test selection and variation phases. For example, perform directed test selections to pass on tests that enhance a particular objective, but do not degrade other objectives significantly in future evolutions. Also, conduct GEA variation amongst more selective snippets to produce tests that can enhance more than one coverage objectives.

6.12 Benefits, shortcomings and recommendations

This section outlines the benefits and shortcomings of multi-objective GEA test generation, and the four test generation methods used in the comparative study. Recommendations on how to best utilise these test methods are also provided.

Manual applications driven testing is to complement automated test generation methods. It provides directed testing, focusing on specific test functions that are difficult to test with automation, or are critical to the design and must given extra test attention. For this reason, manual application testing is tedious to perform and cannot verify the SoC by itself.

Random automated test generation is simple to implement for automated verifications. However, it is loosely directed. At best, it uses some form of constrained biasing, and is not always efficient in the number of tests it employs to achieve coverage. As demonstrated by experimental results, it is not always possible to achieve the same levels of high coverage, especially without additional user guidance.

μGP conducts test generation guided by GEA methods. It operates at the assembler instruction level. μGP is most useful for microprocessor testing. The test program code can be hand-crafted to provide efficient test sizes. However, the composition of tests at the assembler level limits some test functional interactions. As integration of snippets is only possible at the higher software application levels, the range of snippets for use is restricted. In addition, there are inherent dependencies on the instruction set architecture of the target SoC processor to be verified. Portability issues exist when applying the test composition building blocks of μGP from one hardware design to verifying another.

SAGETEG, which employs single objective GEA and snippets test program composition at the software application level, is able to achieve high coverage at reasonably fast rates. It is driven by one coverage objective at a time, and does not take into account test sizes or test resources utilisations explicitly. SAGETEG is more appropriate for use if the verification is to focus on one coverage objective, and there are no restrictions such as test sizes. This method provides the fastest rate of coverage progression.

Multi-objective GEA test generation provides another verification method at the software application level. It acquires equivalent or greater coverage compared with other methods, and is extended with additional techniques to optimise multiple coverage objectives and minimise test sizes. For multi-objective GEA testing, the requirements conducive for such test generations are maximisation of more than one coverage metrics at a time, or whenever there are restrictions (i.e., test sizes) or other test

objectives. As demonstrated by our experiments, for simultaneous coverage of line, toggle and conditional metric (and if coverage attainment rate is not an important criteria), multi-objective GEA testing is favoured.

Multi-objective GEA testing achieves the best coverage for all line, toggle and conditional metric, using minimal tests and least simulation elapsed time. This is especially useful if tests are to be re-used for testing at later phases of the design process, such as post-silicon hardware bring-up testing. In this case, there may be restrictions by physical hardware testers on the available memories to hold tests. The efficiently composed tests from multi-objective GEA test generation can be applied to fit onto these hardware testers, and achieve high coverage concurrently for a variety of coverage metrics using these best tests.

Given various pros and cons of single objective SAGETEG and multi-objective GEA testing, another alternative is to incorporate elements of both approaches and exploit their benefits together. For instance, rather than optimise all multiple objectives evenly, in the multi-objective GEA test generation, higher priority can be given to optimising one particular coverage or some other test objective deem more critical.

In our case, line coverage is often viewed as a prerequisite for initial testing before conducting further in-depth verification. This is because it is an easier metric to exercise and provides prompt feedback on the success of tests. Usually, a test strategy must exercise all lines in a design code. This often acts as a basic test criterion. For multi-objective GEA test generation, line coverage can be given higher priority so that the optimisation process is biased toward achieving higher line coverage first. This allows for preliminary assessment of the multi-objective GEA verification process earlier.

If it is apparent that line coverage is unable to achieve the required target levels, then the verification process can be terminated and prevented from continuing needlessly. Instead, the test process and strategy can be revised or refined before being executed again. Otherwise, if it is clear the test generation process successfully achieves adequate line coverage, then the multi-objective GEA process can be permitted to continue optimising more thoroughly the toggle, conditional coverage or other test objectives at the same time as attaining the remaining line coverage.

Finally, our multi-objective GEA method is versatile, to incorporate other types of test objectives in addition to the objectives employed in this chapter. This would enhance the test generation method further and expand its applicability to other verification strategies.

6.13 Conclusions

This chapter presented a test generation technique that incorporates multiple test goals using genetic evolutionary algorithms and multi-objective optimisations. The key innovation with our multi-objective GEA method is realised using Pareto front sorting and multi-phase divide-and-conquer GEA selection strategies. The technique expands the applicability of the GEA test generation method to cater for wider range of verification challenges. Rather than simply covering the hardware design test space extensively, other test objectives or restrictions can be targeted as well.

The multi-objective GEA test generation was applied to the Nios SoC to maximise multiple coverage metrics whilst minimising sizes of test programs executed. When compared with existing test generation methods, the multi-objective GEA scheme provided higher or equivalent coverage across all coverage metrics, and utilised much lower test sizes. To extend our multi-objective GEA technique further, the GEA selection and variation can be revised to more efficiently manage multiple objectives; and with parallelism, our multi-objective GEA process can be exploited to enhance the speed of the overall verification.

CHAPTER 7. ATTRIBUTES BASED FUNCTIONAL COVERAGE

This chapter examines our coverage method specially devised for SALVEM. We focus on evaluating the SoC application and design functions exercised. The goal is to quantify measured coverage data into a metric and estimate the comprehensiveness of SALVEM testing from a design functionality perspective. Our method is based on a functional coverage approach. It is novel because it employs partially ordered symbolic domains and symbolic trajectory evaluation to abstract design elements for coverage measuring.

7.1 Introduction

Common code coverage methods like line, toggle or conditional coverage, and other methods like finite state machine or assertion based coverage were developed for test simulations that employ lower level conventional verification and test creation techniques. Whilst they give preliminary assessment of verification effectiveness and are prerequisites for many verification processes, code coverage does not relay much functional information on what has been tested. They focus on the semantics and syntactical analysis of the design code description, or in-depth hardware design characteristics such as pin signal transitions and design processing states. Additionally, some of these methods originated from the software domain and do not account for the concurrent nature of hardware designs. Coverage from a functional perspective is lacking.

We propose functional coverage to complement existing simulation based coverage methods. In our technique, a functional coverage model is developed specifying the coverage goals and SoC design variables to be measured. These variables are termed *attributes* and are chosen specifically to be monitored because they indicate how the design was exercised. Our coverage method can identify and quantify the design functions verified in terms of various combinations of SoC attribute values measured. We denote our coverage method as *attribute combinations* coverage.

Given the size and complexities of SoCs, the number of attributes can be high resulting in large coverage models. In order to measure functional coverage verification, we apply other innovative

methods to constrain the coverage model, represent design attributes, and conduct the measurement process. We adapt techniques from formal verification in our coverage method to realise a semi-formal approach. Specifically, to constrain the size of the coverage model and represent our design attributes in a more efficient and abstract manner. We exploit abstraction mechanisms from *partially ordered* symbolic domains and symbolic trajectory evaluation (STE) techniques [BBS91, BS90, Jai97]. The amount of attributes is reduced by abstracting large sets of attribute values that share common characteristics, and representing them as symbolic partially ordered domains. Essentially, we are able to specify coverage goals and conduct coverage measurements using a reduced set of interesting and important attributes and values only.

In order for a verification methodology to be truly effective, an essential ingredient of the testing process is to evaluate the coverage of the design that has been verified, and what remains untested. For SALVEM, to assess verification comprehensiveness, the type of coverage information measured must be directly related to design operations invoked by our snippets. Existing coverage methods are not as compatible because they do not provide useful information about the SoC applications or design functionalities exercised.

SALVEM requires a coverage method that is able to measure the SoC design functionalities and operations tested. The coverage method should operate within the application domains of the SoC because this is where SALVEM test programs are sourced from. With attribute combinations functional coverage, a mapping between application based test programs to the design functions and operating scenarios tested becomes possible. SALVEM is able to determine what design functions remain untested and how effective the snippets have been utilised, by measuring and examining the combinations of SoC design attributes exercised by the test programs. Next, an outline of the goals of our coverage method is presented.

Goals of the coverage method

With the attributes based functional coverage, we aim to achieve the following.

- Measure the types and range of SoC functional operations and design behaviours exercised, in order to evaluate how comprehensive the SoC was tested.
- Provide traceability between snippets to the functional attributes, and map the test programs to the SoC design functions being exercised. This enables SALVEM to ascertain what SoC features were tested or overlooked, so that the functional coverage can drive verification and test generation.

The coverage measurement process should also be efficient and not require overly intensive resources. The coverage method should be scalable and be usable for coverage measurement of different (possibly larger) SoC designs. The next section presents a high level overview of the coverage measuring process, followed by in-depth formalisation and description of the method in Sections 7.3 to 7.11. In Section 7.12, the usability and effectiveness of the coverage method is demonstrated via experiments conducted during SALVEM verification of the Nios SoC.

7.2 Overview of attribute combinations coverage

The coverage method relies on the identification of measurable attributes of the SoC design to reveal functional operations and behaviours conducted. These attributes are combined with other SALVEM testing information such as snippets to create a graph based coverage model. The graph model is employed by the coverage method to process exercised attribute values in an efficient manner. The extent and types of SoC functions tested are measured by the range of combinatorial values of the attributes exercised by testing. Representation and manipulation of attributes are facilitated by partially ordered domains to focus on critical attribute values only; whilst the graph coverage model and measuring method are based on STE techniques.

Figure 7.1 shows how the succession of various test and coverage building blocks such as snippets, attributes, and domains are progressively build up to form the coverage measuring components of *antecedent consequent* specifications, *attribute combinatorial sets*, and graph based coverage model; before coverage can be measured.

Figure 7.1 Usage and flow of coverage measure components

7.2.1 Control graph

Our coverage model encapsulates all SoC functionalities that must be verified. It describes the possible sequences of snippets and SoC operations that can be carried out with our snippets library. Our coverage model is implemented with control graph structures from symbolic trajectory evaluation (STE). A control graph is used by STE to describe design circuit behaviours to formally verify. For coverage, our control graph depicts test execution flows and operations that can be performed on the SoC by SALVEM. Graph nodes correspond to and represent major functions executed by snippets. Graph edges direct the chain of SoC functions performed.

Like STE, each graph node is tied to an antecedent consequent specification, and a combinatorial set of attributes which acts as measurable coverage goals to attain for that node (and thus snippet). The greater the attribute combinations exercised according to these coverage goals, the higher coverage achieved. Appendix H.2 details our definition of the control graph for coverage measuring. In later sections, Figure 7.7 and Figure 7.9 show an example of a control graph being used for coverage measuring.

7.2.2 Antecedent consequent specifications

Antecedent consequent specifications (denoted as $an \rightarrow cq$) state the conditions that must be triggered during test simulation in order to demonstrate functionalities specified by the graph node was conducted. The $an \rightarrow cq$ denote pre (an : antecedent) and post (cq : consequent) conditions on attributes of the SoC design, similar to how circuit nets or wire connectors are stipulated with logical data in STE. The usage of control graphs and $an \rightarrow cq$ specifications allow graph traversal and $an \rightarrow cq$ specification checking methods to be adapted for coverage measurement. Our coverage graph model is traversed according to the snippets executed from test programs, and the combinations of attribute values exercised as governed by $an \rightarrow cq$ specifications reflect the functional behaviours exercised.

7.2.3 Coverage attribute combinatorial sets and goals

The $an \rightarrow cq$ nodal specifications make possible the concept of coverage goals. We convert $an \rightarrow cq$ into attribute combinatorial sets. Depending on the conditions enforced on attribute values, the range of attribute combinatorial values that must be realised are captured in these sets; which act as coverage

goals as well. During simulation, exercising the required number of attribute combinations from these combinatorial sets indicates whether sufficient SoC functions was carried out by snippets test programs to satisfy coverage goals. The number of attribute combinations exercised also provides the quantitative functional coverage metric with respect to the goals of combinatorial sets.

7.2.4 Attributes

The attributes specified in $an \rightarrow cq$ and combinatorial sets are generally configuration registers of the SoC or other operational data elements of the design, such as processor cache settings or bus data or signal lines. Configuration registers play a major role in the development and running of snippets. They are highly suitable to indicate how snippet operations were performed and what SoC functions were exercised. Register based attributes bridges the test program stimulus with coverage information measured. The realisation or absence of various attribute combinations can be traced to respective SoC functions, and back to snippets that invoke these (exercised or missing) functions.

In our coverage method, identification of SoC functions tested is straightforward using attributes. For example, a simple coverage implementation is to use a set of three attributes such as the direct memory access (DMA) device read, write, and transfer size registers, and their relevant snippets. These attributes indicate what transfer operations were tested amongst which devices, and the amount of transaction data. Using attributes, the aim is to exercise many range and number of combinations of these registers; and measure coverage of attributes to confirm a comprehensive set of SoC functions was verified. However, the difficulty with this approach is the large number of attribute combinations.

7.2.5 Domains

To address large combinations, the coverage model and coverage goals are constrained by directing tests to focus on only the interesting and important SoC operating scenarios. Specifically, we abstract attributes by combining their values into semantically equivalent sets, and performing coverage measuring in terms of these equivalent attribute sets instead of their raw values. To do this, partially ordered domains are defined to represent and group together attribute values. Each domain has a semantic meaning. For example, *boundary* domain values or *intentional-error* values. For each attribute, the values that correspond to these meanings are identified and assigned to their appropriate domains. This reduces substantially the number of combinations to target.

7.2.6 Partially ordered structure for domains

In order to manage domains and attribute combinations effectively, attribute domains are organised into a partial order [BBS91, BS90, RW03]. The partial order structure (detailed in Figure 7.3 later) is applied uniformly for all domains and attributes. It classifies and arranges the domains into a hierarchy. Domains ordered higher up the hierarchy imply attribute values with more specific meanings. The aim of testing is to target higher partial order domains to begin with, and by doing so, exercise more general domains beneath the hierarchy concurrently. This establishes a reduction scheme whereby coverage measuring requires less resource because it aims to use lower domains that represent larger sets of values as more values from higher domains are exercised.

7.2.7 Coverage measuring process

We summarise the measurement process and its interaction with each other. The process consists of two parts, (1) traversing the coverage model graph to match up against nodes, and (2) evaluating the coverage metric as each node is encountered. Figure 7.2 shows the flow of this process. The operations conducted for parts (1) and (2) are described fully in Sections 7.8 and 7.9.

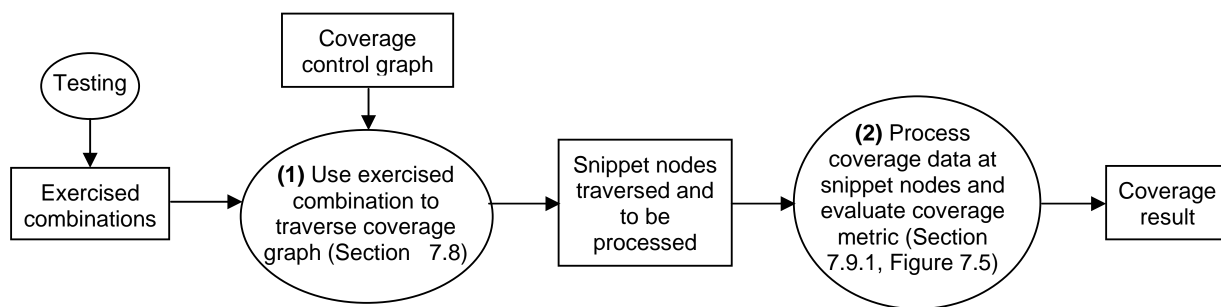


Figure 7.2 Two phase coverage measure process

(1) Graph traversal and snippet node matching

The main components into the coverage measurement process are the coverage model graph and the trace of attribute combination values. Each simulation of a SALVEM test program produces a trace of the attributes' values that were realised during test execution. The sequence of attribute combinatorial values exercised is used to traverse through the coverage control graph. Each exercised attribute combination will match at least one graph node, before the next attribute combination is used to

traverse to and match up against another graph node; and so forth until the trace of exercised combinations are exhausted.

Using partial order domains and operators, our techniques matche graph nodes and traverses the control graph. This process is detailed in Section 7.8. When a node match is successful, we update the graph node to reflect the additional attribute combinatorial values (and implicitly, the test functionalities) covered. In addition, we store information regarding the exercised combination that triggered the node match. Again, a partial order mechanism is employed to process, extract information from, and detect only unique combinations contributes toward the coverage metric. These operations conducted at each node are summarised next in (2).

(2) Coverage measured data processing and metric evaluation

The update and storage of newly exercised attribute combinations at a graph node are facilitated by two types of combinatorial sets, the target combinatorial set (targ_comb) and the cumulative combinatorial set (cumu_comb). The targ_comb specifies attribute combinatorial values that should be exercised during testing. In this sense, the targ_comb act as coverage goals. targ_combs are expressed using partial order domains for each attribute, they capture a sub portion of the coverage space corresponding to SoC functions that must be tested.

During testing and subsequent coverage measurements, progress of coverage goal satisfaction (and coverage metric increments) is monitored by updating the cumu_comb, and comparing it against the targ_comb. The cumu_comb uses domains to represent for each attribute, the combinatorial values exercised during testing thus far. Domains allow for more efficient storage and comparisons as multiple attribute values that are semantically equivalent can be grouped and represented together with a single symbolic variable. As more tests and attribute combinations are exercised, the cumu_comb is updated with appropriate domains to reflect the attribute values exercised. We denote this process as *update and reduction* which is described in Section 7.9.3.

When sufficient tests and attribute combinations have been executed, the domains in the cumu_comb and targ_comb will match each other. This implies that the combinations captured by the cumu_comb and targ_comb are identical, satisfying the coverage goal. From Section 7.4 onwards, the coverage measuring components, and cumu_comb and targ_comb update process are described fully. But first, the coverage metric is defined next.

7.3 Quantitative metric

Our coverage technique defines the quantitative metric based on the number of combinations of the attribute values exercised. From a generalised and simplistic viewpoint, our coverage metrics are calculated as a percentage by the following formulation,

$$\frac{Number\ \ of\ \ exercised\ \ combinations}{Total\ \ number\ \ of\ \ possible\ \ combinations - Number\ \ of\ \ illegal\ \ combinations} \times 100$$

Given the attribute and combinatorial state explosion problem, depending on the SoC, evaluating the coverage metric as above may be impractical, whereby full coverability cannot be achieved. Hence, STE and partial order domain based solutions are proposed in our coverage technique to address this shortcoming. As part of the coverage technique, two quantitative coverage metrics that perform the above coverage metric calculation are defined to approximate and report the final coverage result.

The Best Combinations Count (BCM) and Worst Combinations Count (WCM) are coverage metrics that are calculated from raw coverage measurements of attributes and their combinatorial values exercised during testing. They are both calculated according to the above formula, but are distinguished by the different kind of exercised combinations which can be measured and employed for metric evaluation. The different types of exercised combinations measured can vary based on their uniqueness or duplication during the coverage evaluation process. In our measurement procedure, the BCM and WCM differ in how they are calculated with respect to estimating coverage of unique combinations, handling of repeating combinations, and reducing the effects of over-inflated coverage results.

Instead of a single coverage metric as is typically the case, the WCM and BCM provide a worst and best coverage attainment quantity, indicating possible error margins. This allows design and verification teams to demonstrate satisfaction of test goals whenever the WCM to BCM coverage range overlaps the specified coverage goal. Section 7.9.2 describes BCM and WCM in more detail.

7.4 Attributes and combinations

Our coverage method relies on both on and off chip registers of the SoC hardware design to act as attributes. Alternatively, these attributes could also be hardware design signals. For the Nios SoC, Table H.2 in Appendix H.1 shows the entire set of attributes and their domain range of possible values employed for their coverage measuring. An attribute is defined as follows.

Definition 7.1 : Attribute definition

An attribute a is identified from the SoC design and holds different values during SoC operation.

The domain set of values that an attribute can exercise is $D \subseteq \mathbf{N}$, such that an attribute holds a subset of the positive integers at any time during running of the SoC. For example, if the attribute a is a register of size s bits, then its domain is $D \in \{0, 1, \ldots, 2^s-1\}$ and the number of values to exercise is $|D|$.

□

During testing of the SoC, the goal is to exercise as many combinations of the attributes with as many of their domain values as possible. Attributes are selectively identified to make up different combinatorial set of attributes. For example, to exercise and gauge the comprehensiveness of universal asynchronous receive transmit (UART) testing, a set of three attributes, the UART transfer read/write mode, UART transaction counter, and UART termination condition control registers can be included into an attribute set to indicate what type of UART operations were carried out.

Each combination of attribute values indicates a specific functional behaviour of the SoC being tested. For instance, consider exercising the functionality of reading x number of byte message characters from the UART until an appropriate termination character is transmitted, this operation can be deduced from the three UART attribute combinatorial set above. In our coverage method, numerous combinatorial sets of attributes are defined to measure testing of particular SoC functionality types or focus on specific SoC devices. For example, attribute sets that evaluate interrupts handling, or attribute combinations that examine DMA, Ethernet, and other inter device operations such as processor and off-chip hardware functionalities.

The combinatorial set of attributes that can be utilised to measure SoC functions are defined as follows.

Definition 7.2 : Attribute combinatorial set definition

Let the combinatorial set of attributes used for measuring some portion of SoC testing be defined as a tuple of m attributes, $\langle a_1, a_2, \ldots, a_m\rangle$.

The combinatorial set CMB of possible combinations of attribute values that can be realised during SoC testing is defined as,

$$CMB = \{\langle a_1, a_2, \ldots, a_m\rangle \mid a_i \in D_i \;\forall\; i = 1, .., m\}$$

□

Based on Definition 7.2, the number of combinations that can be exercised for a set of attributes is given by the product of the domain set of values of each attribute making up the combination. Quantitatively, this is expressed as $|CMB| = \prod_{i=0}^{m} |D_i|$, which is an exponentially growing number as more attributes are added.

If one was to follow the rudimentary cross-product approach employed by [UY99, UZ02, Ziv03], attaining full coverability by exercising all combinations would be impractical. Such an approach is not scalable for increasing sizes SoC designs. To address this, our coverage method restricts the number of combinations that must be exercised, but still indicates what types of functions were tested.

To this end, in the following sections, partially ordered domains are defined to contain the range of attribute values to exercise. Additionally, control graphs are employed for coverage measuring to only evaluate attribute combinations that can arise from possible SoC execution. This eliminates many unrealisable combinations which can skew coverage, and prevents needless measuring of combinations that are impossible.

7.5 Partially ordered domains and structure

Our first strategy for containing coverage measuring is to reduce the number of values each attribute is required to exercise, and by doing so, reduce the number of attribute combinations to measure. The attribute values and combinations to exercise are chosen such that only important and interesting SoC operating scenarios are tested. To facilitate this, partially ordered domains are defined.

