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

APPENDIX A. The Nios System-on-Chip

In this research thesis, the software application level verification methodology (SALVEM) was implemented and applied to the Nios system-on-chip (SoC) [Alt03] from Alterna Inc. This SoC was chosen for establishing the feasibility of SALVEM because it provides reconfiguration features and can be used for different applications. The SoC can be configured with different Altera or user-provided peripherals. Application specific or more general purpose SoCs can be easily configured. The Nios SoC design was also provided in application specific integrated circuit (ASIC) synthesisable and field programmable gate array (FPGA) compliable Verilog behavioural code, which is suitable for simulation.

The Nios SoC has been used in different applications. It was used in Alcatel network platforms, Artesyn Technologies telecommunication network cards and various transmission control protocol/internet protocol (TCP/IP) based servers. Compared to Freescale networking SoCs, the Nios SoC can be used in similar applications and executes similar on-chip transactions. The Nios SoC is ideal for prototyping the SALVEM system, and was expected to provide the same benefits as previous Freescale verification projects.

Additionally, the size and complexity of the Nios SoC can be controlled using configuration options. The SALVEM system is established using a smaller SoC that still provides sufficient application transactions to verify. After proving feasibility of the SALVEM technique, the Nios SoC is expanded with additional peripherals and complexity. Eventually, the SoC can be configured to mimic other common-off-the-shelf vendor SoCs.

For our SALVEM investigations, the SoC was configured as multi-application to reflect common SoCs today. This enables a greater variety of use-cases and associated snippets to be developed. The Nios SoC is configured with SoC devices and settings in Table A.1.

Figure A.1 shows the layout of the Nios SoC. On-chip devices and external memories communicate and transfer data using the Altera Avalon bus. The SALVEM system controls test simulation loading of test programs and other data directly into memories. Universal asynchronous receive/transmit (UART) and parallel input/output (PIO) data transactions are initiated across the SALVEM environment and Nios SoC interface during test execution.

Table A.1 Nios SoC specifications summary

Nios processor	32 bit architecture; 16 bit instructions; Harvard architecture 4KByte data and instruction caches; 256 windowing registers; 5 stage pipeline
Avalon bus	Byte, halfword, word transfers; single or multiple identical masters and slaves arbitration
ROM	64KByte; 32 bit data size
RAM	2KByte; 32 bit data size
SRAM	1MByte; 4-way; 32 bit data size
Flash	8MByte; 8 bit data size
DMA	Streaming transfer; Fixed length or variable end-of-packet terminated transfers; byte, halfword, word transactions; interruptible
UART	8bit port; RS-232; transmit/receive/duplex transfers; end-of-packet termination; interruptible
PIO	8 pins port; R/W/RW direction configurable pins; interruptible;
Timer	32 bit interval timer; internal counter; snapshot based and interruptible

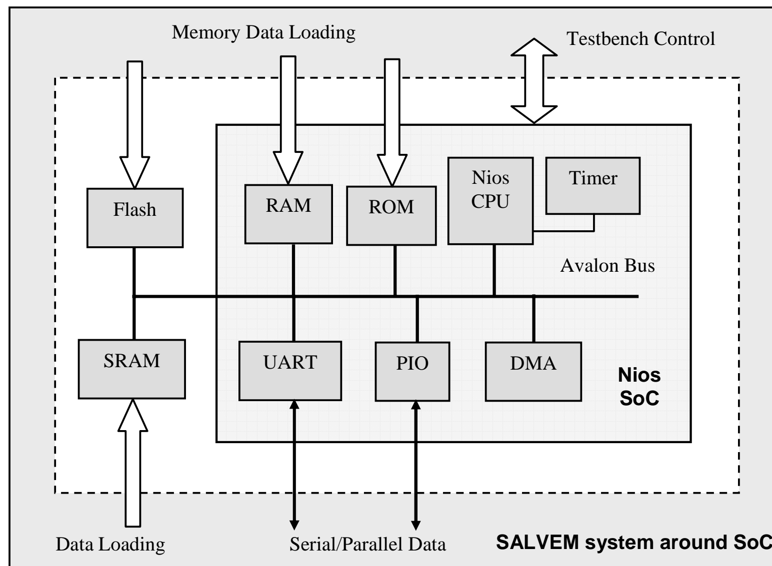


Figure A.1 Nios SoC architecture

APPENDIX B. Additional Literature Survey

This appendix is a supplement to the literature survey of the thesis in Chapter 2. It provides additional information regarding research in other areas that are not directly related to the research thesis, but are still relevant and applicable to our research in some way.

B.1 Other design verification methodologies

B.1.1 Formal verification

Formal verification (FV) techniques certify the correctness of a design by using formal mathematical methods and proofs [CW96]. FV examines the states in an abstracted model of the design to deduce, prove and disprove various circuit properties. A formally verified hardware design is deemed functionally correct for all conditions (i.e. for all possible execution sequences, all possible combinations of inputs and internal states, and all combinations of outputs). Hence, FV is considered a complete verification technique. A correctness guarantee from a formally verified design makes FV is invaluable. However, adoption of FV has been hindered by a number of issues.

In order to verify all hardware operating conditions, the FV tool must be supplied with an abstracted model of the design that is of appropriate size. Despite various abstraction techniques available, usage of FV is often hindered by the size of the designs. Binary decision diagrams (BDD) [Bry86] are commonly used to represent the abstracted design. But when evaluating the design through all possible states, computational resources may not be sufficient. Subsequently, FV is mainly used for small, modular, and specially identified sub-portions of a design only. Simulation is still the primary solution for verifying the overall hardware design. At this stage, FV is still unable to tackle large scale industrial designs in its entirety. Some FV techniques are also considered highly theoretical and difficult to use, especially for users without any prior domain background of the formal methods employed.

Despite design size limitations and usage complexities, research into FV has continued at a steady rate. FV techniques such as symbolic model checking [BCRZ99, McM93, McM99], equivalence checking [HWA99, KK97, Mat96] and symbolic trajectory evaluation [BBS91, BS90, HS97, Jai97, Mar00, SB95, Seg00, YS02] have all been successfully used on industry design projects to verify sub-portions of a chip design.

B.1.2 Semi-formal verification

Semi-formal techniques combine methods from simulation and FV methods. Historically, design verification has been carried out using either simulation or FV. In the last two decades, semi-formal verification has become more common and many proposals for intermixing techniques from one domain to the other have been presented by academia. The aim of semi-formal verification is to compensate for the weaknesses and bring together the strengths of both techniques; by exploiting the design scaling capability in simulation and the abstraction methods that enable complete verification approach in FV.

Usually, semi-formal techniques incorporate FV concepts into a simulation based approach. For example, in [RSU02, RUW01], symbolic model checking is used in coverability analysis to determine what portions of a design can be covered by the test suite and simulation setup. Formal methods are also employed to generate tests for simulation. Banerjee et al. presents an automated test generation method using formally extracted properties and specifications of the design [BPD+06]. Model checkers, path enumerations, formally specified constraints and constraint solvers are also popular for test generation. This is described in Section 2.2.1 Chapter 2 and Section B.4.2.

In fact, our verification research does not operate solely within the simulation domain. For our application driven verification methodology, as far as we are aware, no suitable semi-formal abstraction method has been applied to constrain the size of coverage models and reduce coverage measuring complexities. In this thesis, we apply methods from the FV field of symbolic trajectory evaluation (STE) to represent and facilitate our coverage measuring technique under the software application level verification methodology. We exploit the abstraction mechanisms and trajectory graph checking methods from STE to provide a coverable coverage metric. The size of our coverage model is contained to prevent the state-explosion phenomenon and facilitate more efficient coverage measuring.

B.2 Unit, block, and system based testing

Unit testing

Unit testing concerns verification of small partitions of the design. Each unit is highly modularised and performs specific and well-defined functions. Testing is usually conducted individually in an ad-hoc fashion by designers of the unit. Tests are usually short, directed, and signal pin stimulus based; and are created manually by designers to verify only the basic logical functions and simple operations of

the unit. The aim is to simply produce a working unit that can be combined with other units to perform more thorough testing at the block level. Hence, unit verification effort minimal and involves simple checks of the design.

Block testing

Block testing is the verification of standalone design blocks to ensure they are functionally correct before being integrated with other intellectual property (IP) blocks to form the complete hardware chip design. Given the high levels of reuse in the design process, a chip design can be made up of numerous IP blocks, including IP blocks supplied by external vendors or legacy designs from previous projects. The diversity of these IP blocks makes block testing a critical phase in design verification. However, the focus in block testing is highly individualistic, additional methods are needed to further verify the overall hardware design.

System testing

System testing targets the entire design, which consists of all the separate design modules, on-chip peripherals, and other IP blocks. Given that the internal behaviours of these design blocks should have been validated previously, system testing examines the external response and control signals of these design blocks instead. Assumptions are often made about external interfaces when individual design blocks are tested; system tests ensure any incorrect assumptions are caught.

System tests can be created in different ways, but they must all initiate functional operations throughout the entire chip design, including transactions with the external design environment. The design will be put through a range of operational scenarios including error conditions involving multiple design blocks.

The prerequisite for system testing is that all design blocks making up the system have each undergone prior verification. Otherwise, design bugs uncovered will predominately be those internal to these design blocks, and system testing simply reverts to block or unit testing. Likewise, the post-requisite for unit and block testing is for system testing to follow. In system testing, the greater size and complexities from taking on verification of the entire chip design implies more difficulties in uncovering bugs and attaining high coverage at this level. Much effort and technical resources are

needed in developing and setting up thee testbench simulation platform, test generator, and coverage measuring. Our research focuses on the system test level.

B.3 Coverage driven verification by construction

CDV by construction verifies specific areas of a design based on pre-determined coverage information before any testing is conducted. The coverage information is usually extracted from an abstract model of the design, and specifies the tests needed to cover the design in order to satisfy the coverage goals. This approach guarantees coverage beforehand as tests are directly constructed for coverage of the abstract model.

The model employed under this method is largely based on an extracted state machine or graph description of the design. Abstraction is necessary to condense the model into a manageable size for verification. It removes design information irrelevant to coverage and test generation. To satisfy coverage, the states and transitions of the abstract model must be covered. Abstract tests are constructed beforehand to traverse through this design model. The set of abstract tests are then transformed actual tests that can be simulated on the actual hardware design.

Design modelling, abstraction, and formal verification methods are prominent in CDV by construction. Ho et al. [HYHD95] employed CDV by construction to verify pipelined processor designs in behavioural Verilog. They extracted a finite state machine (FSM) description of the microprocessor and abstracted the FSM to retain only control logic behaviours. Given the abstract state machine model, a transition tour is conducted using FV methods to fully enumerate the design model. Tests are then created to exercise all reachable states and transitions from reset, hence ensuring coverage of the design model. Despite employing abstraction, the scalability of their approach is unknown, even for superscalar processors.

Another CDV by construction technique involves the use of binary decision diagrams to capture the design's functionalities [GFL+96]. In their technique, Geist et al. employs a symbolic model checker as a counter example engine to generate tests. To construct tests that cover the model, false assertions are supplied to the model checker to trigger counter example traces that are concretised into test vectors. The constructed tests ensure coverage as these false assertions cover the functionalities of the model. The Geneieve methodology [DBG01] also creates tests that cover specific coverage events by falsely supplying unreachable behavioural transitions to the model checker.

In [UY99], a similar approach was adopted for verifying pipelining mechanisms in processors. This technique differs slightly where counter example traces are converted to constraints first. These constraints are then resolved by the Genesys [FAL99] external test generator tool adding randomisation as well to create assembler instruction test programs. Despite attaining excellent coverage results, the drawback with these techniques is the limited design sizes can be handled, which is an inherent limitation from applying FV methods such as model checkers. Furthermore, experience and expert knowledge is often needed to model the hardware designs into abstracted representation, derive false assertions, and apply the counter example engine correctly.

Verifying a design using CDV by construction ensures coverage upfront. Simulation is more efficient as testing targets only the required design functions demanded from coverage specific to the design model. The proviso with this approach is to ensure the extraction of the design model represents the actual design appropriately and accurately. Abstraction removes design details and may also conceal bugs by removing them from the model completely. Such hidden bugs would not be detected under this verification scheme. The risks involved with hidden bugs in CDV by construction often make CDV by feedback the more favourable CDV strategy. Furthermore, CDV by construction assumes that all coverage events that need testing will be correctly covered by appropriate tests beforehand. CDV by feedback does not rely on any such assumptions and creates tests for any coverage events when necessary.

B.4 Test generation techniques

B.4.1 Random test generations research domains

Model based

After IBM's RTPG tool [ABD+91], the next generation of random testers were model based [LMA94]. Whilst certain elements of RTPG can be considered model-based, the Genesys [AGL+95, FAL99] and Genesys-Pro [AAF+04] tools derived from RTPG are true implementations of the model-based approach. These tools 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 can be reused to verify other designs. Devising a model of the design along with some other configuration information takes less effort than redeveloping a test generator. The beneficial side effect of having a model of the design is that it can be used for results-checking against actual test simulation as well.

Like the model based approach, our initial random test generation for the software application verification level is also highly reusable. The strength of its reusability comes from the library of code segment building blocks for creating tests. The library of building blocks provide generic functions that simulate the hardware design, which can be reused for different tests and designs of similar applications. The model based approach however, requires a new model of each hardware design to be created for verification each time.

Biasing

In biasing, test generation parameters that shape the random test set are supplied with various values to steer testing toward desired design scenarios. Such user input enables interesting interactions, corner cases, and any pre-identified design areas of concern to be tested earlier and more often; reducing the occurrence of untested design holes. Biasing enhances testing by incorporating additional knowledge of design functionalities and hardware architecture details from the user's perspective (i.e. design or verification engineers). The Genesys [AGL+95, FAL99] and Genesys-Pro [AAF+04] tools are biasing test generators that allow user influence on instruction selections into test programs. It enables various sequences of instructions that target interesting corner cases to be inserted into the generated tests more often.

Our test generations take biasing to another level by controlling both the parameters within test programs internally (i.e. within the domain of individual code segment building blocks), and the overall test suite during the entire verification process. Furthermore, in our algorithmic (i.e. genetic evolutionary) test generation, we extend such equivalent form of biasing from a user manually based approach to an automated scheme; whereby test generation parameters are automatically adjusted to maintain best effective test creations and efficient verifications.

Templates

Template enhanced testing also provides 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. These user devised templates outline the types of tests that will be created. With templates, corner cases,

previously identified design holes, or specific design functions that need special attention can be formally specified in a convenient manner.

In our test generations, our application code segment building blocks function as templates to provide equivalent expressive features for the test creation process. The code segments are in terms of high level software descriptions such as ANSI-C. Such software language descriptors are already powerful mechanisms for laying out the types of SoC functionalities run by eventual software controlling the hardware design. Our code segment building blocks templates are more effective because they are devised and comprise of test elements from a verification engineer's input, and also from actual application software for the SoC.

The other benefit of templates is their reusability for regressions, to verify derivatives of a chip design or even for other verification projects with similar design characteristics. For example, the random test generator in [FUZ04] injects design specification information into templates, and then conducts probabilistic analysis to create regression suites.

The Genesys-Pro [AAF+04] and XGEN [EJN+02] test generators are derivatives of the Genesys tool [AGL+95], and are examples of template based test creations. Each test generator defines their own language to specify test templates, enabling much greater control over the range of test interactions tested. The templates are highly expressive, and can cater for full user directed testing, complete randomisation, or any mixture of both. In [AEM02], the test generator uses templates and pseudo randomness to cater for unexpected operational conditions. They introduce the concept of adaptive test generation that simulate unanticipated test specifications, by incorporating additional test events such as unexpected interrupts, illegal data access, or other unimplemented features in the templates.

Other examples of template-based test generations are described in [WBA05], whereby Markov modelling and randomisation are combined into a test generator called StressTest. User specified templates are supplied to StressTest to generate instruction test programs that targets corner cases. The Raptor and Petra test generators are also examples of user template and biased random test generators for microprocessor and SoC testing [Kah01]. In fact, the successes of these internally developed test generators at Freescale Semiconductor were partially responsible for motivating further test generation research in this thesis.

In our test generations, we do not define our own unique language for equivalent template based test creations. Instead, our equivalent test creation templates are based on common high level software

languages which is the same language used to describe the tests. Hence, our approach can be considered more portable for other designs and verifications as well.

Constraints

Increasingly, random test generators are being enhanced with techniques from formal verification. A common example of semi-formal random test generation is to adopt constraint solving capabilities. 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.

The Genesys-Pro [AAF+04], Piparazzi [ABPZ03], and FPGen [AAF+03] test generators are derivatives of the Gensys tool [AGL+95] employing constraint solving methods. Using test templates as inputs, the test generation process is converted into a constraint satisfaction problem. Pseudo random tests are created by resolving the constraint using a dedicated constraint solver from [BESZ02]. For more directed testing, user requests can also be represented as constraints. Solving the constraint, sub-portions of the test are created to exercise user specified test events. The remaining test creation process is completed by employing randomisation. Behm et al. compares the Genesys and Genesys-Pro for processor verification, and demonstrates the benefits of templates and constraint solving for test generation [BLL+04].

In [CGW+95, CIJ+94], Chandra et al. formalises the use of constraint solving for random test generation by introducing a symbolic instruction graph language. Their AVPGEN test generator also represents test parameters using symbolic values. Users can then create test templates using symbolic parameters and instruction graphs. The graphical templates specify instruction sequences that must satisfy various constraints over the test parameters. A constraint solver is employed on the symbolic instruction graphs to create actual instructions that comply with constraints. Any unconstrained test parameters are chosen randomly. This approach interleaves random decisions with user defined choices given from the constraints and symbolic graph templates.

Yuan et al. [YSP+99] employs input biasing and binary decision diagrams to handle constraints imposed on the types of tests that can be created. In [NZE+06], a system-level SoC random test generator, SoCVer is employed to verify multimedia SoCs. The test generation process is described as

a scheduling problem, whereby various tasks must be allocated to the cores on the SoC appropriately. SoCVer uses randomisation and solves scheduling constraints imposed by the SoC to create software test programs. These test programs run from the SoC's main processor to manage overall execution of test operations carried out throughout the system.

Whilst our test generation process also employs constraints, they function primarily to ensure correctness of the created tests is sound for simulation on the hardware design. For enhancing randomisation, rather than constraint solving, we employ genetic evolutionary techniques as part of the eventual automated and algorithmic test generation procedure. Therefore, our test creation techniques do not require any formal methods, tools, or explicit constraints to be created each time for a new verification project; reducing any complexities associated with complicated formal techniques.

B.4.2 Algorithmic test generators

Semi-formal test generators

A common strategy for enhancing the effectiveness of automated test generators is to adopt formal verification (FV) methods. Like random test generators in Section 2.2.1 Chapter 2 that adopt constraint solving, these semi-formal test generators exploit formalised methods to create tests that can cover specific design functions more effectively.

In [HSH+00], the Ketchum test generator tool is based on both randomisation and multiple FV techniques such as symbolic fixpoint computation, symbolic simulation [Bry91], abstraction, and bounded model checking using satisfiability (SAT) engines [SS96]. At the beginning, Ketchum employs randomisation to create tests that exercises the design through many behavioural states. Once randomisation cannot uncover any new states, formal methods are used to perform unreachability analysis and more exhaustive examinations on the design under verification. Ketchum was successfully applied to industrial designs and attained improved coverage performance compared to randomised tests only. The issue with employing formal search methods together with simulation is the time and computational resources needed to maintain verification performance as the design size increases.

Another popular semi-formal test creation technique is to use model-checkers to derive tests that will exercise specified design behaviours completely. The test generators by Dushina et al. [DBG01] and Geist et al. [GFL+96] are such examples. Design behaviours of interest that need testing are falsely

asserted to be unreachable. A model checker is then used to automatically generate counter example state sequences, which act as tests, to exercise these behaviours. This method can successfully verify a range of design behaviours efficiently, including corner cases. However, it is dependant on the design handling capacity of the model checker. It suffers the same limitations as traditional FV methods.

Another semi-formal test generation method involves using formal symbolic techniques to create test sequences to exercise the design control path, whilst using genetic algorithms to evolve data assignments for the generated tests [FBYR01]. The combination of adopting symbolic BDDs representation and genetic methods is highly novel, but this method is restricted by the common BDD blow-out phenomenon and large search space spanned by GA methods. As design size increases, the BDDs needed to represent control logic grows beyond computational capabilities, and the GA search space becomes too large severely reducing the generator's efficiency.

Finite state machine and state based test generators

Another class of automated test generators involves extracting test vector sequences or mapping assembler instructions from a finite state machine (FSM) representation of the design. Tests are created by traversing the states and transitions of the FSM. This traversal is equivalent to exercising the design behaviours of the actual design when tests are simulated. Design behaviours that require testing correspond strongly to nodes and edges of the FSM. Given the size of chip designs today, the challenge with this method is to create an accurate and compact FSM within the handling capacity of the test generator and simulator.

A number of techniques have been devised for FSM based test generation. In [MAH96, MAH98], Moundanos et al. applies abstraction techniques [Mel87] from FV to create their extracted control flow machine (ECFM). The ECFM is a generalisation of a design's FSM, but uses much smaller state space to encode control flow behaviours only. Test vectors are created to fully enumerate and exercise transitions on the ECFM. The ECFM is smaller than a typical FSM because it focuses solely on the design's control behaviours. Moundanos et al. asserts that covering the control path regardless of data flow is sufficient to detect majority of design errors. Despite abstraction, the ECFM was only demonstrated for small circuits. It fails for larger real-life designs, even when the design circuitry is flattened to reduce complexity.

Shen et al. [SA00, SA98, SA99] extended abstraction methods to handle full microprocessor validation. The novelty of their abstracted FSM is that it can be extracted directly from a design's HDL

description automatically. Their FSM also incorporates mechanisms to preserve cycle accurate behaviours from the design. Test generation algorithms to traverse FSM states and analyse temporal events are applied to create assembler instruction sequences as tests. The effectiveness of this method has been proven on industrial processors from Intel and ARM. Liu and Jou [LJ99] also abstract a design's FSM from the design HDL. To contain the state explosion problem, the size of their FSM is reduced by identifying and grouping states that display the same behaviour into one semantic state. However, their semantic finite state machine (SFSM) was only applied to small to medium size circuit designs. It is unclear from [LJ99] how well the SFSM would scale for larger industrial designs.

Other similar FSM abstraction based test generators by Ho [HYHD95], Ur [UY99] and Vemuri and Kalyanaraman [VK95] have also been proposed. In [HYHD95], control flow paths are identified from the design code description. Each path is translated into constraint equations. By solving constraints, tests are created to complete transition tours of all control paths in the FSM. Any unresolved constraint implies the path is unrealisable and cannot be tested under this technique. Despite the excellent mathematical foundations for their methods, the applicability of these techniques on real world designs was not discussed.

In [IKNH94], Iwashita et al. devised a method whereby processors are modelled as FSMs using binary decision diagrams BDDs. Tests are created by conducting formal reachability analysis, enumerating through all reachable states of the FSM, and then mapping instruction sequences to every traversed BDD node. Regardless of the abstraction, path enumeration, or other FV techniques adopted, the fundamental limitation in FSM based test generation is that the size of designs will eventually exceed the test generator's representative and algorithmic capacity to manage large designs.

Other types of algorithmic test generators

Other algorithmic based methods have been proposed to tackle the test generation problem. Campenhout et al. [CMH98] uses a three phase test generation algorithm. The algorithm consists of searching through a model of a design's control space, employing datapath information to guide further control space searching, and concretizing the tests derived from earlier control space traversal. Their approach was still not yet fully developed, however preliminary experiments shows promising results. Hosseini et al. [HMK96] combines a number of test code generation tools for functional verification of microprocessors. Their set of tools incorporates many of the above methods including heuristic methods, constraint solving, user templates, and randomisation. These tools however, are

designed specifically for testing specific functional features and units of a microprocessor only. It would be useful to assess how these integrated generators perform at testing other designs.