Domains represent and group together the essential attribute values that must be realised. Each domain encapsulates certain semantic characteristics that are common to all its values. For example, important attribute values that should be exercised during testing are identified to be *Boundary* domain, *Non-Boundary* domain, or *Illegal* domain.

A Boundary domain infers attribute values that exercise corner case scenarios. As an example, for a DMA read address register attribute, the boundary values would cover the start and end address of various memory devices such as Flash memory, and their row configurations or paged segments. Non-boundary domain values could infer alternating bit values address, addresses with upper or lower halves filled with bit zeros or ones, or simply I/O device UART or parallel input/output (PIO) port addresses. Illegal domain signifies attribute values that assert or indicate erroneous conditions on the

SoC. For the DMA attributes, this refer to values such as out-of-bounds read or write DMA addresses, or specifying word-sized transactional size when reading from a byte-size I/O port.

For each attribute, the possible values that it can exercise are sorted and assigned into one of these domains. Therefore, every attribute owns a unique set of partially ordered domain values, which requires less effort to measure because fewer values must be exercised. Table H.2 in Appendix H.1 shows the full list of attributes and domains for the Nios SoC.

Besides restricting the attribute values, the representation and manipulation of these attribute domains can be further enhanced by arranging them into a partially ordered structure, as shown in Figure 7.3.

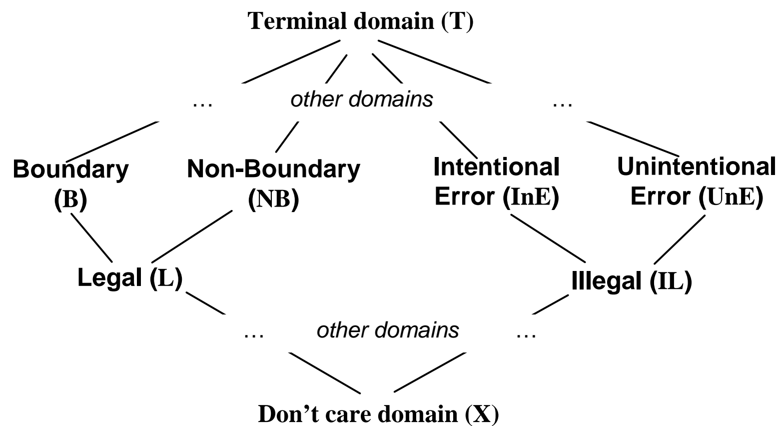


Figure 7.3 Partially ordered domain structure

The goal of the partial order structure is to classify and organise domains into a hierarchy. The partial order is ordered by information content. Domains ordered higher in the hierarchy imply attribute values with more specific meanings. For example, Boundaries or Non-Boundary domains for memories addresses, or interrupt handling or arithmetic overflow error values in the Intentional Error domain. Domain values lower in the hierarchy contain more general attribute meanings such as Legal or Illegal domains – i.e. an Illegal domain represents commonly occurring errors and may include more specific domains such as overflow error Intentional domains.

According to the partial order, an attribute value from a particular domain inherits the semantic meanings of the domains below it. Therefore, lower order domains contain larger sets of attribute values with multiple meanings from the domains above. For example, a Legal domain includes values from the Boundary and Non-Boundary domains

Based on the partial order domain structure of Figure 7.3, the partial order is associated with a relational operator $\leq$ which is used throughout coverage measuring. This operation is used for comparison and evaluation of domains with respect to their ordering in the partial order structure. It is defined in Definition 7.3.

Definition 7.3 : Partial order relation operator ≤

Let (P, ≤) be a partial order, where P is the set of domains of the partial order, and ≤ governs the relations and ordering of these partially ordered domains. Given domains $d_1 \in P$ and $d_2 \in P$,

$d_1 \leq d_2$ if d_2 contains greater than or equal ranges of attribute domain values than d_1, and d_2 is ordered below d_1 in the partial order structure.

□

For example, Boundary (B) ≤ Legal (L) because the Legal domain has greater attribute information content. L covers all legally allowed attribute values during SoC test operations, which includes boundary and non-boundary attribute values. Note that P is a subset of all the domains and domain values D for an attribute a, as described in Definition 7.2; that is, $P \subseteq D$.

Using ≤ during the measurement process, the aim of test and coverage is to target higher partial order domains to begin with, and by doing so, exercise the more general domains below concurrently. This lowers testing effort in later stages for those lower domain attributes that still need to be exercised. The usage of the partial order domain structure is to enable more efficient coverage measuring as described further in Sections 7.7 to 7.11.

In our definition of attribute domains, certain attributes may be denoted as don't-care and assigned to the X domain. The X domain, like many lower ordered domains, are used in attribute combinations to capture multiple attribute values and multiple combinations so they can be represented into an equivalent attribute combination. By exercising and measuring combinations with X attribute domains, this implies other equivalent combinations are also covered and do not need to be specifically targeted any further.

In other contexts, the X domain can also imply the attribute to be ignored for coverage purposes. During test and coverage measurement, at any one stage, certain SoC functions may only involve several devices, other SoC modules are not required. Therefore, any attributes belonging to these dormant devices can be ignored; thereby reducing the number of attribute combinations to examine even further.

The X domain is placed at the bottom of the partial order structure in Figure 7.3 because these don't-care values are either (1) representative of all the remaining values the attribute can exercise, or (2) has already been sufficiently covered by higher order domains.

In Figure 7.3, note that the top level domain is the terminal domain, denoted as T. This domain represents any single attribute value in the entire domain set of the attribute. It is employed as the top level domain simply for completeness purposes in the partial order structure. Figure 7.3 also indicates that other domains can be inserted below the T domain or above the X domain to expand the partial order if required. Currently, the partial order structure employs the domains in Figure 7.3, which is sufficient for our research needs. Table H.1 in Appendix H.1 states these domains and their meanings.

Despite the same partial order structure in Figure 7.3 being employed for all attributes during coverage measuring, note that every attribute has its own unique set of values for each partially ordered domain.

Usage of partial order domains – an example

The effectiveness of partially ordered domains can be demonstrated graphically by a simplistic example examining the coverage measuring test space. The constrained attribute target coverage space for a simple DMA set of three attributes is shown in Figure 7.4. The coverage space includes illegal combinations that must be identified before coverage measure. The measurable coverage space is reduced significantly by considering only three attributes applicable during DMA testing for a DMA based attribute combinatorial set. Other attributes are ignored as don't-care domains.

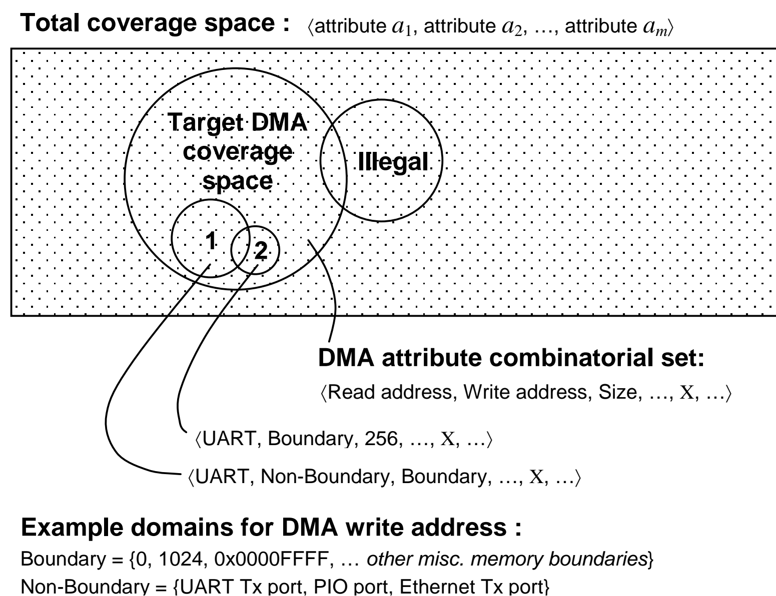


Figure 7.4 Coverage test space visualisation example with partially ordered domains

The DMA attribute combinations space can be reduced further by restricting attributes to values from partial order domains defined in Table H.1 Appendix H.1 and Figure 7.3. For example, constrained space 2 captures attribute combinations corresponding to DMA transfer tests of 256 bytes from the

UART device to any destination address from the Boundary domain. Instead of transferring 256 bytes each time only, constrained space 1 is less restrictive, and refers to DMA operations between the UART to any Non-Boundary domain destination, transferring data amounts of boundary sizes. There is overlap between these coverage regions because both transfer operations originate from the UART.

The use of specially designed attribute combinatorial sets and partially ordered domains reduces the target attribute space significantly. Our aim is to exercise attribute values from these partial order domains only. By doing so, the number of attribute values and combinations that must be exercised to satisfy coverage goals is lower. Instead of a large set of raw attribute values, the coverage method can specify coverage goals and represent exercised attribute combinations using reduced valued domains.

7.6 Specification and manipulation of coverage goals

In order to specify the coverage goals, we employ antecedents and consequents along with their formulation elements. At each node of the control graph (described in Section 7.2.1 and Appendix H.2), individual coverage goals are prescribed to capture what attribute combinations can and should be exercised by that snippet node. Antecedents and consequents are formalised specifications of attributes and partially ordered domain value combinations. Antecedents and consequents allow for attribute combinations that are desired or expected during testing to be measured. We adapt them specially from STE methods for coverage measuring.

To describe antecedent and consequent coverage goals, simple predicate formulations are employed. A simple predicate denotes an attribute to hold a particular domain value; so that values covered by the domain must be exercised by the attribute in order to satisfy the coverage goal. For example, the simple predicate (a is B) signifies the attribute a should only exercised boundary domain values. In order to combine simple predicates of multiple attributes in a coverage goal, the conjunction $\wedge$ operator is employed. This enables different attributes to specify different domains to make up the coverage goal. Appendix H.4 describes our motivations, development, and usage of antecedent and consequent specifications. In addition, the simple predicate and conjunctions are formally defined in this appendix. An example of coverage goal specification is given at the end of this section.

7.6.1 Least upper bound operator

Given the specifications of attributes using antecedents and consequents, the graph traversal and attribute combinatorial measurements at each snippet node can be conducted using various STE related operators. For example, the least upper bound function can be employed to combine partially ordered domains and multiple attributes together. Such an operation is useful for describing the attribute combinatorial set for each snippet arising from antecedents and consequents. That is, the required set of attributes and combinations of attribute domain values can be captured informally with a *CMB* tuple (Definition 7.2) derived from the antecedent consequent specifications.

First, we describe the least upper bound operator for mapping antecedents and consequents to a *CMB* attribute combinatorial set. The least upper bound operator, LUB is defined and utilised according to conventional domain set theory; based on the partially ordered domain structure in Figure 7.3. Namely, given a subset of domains Q from the set of partially ordered domains P, the least upper bound of Q if it exists, is the least domain of Q that is greater than or equal to each element in P.

Recall from Section 7.5 that domains are ordered by information content and arranged according to the partial order structure in Figure 7.3. Therefore, a domain is considered greater than or equal to another domain if it covers more or equivalent range of attribute values. For example, for a subset Q = {B, NB}, the least upper bound of this subset is the legal partial order domain L. According to Figure 7.3, the domains L and X contains attributes values that are greater than or equal to both B and NB. This is because L and X domains contain attributes values from both B and NB, and other domains above it in the partial order structure. The L domain is chosen over the X domain as the least upper bound because it is the lesser domain of the two. L covers less attribute values compared to X, it contains less information content. Diagrammatically, the least upper bound of a subset of domains can be determined from its partial order structure (i.e. Figure 7.3) by identifying the most shallow common lower root domain that is shared by all domains in the subset.

For our purposes, we define the LUB operator to operate on two domains at a time. That is, to attain the least upper bound domain from a subset of two domains only. The LUB is defined in Definition 7.4.

Definition 7.4 : Least upper bound operator (LUB)

Let $Q = \{d_1, d_2\}$ be a subset of two partially ordered domains from the set of attribute partial order domains $(P, \leq)$.

P = {T, B, NB, InE, UnE, L, IL, X} is described in Table H.1 Appendix H.1 according to Figure 7.3, and the partial order relation $\leq$ from Definition 7.3 is used to define the relations between domains. d_1 and d_2 are the two domains for the LUB to operate on.

The least upper bound of d_1 and d_2 is a domain $p \in P$ such that,

(1) $d \leq p$ for all d in Q, and

(2) for any other r in P where $d \leq r$ for all d in Q, it holds that $p \leq r$.

Operator wise, LUB is a commutative operator whereby,

$$d_1 \text{ LUB } d_2 = \begin{cases} p & \text{if } (d_1 \leq p) \wedge (d_2 \leq p), \text{and } p \leq r\ \forall r \in P \text{ where } (d_1 \leq r) \wedge (d_2 \leq r), \\ d_1 \cup d_2 & \text{otherwise} \end{cases}$$

If no least upper bound p can be found such that (1) and (2) above are satisfied, then the union of domains is used instead. This indicates that the domains d_1 and d_2 are disjoint, so no common domain can be employed to represent them. Instead, the attribute value ranges of both domains are maintained and employed for further coverage measuring needs.

□

In Appendix H.4.1, Table H.3 describes a truth table of the LUB operation amongst domain defined for our partial order structure in Figure 7.3. The strength with conjuncting simple predicates and defining a LUB operator is so that formalisation of coverage goals can be easily converted into attribute combinatorial sets. For more efficient manipulation, we denote the converted attribute combinatorial set as the target combinations (targ_comb). The targ_comb is a representation of the coverage goal used for coverage measurements.

7.6.2 Coverage goal conversion to attribute combinatorial set – an example

As an example, consider a snippet node capturing coverage information using four attributes. Let an antecedent or consequent specification using f_1 to f_6 to represent conjunction of simple predicates be as follows.

$$(f_1 \wedge f_2) = ((f_3 \wedge f_4) \wedge (f_5 \wedge f_6)) = (((a_1 \text{ is B}) \wedge (a_2 \text{ is InE})) \wedge ((a_3 \text{ is NB}) \wedge (a_2 \text{ is UnE})))$$

where $f_1 = (f_3 \wedge f_4)$, $f_2 = (f_5 \wedge f_6)$, $f_3 = (a_1 \text{ is B})$, $f_4 = (a_2 \text{ is InE})$, $f_5 = (a_3 \text{ is NB})$, and $f_6 = (a_2 \text{ is UnE})$.

We define the δ function to break down the conjunctions of f_1 to f_6 first. The δ function is formally defined in Definition H.5 Appendix H.4.2. It operates on conjunctions of simple predicates to produce the attribute combinatorial tuple for each simple predicate; before they are combined using the least upper bound operator according to Definition 7.4 for each of the positional matching attributes of each tuple. This process is shown below.

$$\begin{aligned}
\delta(f_1 \wedge f_2) &= \delta(f_1) \text{ LUB } \delta(f_2) \\
&= \delta(f_3 \wedge f_4) \text{ LUB } \delta(f_5 \wedge f_6) \\
&= \delta(f_3) \text{ LUB } \delta(f_4) \text{ LUB } \delta(f_5) \text{ LUB } \delta(f_6) \\
&= \delta(a_1 \text{ is B}) \text{ LUB } \delta(a_2 \text{ is InE}) \text{ LUB } \delta(a_3 \text{ is NB}) \text{ LUB } \delta(a_2 \text{ is UnE}) \\
&= \langle \text{B, X, X, X} \rangle \text{ LUB } \langle \text{X, InE, X, X} \rangle \text{ LUB } \langle \text{X, X, NB, X} \rangle \text{ LUB } \langle \text{X, UnE, X, X} \rangle \\
&= \langle \text{B, IL, NB, X} \rangle
\end{aligned}$$

At each snippet node, the complete attribute combinatorial set can be extracted by combining both antecedent and consequent specifications using the least upper bound operator. For instance, if f_{an} and f_{cq} are the antecedent and consequent respectively, the combined attribute combinatorial set is given by $\delta(f_{an})$ LUB $\delta(f_{cq})$ to attain the targ_comb.

Employing X and other partially order domains promotes abstraction. The use of X and lower level domains encapsulates all possible concrete attribute combinations as if these domains were replaced with specific values. By exercising and measuring attributes that match these domains, many other equivalent attribute combinations capturing similar SoC tested functions can be considered covered at one go.

The use of domains in antecedent, consequent, and combinatorial set specifications restricts and reduces the realisable attribute combinations that must be exercised as depicted in Figure 7.4. The abstract combinatorial attribute set defines a searchable space where testing should focus on, and how this space can be covered in a minimal approach. The verification effort can then focus on exercising attribute combinations that will add value to coverage toward these test spaces rather than an ad-hoc random manner. The targ_comb attribute combinatorial sets converted from antecedents and consequents are used for control graph traversal purposes, which we described in the next section.

7.7 Coverage measure

Given the coverage components of Sections 7.4 to 7.6, the coverage measurement process and how these coverage elements are employed is explained in this section. Together, the set of attributes, partial order domains, control graph, antecedent consequent specifications, and attribute combinatorial

sets, form the coverage model in our coverage measuring method. The coverage model is intended to be specifically devised for coverage measurement of different SoCs. Hence, it is customised to manage minimal attributes, domain values, and combinations, but still provide useful coverage feedback. Tailoring the coverage model for different SoCs is simply a process of identifying the attributes and relevant domain values for the hardware design.

The coverage measurement process consists of two main operations, (1) control graph traversal via snippet node matching, and (2) evaluation of the coverage metric and coverage contribution of exercised attribute combination for each traversed snippet node. The following two sections describe these operations.

7.8 Coverage measure: graph traversal and node matching

The first stage of coverage measurement involves matching the exercised trace of attribute combinations to snippet nodes in the control graph; and by doing so, traversing the graph. During graph traversal, whenever a snippet node matches, this signifies the exercised attribute combination contributes to coverage of that snippet node. The attribute combination must be examined further to update the coverage metric. The attribute combination evaluation and metric update is described in the next section.

For graph traversal, the coverage model is designed such that only one snippet is able to match an exercised attribute combination. That is, selection of only one outgoing edge from a node to the next matching node is guaranteed. Appendix H.5 describes our graph traversal procedure fully. In this section, we focus on snippet node matching because it is most crucial to the traversal process.

The snippet node matching checks only minimal set of attributes needed to facilitate graph traversal. Our snippet node matching is based on mechanisms from STE circuit states checking. STE constructs multiple sequences of circuit states based on antecedents and consequents specifications of circuit node values into a simplified circuit state, called a defining trajectory. Using this trajectory, multiple circuit states exercised can be checked against the antecedent asserted states and expected consequent states using a single check operation, reducing the number of circuit states that need to be checked. An example of the STE checking for a simple inverter circuit is shown in [Che03].

In a similar vain to STE trajectory circuit checking, for coverage, multiple combinations of attributes values are combined into a single attribute combinatorial set specification, in the form of the target

combination (targ_comb). Recall that the targ_comb is an antecedent and consequent derived attribute combinatorial set expressing the coverage goals at a snippet node. It captures attribute combinations that must be exercised during testing for coverage measurement, and is discussed further in Appendix H.7.

Using partially ordered domains in the targ_comb, many combinations of attribute values can be specified from corresponding attribute antecedent and consequents into a single combinatorial set using the δ operator demonstrated in Section 7.6.2. Then based on the single targ_comb at each snippet node, like STE, many exercised attribute combinations can be similarly checked and matched against the targ_comb in one go.

The targ_comb captures many attribute combinations to exercise, and at the same time, reduces the range of attribute combinations that need to be realised to those restricted by the partially ordered domains only. This allows many exercised combinations to be matched against the snippet nodes using less checking of individual attributes. A single targ_comb can be matched against many combinations. Using one targ_comb at each node for comparisons implies fast snippet node matching and faster graph traversal overall. The snippet node matching process is defined in Definition 7.5.

Definition 7.5 : Control graph node matching operation

In a control graph g (Definition H.1 and Definition H.2 in Appendix H.2), for any snippet node $n \in N$ with m attributes $\langle a_1, a_2, \ldots, a_m \rangle$, an exercised combination of attribute values $ex = \langle e_1, e_2, \ldots, e_m \rangle$ matches the targ_comb attribute combinatorial set targ_comb = $cmb = \langle d_1, d_2, \ldots, d_m \rangle$ if,

$$\forall\ a_i \text{ in } \langle a_1, a_2, \ldots, a_m \rangle,\quad e_i \leq d_i \text{ for } i = 1, .., m$$

where cmb is an alias for targ_comb, d_i is the specified partially ordered domain for attribute i in the targ_comb, and ≤ is the partial order relation operator from Definition 7.3.

□

Implementation-wise, to execute control graph traversal, the pseudo code for conducting snippet node matching as stated in Definition 7.5 is shown in Appendix H.6. An example of the control graph traversal according to exercised combinations from multiple tests is shown later in Figure 7.7 Section 7.9.3.

7.9 Coverage measure: coverage metric evaluation

Following on from control graph traversal, this section details the procedures conducted for coverage metric evaluation at each snippet node. After an exercised attribute combination is deemed to have matched the targ_comb attribute combinatorial set, the exercised combination is processed to (1) update the coverage metric (Section 7.9.2), and (2) capture its coverage contribution in an abstracted manner (Section 7.9.3).

To aid metric evaluation and capture coverage information of the test functions conducted, the concept of a cumulative attribute combinatorial set (cumu_comb) is introduced. For each snippet node, a unique cumu_comb keeps track of the range of attribute combinations exercised during testing, to assist in the evaluation of the quantitative coverage metric. During testing and coverage measure, progress at achieving targ_comb coverage goals at each node is monitored using the cumu_comb. Like the targ_comb, the cumu_comb is defined as a tuple of attributes that are applicable to the snippet node. That is, cumu_comb $\in CMB$ from Definition 7.2.

Each tuple element holds a partially ordered domain equivalent to the attribute values exercised thus far belonging to that domain. The cumu_comb employs the same domains (described in Figure 7.3) as the targ_comb to represent for each attribute, the values and combinations exercised during testing. As more tests are executed and more attribute combinations exercised, the cumu_comb is updated with appropriate domains to reflect the set of attribute values covered. When sufficient tests and attribute combinations have been exercised, the domains for attributes in the cumu_comb will eventually match that specified in the targ_comb. This implies the attribute combinations required by the targ_comb are covered by the cumu_comb, and the coverage goal for that snippet is considered fulfilled. The next section examines in detail how cumu_comb is employed for coverage measuring, whilst Appendix H.8 describes an example of cumu_comb usage.