B.4.3 Genetic evolutionary test generators for automatic test pattern generation and other verification test domains

GEA was first proposed and studied extensively as a possible solution for automatic test pattern generation (ATPG) to detect physical hardware faults. Given the extensive vector test space of designs today, the need for generating effective test patterns quickly and effectively makes GEA an ideal optimisation solution [CPRR96a, CPRR96b, CRR98, PR97, PRR94, PRRV94, RPGN94, RPGN97]. Test vectors, patterns and circuit design state elements can be easily mapped to equivalent biological populations, chromosomes, and genome representation to realise an evolutionary pattern generation process [RPGN97]. Results have shown ATPG can be significantly enhanced to stress test large sequential designs when GEA methods are used [CPRR96a, CPRR96b, PR97]. Benefits include better test coverage, more efficient test execution times [PRR94, PRRV94], compact test patterns [CPRR96b], more extensive reachability depths in sequential circuits, and parallel test generation and executions [HT04, KHS+97, LYMT97, She99]. GEA can also be applied for build-in-self-test (BIST) [CCRS01a, CRS01, SRSV05] and software based self-test [SRSV05]. Post-production fault diagnosis of microprocessor devices employing evolutionary refinement techniques have also provided some limited success [BSS+06].

The early successes of GEA for ATPG led to the proposal of adopting genetic evolutionary techniques for formal verification (FV). GEA was first suggested as a possible solution to speed up FV methods during early stages of the design verification cycle. For example, Habibi et al. applied genetic algorithms to improve assertion checking and coverage of designs described in a mixed SystemC and property specification language (PSL) [HT04]. In [SDG00], GEA is applied during property checking and derivation of false counter example states in gate level designs.

B.5 Additional coverage methods surveyed

B.5.1 Other variants of code coverage

In [FFS+01], Ferandi et al. introduces bit coverage which targets both statement and conditional branch coverage. Using stuck-at fault modelling principles from the physical test domain, design variables (e.g. signals or ports) are forced to a stuck-at zero or one value, and branch conditions are stuck-at at true or false. Tests are then created to oppose these stuck-at scenarios. This approach is highly novel as the metric accounts for both statement and conditional coverage, and also employs physical testing and formal techniques for test generation. However, it suffers the same fundamental shortcomings as statement and conditional coverage described above.

Zhang and Harris [ZH00] propose a coverage metric which is a compromise between statement and path coverage. Their method calls for the identification of data flow paths that can trigger erroneous behaviours. By targeting only this subset of data flow paths and their associated statements, coverability can be satisfied. This approach requires further research on how to accurately identify the critical data paths to traverse.

B.5.2 Assertion coverage

Assertion coverage [Piz04, Syn03] measures the number of satisfied assertions out of all assertions devised for the design under test. Assertions are now commonly used for design verification, and can be specified with a number of standardised languages such as SystemVerilog [Haq07, IEEE05, SVWG] and OpenVera [Syn03]. Assertions declare various structural, logical or temporal properties regarding the behaviours and functionalities of the design. For example, in a divide operation, an assertion declares the divisor must never be zero. Like functional coverage in Section 2.3.5 Chapter 2, assertions ensure critical properties or events of the design are exercised. High assertion coverage is to conduct test simulations that trigger a high proportion of these assertions. Despite various tools from EDA vendors that automatically reason assertions for a design, additional design and verification knowledge from engineers is needed to create a complete set of assertions. Otherwise, high assertion coverability would be an inaccurate measure of verification completeness. In order to monitor assertions, simulation overhead is incurred reducing overall verification efficiency.

APPENDIX C. Snippets Test Program Building Blocks

This appendix is a supplement to Chapter 3. The first section summarises comparisons of our SALVEM test creation methods using snippets against instructions based microprocessor test generations. Section C.2 outlines the formalisation technique to describe snippets implementation. Following snippet formalisation details, an example snippet implementation based on this formalised description is provided for initialisation of the direct memory access (DMA) device. Other snippets are formalised, described and implemented in similar manner. The extended Appendix C on the CD-ROM version of this thesis contains examples of formalisation of other snippets after Appendix C.3.1 onwards; such as the parallel input output (PIO), and universal asynchronous receive transmit (UART) devices snippets.

C.1 Comparisons to assembler instruction testing

Our SALVEM snippets based test creation method (Section 3.6 Chapter 3) bears some similarities to test generation of assembler instruction test programs for microprocessor testing. SALVEM test programs are composed of dynamic sequences of snippet code segments whereby snippets perform differently depending on their input parameters. Like our test programs, randomised sequences of assembler instructions and their operand values are employed to test processor cores. From a test creation perspective, the intentions of our snippets and snippet parameters are closely coupled to that of assembler instructions and operands respectively.

In terms of design verification, our test programs are at a higher programming level, invoking system-wide operations that collectively exercise many device functions throughout the SoC. Additionally, snippets are carefully crafted to invoke a range of SoC operations. Snippets must allow for diverse but concise parameterisation to initiate different device functions that can integrate seamlessly into snippet sequences. In contrast, an assembler test generator simply uses whatever instructions are available from the processor’s instruction set architecture.

C.2 Formalisation of snippets

This section provides a template for describing snippets in a formalised manner. The remainder of this section outlines the main headings, and snippet description and implementation details required for each heading, which is needed to formally specify and implement a snippet.

C.2.1 Snippet formalisation and specification elements

A. Description

A1. Designation and label

The snippet's designation should be a name that summarises the goal and main SoC operations conducted by the snippet for verification purposes. The context in which SoC applications perform the snippet operations must also be encapsulated concisely by its designation. So that when it is referred to, the snippet can be recognised and understood immediately for use in different SoC testing. The snippet label is simply a shortened and abbreviated form of the snippet designation, which is easier to use for snippet implementation, documentation and communication purposes between verification engineers.

A2. Purpose

The goals and intentions of the snippets are detailed here. Specifically, a description of what the snippet does, what SoC functionalities in particular does the snippet test, and the verification problems it tackles must be addressed. The types and coverage of SoC testing tasks the snippet is responsible for may also be outlined here. Finally, the priority of the snippet for inclusion into test programs, and its relative importance to other snippets in the snippets library may be discussed and quantified. In this thesis, we use a numerical ranking between one to five, where five is considered highest priority and most important.

A3. Example applications and usage

SoC application use-case scenarios in which this snippet was identified and derived from is presented. These example scenarios will show the context in which the snippet operates within in order to

demonstrate the real-life applicability of testing SoC functions from this snippet. A discussion on how to identify other scenarios which may apply to this snippet may also be given. By presenting example application usages where the snippet originated from, the motivation, rationale, and reasons for creating the snippet (i.e. from A2. Purpose) will be reinforced and better understood. In particular, how the snippet tackles specific SoC verification problems posed by these application use-cases.

B. Implementation

B1. Operations

The main operations, tasks and processes conducted by the snippet in order to test pre-stated SoC functions (i.e. from A2. Purpose) are described in this section. The execution of snippet actions and various control flow processes invoked by the snippet are summarised here with reference to representative, flow-chart and interactive diagrams in B5. Diagrammatic representations. The primary snippet operations should be outlined sequentially in bullet form, stating for each operation, the inputs, outputs and tasks to perform. Appendix C.3.1's B1 sub section shows how to specify snippet operations required by B1 using the *InitDMA* snippet as an example.

B2. Hardware devices and registers

The SoC hardware devices and configuration registers that are involved in the SoC operations initiated by the snippet are listed here; along with a short description for each device and register concerning how they participate (and their responsibilities) within the SoC operations conducted. An accurate listing is essential because these devices are the principle hardware design elements that the snippet is testing, and registers are the facilitator for initiating SoC functions for verification purposes (Section 3.4 Chapter 3).

B3. Drivers

The set of device drivers needed to access registers or perform low-level hardware operations by the snippet is given here. Identifying the necessary drivers beforehand assists in identification of what drivers are available for use or which drivers can be modified for use when the snippet tackles new

SoCs. Otherwise, a plan must be put in place to develop the required drivers. The device drivers are to be listed in ANSI-C API library function call header style.

B4. Snippet relationships

The relationships with other relevant snippets in the snippet library and any interactions with these snippets are described in this section. For example, before snippets can invoke certain operations on a SoC device, the device often needs to be reset and configured appropriately. The collaborations between the snippet and other snippets must be given. For the purposes of snippet composition into the test program, and its reusability for other SoC verification, this information shows what other snippets are needed for inclusion and their roles with respect to the SoC operations invoked. Other snippets closely related to the types of operations invoked by this snippet are also listed here, comparing their similarities and differences. This provides alternatives when choosing and reapplying snippets for other SoCs. For example, other snippets more appropriate for another SoC that perform similar testing can be reused and modified as needed instead.

B5. Diagrammatic representations

This section provides the diagrammatic representations of the snippet to aid the specification information already provided. The three common diagrams employed for our snippet are (1) snippet control and register access flowchart, (2) snippet and devices collaboration diagram, and (3) test composition snippet usage diagram.

The snippet control and register access flowchart shows the control flow of SoC tasks initiated by the snippet in terms of device driver calls and register accesses. This flowchart captures the snippet internal processes and forms the template from which the snippet implementation code will be based upon.

The snippet and devices collaboration diagram outlines the other snippets, SoC devices, or external units that are required during operations invoked by the snippet. It shows the requests and interactions initiated by the snippet in the SoC hardware domain to perform testing. Such a diagram is useful when identifying if the snippet can be reusable on other SoCs; by checking whether other SoCs or verification platforms contain similar devices needed for this snippet to operate with.

The test composition snippet usage diagram shows the various patterns and sequences in which the snippet can be chosen into a test program. Such a diagram is useful to SALVEM test program creation and applying test generation algorithms as it outlines the ways in which the snippet can be used (or restricted) within test programs. The structure of test programs is largely affected by these diagrams for each snippet from the snippet library.

B6. Return variables

The outputs returned as a result of performing the snippet must be specified. Any result, status or control variables are listed so their values can be examined. The return variables may be essential to check for correct operation of the snippet, or may be required for subsequent snippets invoked later on in the test program. The set of return variables, its possible return values, and their meanings are given in this section

B7. Parameters

Parameters are one of the most important specifications for any snippet because it controls the internal snippet operations conducted and the types of SoC functions tested. The complete set of snippet parameters must be specified, detailing for each parameter, its name, value type, range of possible values, any restricted values, and the critical or important values that should be given higher priority. For each parameter, a short statement should also be given discussing how that parameter affects the snippet internal operations conducted, or the types of variation in SoC functions executed.

B8. Constraints

Constraints are an essential element of any snippet because there are certain snippet conditions or settings that need to be maintained or restricted in order for snippet operation and SoC functions to be conducted. Constraints are specified formally using boolean expressions and operators that combine various snippet or test program identifiers together (e.g. snippet label, parameter or variable names, return variables, etc). For example, if a snippet is to initiate DMA transfers from an I/O device, then the source address incrementing mode must be set to non-increment. The address to read data from is fixed to an I/O port. In this case, the constraint might be,

(*source_address* = I/O device) $\wedge$ (*incrementing_mode* = false)

where $\wedge$ is the conjunctive *and* operator.

Any constraint involving the snippet, whether internally or in relation to other snippets or the test program must be specified.

B9. Dependencies

Dependencies specify which other snippets that the snippet currently being specified is dependant upon. Before a snippet can be chosen into a test program, other snippets may need to be executed beforehand or afterwards. For example, before a snippet can initiate operations on a UART device, the reset UART snippet must be selected into and executed by the test program beforehand. Dependencies information must describe what the dependant snippets are, and in what way the snippet is dependant upon them.

Dependency rules ensure only legal snippet sequences are generated. The notation for specifying snippet dependencies are shown in Table C.1. The *targ_snippet* refers to the target snippet that dependency information is being specified for (i.e. the snippet currently being specified). The *dep_snippet* is the snippet that must be executed before or after *targ_snippet* to support its inclusion into the test. SALVEM defines two types of dependencies. A strict dependency requires a dependent snippet (*dep_snippet*) to be executed immediately before (or after) the target snippet. A non-strict dependency implies the *dep_snippet* can be generated amongst other snippets before (or after) the *targ_snippet*.

Table C.1 Snippet dependencies specification notation

Strict dependency	
$\text{targ_snip}\rightarrow^{S}\text{dep_snip}$	Dependent snippet required immediately after target snippet
$\text{dep_snip}^{S}\leftarrow\text{targ_snip}$	Dependent snippet required immediately before target snippet
Non-strict dependency	
$\text{targ_snip}\rightarrow^{NS}\text{dep_snip}$	Dependent snippet required after target snippet
$\text{dep_snip}^{NS}\leftarrow\text{targ_snip}$	Dependent snippet required before target snippet

For example, in DMA transfers using blocking execution mode, DEP(*ExecDMA*$\rightarrow^{S}$*TermDMA*$\rightarrow^{S}$*CheckDMA*) require the DMA termination and check snippets to execute immediately after a DMA transfer execution snippet. This implies dependencies between snippets can also be cascaded.

Similarly, DEP(*ResetUart*$^{NS}\leftarrow$*TxUart*|*RxUart*|*RxTxUart*) imply the UART device can be initialised in between other snippets before it is used to transmit or receive data. A comprehensive set of dependencies specified for each snippet governs the types of snippet sequences that can be composed by our test generator. Figure 3.9 Chapter 3 showed examples of strict and non-strict dependencies.

The snippet dependency information provided here is a subset of the information from B4. Snippet relationships specifications. The test composition snippet usage diagram from B5. Diagrammatic representations, illustrates the snippet dependency information as well.

B10. Pre and post conditions

Pre and post conditions specify requirements that must be satisfied before and after the snippet is executed during test execution. For example, to perform a simple memory data transfer, the pre condition requires that data be available for transfer beforehand. If non-sharing devices are used by the snippet operation, then the post condition must check that these devices are freed for usage by other snippets afterwards.

B11. External requirements

The off-chip requirements needed for snippet operation are specified here. These requirements refer primarily to devices or software external to the SoC, in particular, requirements relating specifically to the testbench or verification platform. For example, snippets involving I/O devices on the SoC usually require some testbench module that can handle the data output by the SoC or provide appropriate stimulus to the I/O ports. External checker modules may also be implemented to check for correct operation of the SoC functions invoked by snippets.

B12. Restrictions

Restrictions refer to the SoC design functionalities that are not tested by the snippet and are not explicitly obvious. It addresses the question of “What is not covered by the snippet?” or “What doesn’t the snippet do?” For example, there may be certain modes of operations not tested explicitly by snippets. Restriction information is useful for keeping track of unimplemented features of a snippet and what SoC test functions are not handled yet, especially during snippet development.

B13. Checking mechanisms

Checking mechanisms outlines what procedures are undertaken to check that the SoC functions invoked by snippets are carried out correctly. It details the results checking features employed by the snippet. For example, is checking operations performed after the SoC operations and snippet execution has completed, or are checks performed on-the-fly whilst the SoC functions are being carried out, or a combination of both? Also, does the snippet perform self-checking procedures or are the checks performed externally by an interfacing checking module from the testbench.

B14. Example test program usage

An example of how the snippet is inserted within the structure of the test is shown. Specifically, the snippet ANSI-C function call that will be used within the test program, including function parameters and return values, must be given.

B15. Snippet pseudo/implementation sample code

The snippet implementation code shows how the snippet is put into practice for SALVEM. Generally, the implementation code is in ANSI-C and should be applicable for verifying the SoC under SALVEM, as long as the appropriate snippet drivers are available. Otherwise, the snippet sample code may be implemented as pseudo code abstracting the SoC specific details until the snippet is to be implemented for verification. The sample code provided may either be code fragments of the important operations encapsulated by the snippet, or complete code implementation of the entire snippet operations.

C. Discussion

C1. Variation possibilities

The manner in which the snippet can be varied, and the types and range of variation in the SoC functions that can be tested are outlined here. For each variation possibility, the outcome from the variation, and the inputs (e.g. parameter settings) and conditions needed in order to produce the variation must be specified.

C2. Attributes

This section lists some of the quantitative attributes of the snippets, for example, the size of the snippet and execution performance times when it is simulated on the SoC. Depending on the available executable program memory size on the SoC under test, the number of lines of code or memory footprint of the snippet affects how many different snippets in total can be composed within a test program. Similarly, certain variation options and SoC functions initiated by the snippet may require different execution times – influencing the types of variation and way in which the snippet is configured in the test program.

C3. Advantages and disadvantages

The benefits and shortcomings from applying the snippet must be discussed. Any potential trade-offs or issues from implementing the snippet must also be listed. These factors are important considerations when deciding whether to use the snippet for verifying the SoC or how often the snippet should be chosen for inclusion into the test program. For example, whilst a snippet may be able to invoke many variations to test a particular SoC function, the resultant size of the snippet needed to implement such variation range may be too large. A compromise between reducing variation (or test effectiveness) and snippet size is required, and the snippet implementation may differ from the standard specification given. The snippet may even be broken down into multiple but smaller snippets.

C4. Other considerations

Any other remaining characteristics or issues regarding the snippet are supplemented in this last section. The information to be given here is optional, but may include aspects such as the snippet's level of reusability, portability, extensibility or flexibility. Specific recommendations on how to implement the snippet, or hints regarding how the snippet should be used by the test creation process may also be given.

C.3 Example snippets formalisation, specification and implementation details

C.3.1 InitDMA snippet

A. Description

A1. Designation and label

Name : DMA initialisation

Label : *InitDMA*

A2. Purpose

Intention : The *InitDMA* snippet configures the DMA device in readiness for executing DMA transfers. It accesses DMA control registers to set up the DMA device such that different DMA data transfers can be conducted later depending on the values written to the registers. If the DMA transfer involves read or write destination devices that require configuration as well, the *InitDMA* snippet is also responsible for setting up those devices.

Test goals : This snippet tests the DMA transfer configuration possibilities that can be realised throughout the entire hardware system. It also tests if illegal types of data transfers are allowed to be configured on the DMA device, and how it handles such error settings.

Priority : 5

A3. Example applications and usage

Scenario : The SoC can be used for DSP purposes. This requires the DMA to be configured so that DMA transfers can be initiated immediately in the background whilst the main CPU (or DSP engine) is performing high intensive DSP mathematical calculations. The configuration of DMA transfers is important for transferring large amounts of input, output and temporary signal data needed by various DSP operations. Configuring non-blocking DMA transfers frees the rest of the system so that DSP tasks are the primary processes conducted on the SoC. The configuration of various memories to act as DMA read and write devices is particularly useful for this scenario. Figure 3.2 Chapter 3 represented this scenario diagrammatically.

Generally, configuration of DMA transfers by this snippet is particularly applicable in scenarios where data transfers are required to be conducted independently of other operations.

B. Implementation

B1. Operations

The main operations conducted by the *InitDMA* snippet are, to perform various sanity checks on the configurations selected for the DMA transfer, use device drivers to access DMA configuration registers to set up the DMA transfer, and prepare the DMA device or any other relevant device in readiness for transfer execution.

The steps taken to perform snippet operations are as follows.

Step 1 :

> **Input :** DMA transfer status register value.
>
> **Task :** Check that it is appropriate to configure the DMA device for transfers, i.e. the non-sharing hardware devices required are not reserved.
>
> **Output :** True or false indicating if a DMA transfer is in progress.

Step 2 :

> **Input :** All DMA registers.
>
> **Task :** Clear and reset all settings on the DMA.
>
> **Output :** None.

Step 3 :

Input : DMA read and write address values chosen.

Task : Check that the DMA read and write address is assigned to an I/O device port, or within a valid memory device address range.

Output : True or false indicating if addresses are valid.

Step 4 :

Input : DMA transfer size, and read and write address values chosen.

Task : If the DMA transfer involves memories, check that the data transfer will not overwrite the same address range or exceed memory device boundaries.

Output : True or false indicating if addresses are valid.

Step 5 :

Input : DMA transactional unit size, read and write address values chosen, SoC data bus specifications.

Task : Check that the transactional unit size chosen for each transfer cycle is appropriate for the source and destination devices involved, and the available data bus bandwidth.

Output : True or false indicating if the transactional unit size is valid.

Step 6 :

Input : DMA transfer size and transactional unit size chosen.

Task : Check that the total amount of data to transfer is a multiple of the transactional unit size chosen, particularly if the DMA device cannot handle data that need to be broken down into halfword or byte size units.

Output : True or false indicating if the transfer size and transactional unit size is valid.

Step 7 :

Input : DMA address incrementing modes chosen.

Task : Check that the address increment mode is enabled only for reading or writing between memory devices.

Output : True or false indicating if address incrementing modes are valid.

Step 8 :

Input : The chosen check DMA error handling mode, and checking results from steps 3 to 7.

Task : If any of the DMA configuration checks are violated, flag an error and recommence snippet selections. However, if the intention of testing is to explicitly test SoC error handling, then proceed with the tasks in this snippet to set up illegal hardware conditions.

Output : Error flag settings.

Step 9 :

Input : DMA end-of-packet register value chosen, and driver to access the register.

Task : Write to the end-of-packet register to configure streaming DMA transferring or a fixed data amount transfer.

Output : None.

Step 10 :

Input : DMA transfer size register value chosen, and driver to access the register.

Task : Write to the transfer size register to specify how much data to transfer.

Output : None.

Step 11 :

Input : DMA read and write incrementing mode control register values chosen, and drivers to access the registers.

Task : Write to the read and write incrementing mode register to enable or disable incrementing of the read and write address after every transfer cycle.

Output : None.

Step 12 :

Input : DMA transactional size control register value chosen, and driver to access the register.

Task : Write to the transactional size register to configure the DMA to execute data transfer in terms of byte, halfword, word, or other appropriate sized units.

Output : None.

Step 13 :

Input : DMA read and write address register values chosen, and drivers to access the registers.

Task : Write to the read and write address register to set up the starting addresses to read and write data between.

Output : None.

Step 14 :

Input : Snippets or drivers to configure DMA read and write destination devices chosen.

Task : Call other snippets or device drivers to set up source or destination devices if such configuration is required by these devices to participate in DMA transfers.

Output : None.

Step 15 :

Input : DMA transfer termination mode control register value chosen, and driver to access the register.

Task : Write to the transfer termination mode register to establish how the DMA will terminate data transferring.

Output : None.

Step 16 :

Input : DMA execution transfer mode chosen, DMA interrupt service routine, and driver to install the routine.

Task : Set up the DMA transfer operation to be blocking or non-blocking (interruptible). If non-blocking, set up interrupt routines and priorities to handle DMA termination notification.

Output : None.

Step 17 :

Input : None.

Task : Save and queue up the configuration of DMA transfer for execution.

Output : None.

Step 18 :

Input : None.

Task : Return status of snippet operation.

Output : Numerical value indicating the outcome from executing this snippet.

B2. Hardware devices and registers

Device	Purpose
DMA	To be configured to perform DMA transfers.
Memories	To act as read or write devices for the DMA data transfers.
UART	To act as read or write devices for the DMA data transfers.
PIO	To act as read or write devices for the DMA data transfers.
Registers	**Purpose**
DMA status register	To check for any ongoing DMA transfers.
DMA control register	To configure the transactional unit size, address incrementing mode, termination mode, and block or non-blocking DMA transfer execution mode.
DMA length register	To specify the amount of data to transfer.
DMA read address register	To specify the start address to read data from.
DMA write address register	To specify the start address to write data to.
UART status register	To check if the UART device is free to participate in DMA transfer.