7.9.1 Coverage metric evaluation scheme

The coverage measuring process updates the coverage metric with every uniquely exercised attribute combination. A combination is considered unique if it was not previously exercised by any of the current or preceding coverage measured tests. Whenever a newly exercised attribute combination has been processed, information regarding the attribute combination is recorded in a database; because accurate coverage measuring requires comparisons against previous combinations, in order to reduce false measurements of duplicate combinations. There is little gain in repeatedly stressing the same

design functions and inflating coverage results misleadingly. Appendix H.9 discusses the common approach of processing an exercised combination against every previous combination, and the associated drawbacks with this strategy.

To address these shortcomings, our coverage method reduces the amount of combinatorial information required to evaluate the coverage metric. Rather than examine raw combinations of attribute values directly, the partial order domains and only attribute values belonging to those domains are manipulated. In particular, the targ_comb and cumu_comb are employed extensively to avoid raw attribute combinatorial values where ever possible.

Many previously exercised combinations captured in the cumu_comb can be compared simultaneously. Rather than compare an exercised combination against all prior combinations, the exercised combination can be evaluated against the cumu_comb with a single ≤ partial order operation; to determine if it exercises any new attribute domain values. The detection of any new domain values signifies immediately the uniqueness of the combination. In such cases, comparisons against previously exercised combinations can be avoided. Not all exercised combinations need to be stored if they are already captured by cumu_comb. Our attribute combinations measuring is outlined in Figure 7.5.

In Figure 7.5, the shaded components represent our unique operations that are conducted, compared to a typical combinations evaluation flow described in Appendix H.9. Exercised combinations are first examined against the cumu_comb using ≤ operator (Definition 7.3). If the operator returns ‘NO’, then comparisons against individual combinations are still required. But comparison is performed against a subset of prior combinations stored only. The capture of information from previously exercised combinations and their abstraction into the cumu_comb eliminates the need to store all exercised combinations into the database. Comparison of subsequent new exercised combinations is then reduced, by only checking against the most recent set of prior combinations, which is much less than continually accumulating all combinations.

At pre-determined stages during test and coverage measure, the cumu_comb will take in database information from all exercised combinations thus far; to facilitate future efficient combinations comparisons. This process is called *update* and *reduction* and is described in Section 7.9.3. But next in Section 7.9.2, we describe the conditions under which the coverage metric is calculated to provide a quantitative measure.

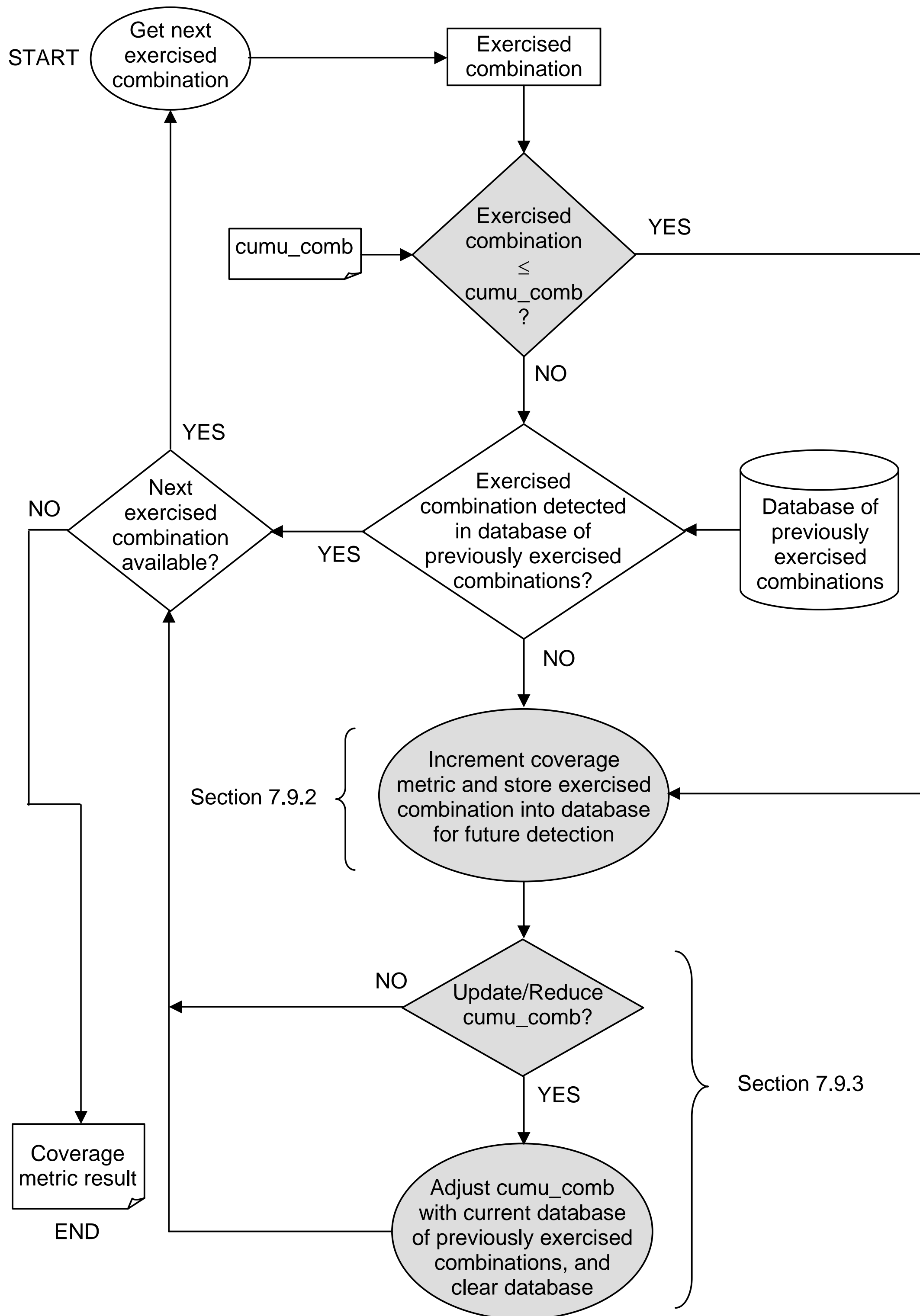


Figure 7.5 cumu_comb coverage measure and metric evaluation

7.9.2 Coverage metric update and calculation

Recall from Section 7.3 that the coverage metrics, Best Combinations Metric (BCM) and Worst Combinations Metric (WCM), are defined to quantitatively assess coverage. The evaluation of these coverage metrics are based on Figure 7.5. Uniquely exercised combinations are detected to update the BCM and WCM.

The BCM is considered an optimistic count of the maximum number of exercised combinations that contributes to test coverage. Combinations that are not detected from the database of the most recent set of previously exercised combinations triggers the BCM to be incremented regardless of the cumu_comb. Hence, the BCM is optimistic in the sense that the coverage from an exercised combination is included despite the fact that the combination may have been exercised by other earlier combinations (but not within the most recent stored combinations database). Appendix H.10 discusses further the uniqueness of combinations and their effects in BCM evaluation.

In contrast, the WCM is a pessimistic coverage metric of the minimum number of exercised combinations that satisfies targ_comb. The combinations calculated by the WCM must be strictly unique. Not only must the combination be unique within the stored combinations database, it must also exercise at least one new attribute domain value. If the combination exercised an attribute domain value that is not captured by the cumu_comb, this guarantees the combination and any other combinations with this domain value were not previously exercised at any stage during testing; even prior to the recently exercised set of combinations stored.

The conditions for evaluating WCM and BCM are outlined as follows.

Increment,
(i) WCM, when exercised combination exercises at least one unique attribute value in cumu_comb,
(ii) BCM, when exercised combination has not been previously exercised within current test window of database of most recent combinations.

The comparison and evaluation of many exercised combinations for BCM and WCM update is efficient by using a single partial order operation against the cumu_comb. The BCM evaluates the greatest number of combinations exercised under the assumption that none of the exercised combinations were previously exercised, even prior to the current set of stored exercised combinations during test and coverage measure. The WCM provides the lowest percentage of combinations exercised in the worst case, whereby all combinations must be detected as strictly unique. The detailed procedure to calculate the BCM and WCM is defined in Definition H.6 at Appendix H.10, and is part 3

of the complete coverage measurement process flow chart in Figure 7.10 in Section 7.11. The pseudo code implementation of the coverage metric calculation is given in Appendix H.11.

The conditions under which the BCM and WCM are evaluated rely on update and storage of the cumu_comb from exercised combinations, and the partial order domain operations conducted. The effectiveness and details of these operations, termed *update* and *reduction* is described next.

7.9.3 Update and reduction of the cumu_comb

Update and reduction of cumu_comb facilitates monitoring of coverage goals attainment with respect to the targ_comb. By updating the cumu_comb, previously exercised combinations that are currently stored for future coverage measuring can also be discarded. Thus, besides indicating coverage progress, actively updating each attribute in the cumu_comb reduces the amount of prior information needed for further coverage measuring. This provides more efficiency for the coverage metric evaluation process.

Updating the cumu_comb with most recent and accurate coverage data is crucial for coverage measuring. In particular, we focus on the process of replacing for each attribute, up to date domains that covers the set of exercised attribute values at any stage during coverage measure. The outcome is to retain data from exercised combinations in the cumu_comb in an abstract manner. A two stage process is employed, (1) *update* and (2) *reduction*, which are performed for each snippet node traversed.

During update, every exercised combination that satisfies the targ_comb (as described in Definition 7.5) is analysed to determine its contribution to the attribute combinatorial test space. The exercised combinations are approximated in the cumu_comb by employing partial order domains to represent many exercised attribute values belonging to these domains. Initially, every attribute in the cumu_comb is assigned the T domain to represent at most one exercised attribute value from the first combination extracted from testing. As more attribute combinations are exercised and examined, the goal is to update and eventually replace the attribute domains in the cumu_comb with other partial order domains lower in the partial order structure (Figure 7.3), to represent greater number of attribute values exercised. The replacement of attribute domains with lower ordered domains forms the second stage of the cumu_comb update process termed reduction.

The goal of reduction is to reduce the attribute domains in the cumu_comb with lower ordered domains until they match corresponding attribute domains specified in the targ_comb. In this way, the test and coverage process is guided toward the targ_comb coverage goals at the snippet node. The continual reduction of domains adheres to the partial order structure in Figure 7.3. When both the

cumu_comb and targ_comb match up to each other with the same domains for each attribute, this implies all desired attribute values and majority of the combinations exercisable by the snippet (as specified by its targ_comb) has been realised. Note that domain reduction of attributes in the cumu_comb is conducted by reducing domains one level at a time. This is detailed in Appendix H.12.

Update and reduction process at each attribute

The update and reduction process of the cumu_comb is summarised in Figure 7.6 and is performed for each attribute in the cumu_comb. If the existing domains of an attribute in the cumu_comb cannot be updated or reduced with other domains identified from corresponding attribute values in the exercised combination, then the process is repeated for the next attribute in the cumu_comb.

Update of domains can only be carried out if the exercised attribute domain values were not previously realised, and are uncovered by the cumu_comb. Also, reduction is possible only if the previously unexercised domain values enable all applicable domains of the root domain to be completely exercised. For any attribute undergoing reduction, the root domain to reduce to is identified by taking the least upper bounds of existing domains in the cumu_comb.

Update and reduction definitions

The update and reduction process is formally described as follows. In the first stage, the cumu_comb is updated with newly exercised attribute values, thereby capturing their coverage contribution. The cumu_comb update process is defined in Definition 7.6.

Definition 7.6 : cumu_comb update process

For any attribute in the cumu_comb, let d be the current domain in the cumu_comb capturing previously exercised values of that attribute. Let e be the corresponding attribute domain value of an exercised combination.

The domain d in the cumu_comb is updated with the exercised attribute domain value to be $(d \cup e)$ if,

(i) $e \leq d$ is false, where $\leq$ is the partial order operation defined in Definition 7.3, operating on the partial order structure in Figure 7.3.

If the above condition $e \leq d$ is true, this implies the exercised domain value was previously realised by prior testing, and is covered by the greater set of values that d in the cumu_comb already captures. That is, d is ordered lower in the partial order structure and represents many attribute values from the domains above it, including e.

□

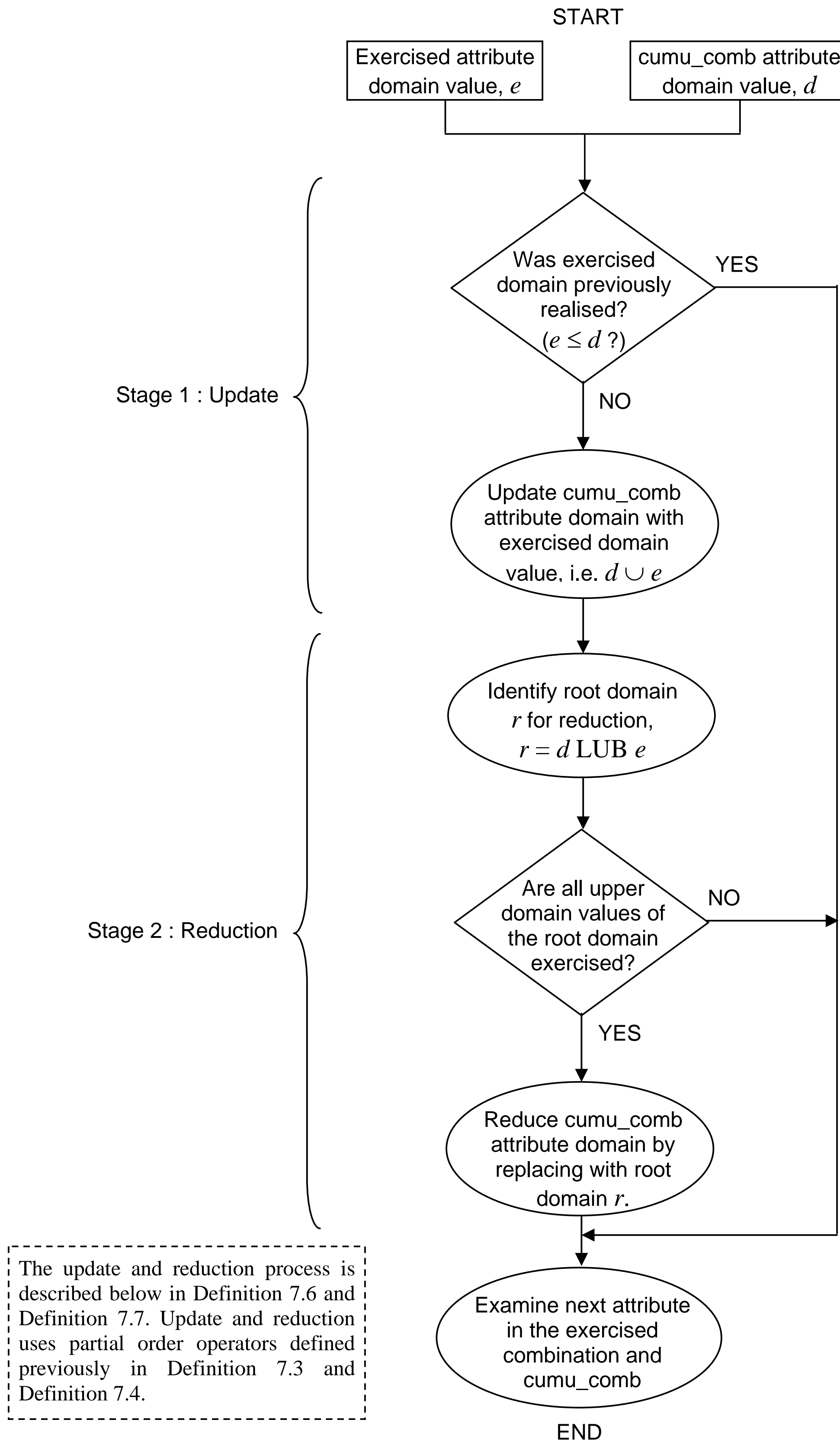


Figure 7.6 Update and reduction flow per attribute

The second stage, reduction, is dependant on the update stage, and is only carried out if previously unexercised attribute domain values were detected by the first stage (i.e. condition (i) in Definition 7.6 holds). In the second stage, for every attribute, newly realised domains from those attributes are examined with existing domains in the cumu_comb. If all these applicable higher level domains have been exercised, then the domain assigned for the attribute in the cumu_comb is replaced with the lower level root domain. The reduced root domain is attained by the least upper bound of the higher level exercised domains. The reduction operation is defined as follows.

Definition 7.7 : cumu_comb reduction process

Let r be the root domain of the exercised attribute domain e and current domain d assigned in the cumu_comb such that,

$r = e$ LUB d where LUB is the least upper bound operation in Definition 7.4.

Let U be the set of upper level parent domains ordered one level immediately above the root domain r.

The currently assigned domain d in the cumu_comb is reduced by replacing d with r if,

$\forall\ u \in U, (u \leq e) \cup (u \leq d)$

That is, all attribute values of the next higher level domains above the root domain r and beyond have been exercised previously, and are captured by either the most recent exercised attribute domain e or the existing cumu_comb domain d. When all the attribute values captured by r have been exercised, this implies the domains above r which were assigned in the cumu_comb can be reduced and represented by r solely.

□

The implementation details to conduct cumu_comb update and reduction is given in Appendix H.13.

Update and reduction example

The concept of cumu_comb update and reduction can be best demonstrated by a simple coverage measuring example in Figure 7.7. The figure shows a sub portion of the control graph and some snippet nodes used for coverage measuring. Each node is specified with the antecedent consequent specification and the associated targ_comb. During testing, the exercised combinations are examined to match against the targ_comb of each node to facilitate graph traversal (Section 7.8). The coverage provided by every exercised combination is processed and captured by the relevant snippet node following each graph traversal step. To evaluate the coverage metric, each snippet node contains a set

of the most recent previously stored combinations and its cumu_comb. To demonstrate how cumu_comb undergoes update and reduction, we focus specifically on the InitDMA snippet only.

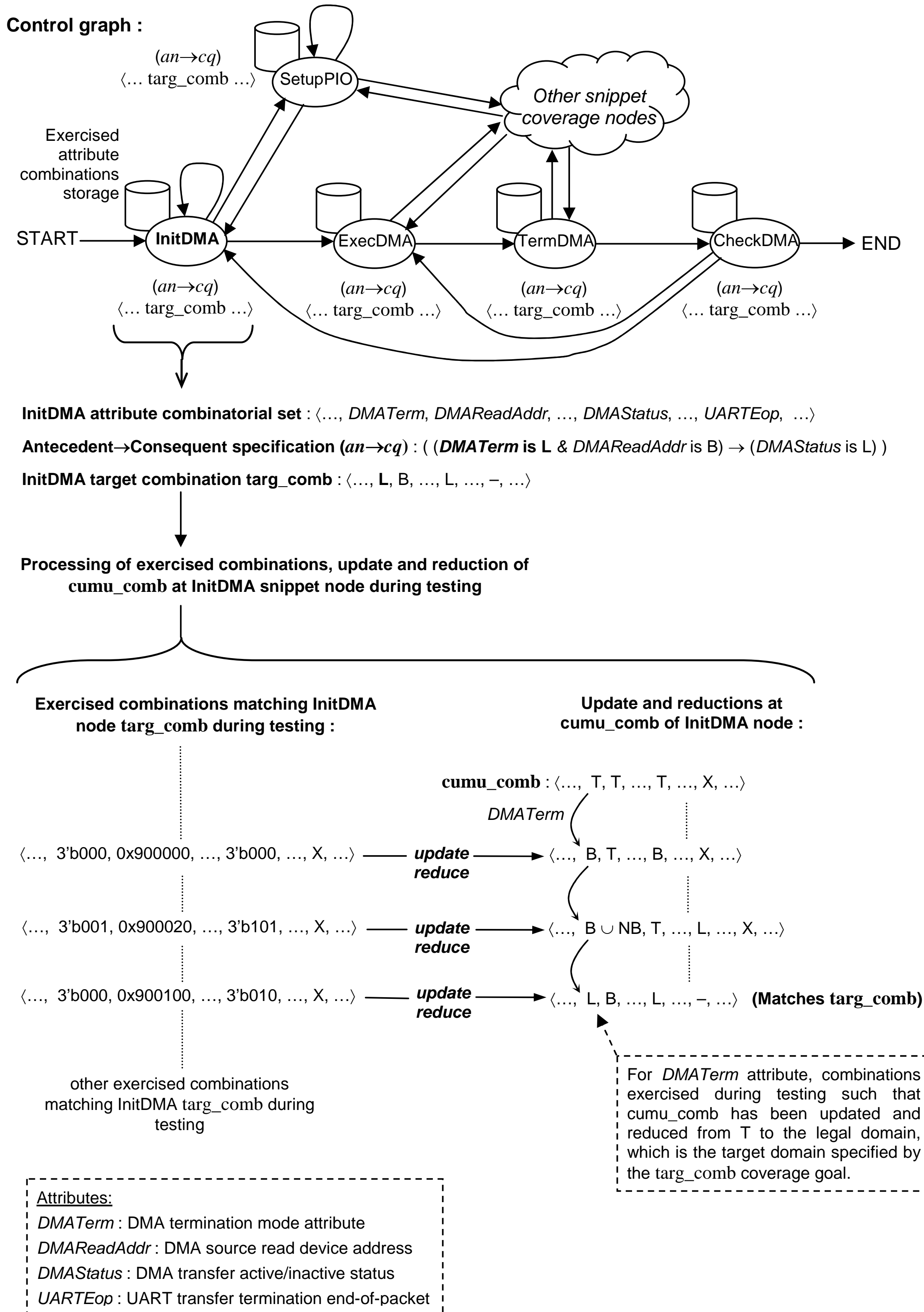


Figure 7.7 Example of coverage measuring cumu_comb update and reduction

During testing, whenever graph traversal arrives at the InitDMA snippet node with an applicable exercised combination, the domains of each attribute in the InitDMA cumu_comb is evaluated against corresponding attribute domains in the exercised combination. Beneath the InitDMA node, a selection of the possible exercised combinations that traverses the InitDMA node during testing is shown. At various stages during testing, whenever another exercised combination from either the same test or another test is encountered, the cumu_comb is updated and if possible, reduced. The update and reduction of the InitDMA cumu_comb is also shown next to the stream of exercised combinations that match InitDMA.

As more combinations are exercised, it is clear that more of the applicable domains get covered. Subsequently, this is updated for any of the applicable attributes. Whenever all attribute values of a domain are exercised, this presents the opportunity for this domain and existing domains of the corresponding cumu_comb attribute to be reduced to a lower level domain. Eventually, continual update of higher level domains with exercised attribute values will lead to realisation of lower level domains in the cumu_comb that matches corresponding domains in the targ_comb.

Specifically, note that the InitDMA snippet targ_comb specifies the Legal domain for the DMA termination mode register (DMATerm). This implies SALVEM tests must exercise DMATerm attribute values from the Legal domains and domains above it. As more combinations are exercised, lower level domains will replace the current domain representing DMATerm. For example, the cumu_comb is updated initially with attribute values from the Boundary domain, then Non-Boundary domain, and eventually reduces to the Legal domain as required by the targ_comb.