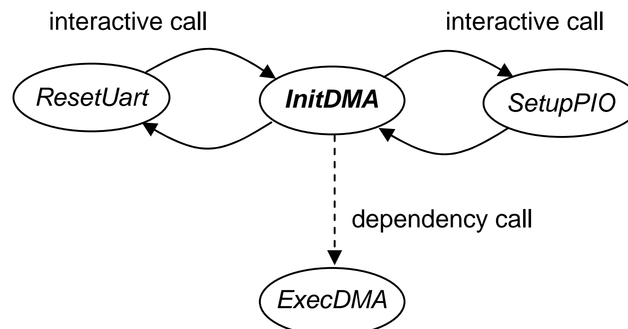
B3. Drivers

Device driver functions and purpose
// Checks if the DMA is idle to perform new DMA transfers. statusEnumType CheckDMAStatus (int DMAdeviceID);
// Clears all existing settings on the DMA. int ResetDMA (int DMAdeviceID, int mode);
// Sets the amount of data to transfer. int SetTransferSize (int DMAdeviceID, int size);
// Sets the end-of-packet byte character for streaming transfer. int SetEOP (char EOP, int size);
// Sets the address incrementing mode during transfer. int SetRWIncrementingMode (int DMAdeviceID, int readIncMode, int writeIncMode);

// Sets the transactional unit size for transferred in each cycle. int SetTransactionSize (int DMAdeviceID, int unitSize);
// Sets the starting read and write addresses. int SetRWAddresses (int DMAdeviceID, int readAddr, int writeAddr);
// Specifies the conditions for termination of DMA transfers. int SetTerminationMode (int DMAdeviceID, int termMode);
// Installs the DMA interrupt handling routine. void nr_installuserisr (int DMAinterruptID, void (*ISR)(), int DMAdeviceID);
// Sets the block or non-block (interrupt) transfer mode. int SetTransferInterrupt (int DMAdeviceID, int transferMode);
// Stores the DMA configuration settings for DMA transfer execution. void SaveDMAState (DMAconfigEnumType DMAconfig, int DMAdeviceID);
// Store the other device settings involved in DMA transfer execution. void SaveExtDeviceSettings (int DMAdeviceID, *external device setting parameters*, …*etc*…);

B4. Snippet relationships

Collaborative snippets : *ResetUart, SetupPIO, ExecDMA*



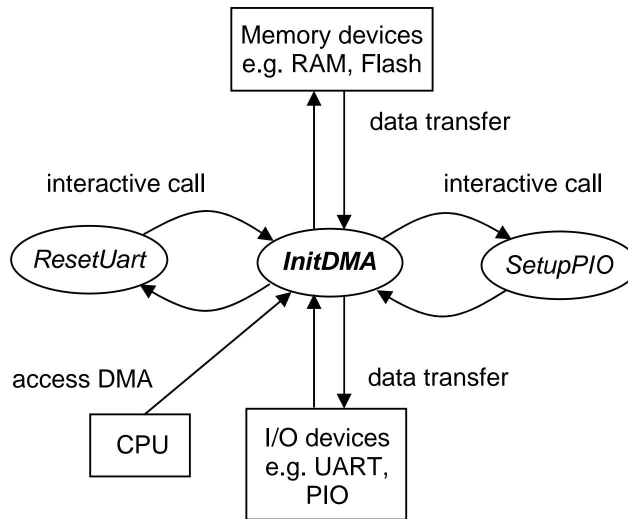
Related snippets : None.

B5. Diagrammatic representations

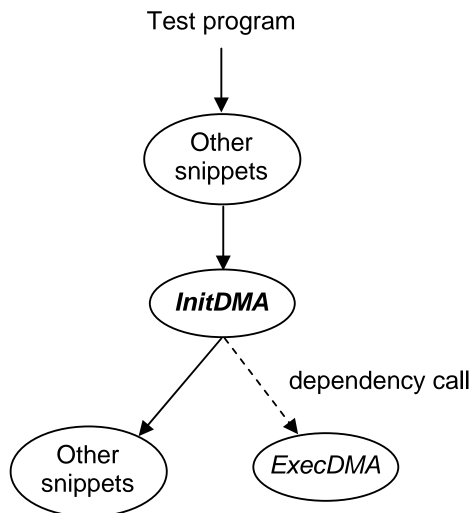
Snippet control and register access flowchart :

This diagram was shown previously in Figure 3.7 Chapter 3.

Snippet and devices collaboration diagram :



Test composition snippet usage diagram :



B6. Return variables

Variable	Value	Meaning
dmaId	> 0	DMA configuration successful, the positive integer returned corresponds to the ID of the DMA transfer that is being queued for execution.
	< 0	DMA configuration unsuccessful, the negative integer returned corresponds to one of the errors that caused the configuration to fail.

B7. Parameters

Parameters	
Name	dma
Type	enumerated SoC device type
Range	list of SoC devices
Critical values	Not applicable
Restricted values	Not applicable
Description	Identifies the type of device this snippet is primarily responsible for, in this case, the DMA device.
Name	dmaId
Type	integer
Range	Identify numbers of available DMA devices
Critical values	Not applicable
Restricted values	Not applicable
Description	Identifies which DMA this snippet operates on.
Name	rAddr
Type	integer
Range	0 to maximum readable memory address
Critical values	boundary values of memory devices
Restricted values	None
Description	Specifies the start address to read data from during DMA transfers
Name	wAddr
Type	Integer
Range	0 to maximum writeable memory address
Critical values	boundary values of memory devices
Restricted values	None
Description	Specifies the start address to write data to during DMA transfers
Name	length
Type	integer
Range	0 to (2^11)-1
Critical values	0, 1, 2, 4, (2^11)-1
Restricted values	None
Description	Specifies the amount of data bytes to transfer
Name	termMode
Type	integer
Range	0 (length), 1 (read end-of-packet), 2 (write end-of-packet)
Critical values	Not applicable
Restricted values	None
Description	Specifies the termination mode for DMA transfers (e.g. length, read end-of-packet, write end-of-packet, etc termination)
Name	rAddrCon
Type	integer
Range	0 (increment), 1 (constant)
Critical values	Not applicable
Restricted values	None
Description	Specifies whether the read address will increment or remain constant during DMA transfers.
Name	wAddrCon

Type	integer
Range	0 (increment), 1 (constant)
Critical values	Not applicable
Restricted values	None
Description	Specifies whether the write address will increment or remain constant during DMA transfers.

Name	checkError
Type	short
Range	0 or 1
Critical values	0 and 1
Restricted values	None
Description	Specifies whether to check for errors operations during DMA transfers (1: check, 0: do not check).

Name	uartEOPValue
Type	short
Range	0 to (2^8)-1
Critical values	0, 0xFF, 0xAA, 0x55, 0xF0, 0x0F
Restricted values	None
Description	Specifies the end-of-packet byte character to trigger DMA transfers involving the UART.

Name	uartDivisor
Type	short
Range	3 to 10
Critical values	None
Restricted values	None
Description	Specifies a UART divisor value if UART configuration is needed.

Name	pioDir
Type	short
Range	0 to (2^8)-1
Critical values	0, 0xFF, 0xAA, 0x55, 0xF0, 0x0F
Restricted values	None
Description	Specifies the PIO directional pin settings for DMA transfers involving the PIO device.

Name	pioIntMask
Type	short
Range	0 to (2^8)-1
Critical values	0, 0xFF, 0xAA, 0x55, 0xF0, 0x0F
Restricted values	None
Description	Specifies the PIO pin interrupt mask settings for DMA transfers involving the PIO device.

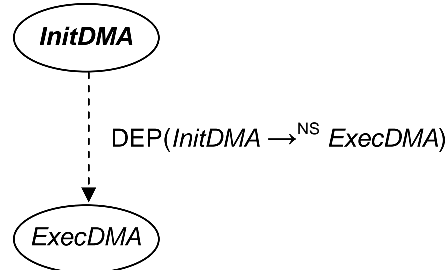
Name	pioEdgeCap
Type	short
Range	0 to (2^8)-1
Critical values	0, 0xFF, 0xAA, 0x55, 0xF0, 0x0F
Restricted values	None
Description	Specifies the PIO pin edge capture settings for DMA transfers involving the PIO device.

B8. Constraints

Constraints and meaning
$(rAddrCon = 0) \wedge (rAddr \leq \text{ROM_end_addr} \wedge rAddr \geq \text{ROM_start_addr}) \wedge ((rAddr + length) \leq \text{ROM_end_addr})$ If the read address accesses the ROM memory, and the address is to be incremented, ensure the transfer does not exceed the boundary of the ROM.
$(rAddrCon = 0) \wedge (rAddr \leq \text{RAM_end_addr} \wedge rAddr \geq \text{RAM_start_addr}) \wedge ((rAddr + length) \leq \text{RAM_end_addr})$ If the read address accesses the RAM memory, and the address is to be incremented, ensure the transfer does not exceed the boundary of the RAM.
$(wAddrCon = 0) \wedge (wAddr \leq \text{RAM_end_addr} \wedge wAddr \geq \text{RAM_start_addr}) \wedge ((wAddr + length) \leq \text{RAM_end_addr})$ If the write address accesses the RAM memory, and the address is to be incremented ensure the transfer does not exceed the boundary of the RAM.
$(rAddrCon = 0) \wedge (rAddr \leq \text{SRAM_end_addr} \wedge rAddr \geq \text{SRAM_start_addr}) \wedge ((rAddr + length) \leq \text{SRAM_end_addr})$ If the read address accesses the SRAM memory, and the address is to be incremented, ensure the transfer does not exceed the boundary of the SRAM.
$(wAddrCon = 0) \wedge (wAddr \leq \text{SRAM_end_addr} \wedge wAddr \geq \text{SRAM_start_addr}) \wedge ((wAddr + length) \leq \text{SRAM_end_addr})$ If the write address accesses the SRAM memory, and the address is to be incremented ensure the transfer does not exceed the boundary of the SRAM.
$(rAddrCon = 0) \wedge (rAddr \leq \text{Flash_end_addr} \wedge rAddr \geq \text{Flash_start_addr}) \wedge ((rAddr + length) \leq \text{Flash_end_addr})$ If the read address accesses the Flash memory, and the address is to be incremented, ensure the transfer does not exceed the boundary of the Flash.
$(wAddrCon = 0) \wedge (wAddr \leq \text{Flash_end_addr} \wedge wAddr \geq \text{Flash_start_addr}) \wedge ((wAddr + length) \leq \text{Flash_end_addr})$ If the write address accesses the Flash memory, and the address is to be incremented, ensure the transfer does not exceed the boundary of the Flash.
$(transSize = \text{halfword}) \wedge (length \% 2) = 0$ If halfword transaction unit size is chosen, ensure the number of bytes to ransfer is a multiple of two.
$(transSize = \text{word}) \wedge (length \% 4) = 0$ If word transaction unit size is chosen, ensure the number of bytes to transfer is a multiple of four.
$(rAddr = \text{PIO} \vee rAddr = \text{UART}) \wedge (rAddrCon = 1)$ If the read address is an I/O device port, ensure that the read address is not incremented.
$(wAddr = \text{PIO} \vee wAddr = \text{UART}) \wedge (wAddrCon = 1)$ If the write address is an I/O device port, ensure that the write address is not incremented.

$(rAddr = \text{PIO} \vee rAddr = \text{UART} \vee wAddr = \text{PIO} \vee wAddr = \text{UART}) \wedge (transSize = \text{byte})$
If the read or write address is an I/O device port, ensure that the transaction unit is byte sized.

B9. Dependencies



B10. Pre and post conditions

Pre-conditions
DMA is idle.
DMA transfer queue is not full.
Post-conditions
DMA is idle but ready to perform transfer.
DMA transfer queue capacity reduced by one.

B11. External requirements

None.

B12. Restrictions

None.

B13. Checking mechanisms

None. The correct operation of this snippet is checked implicitly by performing the DMA transfer when the dependant snippet, *ExecDMA*, is executed later in the test program.

B14. Example test program usage

```
ok = InitDMA  (na_DMA_1,  // DMA device
               1,  // DMA Id
               0x00008000,  // read address
               na_UART_1,  // write address
               0x00000100,  // length
               1,  // transaction size (byte)
               2,  // termination mode
               0,  // read address incrementing mode
               1,  // write address incrementing mode
               0,  // block or non-blocking mode
               1,  // check error mode
               0x1b,  // end-of-packet character
               3,  // UART divisor
               0x0,  // PIO port direction
               0x0, // PIO mask
               0x0, // PIO edge capture
              );
```

B15. Snippet pseudo/implementation sample code

```
1     ////////////////////////////////////////////////////////////////
2     // InitDma snippet
3     //
4     // Initialises a DMA peripheral ready for transferring data.
5     //
6     // These parameters will vary the functional operations
7     // performed by this snippet.
8     //
9     // dma : dma identifier (to identify the dma device)
10    // dmaId : An allocated DMA transfer operation id.
11    // rAddr : Source address to read data from.
12    // wAddr : Destination address to write data to.
13    // length : The number of data units (bytes/halfwords/words) to transfer.
14    // transSize : The size of a data unit (byte, halfword, or word).
15    // termMode : The data transfer termination mode.
16    // rAddrCon : 0 - Increment read addr during transfer, 1 - otherwise
17    // wAddrCon : 0 - Increment write addr during transfer, 1 - otherwise
18    // intEnable : Enable DMA interrupts handling (for non-blocking transfer)
19    // checkError : 1 - do not allow DMA error conditions to be
20    //                  exercised,
21    //              0 - allow DMA error conditions to test error handling mechanisms
22    // uartEOPValue : end-of-packet character value for DMA induced Uart transfers.
23    // uartDivisor : Uart Divisor value to use for DMA induced Uart transfers.
24    // pioDir : PIO directional pin settings to use for DMA induced PIO transfers.
25    // pioIntMask : PIO Interrupt Mask pin settings to use for DMA induced PIO transfers.
26    // pioEdgeCap : PIO Edge Capture pin settings to use for DMA induced PIO transfers.
27    //
28    // Returns  >0 - if the DMA was successfully initialized;
29    //          <0 - if essential DMA settings cannot be made, and
30    //               transfer cannot proceed. The negative integer returned
31    //               corresponds to specific errors.
32    ////////////////////////////////////////////////////////////////
33
34
35
36    #include <stdlib.h>
37    #include <excalibur.h>
38    #include "DMA_drivers.h"
39    #include "mutex.h"
40    #include "uart.h"
41    #include "testbench.h"
42
43    int  InitDMA  (np_dma *dma, int dmaId, int rAddr, int wAddr,
44                   int length, short transSize, short termMode,
45                   short rAddrCon, short wAddrCon,
46                   short intEnable, short checkError,
47                   int uartEOPValue, int uartDivisor,
48                   int pioDir, int pioIntMask, int pioEdgeCap
49                  ) {
50
51       int notOk;
52
53       // Check some pre-conditions :
54       // Transfer in progress? If so, unable to make changes to DMA device.
55       // Wait for any previous transfers to complete.
56       while (CheckDMAStatus(dma) == DMA_TRANSFER_IN_PROGRESS) {}
57
58       // Call device driver to reset DMA.
59       notOk = ResetDMA(dma, 0);
60       if (notOk) {
70          return -1;
71       }
72
73       // Constraints checking :
74
75       // Perform checks of the length, address incrementing modes, transaction
76       // size, and addresses, before setting them in registers
77
78       // If non-fixed addresses to read/write, check for overflow,
79       // (rAddr+length) <= max_peripheral_rAddr_range
80       // (wAddr+length) <= max_peripheral_wAddr_range
81       // and check for Overwriting,
82       // (rAddr+length) < wAddr, if rAddr and wAddr are in the
```

```
83              //                                      same memory range.
84              if (checkError) {
85                 if (rAddrCon==0) {
86                    if (rAddr>=(int)na_Ext_Flash && rAddr<(int)na_Ext_Flash_end &&
87                       (rAddr+length)>(int)na_Ext_Flash_end) {
88                       return -2;
89                    }
90                    if (rAddr>=(int)na_Ext_SRAM && rAddr<(int)na_Ext_SRAM_end &&
91                       (rAddr+length)>(int)na_Ext_SRAM_end) {
92                       return -3;
93                    }
94                    if (rAddr>=(int)na_on_chip_ROM && rAddr<(int)na_on_chip_ROM_end &&
95                       (rAddr+length)>(int)na_on_chip_ROM_end) {
96                       return -4;
97                    }
98                    if (rAddr>=(int)na_on_chip_RAM && rAddr<(int)na_on_chip_RAM_end &&
99                       (rAddr+length)>(int)na_on_chip_RAM_end) {
100                      return -5;
101                   }
102                   if (rAddr<wAddr && (rAddr+length)>wAddr) { // Overflow?
103                      return -6;
104                   }
105                }
106                if (wAddrCon==0) {
107                   if (wAddr>=(int)na_Ext_Flash && wAddr<(int)na_Ext_Flash_end &&
108                      (wAddr+length)>(int)na_Ext_Flash_end) {
109                      return -7;
110                   }
111                   if (wAddr>=(int)na_Ext_SRAM && wAddr<(int)na_Ext_SRAM_end &&
112                      (wAddr+length)>(int)na_Ext_SRAM_end) {
113                      return -8;
114                   }
115                   if (wAddr>=(int)na_on_chip_RAM && wAddr<(int)na_on_chip_RAM_end &&
116                      (wAddr+length)>(int)na_on_chip_RAM_end) {
117                      return -9;
118                   }
119                }
120             }
121
122             // Check that correct transaction length and unit size is chosen,
123             // hw transaction size, (length mod 2 == 0)
124             // word transaction size, (length mod 4 == 0)
125             if (checkError) {
126                if (transSize==2 && (length & 1)!=0) {
127                   return -10;
128                }
128                if (transSize==4 && (length & 3)!=0) {
130                  return -11;
131                }
132             }
133
134             // Call device driver to set the transaction size
135             notOk = SetTransferSize(dma, length);
136             if (notOk) {
137                return -12;
138             }
139
140             // For streaming transfer, call the device driver to set the end of packet register
141             notOk = SetEOP(uartEOPValue, length);
142             if (notOk) {
143                return -13;
144             }
145
146             // Check that address is not incrementing for UART or PIO peripheral.
147             if (checkError) {
148                if (rAddrCon==0 && (rAddr==(int)&(na_PIO_1->np_piodata) ||
149                   rAddr==(int)&(na_Uart_1->np_uartrxdata))) {
150                   return -14;
151                }
152                if (wAddrCon==0 && (wAddr==(int)&(na_PIO_1->np_piodata) ||
153                   wAddr==(int)&(na_Uart_1->np_uarttxdata))) {
154                   return -15;
155                }
```

```
156         }
157         // Call device driver to set read/write address incrementing modes
158         notOk = SetRWIncrementingMode(dma, rAddrCon, wAddrCon);
159         if (notOk) {
160             return -16;
161         }
162 
163          // Check that the data transaction size is appropriate for the
164          // peripherals bus width involved n the transfer.
165          if (checkError && transSize!=1 &&
166              (rAddr==(int)&(na_PIO_1->np_piodata) ||
167              rAddr==(int)&(na_Uart_1->np_uartrxdata) ||
168              wAddr==(int)&(na_PIO_1->np_piodata) ||
169              wAddr==(int)&(na_Uart_1->np_uartrxdata))) {
170                return -17;
171          }
172 
173         // Call device driver to set the transaction size
174         notOk = SetTransactionSize(dma, transSize);
175         if (notOk) {
176             return -18;
177         }
178 
179         // Call device driver to set read/write DMA addresses
180         notOk = SetRWAddresses(dma, rAddr, wAddr);
181         if (notOk) {
182             return -19;
183         }
184 
185         // Set up source or destination devices if needed
186         if (rAddr==(int)&(na_Uart_1->np_uartrxdata) ||
187             wAddr==(int)&(na_Uart_1->np_uarttxdata)) {
188                 if (notOk) {
189                     return -20;
190                 }
191         }
192         if (rAddr==(int)&(na_PIO_1->np_piodata) ||
193             wAddr==(int)&(na_PIO_1->np_piodata)) {
194                 notOk = SetupPIO(pioDir);
195                 if (notOk) {
196                     return -21;
197                 }
198         }
199 
200 
201         // Call device driver to set the DMA transfer termination mode
202         if (termMode!=0) {
203             notOk = SetTerminationMode(dma, termMode);
204             if (notOk) {
205                 return -22;
206             }
207         }
208 
209         // If interrupt is enabled for the DMA, must register
210         // an ISR. The ISR will call the 'TermDma() and CheckDMA()' snippets
211         if (intEnable==1) {
212             nr_installuserisr(na_DMA_1_irq, DMAISR, (int)dma);
213             // Call device driver to set the DMA block or non-blocking (interrupt) execution mode
214             SetTransferInterrupt(dma, intEnable);
215         }
216 
217         // Save the DMA transfer info. so they can be executed by execDMA snippets later.
218         SaveDMAState(dma, dmaId);
219         SaveExtDeviceSettings(dmaId, uartEOPValue, uartDivisor, pioDir,
220                                 pioIntMask, pioEdgeCap);
221 
222         return dmaId;
223     }
```

C. Discussion

C1. Variation possibilities

The main source of variation in the *InitDMA* snippet comes from the snippet parameters. The different values supplied for these parameters every time the snippet is chosen into the test program results in various types of DMA transfers to be configured on the DMA. In particular, the outcome of variation is to configure DMA transfers,

- between different source and destination devices,
- of different data transfer amounts,
- transferring byte, halfword, or word data units per transaction,
- reading from and writing to addresses that increment across memory ranges,
- that terminate depending on the amount of data transferred or when a specific byte character has been transferred (allowing for streaming mode transfers),
- where transfer execution can be conducted in blocking or non-blocking (interruptible) modes, and
- diverse types of erroneous DMA transfer settings can be tested.

Additionally, different combinations of the *InitDMA* snippet can be chained together with other I/O device or similar DMA snippets to test the device sharing, resource allocation, and concurrent process executions capability of the SoC.

C2. Attributes

Size :

ANSI-C code : 223 total lines of code (LOC), 148 effective LOC (excluding comments/line breaks)

C3. Advantages and disadvantages

The main benefit of applying the *InitDMA* snippet is the large range of DMA transfer configurations afforded by its parameters. Such large variation allows for many different forms of transfers to be set up and conducted on the DMA device. Whilst large variation is useful to provide wider coverage of DMA testing, the downside is that it is sometimes difficult to test all permutations of test conditions

possible. The trade-off is to apply variation for the snippet in conjunction with some user-biasing on the parameters to invoke critical corner-case scenarios and discover interesting conditions for DMA transferring.

Using parameter biasing is also important to prevent the snippet from configuring DMA transfers that execute too many transactions. For example, overly long DMA transfers with blocking mode selected can result in bottlenecks during testing whereby the snippet takes up SoC resources and requires many simulation cycles to complete before other snippets can be invoked. Similarly, if DMA read and write addresses often selects non-sharing devices such as the UART or PIO, then many blocking loops may result waiting for these devices to be available again if they happen to be currently in use by other snippets. If so, the *InitDMA* snippet may have to wait for long periods. Such conditions do not add value to testing and are unnecessary.

It is worthwhile to point out that the *InitDMA* snippet is highly dependant on the *ExecDMA* snippet. The *InitDMA* snippet is responsible for configuring different DMA transfers only. Without the *ExecDMA* snippet, the actual data transactions are not executed. We designed the DMA testing functions separately into individual snippets because the set up procedure of the DMA is not simple. There are many checks that are needed for the various set up conditions and procedures to adhere to for performing the configuration. Separating the *InitDMA* and *ExecDMA* snippets independently also provides more diverse execution of DMA transfers. The transfers can be conducted at any given stage after the *InitDMA* snippet in either blocking or interruptible modes, and can be intermixed with other configurations of the DMA being queued up.

Finally, note that the launch of an *InitDMA* snippet is useless if the DMA device is not free to accept new configurations. To perform the *InitDMA* snippet, the DMA must be idle and have completed prior transferring. Therefore, it is important that the *InitDMA* snippet be strategically placed relative to other DMA snippets in the test program. The mechanisms to do this were described via algorithmic test generation in Chapters 4 and 6.

C4. Other considerations

There are no other limitations on how the *InitDMA* snippet can be used. In fact, the structure of the snippet can be easily modified to cater for transferring of new devices if the SoC system is enhanced. It is easy to identify where to add in the appropriate checks and settings for these devices within the snippet. This makes the snippet extensible and reusable.

APPENDIX D. SALVEM Random Test Generation

Appendix D focuses on automated test generation of software test programs for design verifications. The automated test generation uses randomisation to inject diversification into the types of tests created and the range of SoC application functions tested. The test generation method is employed to create test suites for verification of the Nios system-on-chip design, and to demonstrate manually directed coverage driven verification using our verification methodology.

D.1 Introduction

The development of the automated test generation method in this appendix was our first attempt at verifying the Nios system-on-chip (SoC) using the software application level verification methodology (SALVEM) devised in Chapter 3. Our test generator enables a diverse range of tests to be created in order to evaluate the effectiveness of SALVEM. Like other verification schemes, randomisation is the intuitive strategy employed for our test generator. Randomisation is often the initial technique that is applied for automating test creation because it is not difficult to implement, and is able to produce tests quickly.

In addition, randomised test generation can create many variations of tests that are usually 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. The major drawback with employing randomisation is that it is often 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. Randomised test generation must be partially guided, as is the case in the SALVEM test generator in this appendix.

In SALVEM, randomisation is highly applicable because the SALVEM software test creation process consists of several elements that can be varied effectively by randomisation. The opportunities for applying randomisation are available both during the test creation process and within the resultant test program itself. For example, randomisation can be applied to the types of snippets chosen into the test program or the parameters chosen for each snippet. Under certain randomisation conditions, test generation restrictions and user influences, our goal is to develop a stochastic test generator tool to automate SALVEM test creation; and provide many effective test cases for SoC verification.

The next section describes our automated test generation strategy, employing randomisation and other associated test creation elements. Section D.3 discusses issues with the random automation approach, and outlines benefits and shortcomings of our methods. This is followed by experiments and analysis concerning the practicality and effectiveness of the random test technique in Section D.4. Before concluding, Section D.5 describes a coverage driven verification case study using the test generation scheme of this appendix.

D.2 Automated random test generation for SALVEM

In this section, the SALVEM verification concepts from Chapter 3 are extended with automated test program generation. The resultant test generator tool will be developed and integrated within the SALVEM verification platform to create many different test programs for testing the Nios SoC. We begin with a description of the high level test generation procedure.

D.2.1 Overall test generation process

An overview of the random based test creating automation process is shown in Figure D.1.

Test configuration

The first stage of our test generation flow is to configure global settings for the overall test generation process, and the parameters that apply to every test program. For example, using randomisation and user guidance, the number of tests that will be created for the test suite is chosen. The sizes and types of external modules (and their internal set up) are selected in order to perform hardware configuration and prepare the SoC test environment for simulation. These randomised test creation decisions apply throughout the test generation and verification phase. For the test program, the randomised variables asserted for the creation of each test includes the number of *snippets* test building blocks (Section 3.4 Chapter 3), the miscellaneous memory and input/output (I/O) data used by snippets, and other parameters that are applied for the creation of the test.

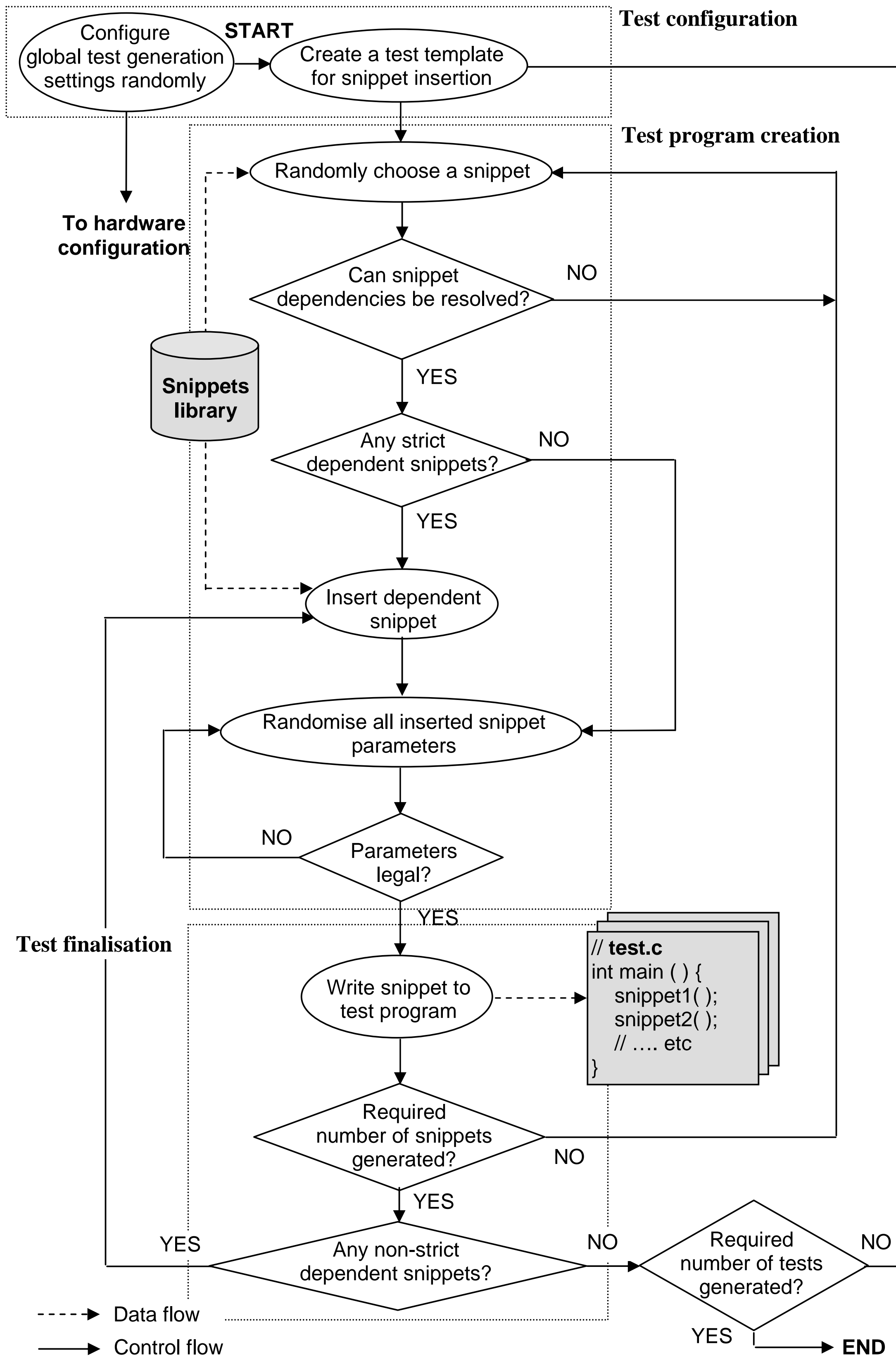


Figure D.1 Test generation flow

Test program creation

Once global test creation parameters are randomised and chosen, the software test creation process to create a test program begins. A test program template is provided based on the global settings. The template shall be populated with test program specific elements such as the test building block snippets and their configurations. The test creation flow iterates, randomly selecting various snippet combinations to compose the sequences of snippets within the test program.

During test generation, use of SoC resources is monitored. A simplistic model of the SoC was developed, and used by the test generator to mimic how the SoC would behave as if each snippet were executed on the actual SoC. Essentially, the test generator maintains a partial SoC state based on the history of snippets chosen. Each chosen snippet must be compatible with the current SoC state, satisfying various dependencies and conditions. For example, if the chosen snippet is dependant on another snippet, then the test generator generates that snippet before selecting another.

If all dependencies can be resolved by inserting dependant snippets into the test, and other snippet conditions are satisfied, the chosen snippet can then be composed into the test. Otherwise another snippet is randomly chosen again. These snippet dependencies and constraint conditions were described in Section 3.4 Chapter 3.

If the randomly chosen snippet is to be included in the test, parameters specific to that snippet are randomised according to various constraints. The parameters and constraints for each snippet were also listed in Section 3.4 Chapter 3 and Appendix C.3. Again, the SoC model maintained by the test generator is consulted to ensure parameter selections are legal.

Test finalisation

Once all configurations for the chosen snippet are established, the snippet function header is then written to the main function routine of the ANSI-C test program file (i.e. `test.c`). If the test program is still not filled with the required number of snippets randomly chosen previously, then the snippet selection cycle repeats again to add another snippet into the test.

Before creation of the test program completes, the test generator checks if any remaining non-strict dependent snippets (Section 3.4, Chapter 3) are required, and inserts them into the test if needed. Non-strict dependent snippets are snippets that do not need to be inserted immediately before or after a specific snippet, but must be inserted into the test at some point earlier or later to ensure the test is

executable. Finally, if another test program is to be generated for the test suite, the randomised test creation cycle is repeated again.

Test creation considerations

Whilst our goal is for all test generator choices to be fully randomised throughout the test program creation process, in fact, the test generator adopts a constrained-random-biased approach. In order to realise executable test programs, the types and range of selection choices that are available for various test generation variables must be restricted. For example, we must restrict the maximum number of snippets in a test so that the number of snippets randomly chosen into the test does not exceed the capacity of the SoC executable memory. Also, if a universal asynchronous receive transmit (UART) device snippet is selected to execute in blocking mode, the transfer parameters must be randomly chosen from a restricted range to ensure serial transfers do not run for overly long periods; taking over SoC resources and causing simulation to hang.

Internally, the aim of test generation and snippet constraint and dependency rules are to ensure a legal executable test program that can be simulated by the SoC is created, under a randomisation environment. Additionally, the random decisions undertaken by the test generator may also be explicitly influenced by external users to target certain corner cases. We discuss this aspect in greater detail in Section D.5 later.

Finally, note that the availability of a *lightweight* SoC model for dependency and constraint checking is also highly beneficial for validation of SoC operations conducted by snippets. The SoC model is considered lightweight because it does not implement the full functionality of the entire SoC design. Instead, it only imitates and keeps track of the SoC operations of devices which the snippets control. The SoC model acts as a reference model for snippet operations simulated on the SoC design. Whenever a snippet is selected for the test, the expected outcomes of operations from that snippet can be predicted by the SoC model, and compared against real snippet run-time outputs during actual simulation. Given the range of test creation elements that can be randomised, checking the correct outcome of testing is essential.

Randomisation possibilities

Before describing the verification system employing the test generator tool, we list the software test creation elements that can be varied by randomisation as follows:

- Types of snippets chosen from the snippets library.
- Snippet insertion position in the test program.
- Snippet parameter values.
- Size of each test (i.e. number of snippets in each test program).
- Number of tests to make up the test suite.
- Types of error checks and operations performed.
- Memory and I/O data creation (i.e. type and size of data used by snippets).

D.2.2 Implementation of automated random based verification system

The SALVEM verification system for automated test generation consists of three main areas of work: (i) snippets library development, (ii) test generator development, and (iii) test simulation environment.

Snippets library development

Developing the test generator concurrently and independent of the snippets library reduced test development time and allowed ongoing addition of new snippets whenever possible. The snippets library was discussed previously in Section 3.5 Chapter 3.

Test generation development

This sub-section describes in greater detail the implementation of the test generator which implements the flow of the test generation process shown in Figure D.1. We also discuss the practical issues and solutions to realise the automated random approach. In particular, our approach in adopting an object-orientated (OO) strategy to create all test generation elements including snippets composition and rules, and the partial state model of the SoC employed for test creation. The OO method enhances re-usability of the test generator tool for application to other SoCs, the details of which we shall discuss in the remainder of this sub-section.

Our test generator implements randomisation mechanisms to automatically create many different snippets based software test programs. The randomisation mechanisms employed by the test generator is facilitated by an underlying random number generation engine. The test generator pseudo algorithm (based on a C++ object orientated approach) is summarised in Figure D.2. It implements a sub flow of the test generator process shown in Figure D.1, after the initial global test configuration stage. The main test generation functions are highlighted in bold.

```
1     snippet_library s_lib = {InitDMA, ExecDMA,  … etc …};
2
3     Gen_Snippets_Test (constraint cons, dependency deps, bias biases, snippet_history s_hist) {
4
5         // Add a snippet into the test
6         snippet_type s_type = Select_Snippet(s_lib, biases);
7         while (Illegal(s_type, deps, s_hist))
8                 s_type = Select_Snippet(s_lib, biases);
9
10        // Instantiate snippet object and parameterise
11        snippet snip = s_type();
12        snip.Parameterise(biases);
13        while (Illegal(snip, cons))
14              snip.Parameterise(biases);
15
16        num_snippet++;
17        Add_to_Snippet_Sequence(snip);
18        Add_to_Snippet_History(snip, s_hist);
19        Update_SoC_State(snip);
20        // Add another snippet into the test
21        if (num_snippet < snippet_test_length)
22              Gen_Snippets_Test(cons, deps, biases, s_hist);
23    }
```

Figure D.2 Test generator pseudo code

Our pseudo test generator code begins by randomly selecting a snippet from the snippet library. Snippet dependency rules are checked to ensure the selected snippet comply, otherwise a new snippet is selected. At least one complying snippet in the snippet library is always available for selection. Next, the snippet object is created. Snippet parameter values are randomly chosen within the limitations enforced on their allowable range. These chosen values are then checked against parameter constraints. If any constraint is violated, parameters are re-selected. The snippet sequence in the main function of the test program is then updated with this new snippet. This test generation routine is called recursively until the desired snippet test length is attained.

The case whereby snippets or snippet parameter values require re-selection raises the issue, *what happens if there are no suitable snippet or parameter values available?* In our test generation set up, we prevent this scenario from occurring as a pre-condition when specifying dependencies or constraint

rules. This averts any deadlocks during test generation. At any stage during snippet composition, there will always be at least one snippet and parameter value appropriate for the test generator to select from.

During test generation, the history of the snippet sequence composed thus far, and the state of the SoC is monitored by our SoC model. This enables the test generator to identify which devices are in use at any stage. Dependency rules can then be checked, and snippets are also chosen based on available SoC resources that will be released by prior snippets in the test during actual test simulation. For example, a DMA snippet cannot initiate streaming UART to memory transfers unless previous UART snippets complete and release the UART device. The expected outcomes and resultant device state predicted by the SoC model are recorded into each snippet object for comparison to actual simulation behaviours during verification runs.

The SALVEM randomised test generator (and snippets library) is implemented using the Python scripting language and object-orientated (OO) techniques. The OO features employed reduces development time and effort, to facilitate fast development and prototyping of automated random testing. Under the OO approach, a class is defined for each snippet. Whenever a new snippet is selected into the test sequence, a snippet object is instantiated by the test generator based on the specifications from the snippets library. The test generator's snippet object is self-contained, and executes internal methods to randomly select parameters and self-checks against constraint rules. The snippet object also creates the ANSI-C function interface code for the main test program to call.

Snippet dependency and constraint rule specifications are also implemented as objects in the test generator. Similarly, the test generator implements the lightweight reference SoC model by defining OO classes for each SoC device. An object is instantiated to represent and maintain the state of each device. Collectively, these objects make up the overall SoC state that is monitored during test generation. As snippets are composed into the test program, each on-chip device object that will be manipulated by the snippet updates its internal set of resources. This ensures illegal conflicts amongst SoC resources caused by snippets are avoided during actual test simulation of SoC operations.

An OO implementation of the test generator is also highly beneficial for future expansion of the snippets library or SoC (and when the test generator is applied to other SoCs). Whenever a new snippet is devised, or other devices are integrated into the SoC, a new class can simply be developed and its object instantiated into the test generator, independent of existing test generator classes. Only the additional specifications of these snippets or devices, and their randomisation variables for test creation need to be implemented.

The randomisation engine employed by the test generator is based on the random number generator module of the implementation code. The various test creation variables available for random selections are all performed using random numbers to choose from their available range of options. With this approach, all randomise selections can be traced back to the seed value used by the random number generator module. Using this seed value, the same random decisions and test creation process can be reproduced if needed to recreate the test (e.g., to perform debugging activities). The seed value chosen for each test generation process is taken from the operating system's current time, thus ensuring that different randomise selections eventuate every time to produce diverse test programs.

The next section focuses on the verification system and the environment for facilitating SALVEM verification and automated test generation.

Test simulation environment

The test generator is supported by various tools and scripts to automate the entire SALVEM verification process. These tools automate the overall SALVEM verification flow by integrating (i) the snippets based test program generation and compilation to executable binaries, (ii) test simulation, and (iii) verification results checking and coverage measure. Figure D.3 shows the complete SALVEM verification architecture and flow, employing the Nios SoC as the target SoC design under test for verification research in this thesis.

The automating test creation components are captured within the large shaded region. In addition to the test generator and snippets library, the SAVLEM verification platform encapsulates the Nios software compilation tool-chain, the Verilog testbench, and miscellaneous Perl scripts to automatically generate batch runs of software tests.

In Figure D.3, the test generator generates the main test program file (test.c) containing the chosen sequence of snippet function calls. The test generator also generates randomised data (Verilog .dat) files to populate various SoC memories and supply the stream of I/O data to the SoC. The test.c file is then compiled and linked by the Nios compile tool-chain to produce the executable binary. To control and monitor test execution, the Verilog testbench.v file interacts with specific testbench administrative snippet functions. These functions execute concurrently with the main software test execution in the background.

The SALVEM verification system uses Synopsys VCS Verilog simulator to execute our software tests and simulate SoC behaviours. The Verilog SoC RTL design and testbench are compiled into a VCS simulation executable, independent from the test generation and build flow in the software domain.

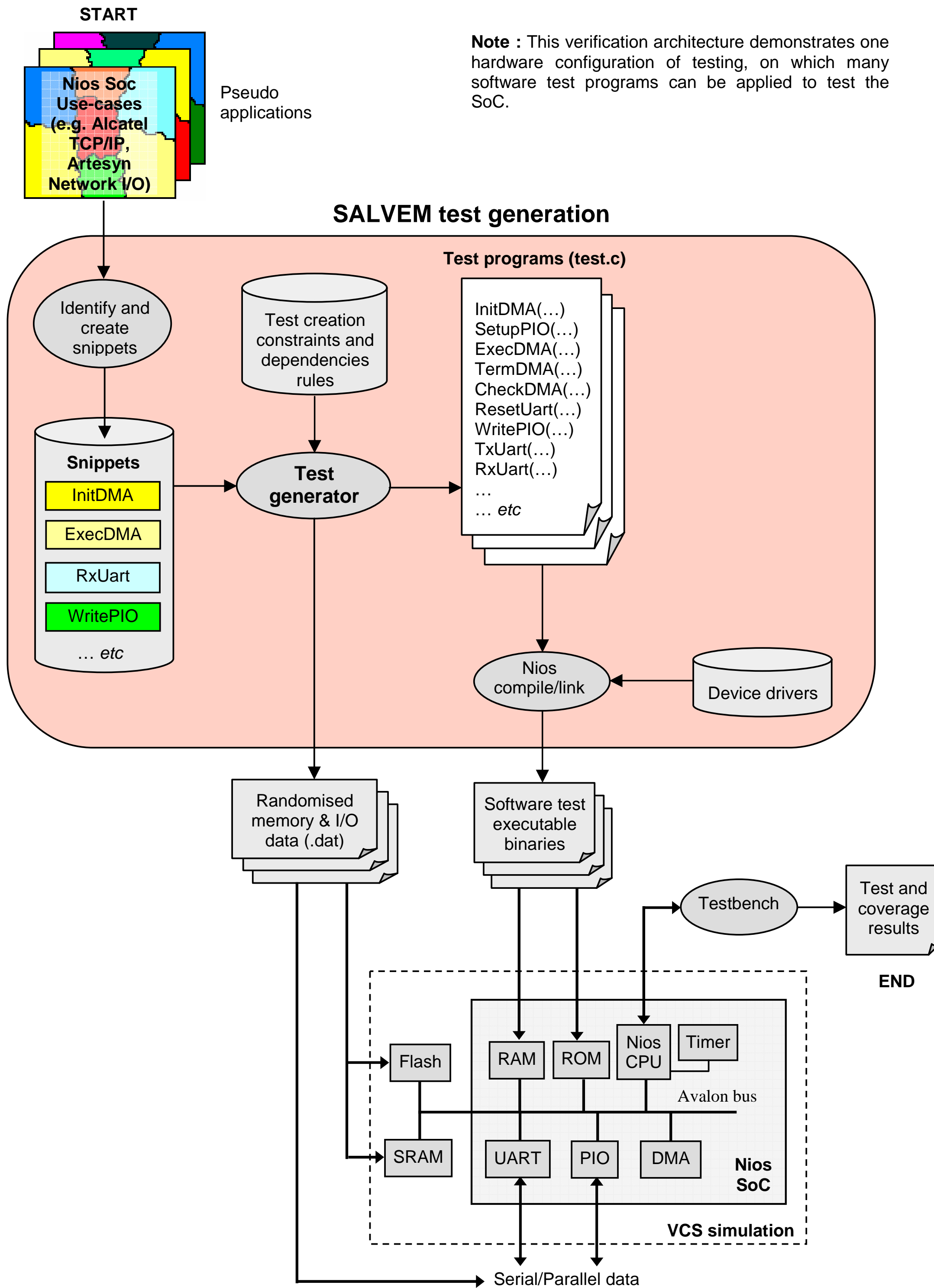


Figure D.3 SALVEM verification architecture and flow

To perform test simulation, the main software test.c binary file is loaded into the on-chip read-only memory (ROM). Immediately after SoC boot-up, the software application test executes from ROM calling the random sequences of snippet functions to initiate SoC device interactions.

During simulation, the snippet sequence is monitored and status messages are logged. Each snippet performs self-checking operations to determine the success or failure of SoC operations it invoked. When test simulation concludes, the results and SoC coverage achieved by the software test are collated. A new software test is then generated and run.

Developing the snippets and the test generator required the most effort for the SALVEM system. The SALVEM implementation with the random test generator required the equivalent of 9 working months of a single engineer. However, once implemented, SALVEM is able to automatically generate many varieties of tests to verify the SoC; including unplanned or unknown corner cases. The effort to manually create directed test cases and achieve the same level of coverage would still exceed the SALVEM development effort.

Currently, the implemented verification system is limited to single processor core SoCs. For other multiprocessor SoCs, more complex multitasking snippets will be required. The verification and coverage effectiveness of the SALVEM prototype is only limited by our set of snippets and the randomised test generator. We discuss this issue in Section D.5, elaborating on how we can overcome this shortcoming and to enhance verification further with other algorithmic test creation methods (Chapter 4).

D.3 Randomisation considerations and test generation issues

This section considers various issues associated with random test generation in SALVEM. We begin by discussing practical considerations that must be accounted for with randomisation, before summarising the benefits and shortcomings of such test generations.

In randomisation, the range of possible test creation selection choices or values must be restricted to only acceptable ranges that yield executable test programs. Identifying these restrictions is not straightforward, requiring in-depth analysis of the design and consultation with designers. In some cases, these selection ranges cannot be determined before-hand, requiring preliminary test runs to narrow down the legally allowed values.

For example, in order to determine the divisor register ranges of the UART, one cannot simply rely on the datasheet information. The divisor value that will be randomly chosen affects the UART’s baud rate and transfer speeds. The I/O transferring rate must consider the simulation speed and capabilities of the verification system, and must be configured to be compatible within both the UART and the

verification system's operating range. Therefore, preliminary test simulations are needed to find proper divisor values to randomise from. Despite the effort needed to calibrate these randomisation ranges, it is an important phase in the implementation of automated random test generators, even for SALVEM.

Besides identifying legal randomisation values, there is often a trade-off associated with the size of these randomisation values (i.e. the number of available selection choices), and the diversity of test programs created (and thus the extent of SoC functions tested). Whilst using the maximal range of selection choices produces greater variety of tests, larger selection ranges are harder to manage. The randomisation process is more complex having to accommodate many more possible values, and at the same time, ensure these larger combinations of random choices still provide legal and executable tests.

On the other hand, using a smaller selection of values for randomisation facilitates faster and less complex test creation process, with less burden on the test generator. However, the variety of tests generated will be low, and previously available selection choices will be eliminated from test creation; possibly preventing crucial SoC functions or corner cases from being verified. In addition, the number of repeatedly tested SoC functions increases.

The desired strategy with randomisation is that neither too large nor small selection ranges should be used. Furthermore, as described in Section D.5, the types and values of selection choices available should be managed from a user verification perspective. For example, steering random decisions towards important values such as memory boundaries when direct memory access (DMA) device snippets configure data transfer operations.

Another criticism of using randomisation in SALVEM is that there is no assurance whether the random selections made will consistently produce effective test programs. Randomisation can be considered a greedy approach, whereby the effectiveness of testing improves only if the number of tests generated increases. But how many random tests are sufficient? For larger designs to verify, this is an ineffective strategy. In Chapter 4, we describe how to expand upon the randomisation approach with algorithmic methods to provide more effective and efficient test generations.