This update and reduction of exercised attribute values applies for all attributes in the coverage combinatorial set of the InitDMA (and indeed all other combinatorial set of other snippet nodes). For example, in Figure 7.7, when all the attribute combination values from a particular domain are exercised, the current higher order domain will be reduced to lower domains after each exercised attribute measurement. This process continues until all updating attribute domains matches the targ_comb domains. This approach targets full coverability for smaller individual targ_combs, before focusing on the full SoC coverage combinatorial set.

In the example, the InitDMA targ_comb did not specify domains for all attributes. This is to demonstrate certain attributes can be specified with the X domain instead. This indicates those attributes can be ignored for coverage purposes (e.g., UART end-of-packet register). That is, the antecedent consequent specifications $an \rightarrow cq$ only need specify attributes whose values are applicable to essential functional operations of this DMA snippet.

The example in Figure 7.7 demonstrates one of the key concepts of our SALVEM test and coverage strategy. That is, to selectively define a restricted target coverage test space that requires specific test focus, and to direct testing toward these important design functions and monitor their progress. The cumu_comb to targ_comb comparison during testing indicates what remaining attribute domain values still need to be exercised to cover the desired test space. And by doing so, these unrealised attribute values reflect the types and range of SoC functionalities that can be triggered by snippets, but are still untested. In this way, the coverage method is able to drive SALVEM testing; by ensuring SoC operations that can be exercised by snippets are fully verified. For example, at a DMA snippet node, unexercised DMA termination mode attribute values that mismatch between the cumu_comb and targ_comb shows the untested DMA transfers which terminate under other conditions.

Expanding on the example in Figure 7.7, Appendix H.14 provides an extensive and more detailed example of the overall coverage measuring, and specifically, the update and reduction process conducted at a snippet node during graph traversal.

Update and reduction issues to consider

Update and reduction of the cumu_comb enable recently stored combinations exercised previously to be discarded; because the coverage contribution from these combinations are captured in the cumu_comb in an abstract manner. Whilst update and reduction of the cumu_comb can reduce the storage requirements of exercised combinations, and lessen the amount of comparison processing to calculate the coverage metric, some drawbacks exists.

First, there is overhead in performing the update and reduction, which become significant for overall coverage measuring if update and reduction is conducted repeatedly or too often. Second, discarding previously exercised combinations reduces the complete set of coverage data to perform coverage metric evaluation, and thus its accuracy as described in Section 7.9.2 may diminish as well. These shortcomings were uncovered during preliminary coverage measuring experiments with our method. For example, extended processing time was required before sufficiently useful coverage measure data was provided. Even if sufficient compute resource is provided, there is still a need to optimise and make better use of such resources to provide coverage results within reasonable time frames.

Therefore, the frequency at which update and reduction of cumu_comb is carried out is an important factor for coverage measuring performance. Too frequent or repeatedly updating and reducing the cumu_comb will slow coverage measuring and give misleading coverage results. On the other hand,

insufficient update and reductions will make the coverage approach of employing partial order domains and its benefits redundant. All coverage measurements will default to examining raw attribute values and large set of previously accumulated exercised combinations again; resulting in coverage state explosion problems which the coverage method was designed to tackle originally.

Given these considerations, there must be a trade off between the intended benefits and shortcomings of cumu_comb update and reduction. With this in mind, the concept of *windowing* is introduced in order to identify and better manage when and how often to conduct update and reduction.

7.10 Coverage windowing

With windowing, every exercised combination updates the cumu_comb, but reduction of the cumu_comb (and discarding of previous exercised combinations) is only carried out when a certain number of combinations have been exercised and processed. That is, reduce the cumu_comb only when a *window* of specified number of combinations is updated. Updating the cumu_comb and discarding exercised combinations affects the accuracy of coverage metric calculation. Hence, by only reducing for each window of combinations rather than reducing and discarding combinations each time, a certain level of accuracy is preserved. Only when exercised combinations storage capacity is filled (which is determined by the window size), reduction will take place to ensure resources requirements and comparison processes do not become excessive.

The windowing procedure

The windowing scheme is based on a divide-n-conquer approach to lessen the computational load; whilst still able to supply useful coverage results. During coverage measuring, the stream of attribute combinations data is divided and processed within individual segments. These segments are denoted as *windows* and they capture a pre-determined number of combinations for measurement each time. The windowing approach is illustrated in Figure 7.8. Attribute combinations exercised during testing are stored and processed to update the cumu_comb within their currently active window stage only. As testing continues, eventually, a pre-designated number of combinations will occupy the window until the window capacity is filled. Reduction is then conducted once as described in Section 7.9.3. Following this, a new window stage is initiated to handle the next set of combinations streamed from test executions.

By employing a windowing scheme, smaller groups of exercised combinations are selected for processing each time. Every new attribute combination is processed against other combinations within its window only, reducing the degree of analysis needed and the computational and memory requirements, compared to analysing the entire stream of coverage combinations all at once.

The effectiveness of windowing is particularly sensitive to the window size. The goal of windowing is not only to ensure all exercised combinations are updated in the cumu_comb each time, but perform reduction only when appropriate. This provides the best balance between the need for storing sufficient exercised combinations to maintain coverage metric accuracy, and discarding these combinations when storage and processing resources become too great to cater for. The experiments in Section 7.12.2 demonstrates the effect of window sizes from actual test runs and coverage measuring, whilst Appendix H.15 discusses window sizes from an analytical perspective.

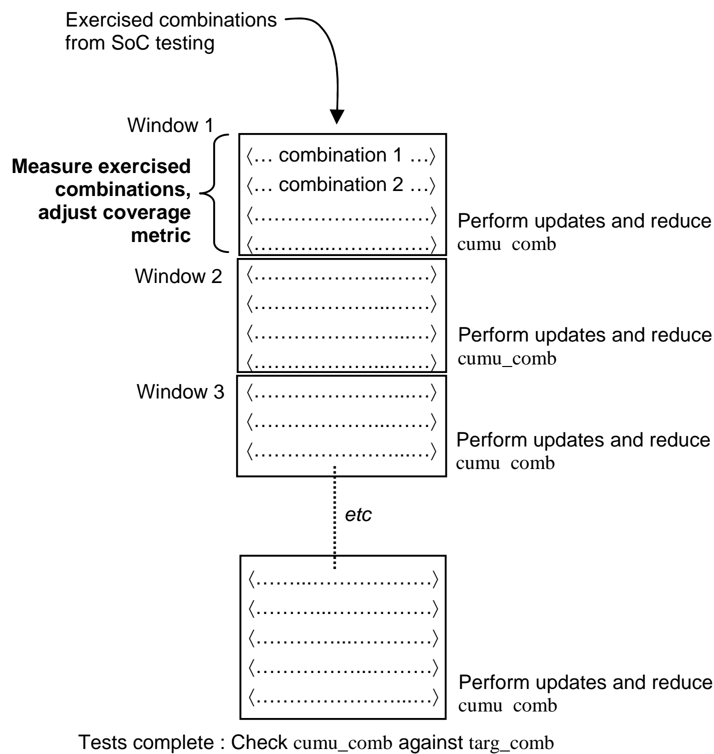


Figure 7.8 Windowing scheme for update and reduction of cumu_comb

7.11 Complete coverage metric evaluation process – a review

This section presents a review of the overall coverage method tying together all coverage measuring components and concepts from Sections 7.3 to 7.10. Figure 7.9 illustrates the notion of processing exercised combinations in order to traverse the control graph and measure coverage contribution to snippet nodes. The exercised combinations from each test invoke a new traversal of the control graph each time. Hence, coverage contribution measured by a snippet node can originate from different exercised combinations from multiple tests, or from exercised combinations of the same test that traverses the control graph and the snippet node multiple times in a cyclic manner.

The coverage measuring process can be broken down into a number of steps corresponding to the coverage descriptions in Sections 7.8 to 7.9 above. From a given test, the first step is to gather the next exercised combination from this test, and the antecedent consequent specification, targ_comb, and cumu_comb of the relevant snippet node to facilitate graph traversal. Next, the second step performs snippet graph node matching (Section 7.8) to traverse the control graph. The quantitative coverage metric evaluation is then carried out according to Section 7.9.2 in step 3. Step 4 updates and reduces the cumu_comb of the current traversed snippet node by adhering to the windowing scheme in Sections 7.9.3 and 7.10. And finally, when all exercised combinations from every executed test and their associated graph traversing coverage measurements are complete, the remaining step is to calculate the final coverage metric value and collate the existing coverage data. Collation of the coverage data involves a final update and reduction of the cumu_comb. By doing so, coverage measuring and evaluation of the same coverage metric with further new tests can resume from where earlier tests runs were completed. Figure 7.10 shows the coverage measuring flow of operations as a control flow chart. The bold highlighted steps correspond to coverage operational steps in Figure 7.9 and the step-wise description above. Appendix H.16 provides a step-by-step description of the flow in Figure 7.10; which summarises our complete coverage procedure from earlier sections.

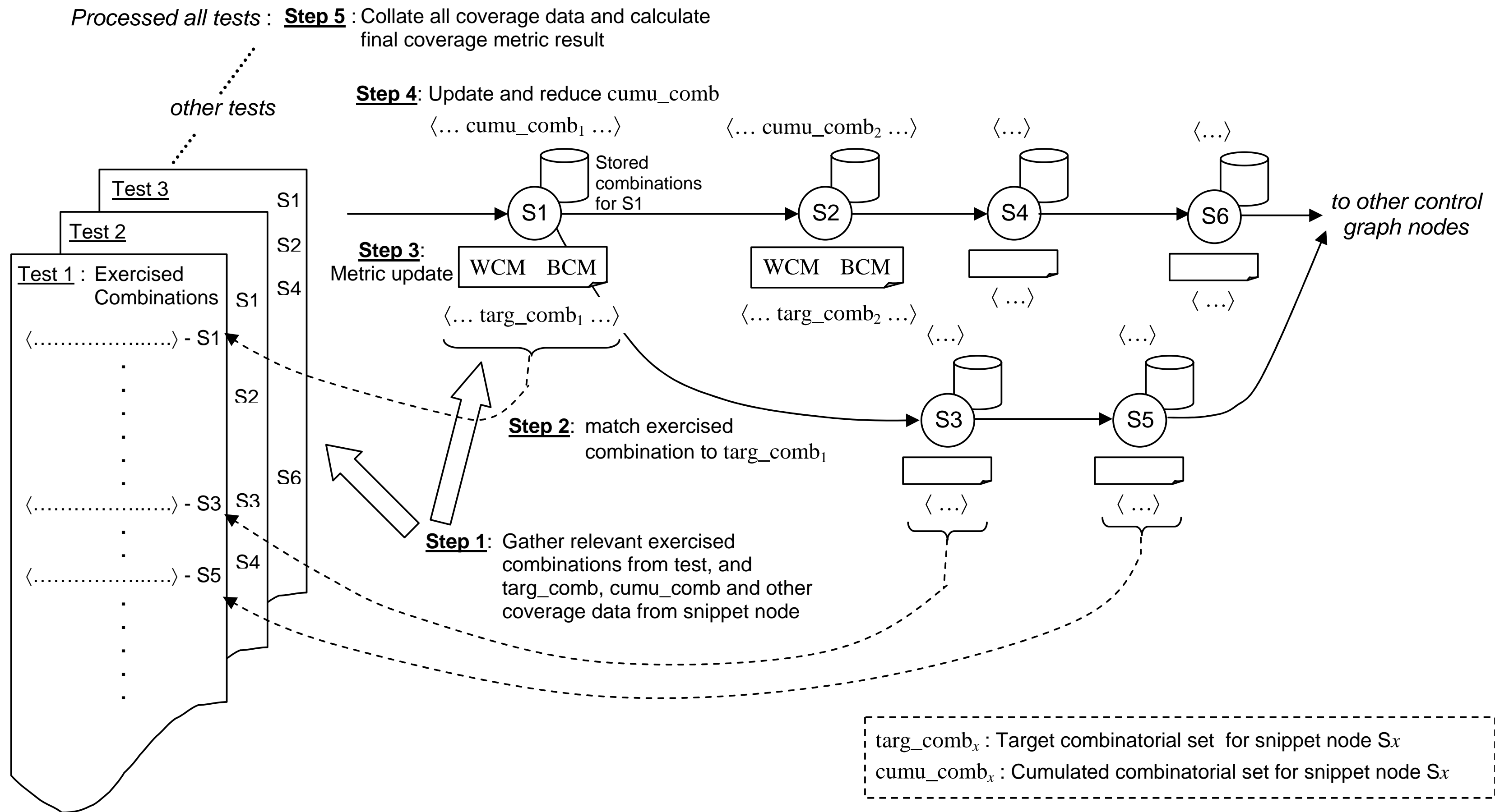


Figure 7.9 Coverage measuring with graph traversal from multiple tests

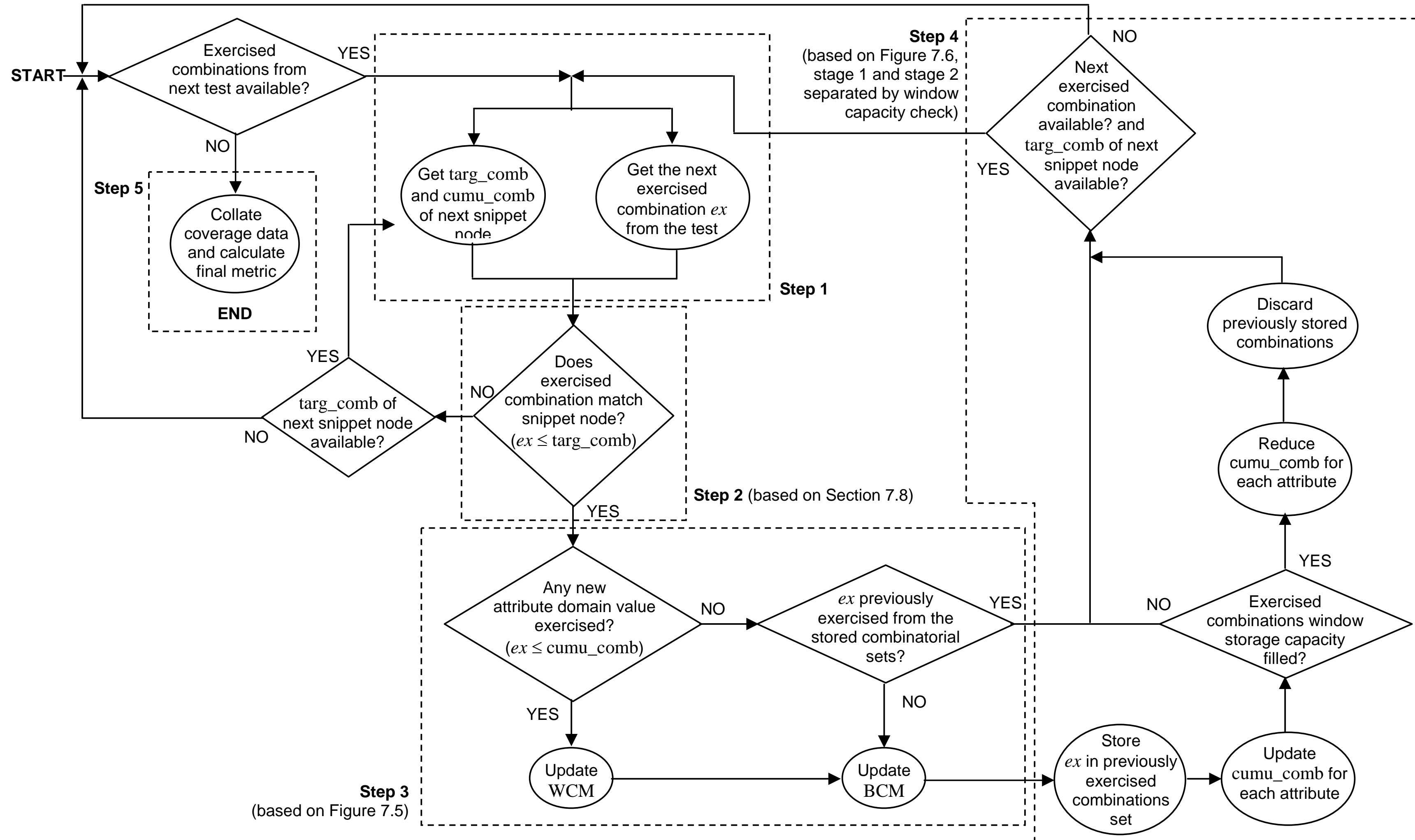


Figure 7.10 Overall coverage measuring control flow

7.12 Experiments and results

Our experiments focus on two main areas of investigations. In Section 7.12.1, we demonstrate the workings and practically of our attribute combinational coverage approach. This section also presents comparative analysis against existing coverage solutions. The other experiments in Section 7.12.2 are to examine coverage windowing. Specifically, we analyse and identify the best coverage window size and favourable coverage measuring conditions for the coverage method. Our experiments are performed with the Nios SoC, and employs randomisation SALVEM test generation and verification. The set of attributes, domain values, and control graph employed in our experiments are detailed in Table H.2 Appendix H.1, and Figures H.2 to H.7 Appendix H.3.

7.12.1 Experiments to study attributes based functional coverage

In our experiments, the use of partial order domain for attributes reduces the combinations and memory requirements significantly. In fact, without attribute domains, if using raw attribute values, the number of possible attribute combinatorial values to exercise is in the order of 6.4×10^9, based on the product of the possible raw values of each attribute. Employing partially ordered attribute domains, the number of possible measurements to cater for is substantially reduced to more manageable 24 million combinations. These calculations were performed according to our reasonings in Section 7.4. Also, when conducting coverage measuring with raw attributes directly, the coverage method became stagnated and did not even complete. The number of combinations was too many to process and no reliable result could be given. Note that our coverage measuring experiments for the remainder of this section employed an un-initialised window size, whereby update and reduction is conducted repeatedly.

Attribute combinations coverage results and comparison with traditional coverage

The coverage technique was employed to measure a Nios SoC test suite of 20 tests of approximately 200 snippets each. The test suite was also measured by traditional methods such as line, toggle and conditional coverage, but they do not provide any useful information on what functional applications were actually verified. From the SALVEM perspective, a high line, toggle, or conditional coverage may not necessary imply the SoC was thoroughly verified – a large percentage of functional operations may still be untested.

Instead, the test suite should be measured by our attribute combinations coverage approach. Table 7.1 provides the number of combinations exercised by the test suite for the overall SoC. The difference between the total number of combinations and BCM is the number of repeated combinations detected, thereby preserving the accountability criteria in our approach. The WCM indicates how many effective and uniquely exercised combinations were guaranteed by the test suite. The final coverage results in the bottom row are calculated by the generalised attribute combinations exercised over total number of possible combinations (less illegal combinations) equation in Section 7.3.

Table 7.1 Attribute combinations coverage data

Combinations ($\times 10^6$)	SoC	CPU	Memory	DMA	UART	PIO
Total	4.50	0.81	3.77	4.24	2.11	1.22
WCM	0.51	0.004	0.46	0.51	0.18	0.17
BCM	4.39	0.77	3.67	4.14	2.04	1.19
$\lvert CMB_{illegal} \rvert$	11.5	0.03	10.4	11.4	4.07	3.77
$\lvert CMB \rvert$	24.8	0.12	22.4	24.7	8.76	8.10
Coverage %	33%	99%	31%	31%	43%	27%

Figure 7.11 plots our combinations coverage results in Table 7.1 alongside traditional coverage measures for the test suite. Compared to previous line, toggle and conditional coverage, combinations coverage is much lower for all devices except the CPU. Most non-CPU snippets make use of the CPU indirectly and add to CPU coverage implicitly. However, the remaining results confirm conventional coverage figures were slightly inflated indicating false coverage that should not be as high. For example, in the DMA device, 100%, 82% and 93% were reported for line, toggle and conditional coverage respectively. However, our combinations coverage show only 31% of the DMA was tested.

The lower DMA coverage assessed by attribute combinatorial values should not be considered as verification failure. Rather it shows that testing is incomplete, and there exists opportunities to further improve the quality of the SoC design; by targeting untested functions uncovered by missing attribute combinations. Lower coverage is desired and should be reported when insufficient tests are executed or inadequate verification is performed. High coverage, especially when there are deficiencies in testing, simply covers up a lack of functional correctness in the design. The lower coverage indicated by attribute combinations shows there are missing functionalities not tested by SALVEM, which traditional coverage could not detect. These uncovered functions can be exposed by analysing attribute combinations coverage further at the snippets level.

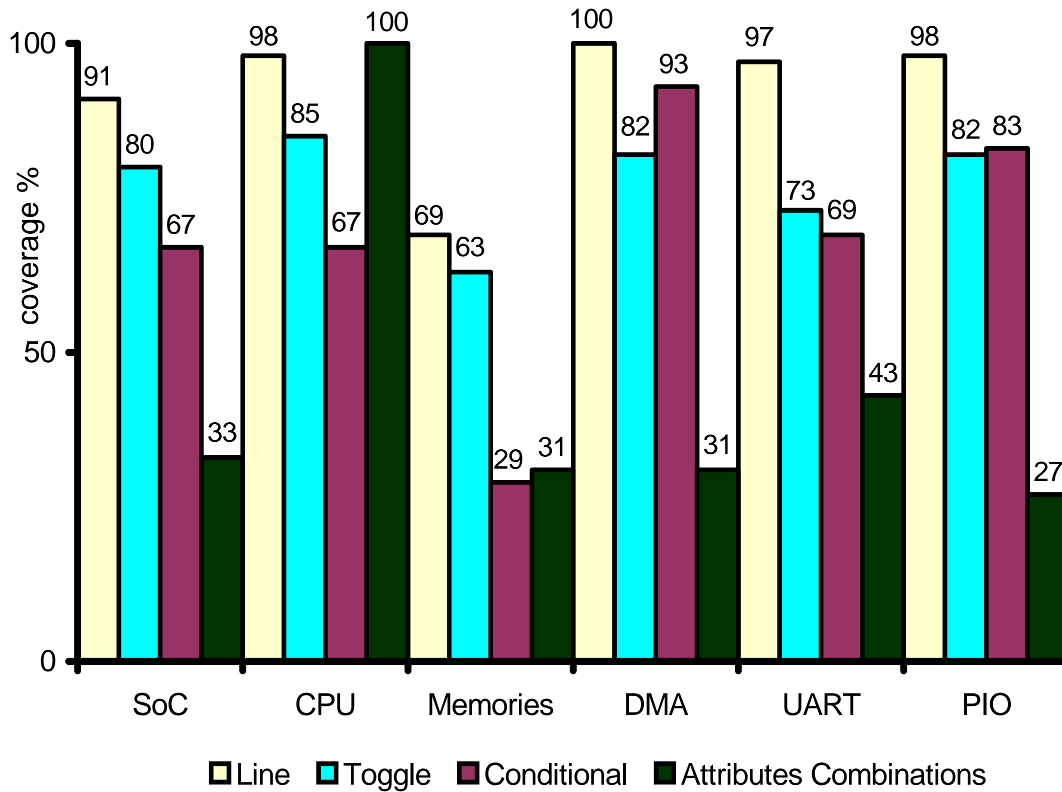


Figure 7.11 Attribute combinations coverage versus traditional coverage

Attribute combinations coverage relating to snippets usage

Table 7.2 shows the utilisation coverage for a selection of the snippets used in our randomised test suite. It examines how many of the possible attribute combinatorial values that can be exercised by each snippet were realised using this test suite. Based on Table 7.2, the amount of attribute combinations exercised corresponding to each snippet can be identified for analysis.