The design of our lightweight SoC reference model for use during test generation poses other questions as well. Our SoC model is maintained by the test generation during the test creation process, therefore it should only impose minimal requirements so as not to slow down the test creation process. Given this, it is not always straightforward to identify what SoC design elements must be monitored whilst ensuring that sufficient SoC state information and predicted behavioural results from snippet

operations are provided. Otherwise, parts of the test creation process (i.e. constraint and dependency checks) and results checking during simulation cannot be carried out.

For our test generator, we monitored the snippets creation history and critical SoC state registers only. This was sufficient for our current test creation needs. However, other SoC operations are difficult to keep track of, especially runtime characteristics such as interrupt occurrences, priorities or handling. Whilst an independent and full SoC reference model can be employed, our partial SoC model was adequate. Also, developing an SoC model was not the focus of this appendix or the research thesis.

Finally, integrating the snippet test program constraint solver and dependency checker can also be problematic with a randomised test generator. The random test creation decisions are often in conflict with constraint and dependency rules, and performing post-selection constraint and dependency checks is inefficient. Also, illegal random selections only become apparent after the SoC model is consulted. If so, the random test creation decisions need to be undertaken again. Next, we discuss benefits and shortcomings of our method in relation to SALVEM.

Advantages

Using randomisation for test generation provides a number of benefits for SALVEM. First, the randomised test generator is relatively easy to implement and can generate many tests quickly. Applying the SALVEM random test generator for other SoCs or enhancing the verification system with new snippets is also not difficult. For example, the test generator can be easily configured to output the test program in other test program language formats other than ANSI-C if desired. The OO implementation of the test generator and graph based representation of the test case internally (Figure 3.9 Chapter 3) makes it straightforward for the test generator to satisfy the syntax or semantics of the target SoC operating software language. Snippets are simply designated as nodes and vertices connect snippets together to form snippet sequences as graphs to represent the test program conceptually.

Second, randomised tests can be applied to the SoC to cover a significant portion of the design, achieving up to 85% coverage easily [Ber03]. This provides a solid basis from which to conduct further verification and attain the remaining coverage using more advanced algorithmic test generation techniques. From a test case perspective, the numerous combinations and lengths of snippet sequences that can be created by randomisation is extremely valuable. In order to exercise corner cases and difficult to realise scenarios, the complex series of SoC behavioural states can only be induced if extensive and highly intermixed SoC operations are conducted, which is the goal of randomising snippet sequences.

Disadvantages

The downside with our SALVEM test generator is that it doesn't employ a true randomisation engine. It relies on the inbuilt random number module, and we use different randomisation seed values based on the current operating system time stamp when the test generator is invoked. Such a *pseudo* random set up requires the test generator to create tests constantly throughout as many different periods as possible. This is so that diverse seed values are applied to enforce varied pseudo random selections for our tests. The randomisation method of our test generator is limited by the range of seed values that are employed. Despite this, the randomisation scheme and range of tests created are still sufficient for our prototyping purposes in SALVEM verification research in this appendix.

Another drawback concerns the extent of randomisation that is permissible for SALVEM test creation. In our approach, the capability and usefulness of randomisation is largely dependent on the possible test creation elements that can be put under randomisation. The main test creation variables available were described in Section D.2.1. Whilst these test creation variables are effective for producing diverse tests, other avenues for enhancing random test generation further would be to provide more test creation options to randomise.

D.4 Experiments, results and discussions

D.4.1 Experimental goals and configuration

The randomised automated test generator described in Section D.2 was employed to generate SALVEM test suites to verify the Nios SoC (Appendix A). The experimental goals in this section are to demonstrate the feasibility and effectiveness of SALVEM with automated test generation; for verifying a typical industrial SoC design. The experimental results will allow for identification of deficiencies in our current verification set up, and enable further enhancements of the SALVEM methodology as described in Chapters 4 to 7 of this thesis.

Our experiments focus on the design coverage attained by the randomly generated SALVEM tests. We employ standardised line, toggle and conditional coverage measures that are used for simulation based design verification. These coverage methods are conducive for designs described by hardware behavioural code, which captures the target Nios SoC. But verification-wise, in Chapter 7, we devised functional coverage methods more suited for SALVEM.

Regardless, line, toggle and conditional coverage are still considered highly useful for design verifications. In many industrial verification projects, attaining high percentages for these coverage measures is usually a prerequisite for conducting further testing and analysis. In addition to coverage, we also recorded other experimental measures such as test execution times to assess the performance of SALVEM.

D.4.2 Experimental procedure

The SALVEM tests were generated and executed on the Nios SoC by the Synopsys VCS simulator using a Linux Redhat 7 platform, powered by an Intel Pentium 4 3GHz CPU and 2GB RAM. The verification system from Figure D.1 and Figure D.3 was implemented but only a subset of snippets from the snippets library in Section 3.5 Chapter 3 were used as the test building blocks for our SALVEM test programs. Specifically, the snippets employed were InitDMA, ExecDMA, TermDMA, CheckDMA, SetupPIO, WritePIO, ResetUart, RxUart, TxUart, and RxTxUart.

We used this basic set of snippets only because we wanted to focus on the SoC devices that are most likely to participate in system wide application functions of the SoC first. Specifically, these snippets are sufficient to explore the practicality of employing SALVEM with automated test generation for large-scale mass production of verification tests suites; before refining the snippets library and test generator later to fully stress test the Nios SoC (in Chapters 4 and 6).

D.4.3 Experimental results and discussion

Coverage results

The test generator was used to create a test suite of randomised SALVEM test programs for verifying the Nios SoC. In total, 100 tests and 21,969 snippets were simulated. Line, toggle and conditional coverage data were collected and the results are shown in Table D.1. Even with a subset of snippets, the SALVEM software tests achieved reasonable coverage results implying a sufficient portion of SoC behaviours was verified.

The number of snippets executed is also broken down for each SoC device. Snippets for the CPU, memories and DMA were the highest because many snippets use at least one of these devices when executing an application function. Table D.1 also shows the individual coverage attained for each

device. The DMA attained the best overall coverage as their snippets were most comprehensive. The full functionality of the DMA were analysed, and snippets were developed to cover their range of possible applications. The additional complexities in the CPU, UART, memories and Nios Avalon data buses imply more extensive snippets for these devices could improve coverage of the SoC.

Table D.1 Coverage and snippets usage results

	SoC	CPU	Memories	DMA	UART	PIO
Line coverage (%)	84.6	97.9	67.4	100	94.0	89.6
Toggle coverage (%)	76.1	83.3	57.4	74.8	71.0	59.7
Conditional coverage (%)	66.5	66.8	26.4	94.0	72.6	60.0
Number of snippets	21,969	19,477	12,214	12,051	9,317	5,176

Despite composing many snippets that may exercise devices such as the CPU or memories, many of these snippets were not specifically targeted for these devices. For example, the DMA or UART snippets are counted toward exercising the CPU, buses, and memories because these devices are needed for DMA or UART operations. However, they contribute primarily to DMA or UART verification only. To verify the CPU, buses, or memories fully, detailed analysis of these devices and specific snippets to exercise their functions would be provided by CPU or memory snippets. Nevertheless, the results here indicate an important characteristic of SALVEM testing. That is, the quality of the snippet in terms of the types and range of SoC design functions exercised outweighs the quantity of the snippets employed in our test programs.

Figure D.4 plots the SoC coverage against the number of snippets (and implicitly, tests). We plot against number of snippets instead of the number of tests because under the randomisation method, the number of snippets in each test is random each time. The size of each test is only constrained by the maximum number of snippets that can be held by executable SoC memory. Overall, the average number of snippets in a test was approximately 220, which will initiate various intricate SoC processes but do not require overly long simulation times. The number of snippets is plotted on a logarithmic scale to show coverage improvement as more snippets are executed and increased rapidly. Note that reasonable SoC coverage is attained immediately using a small number of snippets.

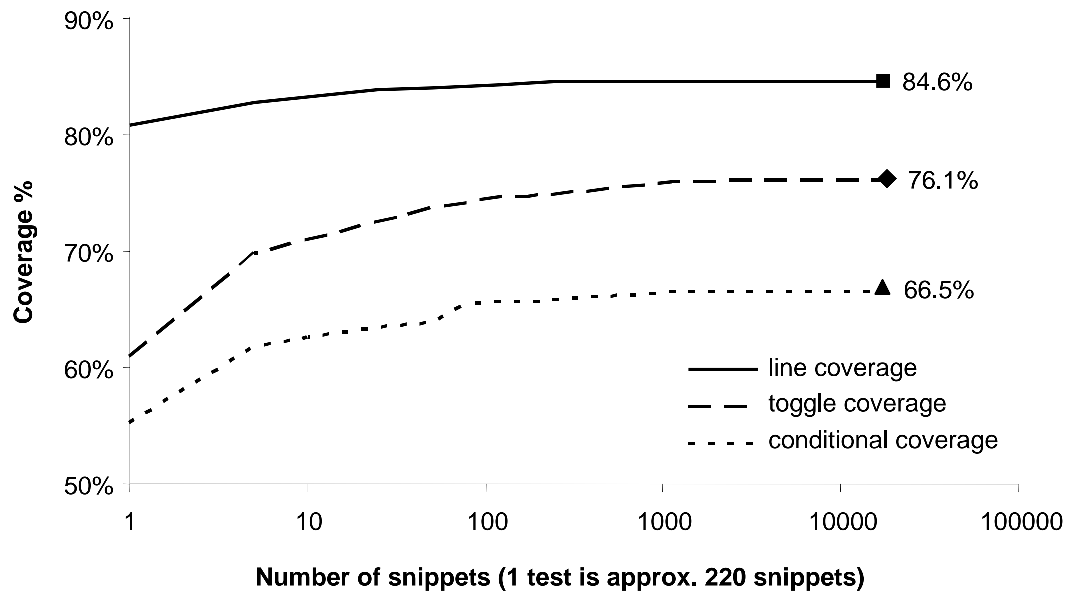


Figure D.4 Coverage progress versus number of snippets

The immediate coverage attained by a single snippet test is due to SoC initialisation. For every software snippet test, the SoC boot-up process performs a sequence of device reset, configuration and self-checking operations. These initialisation tasks perform various SoC operations before snippet test applications are run. Initialisation code provides an existing level of *default* coverage. We executed a software test without any snippets, and confirmed with Figure D.4 the default coverage to be 80.7%, 60.9% and 54.9% for line, toggle and conditional coverage respectively. This implies our snippets test suite contributed an additional 3.9% line, 15.2% toggle and 11.6% conditional SoC coverage from the default initial coverage attainment.

Figure D.4 shows SoC coverage increases marginally after executing an initial set of snippets. Line coverage begins to level out after 30 snippets whilst toggle and conditional coverage do not increase significantly after 200 snippets. The satisfaction criteria for line coverage require each design code statement to be exercised at least once only. Hence, maximum achievable line coverage can be obtained using a simple sequence of snippets. Toggle and conditional coverage involves simulating many execution paths and states. Hence, a larger set of snippets and a wider range of the types of snippets in the snippets library are required.

Analysing further, the snippets employed are unable to exercise the complete range of Nios SoC functions. The aim of the automated test generator and this experiment was to establish a working system for prototyping and feasibility analysis as early as possible. At that stage during our research, given restricted engineering resources, a decision was made to employ snippets for the core set of SoC

functionalities and specific devices only. The initial snippet library would not model overly complicated or uncommon application test scenarios.

In particular, our snippets only handle a subset of applications for the UART and memory devices. The snippet library accounts for approximately 75% and 60% of the complete UART and memory functionalities only. The current snippet library does not exhaustively test all possible types of serial transfers (e.g., full duplex transfers) or Avalon bus transactions between memory devices. Furthermore, this basic snippet library only targets a small number of SoC exceptions. For example, the current UART snippets do not implement various frame error, parity error checking or break character transmissions.

Verifying all error conditions on the SoC is necessary for full coverage. However, developing snippets to test error conditions requires much effort. From our personal experiences, during verification of industrial SoCs such as Freescale networking SoCs, many error conditions were extremely complicated to verify. It was difficult to develop application test code to invoke the SoC into an exception state and recover. Similarly, the Nios SoC consists of many error conditions that must be tested. Designing snippets to initiate erroneous states and recover correctly so test program execution can continue is essential. However, the Nios SoC does not provide explicit or robust error recovery mechanisms. Verifying all SoC error scenarios would require extensive analysis of the SoC and complex snippets. Snippet development for many SoC exceptions was postponed until Chapter 4 in our research thesis. Complete error testing is beyond the scope of the SALVEM snippet library in this appendix.

Whilst our experiments only employed a subset of the eventual full snippets library, the work carried out and subsequent results provided valuable insight into the usage and characteristics of snippets and large scale SALVEM tests execution. This formed the basis for further development of other snippets to enhance the snippets library, and to conduct more effective SALVEM verification on the Nios SoC in Chapter 4. In Chapters 4 and 6, we conduct randomised SALVEM testing again, but with the full set of snippets from the snippets library, showing improved results compared to our experiments here.

Performance results

On average, the CPU time to execute a typical software test of 220 randomised snippets required 856 CPU seconds. When collecting line and toggle coverage, execution time increased to 1,550 and 1,571 seconds respectively. Conditional coverage however, required 10,185 seconds. These test execution

speeds are comparable to previous Freescale SoC verification projects in industry. Without coverage measurement, a typical test suite of many test programs and sufficient snippets length can be run efficiently on the RTL design.

D.4.4 Uncovering design errors

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. From test debugging, we were able to recreate the test failure condition from each of the tests. In addition to the SoC design, we examined the test program itself, the simulator, and the SALVEM verification system to ensure the failure wasn't due to other factors; and that the error was solely from the design.

Essentially, the design error occurs in the UART module, and is an incorrect configuration register declaration and incorrect register assignment of its control values. The erroneous code and the correctly intended design code are shown in Figure D.5.

Verilog design file: UART _1.v (incorrect original design code)

```
In module Uart_1_regs :

800        reg    [9: 0] control_reg;  // UART control register declaration
…
…  and later in the design code  when the control register is used  …
…
873        else if (control_wr_strobe)
874            control_reg <= writedata[9 : 0];
```

Verilog design file: UART _1.v (corrected design code)

```
In module Uart_1_regs :

800        reg    [12: 0] control_reg;  // UART control register declaration
…
…  and later in the design code  when the control register is used  …
…
873        else if (control_wr_strobe)
874            control_reg <= writedata[12 : 0];
```

Figure D.5 Design error in the Nios SoC

This error is likely due to the designer incorrectly interpreting the register's width specification or miscommunication of updated specifications between designers. Hence, resulting in the register size being mistakenly truncated by 3 bits. There are 13 control bits in the UART control registers that manage serial transferring in the UART device. The subsequent effect of the register width truncation is that certain error control bits and the end-of-packet transfer termination triggers could not be configured for serial transfers. Whilst the mechanisms to handle these error and end-of-packet

functionalities were still designed into the UART module, this design bug prevented the UART from using these functionalities as they were not configured properly for usage.

This error was detected by our UART snippets that were to perform serial transfers terminated by a specific end-of-packet byte character. Because the end-of-packet termination could not be configured as expected by the snippet, the transfer could not terminate and a greater number of data than expected were transferred. Hence, test failure was triggered when the number of data bytes processed by the UART was not correct; and also, the last byte transferred did not match the end-of-packet character as required.

We identified this design bug and informed Altera Inc. and they acknowledged this was an actual error which would be rectified in their next release of the Nios SoC. The error support request logged by us and subsequent correspondence are shown in Figure D.6. Figure D.6 also describes other issues with the Nios SOC uncovered by our SALVEM verifications. The detection of this design bug reinforces the practicality and usefulness of our SALVEM method by employing tests that exercise a range of SoC functions commonly requested by typical applications.

Furthermore, another two design bugs involving interrupt priority detection and handling were also uncovered by our SALVEM test programs. When a series of snippets that invoke basic interrupt based operations are composed into a test, it was found that the order of interrupt handling when these operations concluded were not processed correctly. In some cases, the interrupts were not even handled at all, and the simulation would suspend. Common examples include five or more sequential DMA, UART, or PIO snippets that employ interrupts.

Usually, the release of an SoC implies the design would have undergone extensive testing beforehand. Despite the Nios SoC being a certified design released by Altera Inc, the goal of any verification is to uncover bugs. Therefore, while we did not expect to detect any design bugs, the fact that our experiments was able to uncover design bugs using simple randomisation to automate test generation shows SALVEM verification as highly promising, and can be deployed for testing other real-life SoC designs. In verification and test, it is widely accepted that any test method which is unable to detect bugs does not imply the absence of bugs in the design, it is simply that such bugs have not been uncovered. Hence, as a verification methodology, SALVEM’s goal to detect SoC design errors is proven.

Service Request Report for SR 10483248

Service Request Detail

Request No: 10483248 **Status:** Closed
Date Opened (PDT): 1/6/2005 04:33 PM **Date Closed (PDT):** 1/20/2005 03:52 PM
Inquiry Type: Product Question
Device Family: **Device:**
Request Title: Issues with Nios SopC Generated files
Steps to Reproduce / Description: I am using Nios SOPC builder version 4.00 and have encountered a few issues.

1.
The generated Uart verilog file does not include the ieop functionality even though the 'use end-of-packet' bit is set in soc.ptf file.

In the Uart verilog file, (I've called the Uart device Uart_1), in module definition 'Uart_1_regs', register 'control_reg' is only 10 bits ([9:0]), it should be 13 bits ([12:0]) to include up to the ieop bit.

reg declaration should be changed from,
reg [9:0] control_reg
to
reg [12:0] control_reg

and in the following always block, writedata should be changed to,

```
always @(posedge clk or negedge reset_n)
  begin
    if (reset_n == 0)
      control_reg <= 0;
    else if (control_wr_strobe)
      control_reg <= writedata[12 : 0];
    // not 'control_reg <= writedata[9 : 0];'
  end
```

2.
In soc.v, in module definition 'testbench', the module instantiation of the Ext_Flash requires the port vector width of 'Ext_Shared_Bus_data' to be specified for '.data' port. Otherwise, simulation with synoposys VCS hangs when trying to read/write to flash.

```
Ext_Flash the_Ext_Flash
  (
    .address  (Ext_Shared_Bus_address),
    .data     (Ext_Shared_Bus_data[7:0]),
    .read_n   (Ext_Shared_Bus_readn),
    .select_n (select_n_to_the_Ext_Flash),
    .write_n  (Ext_Shared_Bus_writen)
  );
```

The instance connection for the data port requires the vector bit width of Ext_Shared_Bus_data to be specified, otherwise, simulation with synopsys VCS hangs.

3.
When using the DMA to transfer data from UART to a mem device, and setting end-of-packet write termination (ween bit set), the transfer terminates due

Service Request Detail Page 2 of 2

to an EOP encountered even though the UART is the read device, and the mem should not be able to invoke the ween termmintion. Is this correct or intended behaviour? This could/have caused incorrect length termination because the ween terminated the transfer earlier.

thanks,
Adriel
...

Error Message:

Updates 1 - 2 of 2

Date Created (PDT)	Type	Note
1/10/2005 03:50 PM	To Customer	Hi Adriel, If you have not done so, I would suggest to use the latest version of the SOPC Builder, version 4.2. These issues have been resolved in the newest version of the SOPC Builder. Regards, Marlon Mendoza
1/7/2005 04:34 PM	To Customer	Hi Adriel, I will go ahead and look into this and get back to you as soon as soon as a resolutionis made. Regards, Marlon Mendoza

Attachments 1 - 1 of 1

Date Attached (PDT)	Attachment Name	Attachment Type
1/6/2005 04:38:08 PM	10483248_ACHENG	txt

Figure D.6 Nios SoC bug report correspondence with Altera Inc.

D.4.5 Summary

Our coverage results demonstrate SAVLEM is feasible and effective for system verification of SoCs, including uncovering actual design bugs. To improve coverage results further and enhance randomised SALVEM test generation, user directed biasing techniques could be used to focus on specific SoC devices and corner cases. Whilst an obvious and expected enhancement to our SALVEM system involves identification of other SoC applications and development of new snippets, we feel the current subset of snippets can also be further utilised by focusing on these corner cases. The use of user directed feedback and biasing is described in the next section.

D.5 Coverage directed test generation manually – a case study

This section describes a coverage directed SALVEM verification process that is conducted manually. Our goal is to evaluate the possibility and effectiveness of adopting a *coverage driven verification* (by feedback) flow into SALVEM. The manual coverage feedback tasks are carried out in conjunction with the test generator from Section D.2. Using a coverage directed method, we conduct a case study aiming to enhance Nios SoC verification coverage. The case study forms the basis from which automated coverage driven verification for SALVEM is developed in Chapter 4.

Another goal of the case study is to improve upon the randomised verification conducted in Section D.4. We demonstrate that further coverage improvements are possible with some fine-tuning during verification, despite using a subset of the snippets library.

Figure D.7 shows the flow adopted for our coverage directed process. During test generation and execution, coverage and other test execution statistics are collected to characterise the types of test programs generated; and identify what SoC devices and functionalities were insufficiently exercised. Using this information, we direct the test generator to create tests for previously untested functions and improve coverage.

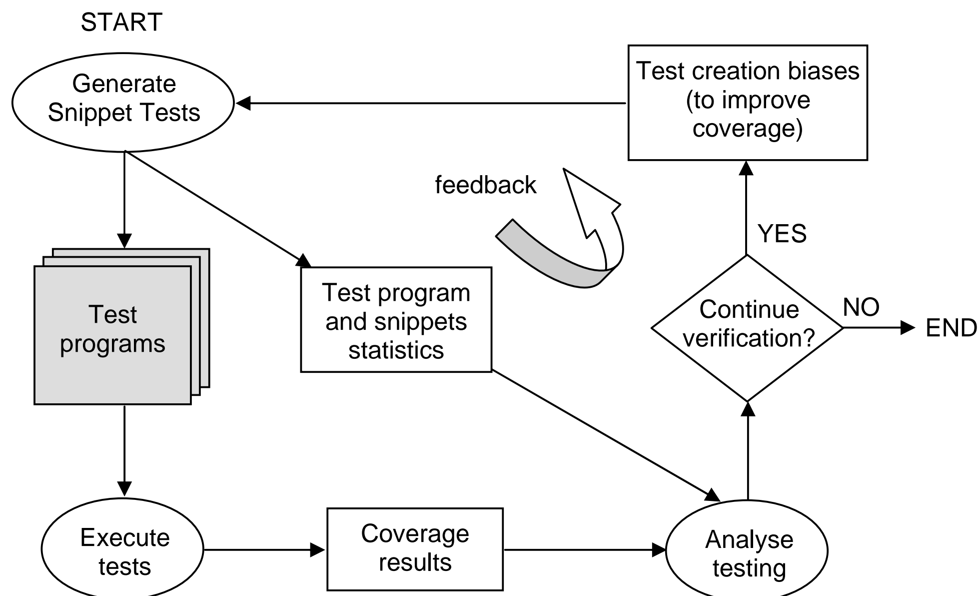


Figure D.7 Coverage directed verification flow

The primary mechanism by which we employ coverage directed verification is manual biasing. Biasing is a form of user control in addition to existing test generator input such as selecting the types of test creation variables to randomise or restricting the range of randomisation. With biasing, a test

generator can be influenced to perform certain forms of test configuration or select particular test creation choices with greater likelihood. Even though randomisation is still employed, the random decisions will lean towards these specified bias settings. Biasing enables a user to exert some control over the randomisation process during test generation. It provides the balance between manually devising all test creation components to generate a test, and using randomisation to fully automate the test generation process.

In our case, the SALVEM test generator allows external user influences to direct verification toward certain SoC devices and important test scenarios. In particular, test generator biases enhance the generation likelihood of certain snippet sequences and parameter values to be composed into our randomised test programs. Biasing generates new tests to target functional corner cases

SALVEM implements two kinds of biasing, *weight* and *range* biases. A weight bias assigns relative probabilities to a particular snippet or parameter choice. The choice with the higher weight value is more likely to be selected. For example, weights can be assigned to particular snippets to influence the snippets chosen in the test sequence. Similarly, weights can be used for manipulating the likelihood of byte, halfword, or word transaction sizes in DMA transfers. Weight biases are useful for constrained discrete choices.