Table 7.2 Individual snippets combinations coverage

InitDMA	15%	ResetUart	100%	DFT	100%
ExecDMA	27%	UartRx	100%	MatrixMultiply	80%
SetPIODir	100%	UartTx	100%	WriteROM	76%
UsePIO	100%	UartRxTx	100%	ReadTimer	67%

In Table 7.1, despite lower coverage for the UART and PIO devices, the test suite attained full coverage for the UART and PIO snippets (Table 7.2). This implies the UART and PIO snippets were thoroughly deployed by the test generator in the test suite. All functional behaviours tested on the UART and PIO using these snippets have been exhausted. New snippets are needed to exercise the UART and PIO more extensively and increase their coverage.

Also, coverage for the DMA snippets shows a small subset of the available snippet functional modes were used. Further investigations reveal certain operational modes of the DMA were not tested even though conventional coverage suggest otherwise (e.g., 100% DMA line coverage). Examining the domain attribute values for the DMA with our combinations coverage, a number of crucial operational values were not assigned to the DMA read, write, length, and termination mode registers. Transactions between the minimum and maximum RAM, ROM, SRAM, and Flash memory address ranges, and the UART ports were not tested at all.

After identifying these untested scenarios, a test case was created to specifically target these functions. The revised DMA coverage from this single test improved to 32%. Coverage for the overall SoC was enhanced to 34%. Further coverage analysis and directed test creation would improve DMA coverage further. Similarly, for the other SoC devices, combinations coverage was low indicating many other missing scenarios can be deduced for additional testing.

Unlike traditional coverage, the above example scenario shows attribute combinatorial coverage provides direct traceability between what SoC functions are tested to the attribute combinatorial values exercised, and eventually, to the snippet test building blocks used to invoke these SoC operations. Based on the unexercised attribute combinatorial values, the untested SoC functionalities can be explicitly identified. Traditional coverage cannot feedback such information easily to the test generation phase. They do not focus on functionalities tested.

Attribute combinations coverage progress trend

Figure 7.12 shows coverage improvement measured by each metric as the number of snippets and test programs executed increases. The trends shown are similar for test and verification duration times. Functional attribute combinations coverage increases steadily with the number of snippet tests. For conventional line, toggle and conditional measures, coverage levels out very early, and achieves peak values using less than 500 snippets. Whilst conventional coverage has reached its maxima, for the same number of snippet tests, attribute combinations coverage is at 33%. Furthermore, the attribute

combinations graph line indicates additional testing would continue to enhance this coverage. Therefore, verification can be conducted using combinations coverage as a means to monitor progress and guide additional coverage analysis and testing.

With attribute combinations coverage, many more snippets and tests can be executed whilst being guided by this coverage metric. This metric and its coverage data enable testing of other functional test regions that were either undetected or incorrectly considered covered by traditional coverage methods. For the randomised test suite in Figure 7.12, because attribute combinations coverage does not saturate immediately when small to medium tests are already executed, many more tests and snippets can be created to facilitate further effective verification. Specifically, coverage improvements can be steadily achieved to provide a higher functionally correct design overall. As the coverage is allowed to improve at a stabilised rate, ample coverage information from attributes can be carefully analysed to direct SALVEM verification toward full functional coverage. Note however, that the combinations coverage analysis and automated directed test generation for the remaining untested SoC scenarios are beyond the scope of the work in this chapter.

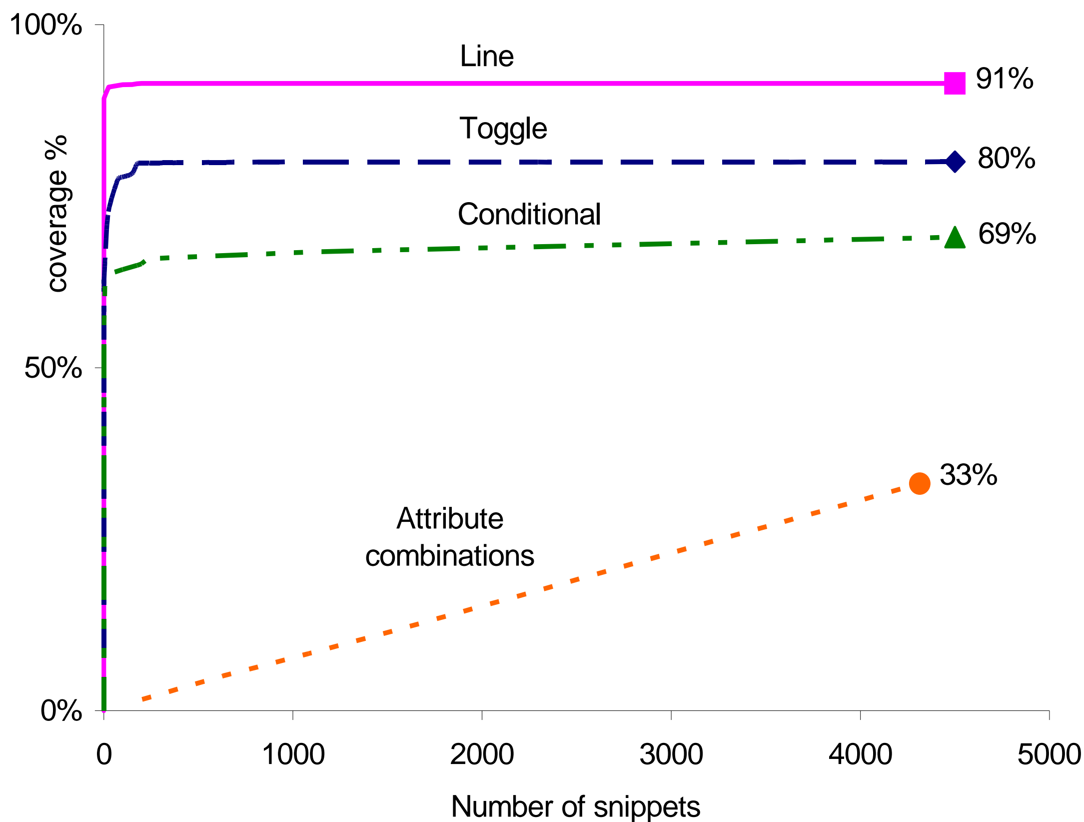


Figure 7.12 Coverage versus number of snippets

Attribute combinations coverage processing times

Another experimental assessment of attribute combinatorial coverage is to compare its measurement execution times with other methods. Table 7.3 shows the timing results. The results are recorded by measuring total execution times of verification when each coverage method is used, and taking the difference between each of these times against a verification run without any coverage measuring. This shows the additional overhead time taken up to perform each of the coverage measurement methods.

Despite processing large amounts of attribute combinatorial values so as to provide functional coverage information, the overhead times for attribute combinations coverage is only greater with respect to line and toggle coverage, but is less than the conditional coverage metric. The results were expected given line and toggle coverage require less complicated measurements and recording of tag-like information in the design descriptive code. In fact, the main overhead with attribute combinations coverage is to record the trace of attribute combinations during test executions only. Coverage measuring can be conducted independently or in parallel to test executions, either during or after the test is completed. For this reason, the overhead time of attribute combinations coverage is lower than conditional coverage, which is the worst offending measurement, and requires more complex assessment of SoC conditional paths executed. Performance-wise, there is no reason why attribute combinatorial coverage cannot be applied effectively.

Table 7.3 Coverage execution overhead times

Coverage method	Attribute combinational	Line	Toggle	Conditional
Average overhead time per test (CPU seconds)	5,412	694	715	9,329

Note 1 : Average test execution time without coverage measuring is 856 CPU seconds.
Note 2 : Average test execution times with coverage measurements are 6,268 CPU seconds for attribute combinational coverage, 1,550 for line, 1,571 for toggle and 10,185 for conditional coverage.

Attribute combinations coverage for other test generation verification schemes

Given the limited timeframe of the research candidature, to gain some initial preliminary results, the attribute combinatorial method was adapted in a minimalistic approach for measuring coverage of genetic evolutionary generated test suites (according to the methods in Chapters 4 and 6). Much of the coverage infrastructure such as the attribute domains and control graph was reused as is without complications. Despite not designing and fine-tunning the coverage method for the evolutionary test

method, the coverage result gained was 52%, which is clearly an improvement from the randomised test suite results discussed above.

The improvement is expected given genetic evolutionary test suites are more effective as shown in Chapters 4 and 6. This demonstrates the attribute combinatorial coverage method is a useful coverage metric that can be incorporated with genetic evolutionary methods to facilitate superior SALVEM verification.

In summary, our coverage method provides more accurate estimation of the SoC applications behaviours tested, and exposes useful functional test information for subsequent coverage investigation and directed snippet testing.

7.12.2 Configuring attributes based functional coverage measuring

The second phase of our experiments is to examine the coverage measuring set up. Specifically, the parameters and operating conditions under which measuring takes place must be selected carefully given they can be conflicting. For example, a large coverage window is beneficial for processing many unique attribute combinations, but this uses up significant portions of memory. The selection of coverage measuring parameters must satisfy a suitable trade-off amongst each other. Otherwise, bottlenecks in the measurement process may eventuate, especially for other larger SoC designs. Appendix H.15 provides an analytical assessment of these trade-off considerations.

In this section, experiments were conducted to characterise the performance and usefulness of attribute combinations coverage results. The aim of these experiments was to determine the best operating conditions for executing attribute combinatorial measurements, in particular, to establish an optimised window size. In our experiments, using a range of window sizes, attribute combinations measurements were taken for a randomised SALVEM test suite. Besides recording WCM and BCM coverage results, memory usage and CPU processing times were also collected to assess the impact from different coverage measure configurations. The results are shown in Figure 7.13 and Figure 7.14.

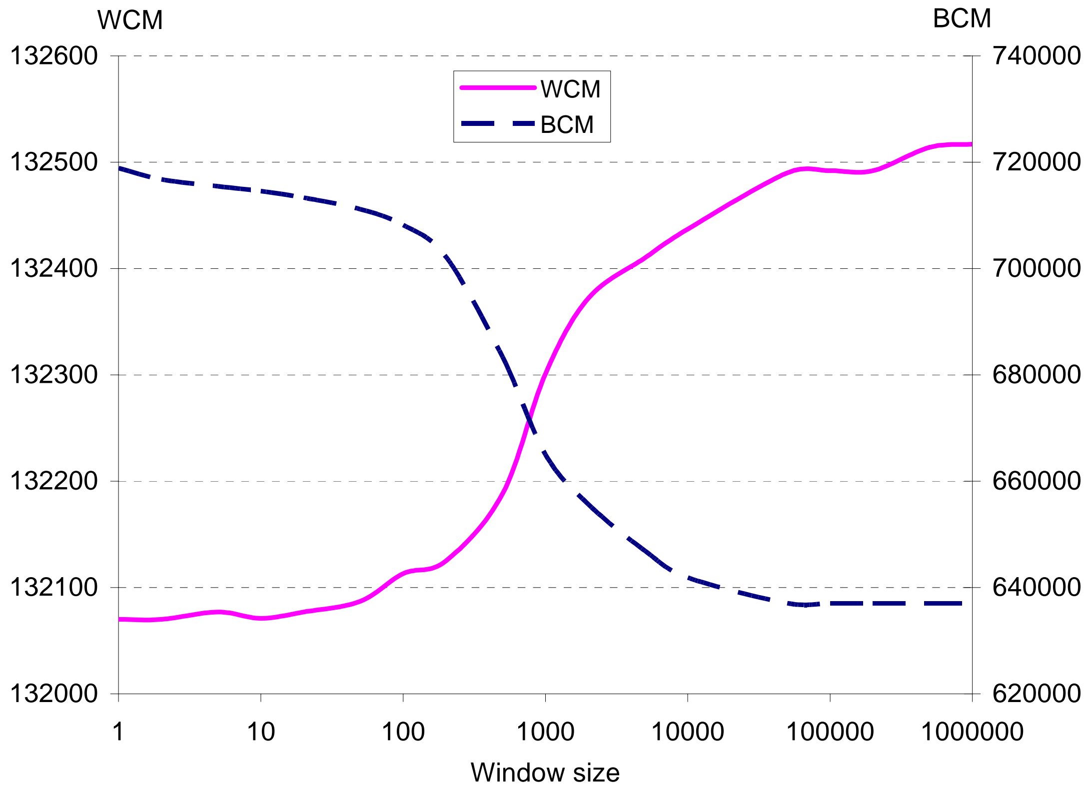


Figure 7.13 Coverage metric results versus window sizes

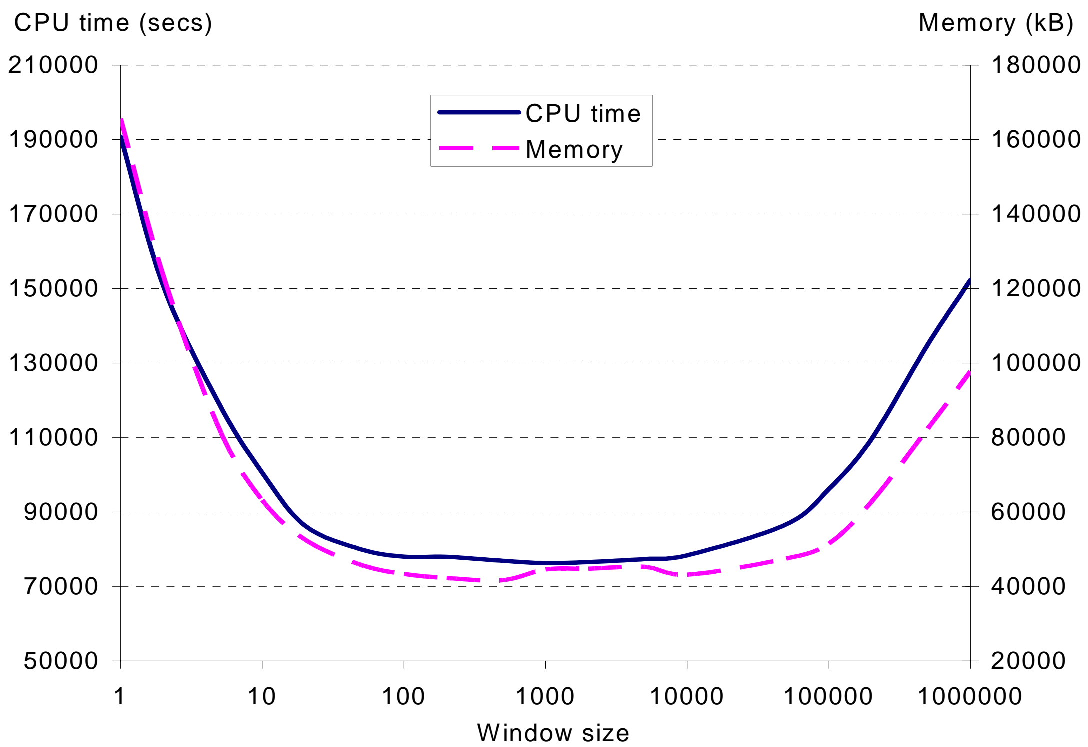


Figure 7.14 Coverage measure and resource usage performance versus window sizes

Coverage measurements were conducted for window sizes up to one million on a logarithmic x-axis to show changes in coverage metric results and performance as windows are rapidly enlarged. From previous experiences with SALVEM test runs, it was unlikely a SALVEM test case would execute sufficient SoC functions to output more than a million attribute combinations at a time for any one test. Therefore, a window of one million is equivalent to a single window, whereby all attribute values and combinations are stored and processed together; - i.e only a single reduction and cumu_comb update is performed at the end of coverage measure. In contrast, a window of one indicates no attribute values and combinations are stored, the cumu_comb is updated and reduction occurs for every combination processed. A window size of one is impractical for coverage purposes, and was used here for experimental purposes only.

In Figure 7.13, the WCM trend line is initially low for small window sizes before rising rapidly at window size 100 and then levelling off for windows above 100,000. The WCM line confirms our analysis in the previous section. For small window sizes, less attribute values and combinations are stored, hence less newly exercised domain values can be detected before they are updated in the cumu_comb. As the window increases, more opportunities for detecting new domain values are available prompting WCM to rise; until all new domains are eventually exercised within the one single window. The WCM stops increasing for windows larger than 100,000 because the large size of such windows provide sufficient storage for all newly detected domain value combinations.

The BCM line is inverse to that of the WCM. For smaller windows, BCM is high and decreases as the window size is enlarged, before levelling off. For larger window sizes, more combinations can be stored, therefore the likelihood of detecting a previously identical combination and repeated SoC test function within the same window is higher. Hence, BCM decreases as these repeated combinations are excluded, providing more accurate BCM results. For window sizes above 100,000, like the WCM, the number of identical combinations detected saturates. Uncovering all repeated combinations within such a large window is guaranteed. In Figure 7.14, the CPU time and memory usage trend lines are similar to each other, and reveal a parabolic pattern. The CPU time and memory usage is high for smaller window sizes, but resource requirements decreases rapidly as the window size increases. Coverage measurement is quickest and uses less memory for windows between 1,000 and 10,000. For windows above 5,000, it increases rapidly again. The rise in compute resource as the window increases is directly attributed to the larger number of attribute values and combinations that can be stored per window. A larger storage of attributes and combinations imply more processing is required to examine and analyse each exercised combination. For example, updating the cumu_comb, or detecting repeated SoC functional combinations by examining a larger set of stored combinations. Similarly, the

increased number of stored combinations and attributes in larger windows require additional storage memory, accounting for the increasing memory graph line.

From our experiments, an unexpected observation was the unusually high CPU time and memory for small windows initially. Upon further investigation, it was discovered the high compute resource usage was due to the increased total number of reductions and cumu_comb updates that were carried out during coverage measurement; despite a lower quantity of stored combinations and attributes to handle in each window. A smaller window size signifies many window stages are needed to process the same number of exercised combinations. And a greater number of windows imply more updates are required for the cumu_comb per number of combinations during coverage measure. Therefore, additional CPU time and memory are needed for more reduce and update operations overall. As window size increases, the total number of updates and reductions decreases and resource requirements are reduced; before extra resource requirements are requested again to cater for the increasing number of stored combinations in larger windows. Using our strategy of capturing abstract exercised combinations information, and update and reduction of the cumu_comb, significant savings in memory and processing of up to 30% can be achieved when the appropriate window size is selected.

The coverage WCM, BCM, CPU time and memory usage graphs was analysed to determine an optimised window size. The graphs indicate a number of constraints. The window size should be within 100 to 10,000 to ensure minimum CPU time and memory usage; thus ensuring the coverage measurement can be completed and executed efficiently in the test flow. However, examining the WCM and BCM coverage graphs, it is clear that a larger window size of at least 50,000 or greater would provide the best coverage results in terms of accuracy and coverage information. Despite the need for effective WCM and BCM coverage results, it is important to apply an appropriate windowing size to limit the processing and storage of too many attribute combinations in large windows. Therefore, we determine the best window size to be 10,000.

Finally, besides windowing, other coverage parameters must also be considered. For example, the types and number of domains and domain values chosen for attributes has a direct bearing on the size of the coverage model. The domains and targ_comb coverage goals are specifically chosen to prevent an oversized coverage model. However, the selection details are beyond the scope of the work in this chapter.

7.13 Conclusions

This chapter described the formulation of a coverage technique to measure the types and range of functional operations verified by SALVEM. The key to our coverage method is to identify and monitor combinatorial set of attribute values in order to measure functional behaviours tested. Using partially ordered domains and abstraction techniques from STE, the combinatorial state explosion problem is addressed. The attribute coverage state space is reduced and the requirements on the measuring process are lower because only the important and desired attribute combinatorial values need to be evaluated.

For each snippet, using specifically devised attributes and their domain values, there is direct traceability between the coverage information measured, to the SoC behaviours exercised and the snippets executed. This allows for identification and measurement of functionalities that the snippet triggers and the SoC utilises. Hence, directing SALVEM testing to verify uncovered test scenarios is possible.

Experiments were conducted applying our coverage method for SALVEM verification of the Nios SoC. The attribute combinations coverage provided useful functional information about what was tested and what test scenarios were not exercised. The coverage method is also adaptable by adjusting combinations window sizes. The coverage measuring can be tuned to operate more effectively as demonstrated from experiments.

The drawback with the method is the upfront investment needed to devise attribute domains and identify a suitable set of attributes. However, under most circumstances, once domains are devised for a design, they have high likelihood for reuse. Attribute sets on the other hand is often more specific to the design under test, because functionalities and features of designs tend to vary. Regardless, once suitable domains are proposed, our coverage method puts in place the infrastructure to facilitate coverage measuring in an efficient flow, reporting valuable functional test information.

CHAPTER 8. CONCLUSIONS

The three key areas of the design verification research in this thesis are: (i) the software application level verification methodology, (ii) test generation, and (iii) coverage.

8.1 Verification methodology research

In system-on-chip (SoC) design verification, during early stages of the design cycle, little consideration is given to the functional interactions invoked by software that will eventually control the SoC. The problem with application based testing during pre-silicon simulation phases is the large size of application test programs, which verification systems are unable to cope with due to simulation performance limitations.

The contribution of the research to address this software to hardware verification gap was to devise a new application based verification methodology that can be applied during early stages of design behavioural simulations. The verification methodology is named software application level verification methodology (SALVEM). The methodology exploits common and essential design functions used by application code of the SoC for hardware design verifications. SALVEM employs numerous software test programs that contain various SoC application usage characteristics, in order to test associated design functions covered by real-life SoC use-cases. Verification from a software application perspective is applied much earlier in the design cycle before application based design bugs are allowed to propagate too far down the design flow.

The key concept of SALVEM is to extract from SoC applications use-cases the test building blocks *snippets* that are used to automatically create systematic sets of tests. In the test creation procedure, different sequences of snippets are selected for the production of test programs. The library of snippets building blocks facilitates many unique compositions of test programs, providing greater test coverage of the SoC and more thorough verifications from an application perspective is realised.

With the snippets concept, SALVEM can apply software application based test programs at the behavioural register transfer level (RTL) simulation phase. Using snippets, the sizes of our application test programs can be managed so that RTL simulations are possible, unlike overly large size applications that require hardware emulation or expensive test equipment to accelerate test executions.

Most significantly for verification project teams, is the capability of SALVEM tests to perform efficient application testing at the pre-silicon stage but with smaller test sizes.

SALVEM was formalised so it may be applied for design verifications of SoCs in general. The snippets concept facilitates re-use across different SoCs and design projects, saving verification efforts. As an example of SALVEM reusability contributions, we adapted and reapplied direct memory access (DMA) device snippets originally developed for the Nios SoC on the Tsinghua digital signal processing SoC. Furthermore, the other innovation and benefit of the SALVEM and snippets approach is that the application tests creation technique can be applied to other levels of test generation and verifications. For example, the concept of snippets can be applied to test pattern generation for post-silicon testing, as long as equivalent snippet building blocks are identified to perform SALVEM-like test creations.

Another contribution of the methodology research is the formulation of automation mechanisms for the snippets based test generation process. Constrained-biased randomisation for software test program generations in SALVEM was investigated and developed. The useability and benefits of our random test generator were demonstrated by its capability to test the Nios SoC with greater coverage than typical *ready-made* application software programs. More importantly, we were able to uncover design bugs in the Nios SoC with SALVEM tests; despite the SoC being release for general use in the engineering community. Besides demonstrating the effectiveness of our methodology, SALVEM's ability to uncover design errors contributes to the greater effectiveness of industrial verification procedures and enhances the quality of the SoC designs.

8.2 Test generation research

The test generation research is partitioned into three main areas: (i) genetic evolution test generation, (ii) test generation parameter selections with Markov chains, and (iii) multi-objective test generation.

8.2.1 Genetic evolutionary test generation

The creation of effective test programs for design verification is a difficult process, even for experienced verification engineers. Automating the test generation is also essential, but this adds further complexity to the test creation flow. In Section 8.1, we alluded to automated test creations via randomisation. Despite the ability to generate many tests automatically, the random approach can be

wasteful of verification resources. Many random tests are needed in the hope that large numbers of tests will cover sufficient portions of the required test space; but even worse, the same functions could be repeatedly exercised whilst other design behaviours remain unverified. In our view, besides automation, algorithmic based test generation was the solution. The test generation research investigated and devised genetic evolutionary algorithmic test creation strategies, which also facilitated coverage driven verifications.

The genetic evolutionary test generation is posed as an optimisation process whose objective is to maximise coverage, subject to constraints on test size, simulation resources, and verification time. The strength of this test creation scheme lies in our encoding of GEA elements and processes to the test generation domain, in order to conduct genetic evolutionary cycles. Employing snippets as test building blocks in the same way biological genes make up chromosomal individuals, and with the verification test suite acting as the population of individuals, SALVEM test programs are evolved like real-life organisms in an evolutionary manner. The contribution of the devised test creation technique is that the test programs generated every cycle are continually enhanced to gain coverage in an algorithmic manner; the same way that stronger biological individuals are acquired every evolutionary cycle. The advantages of GEA are directly realised for application based test creations to acquire higher test coverage, thus bringing about more effective design verifications.

We named our test generator tool SALVEM genetic evolutionary test generator (SAGETEG). SAGETEG facilitates a coverage driven verification process for SALVEM, which is another key contribution of the test generation research. Under GEA test generation, coverage information is continually fed back to explore the SoC test space in a directed and strategic manner. The genetic evolution technique only retains tests with the best coverage fitness, and applies genetic evolutionary variation on this retained test set to create new tests. By reusing higher coverage yielding tests, this approach is guided by coverage information to continue verifying desired SoC test functional areas. The GEA coverage influenced test suites enhances design verification by ensuring the important and required application based test space is covered.

Despite the single objective nature of SAGETEG, a beneficial side-effect is the minimal test sizes that eventuate from higher coverage yielding tests. Our tests are initially empty or contain only few snippets. The tests are progressively cultivated over many evolutions to hold only the required snippets that contribute to the coverage they attain. Thus, the number of snippets and test sizes is not large, contributing to the wider verification effectiveness and efficient use of test resources.

Genetic evolutionary self-adaptation and variation characteristics were also studied and reported, providing valuable insight and enhancement recommendations for the genetic evolutionary test generation tool. The outcome of this particular research study was to propose and implement a revised GEA variation self-adapting technique, which attains superior GEA test creation performance. The GEA snippet variation dependency analysis and our self-adaptation technique also have wider applicability, not just for test generations and snippet characteristics. If similar genome characteristics and equivalent GEA variation is observed in other GEA strategies for solving other problem areas, our revised self-adaptation procedure would be able to provide similar advantages toward those application domains and facilitate more effective GEA flows.

Another important contribution of this test generation research is that the GEA technique to evolve coverage enhancing test suites is re-usable in other test generation domains, in addition to software application test creations. For example, the GEA test generation strategy can be applied for post-silicon test pattern generations or assembler instruction test creations for microprocessor verifications.

8.2.2 Test generation parameter selections with Markov chains

Test generators are predominately controlled by input parameters that influence the test creation process. These parameters are powerful mechanisms that manipulate the types of tests generated and affect the level of verification effectiveness. Despite this, selection of values for parameters can be ad-hoc, either chosen without proper consideration or employing preliminary test generation and calibration runs to identify appropriate parameter selections before actual SoC verifications. Regardless of their effectiveness, conducting such calibrations for the test generator is inefficient, taking up valuable verification time and resources. Our research into this problem led to the proposal and development of an analytical solution based on Markov chains modelling.

Our genetic evolutionary test generation method (SAGETEG) was complimented by a Markov based analytical technique for examining the test generation procedure. Besides revealing important behavioural characteristics of the genetic evolution process, the contribution of this Markov modelling and analysis technique is to facilitate selection of best suitable test generation input parameters that enhances the test generator performance and quality of tests created.

In our method, Markov chains are employed to model sub components and processes of the GEA test generation process. Based on the derivation of these Markov models and their subsequent analysis, information about the GEA test generation can be extracted without the need to perform any

verification runs. Armed with this information, appropriate values are then chosen for test generation parameters to enhance the likelihood of desired characteristics according to outcomes revealed by Markov chain analysis.

From a verification perspective, the contribution of this approach saves valuable verification time and resources directly. The need for preliminary test runs to calibrate parameters is eliminated. With our technique, the selection of parameters are supported analytically from thorough examination of the GEA test creation process; it leads to more accurate and effective use of test generation parameters during verifications. Using our analytical test selections, actual SoC verifications using GEA test creations can also be applied much earlier, with only some fine-tunning of the generator's parameters needed. This allows for immediate test generations of the SoC, and facilitates more efficient verification from the beginning.

Our method also overcomes prior limitations which employ one Markov chain to model the entire GEA process – a strategy that is impractical for GEA processes such as test generation because the resultant Markov chain would be oversized, complex, and intractable to analyse. Instead of a single Markov model, the key to our analytical parameter selection method is to employ a divide-n-conquer strategy for modelling GEA test generation using multiple Markov chains. Each Markov chain can be utilised to select values for certain parameters, or to build up other Markov chains that model more complicated GEA sub processes and select other subsequent parameters.

Given the wide applicability of GEA for diverse problem areas, the difficulties with using Markov chains to characterise any general GEA processes is an inherent problem for Markov modelling and analysis. Using divide-n-conquering with multiple Markov models, and encoding GEA sub-components and test parameters into the Markov model state and transitions, the major contribution of this research is to enable the Markov modelling and analysis of greater range of GEA applications. Our technique is repeatable and reusable for analysing other different GEA sub components and processes of various application domains which also employ GEA, not just GEA test generations. Parameters for other GEA processes can be analysed and selected using the same strategy we have developed.

The Markov chain GEA parameter analysis was applied for GEA test generations for the Nios SoC. Four example Markov chains and analysis of their GEA sub-flows were demonstrated, along with parameter selections. The feasibility of our Markov methods was established, supporting the significance of our work toward design verification research – that is, to facilitate Markov modelling and analysis of the overall test generation process, in which parameter selection was shown to be one possible application of the modelling.

8.2.3 Multi-objective test generation

In many test generation schemes, a single goal drives the test creation process. However, a verification process imposes many requirements and must satisfy various goals and constraints. The test generation process should be driven by multiple objectives to provide more effective testing, satisfying greater scope of verification requirements at once. To address this need, we conducted research into multi-objective optimising strategies for test generation.

We extended our genetic evolutionary test generation process to operate with multiple objectives. Instead of individual test objectives, we incorporate into the verification process, maximisation of multiple coverage metrics concurrently and other objectives such as minimising test sizes; given that limitations on available memory to hold tests are common, and test execution times can be restricted for simulations. By incorporating other objectives into the verification process, the immediate contribution of our work is to provide more efficient testing requiring less test generations and simulations, along with a more thoroughly verified design governed by a range of test criteria.

We design multi-objective genetic evolutionary methods by combining the concepts of aggregation and Pareto optimality for creating and ranking tests. To optimise and assess effectiveness of multiple objectives simultaneously, we also devised unique multi-phase round-robin GEA test selection and termination methods to aid cultivation of evolutionary multi-objective tests. By injecting multi-objective mechanisms into the evolution cycle, we exploited the available parallelism by amalgamating multiple individually-objectified GEA processes into a single flow. SALVEM test generation can then be directed by more than one coverage measure and other test objectives concurrently and efficiently. The significance of our multi-objective GEA is to bring to fruition at the pre-silicon simulation stage, software application testing which is driven by multiple verification goals.

Our goal in multi-objective GEA test generation was to create tests that provide maximal code coverage whilst using minimal number of tests with low test sizes. Such conflicting objectives are a challenge common in multi-objective optimisation, not just test generation. The key to tackling this difficulty is our multi-objective GEA's ability to acquire a set of trade-off solutions which are optimised to give best overall results for all objectives, in spite of conflicts. We sub-divided the conflicting objectives into smaller subset of verification goals, and various phases of the GEA process was performed by partitioning their operations for each subset objective separately and then combining them later. This breaks down the conflicting multi-objective procedural workload into manageable and efficient tasks. Subsequently, the direct benefits of the multi-objective GEA method are to gain higher coverage using much lower test sizes.

Another contribution of the multi-objective GEA is that other objectives or additional goals can be easily incorporated to cater for other verification challenges. We designed our multi-objective GEA method to be versatile, to be able to integrate other types of verifications or more general objectives if desired. This enhances the test generation method further and expands its applicability to other verification strategies. For example, the multi-objective GEA can be used for hardware emulation testing, whereby the size of tests applied becomes an important consideration. For such test platforms, the memory available to hold tests is limited. Our multi-objective test generation provides a set of tests minimised by size that is able to run on the system, and still provide the best coverage despite size restrictions. Such scenarios demonstrate the contribution and general applicability of the multi-objective GEA test generation; for verification and testing of SoCs across multiple design levels from pre-silicon simulation to post-silicon hardware testing. At each level of verification, different test requirements are needed, which can simply be managed by customising other objectives into our technique.

In this thesis, our research and experiments have proven the multi-objective GEA test generations for four objectives, with three sets of conflicting objectives and objective subsets. For greater number of objectives, the method is scalable to take on additional goals concurrently, which is a significant advantage. The only proviso is that greater computation resources could be required to take on additional GEA test selection and Pareto frontal processing.

Besides the ability to take on different or additional objectives, the other significant contribution of the multi-objective GEA is its reusability. The multi-objective GEA strategy to handle conflicting objectives has broader applicability, not only within a verification context. In other domains where conflicting objectives may hinder the problem solving capability of other methods, the formalised and methodical objectives management method can be adapted for use; e.g., to assess and effectively trade-off the performance results of objectives before they are deployed into other problem solving methods.

8.3 Coverage research

Conventional code coverage methods do not provide desired information regarding the functional design behaviours verified. For software application verification, a functional coverage measuring method is needed. The goal is to quantify the types of application functionalities tested according to the application test methodology. Traditionally, functional coverage methods suffer from state space explosion phenomenon and measuring inefficiency. Such state expansion problems also occur in

formal verification. Therefore, our research examined how abstraction and graph based trajectory checking from symbolic trajectory evaluation (STE) formal methods can be applied to tackle the state expansion problem in functional coverage measuring.

The outcome of the coverage research is the proposal and development of a functional coverage method that can evaluate the SoC design functions exercised. Our coverage solution quantifies the measured coverage data into a metric for estimating the comprehensiveness of SALVEM testing from an SoC applications test point of view. In our technique, a functional coverage model is designed specifying the coverage goals and SoC design variables to measure. These design variables are termed *attributes* and are chosen specifically to be monitored because they indicate how the design was exercised. Our coverage method identifies and quantifies the design functions verified by SALVEM in terms of various combinations of SoC attribute values measured. We denote the coverage method as *attribute combinations* coverage.

The value of the attribute combinations coverage method is to report useful functional information about what SoC use-cases and associated design features were tested, and what test scenarios were not exercised during verification. Our coverage method estimates the test comprehensiveness and effectiveness of SALVEM verifications from a functional design perspective.

To contain the state explosion problem in functional coverage, the novel contribution of the attribute combinations coverage stems from the adaptation of partially ordered symbolic domains to abstract design elements and to constraint the size of the coverage model for coverage measuring. STE trajectory checking techniques is tailored specifically to perform realisable and efficient measurements. The amount of attributes to measure is reduced by abstracting large sets of attribute values that share common characteristics, and representing them as symbolic domains. We specify coverage goals and conduct coverage measurements using a reduced set of interesting and important attributes and values only. This ensures only the required application functionalities are captured and explicitly measured to guarantee they are verified. Subsequently, the resultant functional coverage method is able to handle SoC design sizes and measures SoC functionalities invoked from software application test programs.

Another key benefit in our use of symbolic domains for coverage measuring is that once domains are devised for a design, they have high likelihood for reuse. Once suitable domains are identified, they can be applied for other SoCs. Using the same domains, our coverage method puts in place the infrastructure to facilitate coverage measuring in an efficient flow, it is able report valuable functional test information for different designs and verification flows.

The other significant contribution of the coverage method is our *windowing* mechanism. The concept of windowing is to manage the amount of coverage data being stored and processed during coverage measuring. The coverage measuring can be tuned to operate more effectively by adjusting window sizes. This provides savings in the computation requirements needed to measure coverage of different SoCs, and contributes to the effectiveness of the coverage method.

8.4 Recommendations for future research investigations

8.4.1 Verification methodology research

In this thesis, the SALVEM approach was developed and applied to design verifications of SoCs at the pre-silicon simulation level. We believe further progress in overall semiconductor test research is possible if SALVEM is extended and applied for other phases and levels of verifications. For example, testing from a SALVEM applications perspective could be deployed for gate-level test pattern vector testing at later stages of the verification cycle; or even post-silicon testing such as circuit board bring-up tests when the SoC is integrated with other components. The test input stimulus applied can be influenced from an SoC usage perspective by simply extending SALVEM to convert its applications based tests into appropriate format for use at other levels of verifications.

If applying SALVEM for lower level testing, traditional methods used at those levels of test could also be beneficial to SALVEM itself. For instance, hardware acceleration strategies or equipment are often employed for post-silicon testing. If available, such methods can be beneficial to SALVEM by speeding up various SALVEM tasks, such as GEA or multi-objectives optimisation test generation processes. This would immediately overcome any possible computing resource restrictions in any of our methods. As an example, greater number of tests and test populations can be created and evolved faster to produce more diverse and effective test programs. In the author's view, further research is needed to uncover other complementary areas in which SALVEM could be applied to enhance hardware verifications, and also provide corresponding opportunities for improving SALVEM as a methodology further.

Extending research for SALVEM as a methodology need not be restricted to hardware verification. Given the software application nature of SALVEM, and similarities between the hardware design and software engineering fields, investigations to extend SALVEM for software testing itself is another avenue of further research. The same strategies to identify and analyse how software programs, tools or libraries are used can be conducted. Then, reusable test building blocks can be extracted to form

tests, employing similar GEA or multi-objective test creation techniques described in this thesis. The methodology research could also lead to other forms of verification, such as network design validation. For instance, like SALVEM, applying real-life orientated test stimulus to check protocols and reliability of the network design; based on modular stimulus building blocks of actual network data units to form stream of network traffic for testing.

Another good area for investigation is to revisit the representation, extraction, and usage of our snippets test program building blocks. Rather than high level software based functions acting as snippets, snippets could be generalised further into larger test building blocks consisting of multiple application based functions, or even a mixture of high level based software code or lower level assembler instructions – in order to speed up snippets execution and allow finer grain internal snippet manipulation. Given snippets are the most crucial ingredient for testing SoC functionalities, they deserve further research.

8.4.2 Test generation research

For both single objective (SAGETEG) and multi-objective GEA test generations, further progress is possible in the test creation optimising procedure if our attribute combinatorial coverage method is employed as the fitness evaluator to drive the test generation process. Despite acquiring favourable results with code coverage metrics, when employing our coverage method, which is designed specifically for the software applications verifications, we expect the SALVEM test generations to be much more effective. The test creation procedure can be guided more precisely and directly from measurements of the SoC application functions exercised. Additional research should lead to explicit applications driven tests and more effective verifications, by extending both the SALVEM test generations and attributes based coverage. This avenue of research was not pursued due to time limitations of the research candidature.

Given the possibilities for duplicate or similar test programs arising from many tests generated by our optimisation procedures, it would be beneficial to investigate the potential in employing a caching technique to reduce too many repetitive test stimulus. Furthermore, a caching method could also prove useful to capture highly effective types of tests so they can be reused more regularly.

Another interesting area of study is to examine other different GEA representations of test generation elements in the genetic evolutionary domain. GEA test generation representation can be enhanced by mapping other specific components of SALVEM test generations into the GEA test creation process.

The goal of such mappings is to provide greater control over the types and range of tests and associated SoC functions that can be crafted via GEA variation. The different representations could also involve examination of other variants of GEA, such as genetic programming.

By re-encoding the GEA test generation representations, other algorithmic, classification, and learning based schemes such as neural networks, Bayesian networks, or support vector machine could also be injected into the test creation procedures. The study will exploit the many GEA evolutions of test generations and simulations to train and continually enhance the test generations.

8.4.3 Coverage research

We summarise one important research possibility that can be extended from our coverage work. Appendix H.18 describes in greater detail the other possible avenues of coverage research. In our coverage method, the coverage window concept plays an important role because the set of coverage attributes information captured by a window determines how accurate and effective the coverage measurements are. Additional research is needed to investigate how to better utilise windowing. Other windowing concepts include using a variable size window to capture different amounts of coverage attributes data from each test generation and verification cycle, or even using a sliding window strategy whereby portions of old attributes data are progressively removed as new coverage data are captured from newer verification runs.

8.5 Concluding remarks

Besides the software tools developed to implement the verification techniques devised from our research, along with this thesis, the research contributions described in this chapter culminated in 11 peer-reviewed research publications. The research conducted also exposed valuable insight into the design verification problem.

Perhaps the most significant lesson imparted upon the author during the course of pursuing this thesis is that no single solution exists that can tackle the design verification problem reliably and comprehensively for current SoC designs. Instead, complementary verification solutions and test methods that can work well together to form effective holistic strategies are required.

Our research work and contributions in the design verification field complements existing verification techniques, rather than replace them. Many verification methods address different requirements during the extensive process of certifying design correctness. This research thesis presents a strategy that brings together a range of methods from different technical domains. The outcomes of our research toward the design verification problem are to verify correct operation of an SoC design from an application functions perspective, and maximise the likelihood of detecting system design errors before chip fabrication.

The achievements of this research are to devise and develop new verification methodology and associated test generation and coverage techniques for effective verifications of SoCs at the pre-silicon level and from a software applications perspective. The research work described in this thesis and the research publications contribute directly to the advancement of knowledge in SoC design verifications.

BIBLIOGRAPHY

[AAF+03] M. Aharoni, S. Asaf, L. Fournier, A. Koifman, and R. Nagel, "FPgen - A Test Generation Framework for Datapath Floating-Point Verification," in *Proceedings of IEEE International High Level Design Validation and Test Workshop (HLDVT'03)*. San Francisco, California, USA, 2003, pp. 17-22.

[AAF+04] A. Adir, E. Almog, L. Fournier, E. Marcus, M. Rimon, M. Vinov, and A. Ziv, "Genesys-Pro: Innovations in Test Program Generation for Functional Processor Verification," *IEEE Design and Test of Computers*, vol. 21, pp. 84-93, 2004.

[ABD+91] A. Aharon, A. Bar-David, B. Dorfman, G. Gofman, M. Leibowitz, and V. Schwartzburd, "Verification of the IBM RISC System/6000 by a Dynamic Biased Pseudo-Random Test Program Generator," *IBM System Journal*, vol. 30, pp. 527-538, 1991.

[ABPZ03] A. Adir, E. Bin, O. Peled, and A. Ziv, "Piparazzi: A Test Program Generator for Micro-Architecture Flow Verification," in *Proceedings of IEEE International High Level Design Validation and Test Workshop (HLDVT'03)*. San Francisco, California, USA, 2003, pp. 23-28.

[Abr90] M. Abramovici, *Digital Systems Testing and Testable Design*. New York: Computer Science Press, New York, 1990.

[AEM02] A. Adir, R. Emek, and E. Marcus, "Adaptive Test Program Generation: Planning for the Unplanned," in *Proceedings of IEEE International High Level Design Validation and Test Workshop (HLDVT'02)*. Cannes, France, 2002, pp. 83-88.

[AGL+95] A. Aharon, D. Goodman, M. Levinger, Yossi Lichtenstein, Y. Malka, C. Metzger, M. Molcho, and G. Shurek, "Test Program Generation for Functional Verification of PowerPC Processors in IBM," in *Proceedings of the 32nd Design Automation Conference (DAC'95)*. San Francisco, California, USA, 1995, pp. 279-285.

[AIS+77] C. Alexander, S. Ishikawa, M. Silverstein, M. Jacobson, I. Fiksdahl-King, and S. Angel, *A Pattern language*. New York: Oxford University Press, 1977.

[Alt03] Altera Inc, "Nios II Hardware Development Tutorial." http://www.altera.com/literature/tt/tt_nios2_hardware_tutorial.pdf, ver 1, 2003.

[AMF+05] H. Azatchi, E. Marcus, L. Fournier, S. Ur, A. Ziv, and K. Zohar, "Advanced Analysis Techniques for Cross-Product Coverage," *IEEE Transactions on Computers, Special Issue on Simulation-Based Validation*, vol. 55, pp. 1367-1379, 2005.