Other test generator choices such as UART end-of-packet characters or DMA transfer lengths span a larger range to select from; and each value cannot be specified with a weight. Instead, a range bias is used to specify a sub-range of values. The range bias is specified using minimum and maximum selection values that confine this sub-range. For example, a range bias can specify DMA transfer lengths towards the maximal memory block or segment sizes. The test generator will then select values from these bias sub-ranges with greater probability than other values. Such biasing mechanisms enable testing of particular application scenarios to focus on specific SoC devices and operations. Armed with these test generation biasing controls, in the next section, we demonstrate SALVEM when it is adopted into a coverage feedback verification flow.

D.5.1 Experiments for manual coverage directed verification

Experimental goals and configuration

The experiments conducted in this section are to prove coverage driven testing in SALVEM, and improve upon the verification from Section D.4. Based upon the experimental setup and snippets from

Section D.4, we shall exhaust the usage of these snippets to demonstrate that SALVEM verification can be improved using coverage feedback; and also, show the importance and need for further snippets to aid SALVEM verification.

For experiments in this section, multiple verification runs are conducted using coverage and test information from previous verification to direct new generation of tests. The verification statistics and coverage information that are recorded are manually examined to analyse for deficiencies in our test programs. Based on this analysis, once the types of test cases desired are identified, the test generator is configured to realise these test cases.

Compared to Section D.4, the test generation and verification process here requires additional pre-configuration. Specifically, relative weighting values are specified for snippets and their parameter values to control their likelihood of usage. Other biasing can also be applied, for example, error checking set up, number of snippets, snippet sequence combinations, etc. Currently, these external influences are all statically defined and apply throughout each test generation process. Note that not all test creation choices need to be externally influenced. If no biasing is specified for any test generation parameter, then the selection process for those parameters reverts back to full randomisation as before.

Despite conducting multiple test runs, in order to attain fair improvement results over the verification in Section D.4, we configure our verification to execute less tests and snippets overall. Otherwise, it can be argued that our improvements here are due to executing more tests and functionalities. Achieving enhanced verification results with fewer tests also shows efficiency in our approach. Therefore, we configure our test generator to create test suites of 25 tests and software test programs with approximately 125 snippets each.

Another reason for this configuration is because our experiences from Section D.4 indicated that the number of snippets per tests caused simulation with conditional coverage to be quite slow. Given that we will be conducting multiple test generation and verification runs based on prior testing, long test executions are undesirable. We require faster test generate, simulate, and coverage test analysis turn-around times, so test execution does not cause bottlenecks. Hence, the number of tests and snippets composed into tests are reduced.

Initial test generation results and analysis

We applied SALVEM verification on the same platform as Section D.4. On average, a typical test generation and execution required 0.3 and 642 CPU seconds respectively. When collecting coverage information however, execution times increased to 1221, 1321 and 7605 seconds for line, toggle and conditional coverage. This is an improvement from Section D.4; even with coverage measurement, a comprehensive test suite can be created and run within acceptable time frames.

To initiate the SALVEM coverage feedback verification flow, the randomised test generator was used to generate and execute an initial test suite on the Nios SoC. This is followed by an analysis of the coverage attained from this initial verification. Table D.2 shows the initial and analysed coverage statistics. The analysed coverage is considered the true coverage result, and is used as the basis for evaluating against enhanced coverage arising from sequent coverage driven verification. It is obtained by examining coverage data against the SoC design for *dead code* and other un-testable error conditions.

For example, unused timer peripherals and several redundant Nios CPU arithmetic units cannot be exercised in our current SoC hardware verification set up, and are considered dead code. The Nios SoC was intended for many general applications and is applicable for various usages. Hence, many re-configurable features exist in the SoC, and not all design blocks or functions can be used in the system. Furthermore, some error conditions cannot be tested because the Nios SoC cannot recover from certain illegal operations. For example, the SoC enters a deadlock state when executing DMA transfers between UARTs using data unit sizes larger than the UART ports – although this may be classified as a design bug, and is under investigation by Altera Inc. as part of our design bug report from Section D.4.4.

Table D.2 Initial and analysed coverage results for non-biased test suite

Initial coverage	**SoC**	**CPU**	**Memories**	**DMA**	**UART**	**PIO**
Line coverage (%)	84.6	97.9	67.4	100	94.0	89.6
Toggle coverage (%)	76.1	83.3	57.4	74.8	71.0	59.7
Conditional coverage (%)	66.5	66.8	26.4	94.0	72.6	60.0
Analysed coverage	**SoC**	**CPU**	**Memories**	**DMA**	**UART**	**PIO**
Line coverage (%)	91.0	98.0	68.9	100	98.2	89.6
Toggle coverage (%)	78.0	85.0	58.1	77.1	74.1	63.6
Conditional coverage (%)	66.5	66.8	26.4	94.0	72.6	60.0

Testing and coverage analysis for feedback verification

After accounting for dead code and error conditions, testing and coverage of the Nios SoC can then be conducted using the feedback verification prescribed in Figure D.7. Figure D.8 and Figure D.9 show a subset of the test generation statistics collated from the test suite of Table D.2. The histogram in Figure D.8 identifies which particular snippets were deficient in the test programs. For example, compared to other snippets, DMA and WritePIO snippets were not generated and executed as often. The histogram shows more snippets initialised the parallel input/output (PIO) device instead of using it. Indeed, the coverage results for the PIO support this. Additional analysis of the PIO device and what operations were logged during testing shows PIO port accesses and I/O operations were lacking. Therefore, additional WritePIO snippets would improve the current PIO coverage of the unbiased test suite.

Despite low DMA snippets and inadequate parameterised test configurations (Figure D.9), the unbiased test suite achieved full line coverage for the DMA device (Table D.2). This result is misleading because line coverage satisfaction criteria require statements in the design code be exercised once only. The structure of the DMA hardware design code requires only a few transfer scenarios to exercise each line of the DMA device. Lower DMA toggle and conditional coverage confirms other transfers scenarios were untested and the DMA was insufficiently tested. Similarly, higher PIO and UART line coverage may not imply these devices were sufficiently verified. In general, line coverage is always greater than toggle or conditional coverage. However, line coverage alone is not sufficient and other metrics such as toggle and conditional coverage must be considered as well.

Figure D.9 shows the DMA snippet parameter selections. Examining this histogram, the UART, PIO and ROM devices were chosen insufficiently as DMA source and destination transfer devices. Subsequently, this resulted in unsatisfactory coverage for these devices (Table D.2). The ROM was selected once, partly because the ROM can only be chosen by the DMA as a read-only source device. Parameter selections for DMA execution modes also favours non-interrupt (blocking) mode heavily, suggesting insufficient simultaneous SoC operations were executed.

Similarly, for the UART, the uncovered serial transfer functionalities are predominately due to error conditions that were not exercised by existing UART snippets. New or improvements in UART snippets, rather than feedback verification would be more valuable for verification purposes. In fact, our revised results as shown later in Table D.4 show no coverage improvement at all from the feedback processes.

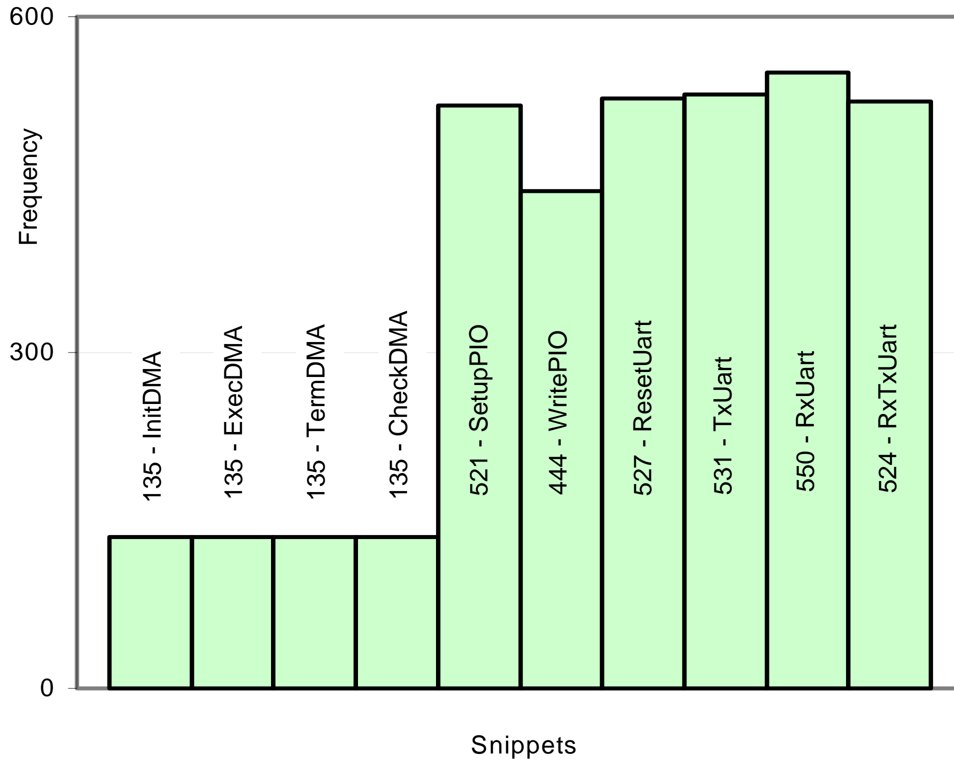


Figure D.8 Snippets usage

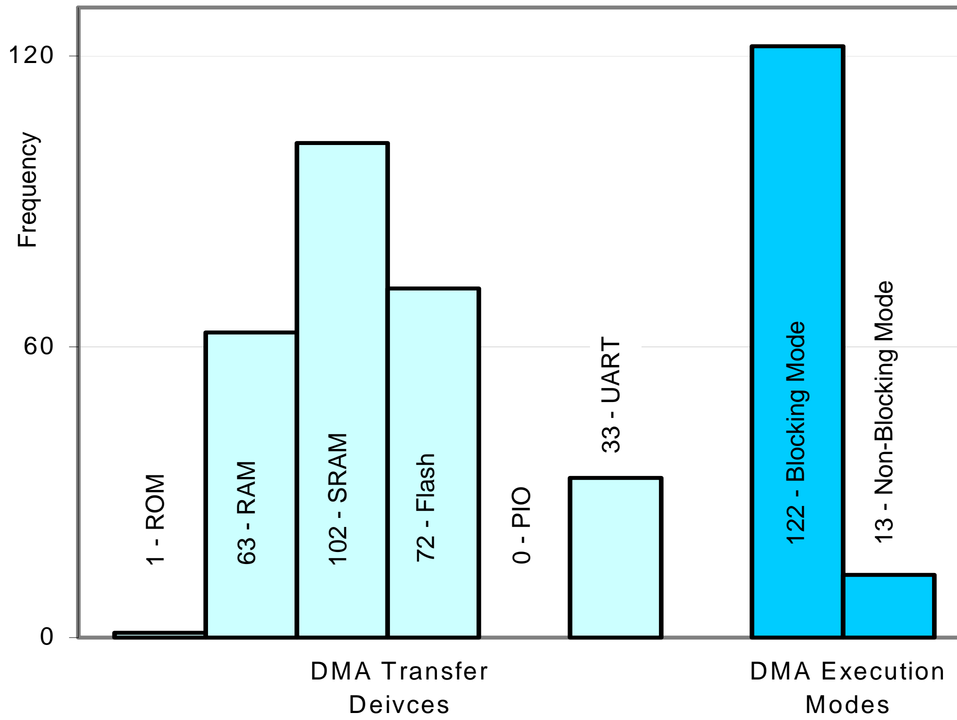


Figure D.9 DMA snippets transfer device and execution mode selections

Based on our analysis, a revised test suite is created by the test generator according to biases in Table D.3. For the DMA parameters, UART, PIO and ROM devices are selected more often as transfer

devices; and we increase the number of DMA interrupt executions. DMA and WritePIO snippets are also weighted with greater selection likelihood. DMA snippets are assigned a greater weight because the DMA exercises other source and destination transfer devices concurrently.

Table D.3 Recommended biases

<table>
<tr><th>Snippets</th><th>Bias</th><th colspan="2">DMA parameters</th><th>Bias</th></tr>
<tr><td>InitDMA</td><td>3</td><td rowspan="6">Transfer devices</td><td>ROM</td><td>2</td></tr>
<tr><td>ExecDMA</td><td>3</td><td>RAM</td><td>1</td></tr>
<tr><td>TermDMA</td><td>3</td><td>SRAM</td><td>1</td></tr>
<tr><td>CheckDMA</td><td>3</td><td>Flash</td><td>1</td></tr>
<tr><td>SetupPIO</td><td>1</td><td>PIO</td><td>3</td></tr>
<tr><td>WritePIO</td><td>2</td><td>UART</td><td>3</td></tr>
<tr><td>ResetUart</td><td>1</td><td rowspan="2">Execution modes</td><td>Blocking</td><td>1</td></tr>
<tr><td>TxUart</td><td>1</td><td>Interrupt</td><td>2</td></tr>
<tr><td>RxUart</td><td>1</td><td colspan="3" rowspan="2">(Note : A weight bias greater than 1 indicates greater selection likelihood.)</td></tr>
<tr><td>RxTxUart</td><td>1</td></tr>
</table>

Table D.4 Coverage results for non-biased initial test suite and final biased test suite

Initial coverage	SoC	CPU	Memories	DMA	UART	PIO
Line coverage (%)	91.0	98.0	68.9	100	98.2	89.6
Toggle coverage (%)	78.0	85.0	58.1	77.1	74.1	63.6
Conditional coverage (%)	66.5	66.8	26.4	94.0	72.6	60.0
Final coverage	**SoC**	**CPU**	**Memories**	**DMA**	**UART**	**PIO**
Line coverage (%)	91.4	98.0	69.0	100	98.2	98.3
Toggle coverage (%)	80.8	86.4	63.7	82.4	74.4	83.1
Conditional coverage (%)	69.4	69.7	29.1	95.0	72.6	83.3
Coverage gain	**SoC**	**CPU**	**Memories**	**DMA**	**UART**	**PIO**
Line coverage (%)	0.4	0	0.1	0	0	8.7
Toggle coverage (%)	2.8	1.6	5.6	5.3	0.3	19.5
Conditional coverage (%)	2.9	2.9	2.7	1.0	0	23.3

The effect of these bias recommendations is an increase in coverage for most of the on-chip devices and overall SoC. Based on this new verification test suite, we conducted further test and coverage analysis to feedback new biasing settings to create further new tests. We performed this coverage directed feedback verification cycle up to five times to improve coverage results until we were certain that additional feedback verification would not enhance verification any further. The final coverage results attained are shown in Table D.4. The PIO coverage gain was best because we biased the PIO

snippet to execute twice as many parallel I/O access operations. In total, we achieved an increase of 0.4%, 2.8%, and 2.9% for line, toggle and conditional coverage respectively.

D.5.2 Summary

The experimental results and our case study demonstrate SALVEM to be feasible as a coverage feedback verification technique. The initial test feedback iterations provided valuable coverage and verification enhancements. Note that even a seemingly small addition to coverage is significant, and one must put this coverage gain into appropriate context. In design verification, when coverage levels exceed 80% to 85%, any improvement is considered extremely beneficial. The remaining 20% of coverage test events are usually hidden deep within the logic of the chip design and cannot be easily exercised, especially in the case of toggle and conditional coverage. In design verification test executions and coverage measuring, it is often a case of diminishing returns whereby the application of more tests results in lower rate of coverage gain as overall coverage levels increase. Therefore, the coverage gain attained in this section is considered highly beneficial.

For these experiments, we chose to focus on the DMA and PIO devices. However, similar analysis and biasing feedback can be applied to the CPU, UART, and memory devices when new or enhanced snippets are included in the verification set up. Given a set of further enhanced snippets, additional coverage feedback iterations will improve verification toward higher coverage. In Chapters 4 and 6, we continued to pursue coverage driven verification in SALVEM, but from an automated perspective – using an algorithmic test generator and the full snippets library.

D.6 Conclusions

This appendix described the development of a randomised test generator to create many SALVEM test programs automatically. The randomisation strategy facilitates many different permutations of long snippets sequences in tests, so as to invoke the SoC into complex and difficult to reach operating states. The favourable coverage results and design bugs uncovered in the Nios SoC prompted further investigations; to conduct a manual coverage directed verification case study with SALVEM. Whilst the subsequent results demonstrated the practicality and improvements in SALVEM verification, it is clear that the method was limited by a lack of automation. In addition, a more suitable form of coverage measuring that is catered for SALVEM would be beneficial for coverage information feedback and directed test creation purposes. In Chapter 7, we described a coverage solution satisfying the needs of SALVEM.

APPENDIX E. Single Objective Genetic Evolutionary Test Generation

This appendix contains supplemental material for Chapter 4, supporting the single objective SALVEM test generation research using genetic evolutionary algorithms (GEA).

E.1 Genetic evolutionary pseudo code

The five main GEA phases: representation, variation, fitness evaluation, population selection, and termination, which are critical to the GEA process, were described in Chapter 4 previously. They form the execution flow of the GEA pseudo code in Figure E.1, which is summarised as follows. Initially, the objective function must be defined. The objective function states the problem to be tackled, and evaluates the quality of each proposed individual solution by assigning a fitness value to each individual. Specifically, for test generation and SALVEM verification, our objective function is to assess how well the test individual exercises and verifies the SoC design (quantified by test coverage). Following this, the evolution process begins with the creation of an initial population of solutions (line 12), usually attained via some form of stochastic method. Fitness is evaluated for each initial individual in the population.

Next, the evolutionary cycle begins (lines 15 to 25). New individuals are created with variation using mutation or recombination of existing parents from $P_{\mu}(z)$ to form child solutions (line 17). The types of mutation and how many individuals are mutated depend on the particular GEA method. Similarly, which parents are chosen, how many children are created, and how parent genes are used to create new children depends on the specific GEA method chosen. The entire population of λ new individuals is then evaluated to assess their fitness (line 19).

Following this, the next generation of best individuals is selected to establish the population for the next evolutionary cycle (line 21). The selection technique – how many individuals are selected and which individuals from parents or children (or both) to select from – is again unique to the GEA approach taken.

The evolution process is repeated by varying individuals, evaluating fitness, and selecting new individual solutions for the next population until the termination condition is met (line 15). The termination condition can be specified in a number of ways. The most common stop condition is when

the target fitness value has been attained by any of the best individuals the population. Alternatively, the process may terminate if there is no improvement in fitness for a pre-selected number of consecutive evolutions. Ideally, at the end of the evolution process, the population provided will contain the best set of solutions for the objective problem.

```
1   // Let Pμ(z)  represent a population of μ parent individuals at
2   //           evolution time index z,
3   //      Q     represent a special set of individuals that may be considered for selection,
4   //            Q = Pμ(z) or ∅, depending on the selection strategy employed.
5   //      Pλ(z) represent the set of λ newly created children offspring
6   //            individuals,
7   //      Pμ(z+1) represent the next generation of successful
8   //              individuals that survived.
9
10  // Begin GEA Process
11  z = 0
12  Initialise [Pμ(z)]     // Randomly search for initial solutions
13  Fitness_Evaluate [Pμ(z)]     // Determine fitness of initial solutions
14
15  while [Terminate is false] {
16
17      Pλ(z) = Variation [Pμ(z)]     // Create new solutions
18
19      Fitness_Evaluate [Pλ(z)]
20
21      Pμ(z+1) = Select [Pλ(z) ∪ Q]     // Retain best solutions
22
23      z = z + 1;
24
25  }
```

Figure E.1 Generalised GEA pseudo code

E.2 μGP integration in SALVEM for genetic evolutionary test generation – a feasibility study

Introduction

Given our conceptual strategy for deploying GEA within SALVEM test generation, a feasibility study was first conducted to establish the usefulness and potential of such GEA test creation for SALVEM; before committing fully down this research path. For the genetic evolutionary feasibility study, our aims were to prototype GEA methods into SALVEM test generation and assess the viability of adopting such a technique.

For this purpose, rather than design and develop an inbuilt GEA test generation engine for SALVEM immediately, we reused the μGP [CCS03] tool and modified certain components of the SALVEM platform to facilitate basic ad-hoc GEA test creation. When adequate viability of SALVEM GEA had been demonstrated, a fully dedicated GEA test generator was then devised to investigate and analyse different applications of GEA techniques for SALVEM, and fully exploit the SoC coverage that can be achieved.

μGP employs mixed genetic algorithm and evolutionary strategy techniques to test microprocessor designs exclusively. The verification operates at the assembler instructions machine code level. The sequences of GEA influenced test instructions from μGP are effective at stressing a processor core, and verifying various microprocessor arithmetic or pipelining units. However, these instructions at present cannot initiate system wide transactions to test SoCs under SALVEM. For example, to test basic system level operations, the SoC and other on-chip devices must be configured with correct sequence of configuration registers accesses and values – a GEA stream of assembler instructions from μGP itself is unable to facilitate this.

To employ some form of SALVEM GEA test generation with μGP, the snippets had to be transformed and mapped into equivalent assembler machine code based test building blocks. The mapping of ANSI-C based snippets to equivalent but simplistic assembler instruction building blocks facilitates some basic SoC functions to be exercised; but additional overhead is introduced into the test creation process. In addition, some supplementary modules and integration code had to be created to use μGP for SALVEM testing. Nevertheless, despite these limitations and complications, restricted forms of SALVEM test programs could be created in a basic and artificial GEA manner for experimentation.

Given this was only a preliminary feasibility study, the entire snippets library was not mapped into equivalent assembler building blocks. Mapping is an extremely costly exercise. Only a small subset of snippets from the snippets library was adapted. The list of snippets employed, snippets mapping, and μGP to SALVEM integration process are described further next.

Adopting μGP for GEA test generation with SALVEM

The integration work required careful pre-analysis and modification of SALVEM platform components to interface with the μGP. In addition, to overcome incompatibilities between SALVEM system level components and low level μGP modules, various integration units and mappings were created. This ensures communication of test information would remain transparent throughout the

modified verification platform, and the test generation flow can be executed efficiently as before in [CPL05b].

Figure E.2 shows the overall GEA SALVEM platform. The μGP generates SALVEM test programs using mixed SALVEM and μGP snippet macros from the instruction library. These snippet macros were originally mapped from the SALVEM snippets library. The test programs are then compiled and linked with snippets API and device drivers to form the executable test binary for SoC simulation. Coverage data gathered during simulation is then fed back as fitness results to drive the test generator during future test evolutions.

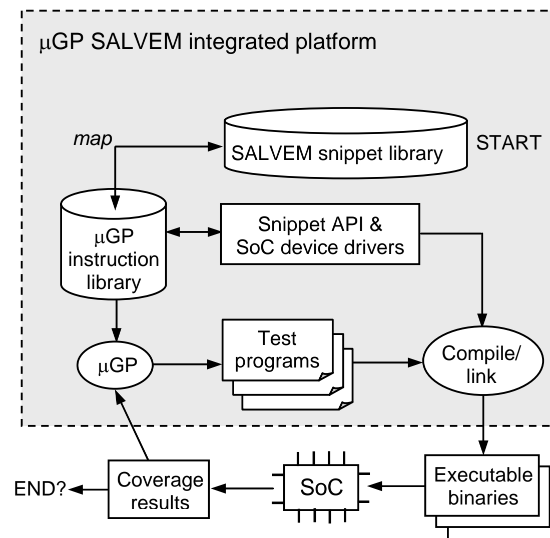


Figure E.2 μGP in SALVEM integrated platform

In μGP, macros are used to specify the types of assembler instructions that make up a test program. μGP macros are very short assembler instruction routines that perform low level operations but are unable to execute SoC functionality. In contrast, SALVEM uses snippets implemented in terms of ANSI-C functions to initial system-wide SoC transactions. In SALVEM test programs, the sequence of snippets is realised by ANSI-C function calls to the snippet library and API. Therefore, in order to facilitate SALVEM GEA using the μGP, new macros were created and mapped to each snippet in the snippets library. μGP can then produce different test programs in terms of these snippet based macro sequences, similar to snippet sequences.