[AMZ04] S. Asaf, E. Marcus, and A. Ziv, "Defining Coverage Views to Improve Functional Coverage Analysis," in *Proceedings of the 41st Design Automation Conference (DAC'04)*. San Diego, California, USA, 2004, pp. 41-44.

[ASR+04] L. Anghel, E. Sanchez, M. S. Reorda, G. Squillero, and R. Velazco, "Coupling Different Methodologies to Validate Obsolete Microprocessors," in *Proceedings of the 19th IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems*. Cannes, France, 2004, pp. 250-255.

[BBS91] R. E. Bryant, D. L. Beatty, and C.-J. H. Seger, "Formal Hardware Verification by Symbolic Ternary Trajectory Evaluation," in *Proceedings of the 28th Conference on ACM/IEEE Design Automation*. San Francisco, California, USA, 1991, pp. 397-402.

[BCG+00] D. S. Brahme, S. Cox, J. Gallo, M. Glasser, W. Grundmann, C. N. Ip, W. Paulsen, J. L. Pierce, J. Rose, D. Shea, and K. Whiting, "The Transaction-Based Verification Methodology," in *Cadence Berkeley Labs, Technical Report # CDNL-TR-2000-0825*, 2000.

[BCRZ99] A. Biere, E. Clarke, R. Raimi, and Y. Zhu, "Verifying Safety Properties of a PowerPC Microprocessor Using Symbolic Model Checking without BDDs," in *Proceedings of Computer Aided Verification (CAV'99), Lecture Notes in Computer Science 1633*: Springer-Verlag Berlin Heidelberg, 1999, pp. 60–71.

[Bei90] B. Beizer, *Software testing techniques*, 2nd ed. New York: Van Nostrand Reinhold, 1990.

[Ber03] J. Bergeron, *Writing Testbenches: Functional Verification of HDL Models*, 2nd ed. London: Kluwer Academic, 2003.

[BESZ02] E. Bin, R. Emek, G. Shurek, and A. Ziv, "Using a constraint satisfaction formulation and solution techniques for random test program generation," *IBM System Journal, Special Issue on AI*, vol. 41, pp. 386-402, 2002.

[BGH+99] M. Benjamin, D. Geist, A. Hartman, Y. Wolfsthal, G. Mas, and R. Smeets, "A Study in Coverage-Driven Test Generation," in *Proceedings of the 36th Design Automation Conference (DAC'99)*. New Orleans, Louisiana, USA, 1999, pp. 970-975.

[BH99] J. Bergmann and M. Horowitz, "Improving Coverage Analysis and Test Generation for Large Designs," in *1999 International Conference on Computer-Aided Design (ICCAD'99)*. San Jose, California, 1999, pp. 580-583.

[BHS97] T. Back, U. Hammel, and H.-P. Schwefel, "Evolutionary Computation: Comments on the History and Current State," *IEEE Transactions on Evolutionary Computation*, vol. 1, pp. 3-17, 1997.

[BLL+04] M. Behm, J. Ludden, Y. Lichtenstein, M. Rimon, and M. Vinov, "Industrial Experience with Test Generation Languages for Processor Verification," in *Proceedings of the 41st Design Automation Conference (DAC'04)*. San Diego, California, USA, 2004, pp. 36-40.

[BPD+06] A. Banerjee, B. Pal, S. Das, A. Kumar, and P. Dasgupta, "Test Generation Games from Formal Specifications," in *Proceedings of the 43rd Design Automation Conference (DAC'06)*. San Francisco, California, USA, 2006, pp. 827-832.

[Bry86] R. E. Bryant, "Graph-Based Algorithms for Boolean Function Manipulation," *IEEE Transactions on Computers*, vol. 35, pp. 677-691, 1986.

[Bry91] R. E. Bryant, "Symbolic Simulation - Techniques and Applications," in *Proceedings of the 27th ACM/IEEE conference on Design Automation*. Orlando, Florida, United States, 1991, pp. 517-521.

[BS90] R. E. Bryant and C.-J. Seger, "Formal Verification of Digital Circuits Using Symbolic Ternary System Models," *Computer-Aided Verification '90, DIMACS Series in Discrete Mathematics and Theoretical Computer Science, ACM*, pp. 121-146, 1990.

[BSS+06] P. Bernardi, E. Sanchez, M. Schillaci, G. Squillero, and M. S. Reorda, "An Effective Technique for Minimizing the Cost of Processor Software-Based Diagnosis in SoCs," in *Design, Automation and Test in Europe Conference (DATE2006)*. Paris, France, 2006, pp. 412-417.

[BT80] A. Ben-Tal, "Characterization of Pareto and Lexicographic Optimal Solutions," *Lecture Notes in Economics and Mathematical Systems, Springer Verlag*, vol. 177, pp. 1-11, 1980.

[Cad] Cadence Design Systems Inc, "NC-Verilog." www.cadence.com/products/functional_ver/nc-verilog/.

[Cad04] Cadence Design Systems Inc, "Accelerated Hardware/Software Co-Verification Speeds "First Silicon and First Software," in *Cadence White Paper*, 2004.

[Cam99] D. V. Campenhout, "Functional Design Verification for Microprocessors by Error Modeling, PhD Thesis," in *Department of Electrical Engineering*: The University of Michigan, 1999.

[CCRS01a] F. Corno, G. Cumani, M. S. Reorda, and G. Squillero, "Devising an RT-Level ATPG for uProcessor Cores," in *2nd Workshop on RTL, ATPG & DFT (WRTLT2001)*. Nara, Japan, 2001, pp. 20-24.

[CCRS01b] F. Corno, G. Cumani, M. S. Reorda, and G. Squillero, "ARPIA: a High-Level Evolutionary Test Signal Generator," in *3rd European Workshop on Evolutionary Computation Applications to Image Analysis and Signal Processing (EvoIASP2001)*. Como, Italy, 2001, pp. 298-306.

[CCRS02a] F. Corno, G. Cumani, M. S. Reorda, and G. Squillero, "Efficient Machine-Code Test-Program Induction," in *Proceedings of the 2002 Congress on Evolutionary Computation (CEC2002)*. Honolulu, USA, 2002, pp. 1486-1491.

[CCRS02b] F. Corno, G. Cumani, M. S. Reorda, and G. Squillero, "Automatic Test Program Generation from RT-level Microprocessor Descriptions," in *3rd International Symposium on Quality Electronic Design*. San Jose, California, USA, 2002, pp. 120-125.

[CCRS02c] F. Corno, G. Cumani, M. S. Reorda, and G. Squillero, "Evolutionary Test Program Induction for Microprocessor Design Verification," in *11th Asian Test Symposium*, 2002, pp. 368-373.

[CCRS03a] F. Corno, G. Cumani, M. S. Reorda, and G. Squillero, "Fully Automatic Test Program Generation for Microprocessor Cores," in *Design, Automation and Test in Europe Conference (DATE2003)*. Munich, Germany, 2003, pp. 1006-1011.

[CCRS03b] F. Corno, G. Cumani, M. S. Reorda, and G. Squillero, "Automatic Test Program Generation for Pipelined Processors," in *The Eighteenth Annual ACM Symposium on Applied Computing (SAC2003)*. Melbourne, Florida, USA, 2003, pp. 736-740.

[CCS03] F. Corno, F. Cumani, and G. Squillero, "Exploiting Auto-Adaptive μGP for Highly Effective Test Programs Generation," in *ICES2003: The 5th International Conference on Evolvable Systems: From Biology to Hardware*. Trondheim, Norway, 2003, pp. 262-273.

[CGW+95] A. Chandra, D. Geist, Y. Wolfsthal, V. Iyengar, D. Jameson, R. Jawalekar, I. Nair, B. Rosen, M. Mullen, J. Yoon, and R. Armoni, "AVPGEN - A Test Generator for Architecture Verification," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 3, pp. 188-200, 1995.

[Che03] A. Cheng, *Coverage Driven Verification using Semi-Formal Symbolic Techniques, PhD Proposal - The University of Adelaide*, *Australia* 2003.

[CIJ+94] A. K. Chandra, V. S. Iyengar, R. V. Jawalekar, M. P. Mullen, I. Nair, and B. K. Rosen, "Architectural Verification of Processors Using Symbolic Instruction Graphs," in *Proceedings of the1994 IEEE International Conference on Computer Design: VLSI in Computer & Processors*. Cambridge, MA, USA, 1994, pp. 454-459.

[CL07] A. Cheng and C. C. Lim, "System Test Generation for System-on-Chips using Genetic Evolutionary Methods," in *7th International Conference on Optimization: Techniques and Applications (ICOTA7)*. Kobe, Japan: Universal Academic Press Inc., 2007, pp. 67-76.

[Cla91] D. W. Clark, "Large-Scale Hardware Simulation: Modeling and Verification Strategies," in *Chapter 9 of the Proceedings of the 25th CMU Computer Science Anniversary Symposium, Computer Science Department, Carnegie-Mellon University*: ACM Press/Addison-Wesley, 1991, pp. 219-234.

[CLS+08] A. Cheng, C. C. Lim, Y. Sun, H. He, Z. Zhou, and T. Lei, "Using Genetic Evolutionary Software Application Testing to Verify a DSP SoC," in *4th IEEE International Worksop on Electronic Design, Test & Applications (DELTA2008)*. Hong Kong: IEEE Computer Society, 2008, pp. 20-25.

[CMH98] D. V. Campenhout, T. Mudge, and J. P. Hayes, "High-Level Test Generation for Design Verification of Pipelined Microprocessors," in *Proceedings of IEEE International High Level Design Validation and Test Workshop (HLDVT'98)*. La Jolla, California, USA, 1998, pp. 1-8.

[Coe99] C. A. C. Coello, "A Comprehensive Survey of Evolutionary-Based Multiobjective Optimization Techniques," *Journal of Knowledge and Information Systems*, vol. 1, pp. 269-308, 1999.

[Col00] Collett International Research, "2000 IC/ASIC Functional Verification Study," 2000.

[Col02] Collett International Research, "2002 IC/ASIC Functional Verification Study," 2002.

[Col04] Collett International Research, "2004 IC/ASIC Functional Verification Study," 2004.

[CoW] CoWare, "CoWare SPW DSP Workbench and AccelChip DSP Synthesis." http://www.coware.com/news/press263.htm.

[CPL05a] A. Cheng, A. Parashkevov, and C. C. Lim, "Verifying System-on-Chips at the Software Application Level," in *IFIP-WG 10.5 Very Large Scale Integration System-on-Chip (VLSI-SoC'05)*. Perth, Australia, 2005, pp. 586-591.

[CPL05b] A. Cheng, A. Parashkevov, and C. C. Lim, "A Software Test Program Generator for Verifying System-on-Chips," in *10th IEEE International High Level Design Validation and Test Workshop (HLDVT'05)*. Napa Valley, California, USA, 2005, pp. 79-86.

[CPRR96a] F. Corno, P. Prinetto, M. Rebaudengo, and M. S. Reorda, "Advanced Techniques for GA-Based Sequential ATGP," in *IEEE European Design & Test Conference*. Paris, France, 1996, pp. 375-379.

[CPRR96b] F. Corno, P. Prinetto, M. Rebaudengo, and M. S. Reorda, "GATTO: A Genetic Algorithm for Automatic Test Pattern Generation for Large Synchronous Sequential Circuits," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 15, pp. 943-951, 1996.

[CRR98] F. Corno, M. Rebaudengo, and M. S. Reorda, "Experiences in the use of Evolutionary Techniques for Testing Digital Circuits," in *Applications and Science of Neural Networks, Fuzzy Systems, and Evolutionary Computation*. San Diego, California, USA, 1998, pp. 128-139.

[Cro99] A. L. Crouch, *Design for Test - for Digital ICs and Embedded Core Systems*. Upper Saddle River, New Jersey, USA: Prentice Hall PTR, 1999.

[CRS01] F. Corno, M. S. Reorda, G. Squillero, and M. Violante, "On the Test of Microprocessor IP Cores," in *IEEE Design, Automation & Test in Europe Conference (DATE2001)*. Munich, Germany, 2001, pp. 209-213.

[CS03] F. Corno and G. Squillero, "An Enhanced Framework for Microprocessor Test-Program Generation," in *EUROGP2003: 6th European Conference on Genetic Programming*. Essex, UK, 2003, pp. 307-315.

[CSRS04a] F. Corno, E. Sánchez, M. S. Reorda, and G. Squillero, "Automatic Test Program Generation: A Case Study," in *IEEE Design & Test, Special issue on Functional Verification and Testbench Generation*, vol. 21, 2004, pp. 102-109.

[CSRS04b] F. Corno, E. Sanchez, M. S. Reorda, and G. Squillero, "Code Generation for Functional Validation of Pipelined Microprocessors," *Journal of Electronic Testing: Theory and Applications*, vol. 20, pp. 269-278, 2004.

[CW96] E. M. Clarke and J. M. Wing, "Formal Methods: State of the Art and Future Directions," *ACM Computing Surveys*, vol. 28, pp. 626-643, 1996.

[Dau05] A. Dauman, "Improved Design Methodology for FPGA-based DSPs," in *Embedded Control Europe*, 2005.

[DBG01] J. Dushina, M. Benjamin, and D. Geist, "Semi-Formal Test Generation with Genevieve," in *Proceedings of 38th Design Automation Conference (DAC'01)*. Las Vegas, Nevada, USA, 2001, pp. 617-622.

[DGK96] S. Devadas, A. Ghosh, and K. Keutzer, "An Observability-Based Code Coverage Metric for Functional Simulation," in *Proceedings of International Conference on Computer-Aided Design*. San Jose, California, USA, 1996, pp. 418-425.

[DHZ+05] X. Ding, H. He, Y. Zhang, Y. Sun, and Y. Yu, "The Implementation Method about Verifying to VLIW DSP," in *Proceedings of the 6th IEEE International Conference on ASIC (ASICCON05)*. Shanghai, China, 2005, pp. 704-708.

[EJN+02] R. Emek, I. Jaeger, Y. Naveh, G. Bergman, Guy Aloni, Y. Katz, M. Farkash, I. Dozoretz, and A. Goldin, "X-GEN: A Random Test-Case Generator for Systems and SoCs," in *IEEE International High Level Design Validation and Test Workshop (HLDVT'02)*. Cannes, France, 2002, pp. 145-150.

[FAD02] F. Fallah, P. Ashar, and S. Devadas, "Functional Vector Generation for Sequential HDL Models Under an Observability-Based Code Coverage Metric," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 10, pp. 919-923, 2002.

[FAL99] L. Fournier, Y. Arbetman, and M. Levinger, "Functional Verification Methodology for Microprocessors Using the Genesys Test-Program Generator Application to the x86 Microprocessors Family," in *Design, Automation and Test in Europe Conference (DATE1999)*. Munich, Germany, 1999, pp. 434-441.

[FBYR01] F. Fummi, M. Boschini, X. Yu, and E. M. Rudnick, "Sequential Circuit Test Generation using a Symbolic/Genetic Hybrid Approach," *Journal of Electronic Testing: Theory and Applications*, vol. 17, pp. 321-330, 2001.

[FDK98] F. Fallah, S. Devadas, and K. Keutzer, "OCCOM: Efficient Computation of Observability-Based Code Coverage Metrics for Functional Verification," in *Proceedings of the 35th Design Automation Conference (DAC'98)*. San Francisco, California, USA, 1998, pp. 152-154.

[FDK01] F. Fallah, S. Devadas, and K. Keutzer, "OCCOM—Efficient Computation of Observability-Based Code Coverage Metrics for Functional Verification," *IEEE Transaction on Computer-Aided Design of Integrated Circuits and Systems*, vol. 20, pp. 1003-1015, 2001.

[FF93] C. M. Fonseca and P. J. Flemming, "Genetic Algorithms for Multi-Objective Optimization: Formulation, Discussion, and Generalization," in *Proceedings of the Fifth International Conference on Genetic Algorithms*: Morgan Kaufmann, 1993, pp. 416-423.

[FFS+01] F. Ferrandi, G. Ferrara, D. Sciuto, A. Fin, and F. Fummi, "Functional Test Generation for Behaviorally Sequential Models," in *Design, Automation and Test in Europe Conference (DATE2001)*. Munich, Germany, 2001, pp. 403-410.

[FKL99] L. Fournier, A. Koyfman, and M. Levinger, "Developing an Architecture Validation Suite Application to the PowerPC Architecture," in *Proceedings of the 36th Design Automation Conference (DAC'99)*. New Orleans, Louisiana, USA, 1999, pp. 189-194.

[Fog94] D. B. Fogel, "An Introduction to Simulated Evolutionary Optimization," *IEEE Transactions on Neural Networks*, vol. 5, pp. 3-14, 1994.

[Fog00] D. B. Fogel, *Evolutionary Computation: Toward a new philosophy of machine intelligence*, 2nd ed. New York: IEEE Press., 2000.

[Fre04] Freescale Semiconductor Inc, "MPC8560 PowerQUICC III Integrated Communications Processor Family," in *http://www.freescale.com/files/32bit/doc/ref_manual/MPC8560RM.pdf*. rev 1, 2004.

[FU99] E. Farchi and S. Ur, "A Case Study on Improving the Test Design Process and Automating Testing using Functional Models," in *7th European International Conference on Software Testing Analysis & Review*. Barcelona, Spain, 1999, pp. 49-57.

[FUZ04] S. Fine, S. Ur, and A. Ziv, "Probabilistic Regression Suites for Functional Verification," in *Proceedings of the 41st Design Automation Conference (DAC'04)*. San Diego, California, USA, 2004, pp. 49-54.

[FZ03] S. Fine and A. Ziv, "Coverage Directed Test Generation for Functional Verification using Bayesian Networks," in *Proceedings of the 40th Design Automation Conference (DAC'03)*. New Orleans, USA, 2003, pp. 286-291.

[GFL+96] D. Geist, M. Farkas, A. Landver, Y. Lichtenstein, S. Ur, and Y. Wolfsthal, "Coverage-Directed Test Generation Using Symbolic Techniques," in *Proceedings of the First International Conference on Formal Methods in Computer-Aided Design*, 1996, pp. 143-158.

[GHJV95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns Elements of Reusable Object-Oriented Software*. Upper Saddle River, New Jersey: Addison-Wesley, 1995.

[GHO+98] R. Grinwald, E. Harel, M. Orgad, S. Ur, and A. Ziv, "User Defined Coverage: A Tool Supported Methodology for Design Verification," in *Proceedings of the 35th Design Automation Conference (DAC'98)*. San Francisco, California, USA, 1998, pp. 153-168.

[Glu03] A. Gluska, "Coverage-oriented verification of Banias," in *Proceedings of the 40th Design Automation Conference (DAC2003)*. Anaheim, California, USA, 2003, pp. 280-285.

[Glu06] A. Gluska, "Practical Methods in Coverage-Oriented Verification of the Merom Microprocessor," in *Proceedings of the 43rd Design Automatic Conference (DAC'06)*. San Francisco, California, USA, 2006, pp. 332-337.

[Gol89] D. E. Goldberg, *Genetic Algorithms in Search, Optimization, and Machine Learning*. Massachusetts: Addison-Wesley, 1989.

[GS06] C. M. Grinstead and J. L. Snell, *Introduction to Probability*, Second revised ed: American Mathematical Society, 2006.

[Haq07] F. Haque, *The Art of Verification with SystemVerilog Assertions*. Mountain View, California, USA: Verification Central, 2007.

[Har04] Y. Hara, "NEC engineers advance HW/SW co-verification," in *http://www.eedesign.com/article/showArticle.jhtml?articleId=21700502*, 2004.

[HC03] A. Hekmatpour and J. Coulter, "Coverage-Directed Management and Optimization of Random Functional Verification," in *Proceedings of the 34th International Test Conference*. Charlotte, NC, USA, 2003, pp. 148-155.

[HDA95] Y. V. Hoskote, D. Moundanos, and J. A. Abraham, "Automatic Extraction of the Control Flow Machine and Application to Evaluating Coverage of Verification Vectors," in *Proceedings of the 1995 International Conference on Computer Design: VLSI in Computers and Processors (ICCD)*, 1995, pp. 532-537.

[HL92] P. Hajela and C. Y. Lin, "Genetic Search Strategies in Multicriterion optimal design," in *Structural Optimization*, vol. 4. Springer-Verlag, 1992, pp. 99-107.

[HME97] R. Hinterding, Z. Michalewicz, and A. Eiben, "Adaptation in Evolutionary Computation: A Survey," in *Proceedings of the 4th IEEE International Conference on Evolutionary Computation*. Indianapolis, USA, 1997, pp. 65-69.

[HMK96] A. Hosseini, D. Mavroidis, and P. Konas, "Code Generation and Analysis for the Functional Verification of Microprocessors," in *Proceedings of the 33rd Design Automation Conference (DAC'96)*. Las Vegas, Neveda, USA, 1996, pp. 305-310.

[HNG94] J. Horn, N. Nafpliotis, and D. E. Goldberg, "A Niched Pareto Genetic Algorithm for Multiobjective Optimization," *IEEE World Congress on Computational Computation*, vol. 1, pp. 82-87, 1994.

[Ho96] R. C.-M. Ho, "Validation Tools for Complex Digital Designs, PhD Thesis," in *Department of Computer Science*: Stanford University, 1996.

[HS97] S. Hazelhurst and C.-J. H. Seger, "Symbolic Trajectory Evaluation," *Formal Hardware Verification - Methods and Systems in Comparison. Lecture Notes in Computer Science*, vol. 1287, pp. 3-78, 1997.

[HSH+00] P.-H. Ho, T. Shiple, K. Harer, J. Kukula, R. Damiano, V. Bertacco, J. Taylor, and J. Long, "Smart Simulation Using Collaborative Formal and Simulation Engines," in *Proceedings of IEEE/ACM International Conference on Computer Aided Design*. San Jose, California, USA, 2000, pp. 120-126.

[HT04] A. Habibi and S. Tahar, "Towards an Efficient Assertion based Verification of SystemC Designs," in *Proceedings of IEEE International High Level Design Validation and Test Workshop (HLDVT'04)*. California, USA, 2004, pp. 19-22.

[Hub98] H. Hubert, "A Survey of HW/SW Cosimulation Techniques and Tools," Vetneskap Och Konst Royal Institute of Technology, Stockholm 1998.

[HUZ99] A. Hartman, S. Ur, and A. Ziv, "Short Vs. Long - Size does make a difference," in *Proceedings of the IEEE International Workshop on High Level Design, Validation and Test (HLDVT'99)*. San Diego, California, USA, 1999, pp. 23-28.

[HWA99] H. Hulgaard, P. F. Williams, and H. R. Anderson, "Equivalence checking of combinational circuits using Boolean expression diagrams," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 18, pp. 903-917, 1999.

[HYHD95] R. C. Ho, C. H. Yang, M. A. Horowitz, and D. L. Dill, "Architecture Validation for Processors," in *22nd Annual International Symposium on Computer Architecture*, 1995, pp. 404-413.

[HZB+03] R. Henftling, A. Zinn, M. Bauer, W. Ecker, and M. Zambaldi, "Platform-based Testbench Generation," in *Proceedings of Design, Automation and Test in Europe Conference (DATE2003)*. Paris, France, 2003, pp. 1038-1043.

[IEEE05] IEEE, "IEEE Standard for SystemVerilog - Unified Hardware Design, Specification, and Verification Language," Design Automation Standards Committee of the IEEE Computer Society 2005.

[IKNH94] H. Iwashita, S. Kowatari, T. Nakata, and F. Hirose, "Automatic Test Program Generation for Pipelined Processors," in *Proceedings of the International Conference on Computer Aided Design*. San Jose, California, USA, 1994, pp. 580-583.

[Int94a] Intel Corporation, "Statistical Analysis of Floating Point Flaw." Intel White Paper, http://support.intel.com/support/processors/pentium/fdiv/wp/, 1994.

[Int94b] Intel Corporation, "FDIV Replacement Program." http://support.intel.com/support/processors/pentium/fdiv/, 1994.

[Jai97] A. Jain, "Formal Hardware Verification by Symbolic Trajectory Evaluation, PhD Thesis," in *Electrical and Computer Engineering*. Pittsburgh, Pennsylvania: Carnegie Mellon University, 1997.

[JGSB92] W. Jakob, M. Gorges-Schleuter, and C. Blume, "Application of Genetic Algorithms to Task Planning and Learning," *Parallel Problem Solving from Nature, 2nd Workshop, Lecture Notes in Computer Science*, pp. 291-300, 1992.

[Kah01] B. Kahn, "Rapter Test Generator Manual," Freescale Semiconductor 2001.

[KB02] M. Keating and P. Bricaud, *Reuse methodology manual for system-on-a-chip designs*, 2nd ed: Kluwer academic, 2002.

[KHS+97] D. Krishnaswamy, M. Hsaio, V. Saxena, E. Rudnick, J. Patel, and P. Banerjee, "Parallel Genetic Algorithms for Simulation-based Sequential Circuit Test Generation," in *Proceedings of the Tenth International Conference on VLSI Design*, 1997, pp. 475-481.

[KK97] A. Kuehlmann and F. Krohm, "Equivalence Checking using Cuts and Heaps," in *Proceedings of the 34th Design Automation Conference (DAC'97)*. Anaheim, California, USA, 1997, pp. 263-268.

[KN96] M. Kantrowitz and L. M. Noack, "I'm Done Simulating; Now What? Verification Coverage Analysis and Correctness Checking of the DECchip 21164 Alpha Microprocessor," in *Proceedings of the 33rd Design Automation Conference (DAC'96)*. Las Vegas, NV, USA, 1996, pp. 325-330.

[KS92] S. Kang and S. A. Szygenda, "Modeling and Simulation of Design Errors," in *Proceedings of IEEE International Conference on Computer Design: VLSI in Computers and Processors (ICCD'92)*. Cambridge, MA, USA, 1992, pp. 443-446.

[KS94] S. Kang and S. A. Szygenda, "Design Validation: Comparing Theoretical and Empirical Results of Design Error Modeling," *IEEE Design and Test of Computers*, vol. 11, pp. 18-26, 1994.

[LJ99] C.-N. J. Liu and J.-Y. Jou, "An Efficient Functional Coverage Test for HDL Descriptions at RTL," in *International Conference on Computer Design (ICCD'99)*. Austin, Texas, USA, 1999, pp. 325-340.

[LJ01] C.-N. J. Liu and J.-Y. Jou, "Efficient Coverage Analysis Metric for HDL Design Validation," *Proceedings of IEE Computers and Digital Techniques*, vol. 148, pp. 1-6, 2001.

[LLR+00] M. Lajolo, L. Lavagno, M. Rebaudengo, M. S. Reorda, and M. Violante, "Automatic Test Bench Generation for Simulation-based Validation," in *IEEE International Workshop on Hardware/Software Codesign (CODES2000)*. San Diego, USA, 2000, pp. 136-140.

[LMA94] Y. Lichtenstein, Y. Malka, and A. Aharon, "Model-Based Test Generation for Processor Design Verification," in *Proceedings of the 7th Innovative Applications of Artificial Intelligence Conference (IAAI'94)*. AAAI Press, 1994, pp. 83-94.

[LMUZ02] O. Lachish, E. Marcus, S. Ur, and A. Ziv, "Hole Analysis for Functional Coverage Data," in *Proceedings of the 39th Design Automation Conference (DAC'02)*. New Orleans, Louisiana, USA, 2002, pp. 807-812.

[LYMT97] W. Long, S. Yang, Y. Min, and S. Tong, "Level-Oriented GA-Based Test Generation of Logic Circuits," in *IEEE International Conference on Intelligent Processing Systems*. Beijing, China, 1997, pp. 563-567.

[MA98] D. Moundanos and J. A. Abraham, "Using Verification Technology for Validation Coverage Analysis and Test Generation," in *Proceedings of the 16th IEEE VLSI Test Symposium*. Monterey, California, USA, 1998, pp. 254-259.

[MAH96] D. Moundanos, J. A. Abrarham, and Y. V. Hoskote, "A Unified Framework for Design Validation and Manufacturing Test," in *Proceedings of the IEEE International Test Conference*. Washington D.C., USA, 1996, pp. 875-884.

[MAH98] D. Moundanos, J. A. Abraham, and Y. V. Hoskote, "Abstraction Techniques for Validation Coverage Analysis and Test Generation," *IEEE Transactions on Computers*, vol. 47, pp. 2-14, 1998.

[Mar00] A. K. Martin, "Principles and Practice of Symbolic Trajectory Evaluation (STE)," in *Motorola Inc. Report*, 2000.

[Mat96] Y. Matsunaga, "An Efficient Equivalence Checker for Combinational Circuits," in *Proceedings of the 33rd Design Automation Conference (DAC'96)*. Las Vegas, Nevada, USA, 1996, pp. 629-634.

[McM93] K. L. McMillan, *Symbolic Model Checking*: Kluwer Academic Publishers, 1993.

[McM99] K. L. McMillian, "A Methodology for Hardware Verification using Compositional Model Checking," *Cadence Report*, 1999.

[Mel87] T. F. Melham, "Abstraction Mechanisms for Hardware Verification," *VLSI Specification, Verification and Synthesis*, Kluwer Academic Publishers Book, pp. 129-157, 1987.

[Men] Mentor Graphics, "ModelSim." http://www.model.com/.

[MH94] C.-Y. Mao and Y. H. Hu, "Convergence Analyses of Simulated Evolution Algorithms," in *Proceedings of Design Automation of High Performance VLSI Systems*. Madison, Wisconsin, 1994, pp. 30-33.

[MHM97] Z. Michalewicz, R. Hinterding, and M. Michalewicz, "Evolutionary Algorithms," in *Fuzzy Evolutionary Computation*, W. Pedrycz, Ed.: Kluwer Academic, 1997.

[Mic94] Z. Michalewicz, "A Perspective on Evolutionary Computation," in *Proceedings of the Workshop on Evolutionary Computation*. University of New England, Armidale, Australia, 1994, pp. 76-93.

[Mic96] Z. Michalewicz, *Genetic Algorithms + Data Structures = Evolution Programs*, 3rd ed: Springer-Verlag, 1996.

[Mic06] Z. Michalewicz, "Personal communication and correspondence," 2006.

[Mol95] C. Moler, "A tale of two numbers." Mathworks newsletters, http://www.mathworks.com/company/newsletters/news_notes/pdf/win95cleve.pdf, 1995.

[NMUZ03] G. Nativ, S. Mittermaier, S. Ur, and A. Ziv, "Cost Evaluation of Coverage Directed Test Generation for the IBM Mainframe," in *Proceedings of the 40th Design Automation Conference (DAC'03)*. Anaheim, California, USA, 2003, pp. 286-291.

[NV92] A. E. Nix and M. D. Vose, "Modeling Genetic Algorithms with Markov Chains," *Anals of Mathematics and Artificial Intelligence 5*, pp. 79-88, 1992.

[NZE+06] A. Nahir, A. Ziv, R. Emek, T. Keidar, and N. Ronen, "Scheduling-based Test-case Generation for Verification of Multimedia SoCs," in *Proceedings of the 43rd Design Automation Conference (DAC'06)*. San Francisco, California, USA, 2006, pp. 348-351.

[OSCI] Open SystemC Initiative, "SystemC." http://www.systemc.org.

[Par96] V. Pareto, *Cours D'Economie Politique*, vol. I and II. Lausanne, 1896.

[Piz04] A. Piziali, *Functional Verification Coverage Measurement and Analysis*: Springer Verlag, 2004.

[PR97] I. Pomeranz and S. Reddy, "On Improving Genetic Optimization based Test Generation," in *IEEE European Design & Test Conference*. Paris, France, 1997, pp. 506-511.

[PRR94] P. Prinetto, M. Rebaudengo, and M. S. Reorda, "An Automatic Test Pattern Generator for Large Sequential Circuits based on Genetic Algorithms," in *Proceedings of International Test Conference (ITC'94)*. Washington D.C. USA, 1994, pp. 240-249.

[PRRV94] P. Prinetto, M. Rebaudengo, M. S. Reorda, and E. Veiluva, "GATTO: an Intelligent Tool for Automatic Test Pattern Generation for Digital Circuits," in *IEEE International Conference on Tools with Artificial Intelligence*. New Orleans, USA, 1994, pp. 57-61.

[PSK94] S. Palnitkar, P. Saggurti, and S.-H. Kuang, "Finite state machine trace analysis program," in *Proceedings of IEEE International Verilog HDL Conference*. Santa Clara, California, USA, 1994, pp. 52-57.

[Rec73] I. Rechenberg, "Evolution strategie: Optimierung Technisher Systeme nach Prinzipien der biologischen Evolution." Stuttgart: Frommann-Holzboog, Verlag, 1973.

[RPGN94] E. Rudnick, J. Patel, G. Greenstein, and T. Niermann, "Sequential Circuit Test Generation in a Genetic Algorithm Framework," in *Proceedings of the 31st Design Automation Conference (DAC'94)*. San Diego, California, USA, 1994, pp. 698-704.

[RPGN97] E. Rudnick, J. Patel, G. Greenstein, and T. Niermann, "A Genetic Algorithm Framework for Test Generation," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 16, pp. 1034-1044, 1997.

[RPS01] P. Rashinkar, P. Paterson, and L. Singh, *System-on-a-chip Verification, Methodology and Techniques*, 1st ed. Boston, MA, USA: Kluwer Academic Publishers, 2001.

[RSU02] G. Ratsaby, B. Sterin, and S. Ur, "Improvements in Coverability Analysis," in *Proceedings of the International Symposium of Formal Methods Europe*. Copenhagen, Denmark, 2002, pp. 41-56.

[RUW01] G. Ratzaby, S. Ur, and Y. Wolfsthal, "Coverability Analysis Using Symbolic Model Checking," in *Proceedings of the 11th IFIP WG 10.5 Advanced Research Working Conference on Correct Hardware Design and Verification Methods*. Livingston, Scotland, UK, 2001, pp. 155-160.

[RW03] K. A. Ross and C. R. B. Wright, *Discrete Mathematics*: Upper Saddle River, N.J. : Pearson Education, 2003.

[SA98] J. Shen and J. A. Abraham, "Native Mode Functional Test Generation for Processors with Applications to Self Test and Design Validation," in *International Test Conference*. Washington, DC, USA, 1998, pp. 990-999.

[SA99] J. Shen and J. A. Abraham, "Verification of Processor Microarchitectures," in *Proceedings of the 17th IEEE VLSI Test Symposium*. San Diego, California, USA, 1999, pp. 189-194.

[SA00] J. Shen and J. Abraham, "An RTL Abstraction Technique for Processor Microarchitecture Validation and Test Generation," *Journal of Electronic Testing: Theory and Applications*, vol. 16, pp. 67-81, 2000.

[SB95] C. J. Seger and R. Bryant, "Formal Verification by Symbolic Evaluation of Partially-Ordered Trajectories," *Formal Methods in System Design*, vol. 6, pp. 147-189, 1995.

[SBF97] J. Smith, M. Bartley, and T. Fogarty, "Microprocessor Design Verification by Two-Phase Evolution of Variable Length Tests," in *Proceedings of the IEEE International Conference on Evolutionary Computation*. Indianapolis, USA: IEEE, 1997, pp. 453-458.

[Sch84] J. D. Schaffer, "Multiple Objective Optimization with Vector Evaluated Genetic Algorithms, PhD Thesis." Nashville, Tennessee: Vanderbilt University, 1984.

[Sch85] J. D. Schaffer, "Multiple Objective Optimization with Vector Evaluated Genetic Algorithms," in *Proceedings of 1st International Conference on Genetic Algorithms and their Applications*, 1985, pp. 93-100.

[SD84] N. Srinivas and K. Deb, "Multiobjective Optimization using Nondominated Sorting in Genetic Algorithms," *Evolutionary Computation*, vol. 2, pp. 221-248, 1984.

[SDG00] Z. Stamenkovic, H. Dahmen, and U.Glaeser, "VHDL Design Validation by Genetic Manipulation Techniques," in *Proceedings of 22nd International Conference on Microelectronics*. Nis, Yugoslavia, 2000, pp. 735-738.

[Seg00] C. J. Seger, "Introduction to Symbolic Trajectory Evaluation," in *Intel Strategic CAD Labs Report*, 2000.

[SG00] L. Semeria and A. Ghosh, "Methodology for Hardware/Software Co-verification in C/C++," in *Proceedings of the 2000 conference on Asia South Pacific Design Automation*. Yokohama, Japan, 2000, pp. 405-408.

[SGK+06] K. Shimizu, S. Gupta, T. Koyama, T. Omizo, J. Abdulhafiz, L. McConville, and T. Swanson, "Verification of the Cell Broadband Engine Processor," in *Proceedings of the 43rd Design Automation Conference (DAC'06)*. San Francisco, California, USA, 2006, pp. 338-343.

[She99] L. Shen, "Genetic Algorithm Based Test Generation for Sequential Circuits," in *Proceedings of the IEEE Asian Test Symposium*. Shanghai, China, 1999, pp. 179-184.

[SHTK06] A. Samarah, A. Habibi, S. Tahar, and N. Kharma, "Automated Coverage Directed Test Generation Using a Cell-Based Genetic Algorithm," in *IEEE International High Level Design Validation and Test Workshop (HLDVT'06)*. California, USA, 2006, pp. 19-26.

[SMK00] I. F. Sbalzarini, S. Muller, and P. Koumoutsakos, "Multiobjective Optimization using Evolutionary Algorithms," in *Proceedings of the Center for Turbulence Research Program*, 2000, pp. 63-74.

[Squ05] G. Squillero, "MicroGP—An Evolutionary Assembly Program Generator," *Genetic Programming and Evolvable Machines*, vol. 6, pp. 247-263, 2005.

[SRS05] E. Sanchez, M. S. Reorda, and G. Squillero, "Automatic Completion and Refinement of Verification Sets for Microprocessor Cores," in *Applications on Evolutionary Computing: Evo Workshops 2005*, vol. 3449. Lausanne: Lecture Notes in Computer Science, 2005, pp. 204-214.

[SRSV05] E. Sanchez, M. S. Reorda, G. Squillero, and M. Violante, "Automatic Generation of Test Sets for SBST of Microprocessor IP Cores," in *18th IEEE Symposium on Integrated Circuits and Systems Design (SBCCI 2005)*. Florianolpolis, Brazil, 2005, pp. 74-79.

[SS96] J. P. M. Silva and K. A. Sakallah, "GRASP—A New Search Algorithm for Satisfiability," in *International Conference on Computer Aided Design*, 1996, pp. 78-83.

[SS04] J. Steele-Scott, "Enhancing the Software Application Level Verification Methodology - Honours Project Report," School of Electrical and Electronic Engineering, University of Adelaide, Adelaide 2004.

[SSR+05] E. Sanchez, M. Schillaci, M. S. Reorda, G. Squillero, L. Sterpone, and M. Violante, "New Evolutionary Techniques for Test-Program Generation for Complex Microprocessor Cores," in *Genetic and Evolutionary Computation Conference (GECCO05)*. Washington D.C., USA, 2005, pp. 2193-2194.

[SSV04a] E. Sanchez, G. Squillero, and M. Violante, "Exploiting HW Acceleration for Classifying Complex Test Program Generation Problems," in *Proceedings of Evolutionary Computing: Evo Workshops*. Coimbra, Portugal, 2004, pp. 230-239.

[SSV04b] E. E. Sanchez, G. Squillero, and M. Violante, "A Local Analysis of the Genotype-Fitness Mapping in Hardware Optimization Problems," in *Congress on Evolutionary Computation (CEC2004)*. Portland, Oregon, USA, 2004, pp. 871-878.

[SVWG] SystemVerilog Working Group, "SystemVerilog." http://www.systemverilog.org/home.html.

[Syn] Synopsys Inc, "Synopsys VCS RTL Verification Solution." http://www.synopsys.com/vcs/.

[Syn03] Synopsys Inc, "OpenVera Assertions White Paper," in *Synopsys White Paper*, 2003.

[TFC+01] S. Tasiran, F. Fallah, D. G. Chineery, S. J. Weber, and K. Keutzer, "A Functional Validation Technique: Biased-Random Simulation Guided by Observability-Based Coverage," in *IEEE International Conference on Computer Design (ICCD'2001)*. Austin, Texas, USA, 2001, pp. 82-88.

[TKK96] H. Tamaki, H. Kita, and S. Kobayashi, "Multi-Objective Optimization by Genetic Algorithms: A Review," in *Proceedings of IEEE International Conference on Evolutionary Computation*. Nagoya, Japan, 1996, pp. 517-522.

[TQB+98] S. Taylor, M. Quinn, D. Brown, N. Dohm, S. Hildebrandt, J. Huggins, and C. Ramey, "Functional Verification of a Multiple-Issue, Out-of-Order, Superscalar Alpha Processor - the DEC Alpha 21264 Microprocessor," in *Proceedings of the 35th Design Automation Conference (DAC'98)*. San Francisco, California, USA, 1998, pp. 638-643.

[Tur04] R. Turner, "Approaches to accelerated HW/SW co-verification," in *http://www.eedesign.com/article/showArticle.jhtml?articleId=22102217*, 2004.

[UY99] S. Ur and Y. Yadin, "Micro Architecture Coverage Directed Generation of Test Programs," in *Proceedings of the 36th Design Automation Conference (DAC'99)*. New Orleans, Louisiana, United States, 1999, pp. 175-180.

[UZ98] S. Ur and A. Ziv, "Off-the-shelf vs. custom made coverage models, which is the one for you?," in *The International Conference on Software Testing Analysis and Review (STAR)*. San Diego, California, USA, 1998, pp. 36-41.

[UZ02] S. Ur and A. Ziv, "Cross-Fertilization between Hardware Verification Software Testing," in *6th IASTED International Conference on Software Engineering and Applications (SEA2002)*. Cambridge, USA, 2002, pp. 286-291.

[Vel03] M. N. Velev, "Collection of High-Level Microprocessor Bugs from Formal Verification of Pipelined and Superscalar Designs," in *Proceedings of the 34th International Test Conference*. Charlotte, NC, USA, 2003, pp. 138-147.

[VK95] R. Vemuri and R. Kalyanaraman, "Generation of Design Verification Tests from Behavioral VHDL Programs Using Path Enumeration and Constraint programming," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 3, pp. 201-241, 1995.

[WBA05] I. Wagner, V. Bertacco, and T. Austin, "StressTest: An Automatic Approach to Test Generation via Activity Monitors," in *Proceedings of the 42nd Design Automatic Conference (DAC'05)*. Anaheim, California, USA, 2005, pp. 783-788.

[WHY03] F. Wang, G.-D. Hwang, and F. Yu, "Numerical Coverage Estimation for the Symbolic Simulation of Real-Time Systems," in *Proceedings of 23rd IFIP WG 6.1 International Conference on Formal Techniques for Networked and Distributed Systems (FORTE2003)*. Berlin, Germany, 2003, pp. 160-176.

[Wir94] B. Wire, "Intel adopts upon-request replacement policy on Pentium processors with floating point flaw." http://findarticles.com/p/articles/mi_m0EIN/is_1994_Dec_20/ai_15939945, 1994.

[WM93] P. B. Wilson and M. D. Macleod, "Low Implementation Cost IIR Digital Filter Design using Genetic Algorithms," *IEE/IEEE Workshop on Natural Algorithms in Signal Processing*, vol. 1, pp. 41-48, 1993.

[WT95] T.-H. Wang and C. G. Tan, "Practical Code Coverage for Verilog," in *Proceedings of IEEE International Verilog HDL Conference*. Santa Cruz, California, USA, 1995, pp. 99-104.

[YFFR02] X. Yu, A. Fin, F. Fummi, and E. Rudnick, "A Genetic Testing Framework for Digital Integrated Circuits," in *Proceedings of 14th IEEE International Conference on Tools with Artificial Intelligence (ICTAI 2002)*. Washington D.C. USA, 2002, pp. 521-526.

[YS02] J. Yang and C. J. Seger, "Generalized Symbolic Trajectory Evaluation - Abstract in Action," *4th International Conference on Formal Methods in Computer Aided Design (FMCAD), Lecture Notes in Computer Science*, pp. 70-87, 2002.

[YSP+99] J. Yuan, K. Shultz, C. Pixley, H. Miller, and A. Aziz, "Modeling Design Constraints and Biasing in Simulation using BDDs," in *IEEE/ACM International Conference on Computer-Aided Design (ICCAD'99)*. San Jose, California, USA, 1999, pp. 584-589.

[ZH00] Q. Zhang and I. G. Harris, "A Data Flow Fault Coverage Metric For Validation of Behavioral HDL Descriptions," in *Proceedings of IEEE/ACM International Conference on Computer Aided Design (ICCAD2000)*. San Jose, California, USA, 2000, pp. 369-372.

[ZHZ+06] Y. Zhang, H. He, Z. Zhou, X. Yang, and Y. Sun, "A Scalable DSP System for ASIP Design," in *Proceedings of the 8th International Conference on Solid-State and Integrated Circuit Technology (ICSICT06)*. San Francisco, California, USA, 2006.

[Ziv03] A. Ziv, "Cross-Product Functional Coverage Measurement with Temporal Properties-based Assertions," in *Design, Automation and Test in Europe Conference (DATE2003)*. Paris, France, 2003, pp. 834 - 839.

[ZT99] E. Zitzler and L. Thiele, "Multiobjective Evolutionary Algorithms: A Comparative Case Study and the Strength Pareto Approach," *IEEE Transactions on Evolutionary Computation*, vol. 3, pp. 257-271, 1999.