An example of a snippet to macro mapping is shown in Figure E.3. We define macros for selected snippets in the SALVEM snippet library. These macros act as test building blocks and perform the same snippet function call to the snippet library, but are implemented using assembler instructions instead. The actual functional test operations performed by the snippets are still interfaced through the snippet API. The macro invokes these snippet functions at a lower level using assembler jump or

branch instructions equivalent to ANSI-C function calls. Using snippet macros, μGP simply chooses which snippets to call and assigns parameter settings for the snippet macro. After mapping, all snippets macros are captured in the instruction library.

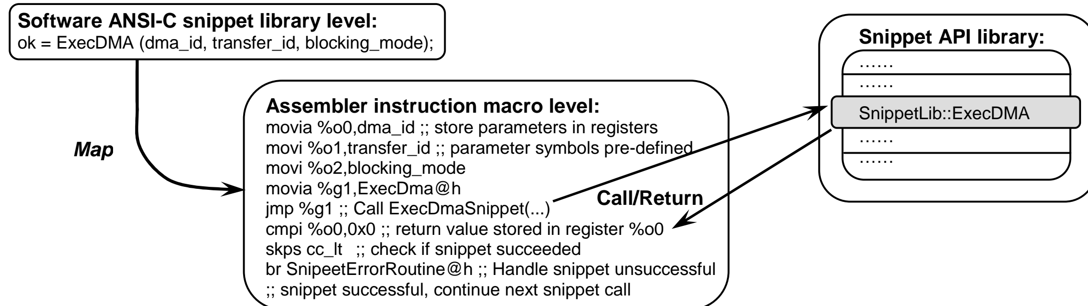


Figure E.3 Example of snippet to μGP macro implementation mapping for Nios SoC

The actual SoC test functions stimulated are still implemented using SALVEM snippet library, API routines and device drivers. The snippet macros simply transfer control to the appropriate snippet API library function. Using the snippet to macro mapping, system wide functionalities can now be tested as the correct sequences of SoC register accesses to invoke these operations are preserved. Previously, the random sequences of instructions from μGP could not do this at all. A subset of the snippets was specially mapped for the feasibility study in this section. Table E.1 lists these snippets.

Table E.1 Snippets mapped for μGP to SALVEM feasibility study

Snippet	Function
InitDMA	Configures direct memory access (DMA) device transfers
ExecDMA	Executes DMA transfers
TermDMA	Terminates DMA transfers
CheckDMA	Validate DMA transfers
ResetUart	Initialise universal asynchronous receive transmit (UART) serial device
TxUart	Transmit serial data
RxUart	Receives serial data
RxTxUart	Duplex serial data transfer
SetupPIO	Initialise or clears input/output (PIO) device pins
WritePIO	Transfers parallel data
MiscCPU	Miscellaneous CPU instructions

Feasibility study experimental results

In this feasibility study, experiments were conducted applying GEA tests on the Nios SoC (Appendix A). The test simulations were conducted on a Linux platform the same as that for experiments of our dedicated SALVEM GEA test generation in Section 4.11 Chapter 4. The GEA test generation employed test population sizes of 30 and 20 test individuals for the parent and children populations respectively. This allows sufficiently large populations and test individuals to mutate and procreate into diverse test suite, but avoid overly long or runaway GEA test processes. The initial number of snippets in tests for the first population was chosen to be small, between 0 to 3. This allows tests to evolve as needed to exercise other SoC behaviours under the guidance of the GEA process.

Line, toggle and conditional coverage were measured to serve as fitness evaluators for the GEA tests. Individual GEA test generation processes were conducted for each of the coverage fitness metric to maximise. On average, this produced 805 tests and approximately 15,300 snippets that were executed within 35 evolutionary cycles. For comparison, a SALVEM randomised test generation run (based on the test generation in Appendix D) was also conducted. Unlike GEA, this random test creation process was not explicitly driven by coverage, hence one randomised test generation was executed measuring all three coverage metric concurrently.

Table E.2 summarises the coverage results. It is clear GEA based test generation achieves greater overall coverage requiring less snippets resources. For conditional coverage however, GEA test creation is slightly below that of random method. This could be due to the nature of conditional coverage measuring whereby many more different kinds and sequences of snippets are required to traverse conditional paths. The GEA method is more conservative in its usage of snippets, and sequences of snippets selection are only retained and expanded if they were previously beneficial. Perhaps the GEA method required more generous composition of snippets for conditional coverage, but maintained judicious usage under GEA guidance instead. For the purpose of this feasibility study, no further refinement to seek higher conditional coverage was conducted. The refinement would be conducted for our dedicated SALVEM GEA test generation investigations instead.

In Table E.2, we show the number of snippets instead of the number of tests for comparison because test sizes between the GEA and the random approach differ considerably. The number of snippets (and tests) created by GEA test generation varies each time because each evolutionary process performs variation differently. Different number of snippets will be added, removed, replaced or recombined during evolution. Therefore, one GEA test is not equivalent to one random-only test.

Table E.2 Feasibility study coverage results

Method	Number of snippets			Coverage %		
	Line	Toggle	Conditional	Line	Toggle	Conditional
Random-only	17,900	17,900	17,900	91.4	82.0	72.0
GEA	15,000	15,100	15,900	97.4	86.4	69.7

In addition to the coverage results above, Figure E.4, Figure E.5, and Figure E.6 shows the coverage progress trends for both GEA and random methods. The coverage graph is plotted along a logarithmic x-axis to show coverage improvement as execution of snippets test increases rapidly, as can be the case during test generations. Comparing coverage progress trend lines, the GEA method outperforms random scheme for line and toggle coverage. The coverage driven process from previous test evolutions in the GEA strategy maintains a higher level of coverage throughout the test generations, whilst coverage improvement rates are relatively similar to the random method in general.

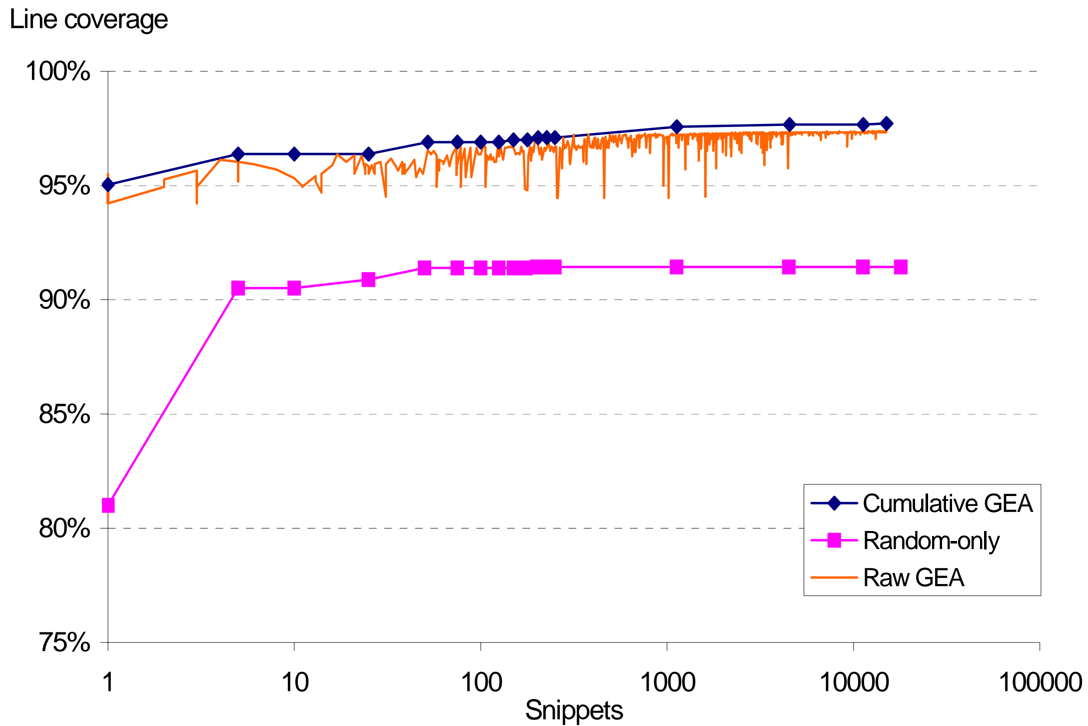


Figure E.4 Feasibility study line coverage versus snippets

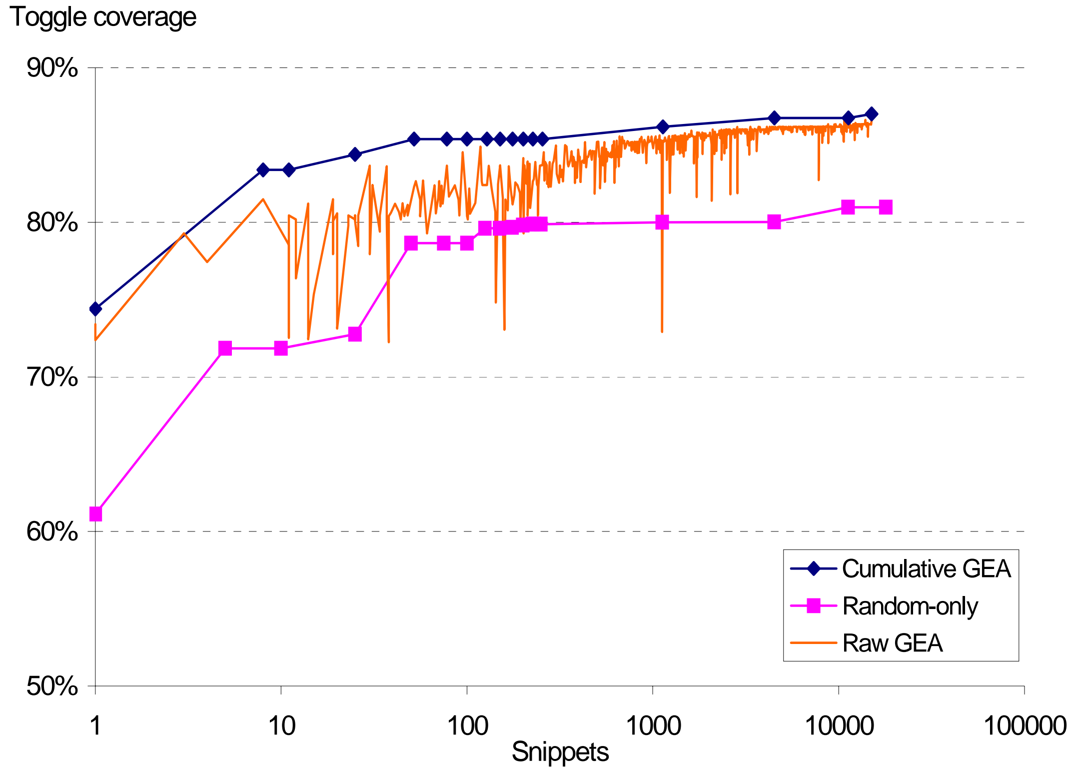


Figure E.5 Feasibility study toggle coverage versus snippets

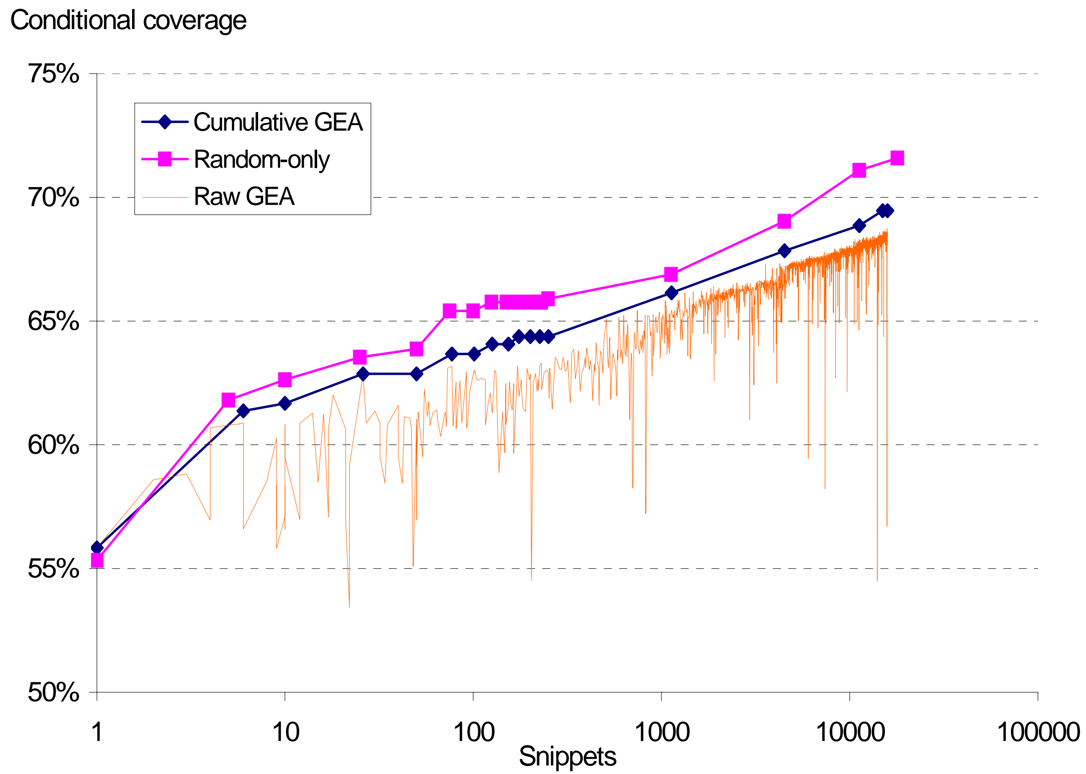


Figure E.6 Feasibility study conditional coverage versus snippets

The line coverage rates are not comparable because line coverage is usually an easy to achieve metric (for the initial 80%), and this was easily achieved by both methods already from the initial single snippet test coverage. The differences in initial coverage attainment from a single snippet test between GEA and random methods are due to the type of snippets used at the beginning of the test process. The initial test program for GEA contain snippet that was simply superior at attaining higher coverage to begin with. As is the case with any GEA process, the success of evolutionary tests depends on the initial test population. For these experiments, both GEA and random testing use randomisation to select their starting tests.

Cumulative coverage trends lines for both GEA and random testing represents the coverage accumulated for all previous and current tests. The raw coverage GEA trend lines represent the coverage value for each test. The oscillating peaks and troughs spikes of the raw coverage are characteristic of any GEA process. Reduction in these spikes indicates the GEA process is close to attaining the optimised population and maximal coverage possible.

For conditional coverage (Figure E.6), large variation spikes still occur at the end of the evolutionary process. This suggests a longer test generation process is required, and further test generate evolutions are needed to stabilise the coverage line and attain the coverage optimum. This is not surprising given that exercising SoC design conditional paths demands much larger number of SoC operations and hence tests. Conducting simulation with conditional coverage is more computationally intensive and requires longer simulation runs, which we did not pursue further given this was simply a feasibility study. The experiments and results were already sufficient for our purposes. We refer the reader to Section 4.11.3 Chapter 4 for full discussion of these coverage spikes, which also occur in the proper experimentation of our SALVEM GEA test generator.

In terms of test generation performance, the test creation time for both GEA and random-only test generators were measured. The results showed not much difference between the two. Compared to test execution time, test creation time from both test generators are negligible. The main overhead of the verification process is test simulation. Table E.3 shows the test execution times per snippet for GEA and random-only methods. The time for conditional coverage is much greater than line or toggle because conditional coverage is difficult to measure. Again, the reason is the number of branch paths in a design is exponential requiring much more computational resource to track. Comparing GEA and random-only methods, it is clear the GEA method is more efficient. GEA provides 2, 1.4, and 1.7 times improvement for line, toggle and conditional coverage testing.

Table E.3 Feasibility study test execution performance times

CPU-sec per snippet	Line	Toggle	Conditional
Random-only	7.0	7.1	46.3
GEA	3.6	5.1	26.6

These basic set of experiments for the feasibility study demonstrates GEA test generation for SALVEM is indeed viable and can provide better test performance. Results here show GEA test generation can be considered more optimal with respect to coverage and test sizes. These enhancements are considerable, demonstrating GEA's ability to explore larger untested test regions on the SoC economically.

However, the μGP based test generation displays lower coverage attainment rate than expected. As compared to our GEA test generator (described in Section 4.11 Chapter 4), it was required to undergo more test evolutions and greater tests before its maximum coverage was attained. This was primarily due to the overhead introduced that facilitate μGP to create SALVEM tests from the mapped snippets. The μGP was never intended to create system level SoC tests, and cannot be considered a solution for GEA test generation for SALVEM.

Despite the incompatibility of μGP for SALVEM, improvements over random methods from this feasibility study confirmed there was potential for the research and design of our own dedicated GEA test generator specifically for SALVEM. The sole purpose of our GEA test generator will be to create SoC test programs composed of snippets directly under GEA, overcoming the limitations and avoiding the unnecessary modifications required for μGP. Our GEA test generator will be fully compatible with snippets test building block characteristics and create tests without any overhead unlike μGP. It will also allow for much more GEA features and options to be researched, so we can refine the test generation for snippets characteristics and maximise coverage over μGP and random methods.

Feasibility study conclusion

The feasibility study demonstrates a GEA test generation process for SALVEM is viable and could return substantial benefits. The feasibility study and integration work conducted formed the basis from which our customised SALVEM GEA test generator was prototyped. Additionally, the deficiencies identified with GEA test generation from this preliminary study was also tackled with our own specially devised GEA test generator.

E.3 SAGETEG variation operators

Figure E.7 shows the diagrammatical representation of GEA variation operators employed by SAGETEG. Replacement variation is a composition of the addition and subtraction variation, hence is not shown.

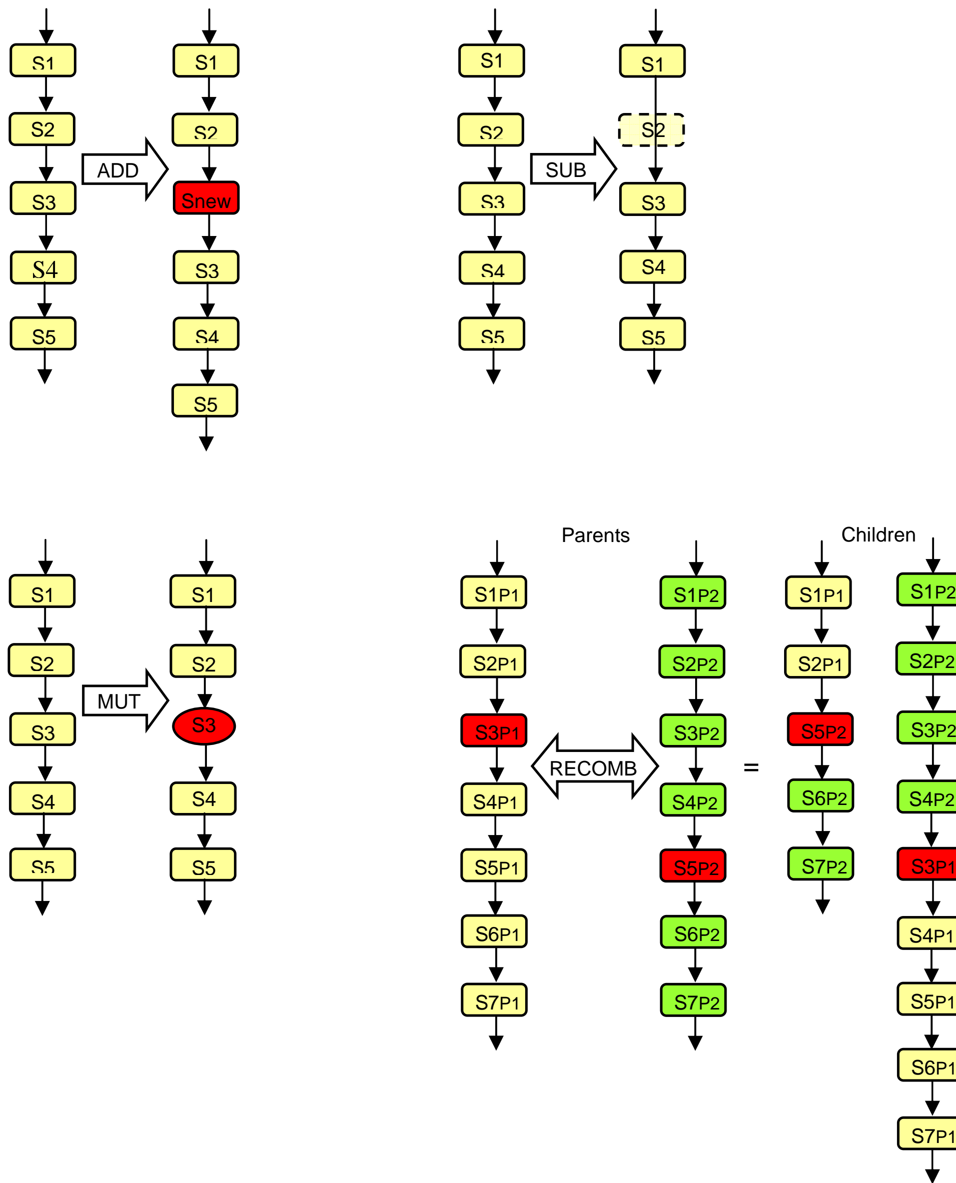


Figure E.7 Variation operations

E.4 Variation constraint checking

The function g defined to check legality of a test which is created by genetic evolutionary variation, employs the following definitions to check constraints of a snippet inserted into the test.

Definition E.1 : Explicit and implicit constraint checking

(i) Explicit constraint

Let $g_{exp} : S \rightarrow \{\text{true, false}\}$ be the function to check explicit constraints, and CT be the set of explicit constraints for the snippet s,

$$g_{exp}(s) = \text{true} \;\; \text{if } \forall\, ct \in CT, \text{ constraint } ct \text{ is satisfied, or false otherwise.}$$

For the Nios SoC, the set of constraints for each snippet is defined in Appendix C.3.

(ii) Implicit constraint

Let $g_{imp} : S \rightarrow \{\text{true, false}\}$ be the function to check implicit constraints, and V be the set of parameters for the snippet s,

$$g_{imp}(s) = \text{true} \;\; \text{if } \forall\, v_i \in V, x_i \in D_i \text{ for } i = 0, 1, \ldots, |V| \text{ , or false otherwise,}$$

where v_i is the i-th parameter from the set V of parameters for the snippet s,
x_i is the value of the parameter held by parameter v_i,
D_i is the domain set of values that should be held by parameter v_i to satisfy implicit constraints.

□

E.5 Dependency snippet insertion for addition variation

The pseudo code implementation of the function d to insert additional dependent snippets into a test varied by GEA addition variation is shown in Figure E.8.

The snippet dependency insertion implementation checks for each of the dependency types one after another. If a dependency snippet is to be inserted into the test, the addition variation operator is employed (lines 9, 16, 24 and 33). This results in a recursive operation of the dependency snippet addition and subsequent dependency checks, given that the insertion of a dependency snippet itself could have unresolved dependencies in the test. Once all dependencies of the original snippet addition variation and any subsequent dependencies of dependent snippets are resolved, the function d called for the initial addition variation will terminate. For non-strict dependencies, the dependency snippet

addition can be called multiple times for every non-strict dependency that is not satisfied (Lines 23 to 25 and 32 to 34). The *rand* function selects a random position to insert the dependency snippet from an interval delimited by two given bounds.

```
1    d [t, s^add, i] {
2        //    t       is the test that has under gone addition variation,
3        //    s^add   is the snippet added into the test,
4        //    i       is the position at which the snippet is added into the test
5
6        // Check for pre-strict dependency using the dependency check function in
7        //  Definition 4.5, Chapter 4.
8        if [ g_pre-strict(t, s^add, i) = false ] {
9            t = Add[t, s^pre-strict, i–1]   // Add is the addition variation operator which
10                                           // is re-used to insert the dependency snippet.
11       }
12
13       // Check for post-strict dependency using the dependency check function in
14       //  Definition 4.6, Chapter 4
15       if [ g_post-strict(t, s^add, i) = false ] {
16           t = Add[t, s^post-strict, i+1]
17       }
18
19       // Check for pre-non-strict dependency using the dependency check function in
20       //  Definition 4.7, Chapter 4.
21       if [ g_pre-non-strict(t, s^add, i) = false ] {
22           // Conduct addition for each unresolved dependency snippet
23           foreach [ s^pre-non-strict ∈ S^pre-non-strict ∧ s^pre-non-strict ∉ t ] {
24               t = Add[t, s^pre-non-strict, rand(1, i–1)]
25           }
26       }
27
28       // Check for post-non-strict dependency using the dependency check function in
29       //  Definition 4.8, Chapter 4.
30       if [ g_post-non-strict(t, s^add, i) = false ] {
31           // Conduct addition for each unresolved dependency snippet
32           foreach [ s^post-non-strict ∈ S^post-non-strict ∧ s^post-non-strict ∉ t ] {
33               t = Add[t, s^post-non-strict, rand(i+1, n+1)]
34           }
35       }
36
37   }
```

Figure E.8 Dependency snippet insertion pseudo code implementation

E.6 Variation self-adaptation analysis

E.6.1 Self-adaptation with Rechenberg's rule

The aim of GEA variation operators is to continually revise test programs to attain as high coverage as possible. During the evolution process, a test program may grow, shrink, mutate internally, combine with other test programs to reproduce new tests, or be replaced by other higher coverage yielding tests. The aim of variation is to construct and maintain the best possible group of test programs. However, overuse of variation operators can lead to poor search of the test space. If test programs are altered too greatly, important test functionalities within local search regions may be overlooked. Therefore, variation should be applied diligently based on how test programs react to their modifications and their influence on test suite fitness.

In our test evolutionary process, each variation operator is assigned a probability weight variable that determines their likelihood of usage. Under normal GEA operating environments, all variation weightings are initially equal. However, for test generation, our initial tests are very small in size with only a few or even no snippets, relying instead on the evolution process to cultivate the test population into a useful test suite. Initially, only addition and mutate variation can be assigned high usage weightings. During the evolution process, these variation weights are then continually monitored and adjusted. The mechanism that facilitates this is a called self-adaptation. Self adaptive schemes are commonly used in GEA and are successful techniques for ensuring appropriate GEA parameters values are used during the test generation process to create and retain the best test individuals. In GEA, the values chosen for various test strategy parameters, such as variation weights, have impact significantly on the effectiveness of the algorithm and the test coverage performance of the test generation process [HME97]. For test generation, the GEA self-adaptive method employed is commonly based on Rechenberg's rule [Mic96, Rec73].

Rechenberg's rule was originally intended for optimising both linear and non-linear mathematical functions. It was first used in (1+1) evolutionary strategy methods involving one parent and one child populations using mutation only. However, in test generation, it can be extended for a general $(\mu+\lambda)$ multi member population using other GEA variation operators as well.

Rechenberg's rule provides 'rule of thumb' guidance in relation to the selection and use of various GEA parameters throughout the evolutionary process. Specifically, for test generation, Rechenberg's rule was employed to control the selection probability weightings of variation operators. By doing so, Rechenberg's rule guides the test generator to create tests based on the relative success or failure of

variation operators applied in previous test evolutions. This ensures usage of more effective variation operators is maintained for future test evolutions, to continue creating successful test populations.

Specifically, Rechenberg's rule states the ratio of successful variation to non-successful variation operations should be approximately 1/5; whereby the variation is deem successful if it creates a test program that yields a higher coverage fitness compared to the original unmodified test individual chosen for variation. If the ratio exceeds 1/5, the variation probability weighting is increased for the next evolution of test creation, to carry on building upon the successes of applying that particular variation. If the ratio is less than 1/5, the variation probability is lowered as current usage of variation is already deemed unsuccessful. Based on Rechenberg's recommendations, before the start of each evolution, the probability weightings of applying each variation operator are adjusted as follows.

Classical Rechenberg's rule

Let $\omega(z)$ be the probability weighting of applying a variation operation at evolution z.

For $z = 0$,

$$\omega(z) \geq 0.5$$

For $z \neq 0$,

$$\omega(z) = \begin{cases} \omega(z-1) \times \sigma & \text{if } \gamma < 1/5 \\ \omega(z-1) \times \dfrac{1}{\sigma} & \text{if } \gamma > 1/5 \\ \omega(z-1) & \text{if } \gamma = 1/5 \end{cases}$$

where γ is the variation success ratio of the population, and
σ is the variation weight change factor that controls how much the variation weight value is increased or reduced.

Test space conceptualisation for self-adaptation

Before discussing the effects and implications in applying Rechenberg's rule, a description of the abstract SoC test space concerning GEA test generation is presented first. Since test generation is essentially a search and optimisation process over the testable SoC design space, it is natural to visualise and explain the use of GEA for test generation using a test space concept. Specifically, the test space shall be referred to when explaining how Rechenberg's rule and test variation influence test

generator search and coverage of the test space. The test space diagram is shown in Figure E.9 and is described as follows.

Assume all possible SoC test scenarios can be captured in an enclosed rectangular area which we denote as the test space. Each point in the test space represents a specific test scenario that can be tested. Test scenarios that exercise similar kinds of SoC functionality are grouped together to establish test regions, as represented by the circles in Figure E.9. For example, within the DMA test region, each point corresponds to a particular form of DMA transfer that must be tested. Each of these DMA points may represent a transfer scenario between one specific source device and a specific destination device involving a particular amount of data units. Many other (possibly overlapping) test regions also can be identified for other related types of SoC test conditions. In Figure E.9, the UART test region overlaps the DMA test region. DMA transfers using the UART as either source or destination device are represented by test points within this overlap region. The GEA test generator will create tests to seek out these test regions and cover a maximal portion of test points within each region before moving on to other regions.

To facilitate this process, large addition and mutate variation is generally used to seek out and identify new and wide-ranging test regions (solid arrows in Figure E.9). Subsequently, subtract, replace and recombination variation are then applied to investigate and maximise coverage of each of these test regions more thoroughly (dashed arrows in Figure E.9).

Using variation under the guidance of Rechenberg's rule, the goal of test generation is to maximise coverage of the entire test space using a minimal test set. Whilst the μ test population size remains constant over successive evolutions, the goal is for each test to contribute as much as possible to coverage, using fewer snippets before the snippet sequence grows too large in future evolutionary stages. This is considered one of the test optimum condition which GEA test generators must pursue.

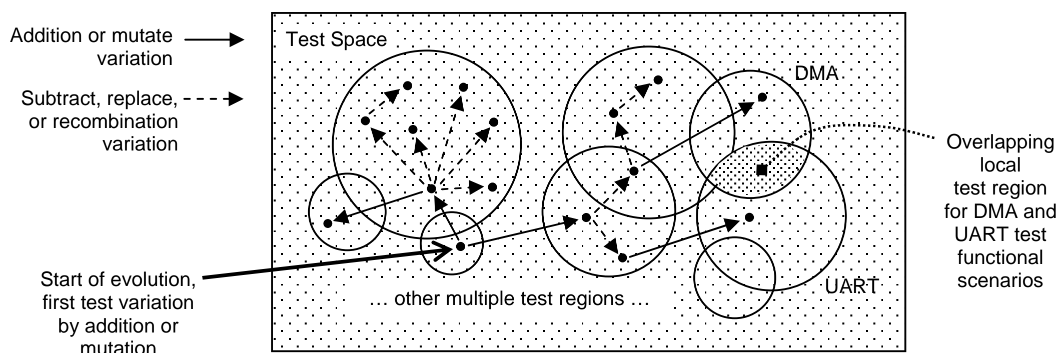


Figure E.9 SoC test space diagram for GEA test generation

Intention of Rechenberg's rule

Based upon the test space representation in Figure E.9, intuitively, Rechenberg's rule implies variation operators should be widely used at the start of the evolutionary process to seek out wide ranging and extensive regions of the test space. To facilitate this, during early evolutions large amounts of variation should be applied to create new tests that are vastly different from previous existing tests. Early evolutionary tests should search for many different test regions in the beginning using minimal snippets before tests become larger in later test evolutions.

After many successive evolutions, when the test creation process is close to the desired test optimum, variation is then reduced according to Rechenberg's rule. This enables the generator to focus within local test regions more thoroughly, to ensure the actual test optimum is achieved. The coverage already attained from local test regions under current investigation must be maximised, by ensuring all test scenarios within those regions are covered. Therefore, at this stage of the evolutionary process, the amount of variation applied will be significantly lower in order to allow for minute fine tuning of the tests covering those test regions.

Employing a success ratio of 1/5 throughout evolutions implies for each variation operator that creates a new test, one out of five tests should be successful in enhancing SoC coverage compared to the previous unvaried test. By maintaining such a ratio, the test generator maintains balance between (1) extensive exploration of new test space to aid potential coverage improvement (solid arrows in Figure E.9), and (2) continual examination to comprehensively cover current local test regions and maximise coverage from those regions (dashed arrows in Figure E.9).

Extensive exploration of new test regions is important to identify new test scenarios that increase coverage. However, over exploration can often lead to tests that do not add to the overall test coverage at all. This is because the variation operators that conduct random exploration may return to test regions where SoC test scenarios were already verified previously at a much earlier evolutionary stage. Hence, the resultant coverage will not exceed what was already attained previously during earlier testing. Additionally, over exploration can cause previously identified test regions to be inadequately examined. Before an existing test region is fully investigated and all test scenarios from that local test space tested, applying large variation too early can force the test generator to move on to other regions of the test space prematurely. Any potential coverage improvements from test scenarios from the current test region will be overlooked, eliminating any opportunities to improve test coverage further.

Similarly, continual examination of a test region should not be over excessive. After a reasonable number of tests have been varied from within a test region, the test generator must be able to move on to investigate other new portions of the test space. Otherwise, the test creation process will generate too many overly similar tests causing coverage test enhancement to stagnate. If such repetitive tests are allowed to dominate and fill the test suite population, the entire test generation process would stall because variation of similar tests restricts further exploration of the test space. Rather, the test generator would end up examining the same test regions repeatedly as a result of varying these similar tests to create further similar tests.

The aim of Rechenberg's rule is to maintain balance between extensive explorations of the test space using stronger variation, versus sufficiently thorough examination of currently traversed local test regions; in order to efficiently find the test space optimum. The 1/5 ratio ensures that at least more than one in five varied tests from a test region must be successful in improving test coverage. If so, larger variation will applied to search further test regions and hopefully seek out more fruitful coverage yielding regions compared to the current search location. The aim is to speed up searching of the optimum, and improve efficiency of the evolutionary process by steering it towards convergence. However, if these test regions do not provide improved coverage, this implies variation and exploration of other test regions may have become too excessive. The success ratio would then decrease falling below 1/5 causing the test generator to revert to lower variation probabilities. This implies that the previous test region was closer to the optimum, hence test space exploration is withdrawn to a smaller search area and the current local test region is thoroughly examined instead.

By adhering to the above rule of thumb, the test generator should be more confident in seeking out new test space and ensure all possible test variants from that space are covered. The maximum attainable coverage from each test region will also be achieved more efficiently using least number of evolutions.

Implementation-wise, applying Rechenberg's rule required a success-to-failure ratio for each variation type. Each ratio is calculated at the end of every evolution and is used by the test generator to adjust variation weights according to Rechenberg's rule. However, for ease of analysis, the averaged ratio amongst all variation operators was used to measure how successful the test generation process was at maintaining the 1/5 ratio during evolution. The aim of Rechenberg's rule is to preserve one in five successful coverage improving test throughout evolution, regardless of how much or which particular variation operator is applied.

E.6.2 Fine tunning and observations of Rechenberg's rule

Preliminary test generations were conducted to check the implementation and evaluate effectiveness of applying Rechenberg's rule in SALVEM GEA test generation. In addition, the various GEA strategic and operational parameter values were also reviewed to ensure they are assigned appropriate values to provide best performance. For example, the initial variation operator probability weights, change factor σ, and other parameters can affect the effectiveness and efficiency of the process in seeking best tests. The process of conducting preliminary runs to fine tune GEA parameters is part of calibrating the genetic evolutionary algorithm for the test generation target application; and is common practice when GEA techniques are employed [HME97].

The initial results from these preliminary test runs showed that a number of modifications were needed. These modifications included minor refinements in various usage parameters and the manner in which Rechenberg's rule was applied during test generation. Additionally, our observations on how test population fitness reacted to Rechenberg's rule led us to revamp the criteria in which variation weightings are adjusted. This sub section discusses the minor refinements carried out, Sections 4.5.10 in Chapter 4 described the changes made to variation weight adjustments.

Traditional Rechenberg's rule assumes all initial variation weights to be equal. However, for SALVEM GEA test generation, subtract, replace and recombination variation should not be applied too often during early evolutionary stages due to the effects of snippet dependency rules. Subtract, replace and recombination variation cannot be properly employed until the snippet sequences in tests have grown to sufficiently large sizes. Given that tests are initially created with very few or no snippets at the first evolution, the initial subtract, replace and recombination variation weights are assigned low values. Instead, addition and mutate variation weights are initially allocated higher values by ratio of up to three to one against subtract, replace and recombination variation.

An initially high addition and mutate weight also supports Rechenberg's recommendation for larger variation to be applied early, so extensive regions of the test space can be found. Afterwards, subtract, replace and recombination weights are then increased according to Rechenberg's rule to examine these discovered test regions more thoroughly. The test generation relies on addition and mutate variation to grow a wide range of different snippet sequences in tests, so remaining variation can be applied more regularly in later evolutionary stages.

The results of preliminary testing also showed variation weights adjustments conducted by Rechenberg's rule can be overly excessive. In some cases, variation weights were increased or decreased to extremely high or very low values. Furthermore, these excessive weight values were

occurring during early evolutionary stages due to a high rate of weight adjustment and large change factor σ from Rechenberg's rule. The test suite during initial testing may not always present a true representation of the population success ratio, as these tests are too small and contains only very short snippet sequences. Therefore, no variation weights should be allowed to over dominate, eliminating opportunities for other variation to be applied. Similarly, variation weights should not be too low as well. Such conditions would drive the test generation into various run-away conditions, whereby generated tests become overly large or small, but cover the same test regions as previous tests and do not add value to coverage.

Furthermore, if variation weights are allowed to reach excessive values, it will require many evolutions before they recover back to appropriate levels and balance out again. By then, the test generation may have already terminated. Therefore, to prevent such conditions, a minimum and maximum weight limit of 10 and 100 respectively should be imposed. This ensures all variation will at least be given some opportunity to be applied throughout test generation.

As discussed above, the test suite during initial evolutionary stages may not always provide an accurate representation of the success ratio. Therefore, immediately applying Rechenberg's rule to adjust variation weights may in fact be premature, and reduce the effectiveness of future weight variations. Instead, application of any self-adaptation should be delayed until a minimum number of test generate evolutions have been conducted. The initial test evolutions will calibrate the test generator first before self-adaptation is enforced. Otherwise blindly adjusting variation weights based on the success or failure of immature tests would be detrimental. In our case, a mature test is one that has undergone sufficient variation during initial test evolutions, to be of sufficient size and considered typical of SALVEM test programs. Applying Rechenberg's rule on such tests will then provide more accurate coverage results to truly reflect the success ratio of the test suite. For SALVEM GEA test generation, application of any self-adaptation can be delayed up to ten initial test evolutions have been conducted.

E.6.3 Motivations for revising variation self-adaptation

Besides refinements from the previous section, the most significant modification to our implementation of self-adaptation was the manner in which variation weights are adjusted. The need for such modification was due to a number of observations and subsequent analysis of the test population response from classical Rechenberg's rule. Specifically, it was not uncommon for the test

suite success ratio to diverge from the 1/5 target, and eventually stagnate at an undesired ratio. There are two types of run away conditions that may cause Rechenberg's ratio divergence.

Whenever the test suite ratio is less than 1/5, Rechenberg's rule decreases variation weights to reduce variation, with the intention of seeking new tests successes again and increasing the overall ratio. Below 1/5, the assumption is that the optimum is more likely to be within the local test space region. Hence, lowering variation will direct the test generator to search within the current test region, and creation of new tests locally would improve test coverage and subsequently drive the ratio back toward 1/5.

However, in some cases, less varied tests may not increase overall success ratio rise at all. If current test regions discovered by previous evolutions were already thoroughly examined, continuing to employ less varied tests from these test regions may in fact decrease or maintain current ratio levels. The test generator will either create similar tests, or seek out remaining variants of SoC test functions from current test regions that were already tested; hence preserving or decreasing coverage instead.

Additionally, by restricting variation, the test generator is prevented from creating tests from other different areas of test space. In certain cases, the identification of other test regions can increase success ratio, especially if these varied tests originates from previously unexplored test regions. By lowering variation, such potential higher coverage yielding tests would not be realised to drive overall test suite ratio up. Under such circumstances above, employing less varied tests will maintain the current ratio levels or even pull it away from the desired 1/5 target. In a number of evolutions during preliminary testing, lowering variation weights and test suite diversity caused stagnant ratio levels initially, which also eventually decreased further. It was obvious that lowering variation slows down the rate of change in success ratio rather than increase it as was expected.

Secondly, Rechenberg's rule demands variation weights be increased whenever the test suite ratio is above 1/5. The intention is to introduce greater variation to continue producing tests from other wide ranging and new test regions. Hence, tests coverage increases and other test regions where the optimum may lie are uncovered; until eventually, the variation applied becomes too excessive and exploration too extensive causing the test suite ratio to fall back toward 1/5. However, the risk for such an approach is that tests from other test regions may not provide improvements in coverage or success ratio compared to previous evolutions.

Because the test generator inherently employs random choices, the new test region may not guarantee better coverage; especially if some portion of that test region overlaps with previously tested test

space. Moreover, by increasing variation weights to shift test generation focus to other regions, the test generator may overlook other fruitful coverage enhancing tests from current test regions. In reality, to increase coverage success test ratio, the current test regions previously under examination should have been thoroughly investigated further, using less varied tests first. Once again, if the above conditions were realised, increasing variation weights according to classical Rechenberg's rule would produce the opposite desired intention.

Furthermore, whenever the test suite ratio is close to 1/5, and either the above two conditions occur, blindly increasing or decreasing the variation weights by the same amount each time will result in significant deviation away from the desired ratio. Because variation weights are linearly adjusted by the same constant change factor σ over successive evolutions, the weight values can reach extremely high or low values quickly; prompting over use of various variations and restricting the diversity of the test suite.

Regardless how close the test suite ratio is to 1/5, adjusting weights by the same amount constantly can be detrimental because a test suite ratio close to 1/5 should not be varied by the same amount as a test suite that is significantly different from 1/5. A ratio close to 1/5 represents a test suite close to the desired goal, and only small variation should be needed to modify tests to achieve that goal. In contrast, a test suite ratio considerably different from 1/5 requires larger change in variation probabilities, to create new and different tests that can provide the desired balance between number of test successes and failures. Recklessly adjusting variation weights by the same amount without considering the relative difference between current test suite and target ratio reduces the likelihood of actually achieving the target ratio.

The occurrence of such above conditions may not be frequent but is highly possible during any stage of the test generation process. Once the test generator enters any of these conditions, a repetitive cycle of continual increase or decrease of variation weights would be initiated. Subsequent evolutions from that point onwards will result in the test suite drifting further away from 1/5. This effectively renders classical Rechenberg's rule useless, causing the remaining test creation process to generate non-coverage optimised test suites. The aim of Rechenberg's rule is to speed up convergence and facilitate more efficient search of the test space optimum. However, whenever such conditions above arise, pre-mature convergence occur, which is not uncommon and a significant shortcoming of Rechenberg's rule. Therefore, it became clear that Rechenberg's rule in its original form was too simplistic for the various types of SALVEM test population and evolutionary test conditions. Instead, we devised our self-adaptation strategy which was described in Section 4.5.10, Chapter 4.

E.7 Population selection formalisation

The population selection is defined in Definition E.2.

Definition E.2 : Population selection

Let Select : $\mathcal{P}(P) \times \mathbf{Z} \rightarrow \mathcal{P}(P)$ be the population selection function to select the next population $P_{\mu}(z+1)$,

$$P_{\mu}(z+1) = Select(P_{\mu+\lambda}(z), \mu) = Retain(Sort(P_{\mu+\lambda}(z)), \mu)$$

where μ is the number of parent tests,

λ is the number of children tests,

$P_{\mu+\lambda}(z)$ is the combined parent and children tests in the current evolution z,

$\mathcal{P}(P)$ is the power set of a population P which represents all possible subsets of the test population and all possible selections of tests from the population,

Sort is a function that sorts the population of tests according to their fitness, and

Retain is a function that retains the selected tests that qualify for use in the next evolutions.

The functions *Sort* and *Retain* are defined as follows.

(i) The function *Sort* : $\mathcal{P}(P) \rightarrow P^{(\mu+\lambda)-\xi}$ sorts a population of tests, to produce a tuple T of tests ordered by their fitness values as,

$T = Sort(P_{\mu+\lambda}(z)) = \langle t_1, \ldots, t_{\mu}, t_{\mu+1}, \ldots, t_{(\mu+\lambda)-\xi} \rangle$ such that $\forall\ t_i \in P_{\mu+\lambda}(z), f(t_i) \geq f(t_{i+1}) \wedge$ lifespan of t_i is less than allowable lifespan limit, for $i = 1, \ldots, \mu+\lambda$

where ξ is the number of tests that are excluded from T because they exceed the allowed lifespan and,

$f(t)$ is the objective function from that evaluates the coverage of a test t.

(ii) The function *Retain* : $P^{(\mu+\lambda)-\xi} \times \mathbf{Z} \rightarrow \mathcal{P}(P)$ retains the appropriate number of tests from the combined population $P_{\mu+\lambda}(z)$ of the current evolution, to produce the selected population $P_{\mu}(z+1)$ of tests for the next evolution as,

$$P_{\mu}(z+1) = Retain(T, \mu) = \{\ t_i \mid t_i = T[i] \text{ for } i = 1, .., \rho\ \}$$

where $T[i]$ is the ith test ordered in the sorted tuple of tests T,

ρ is the number of tests to retain. It is determined as follows,

$$\rho = \begin{cases} \mu & \text{if } \mu \leq (\mu+\lambda)-\xi \\ (\mu+\lambda)-\xi & \text{otherwise, too many lifespan exceeded tests were excluded.} \end{cases}$$

□

In the beginning, the selection process is to sort the population of combined parent and children test individuals, excluding any tests that have exceeded the allowed number of evolutions lifespan. Such tests are considered extinct from future evolutions. Depending on the number of tests excluded, the sorted tuple of tests T may contain than μ tests. If this occurs, then all the remaining $(\mu+\lambda)-\xi$ tests in T are selected for the next evolutionary population. However, the likelihood of T with less than μ tests is extremely low. During each evolution, there are sufficient new tests with superior fitness to replace existing tests, thus preventing tests from surviving many evolutions to begin with. Usually, most tests do not reach its allowed lifespan and will be eliminated much earlier. If a test suite is dominated by tests that have survived many evolutions, this implies the test generation process is about to be terminated regardless due to stagnation of the population.

E.8 Implementing SAGETEG

This section describes the implementation of our single-objective genetic evolutionary test generator for SALVEM. Given the GEA operations defined for our test generation flow in Figure 4.11 Chapter 4, test generation with SAGETEG is summarised in Figure E.10. Figure E.10 shows the pseudo code implementation of our GEA test generation process. The *CreateInitialTest* function (line 10) creates test individuals to fill the initial population of μ tests. The initial test created may contain any number of random snippets, including an empty set of snippets, relying instead on the evolution process to add and vary different types of snippets into the test individual over time.

During each test evolution, the variation stage iterates λ times in order to create at least λ new tests (lines 19 to 81). In each λ iteration, a selected parent test t_{new} may undergo multiple variations to produce a new and different test which is assigned back to t_{new} (lines 26 to 76). Each variation is applied repeatedly on the same t_{new} test and inserted into $P_{\lambda}(z)$. However, if recombination is chosen, unlike other variation operators, two new test children are created each time. In this case, both children tests will be retained in $P_{\lambda}(z)$. If further variation is to be applied, one of the children tests will be selected, and a new copy of this child test will be duplicated to undergo such variation. In this case, three new tests are added into $P_{\lambda}(z)$ – the two children tests, and the duplicated child test that was selected to undergo further variation. If no further variation is to be applied after a recombination variation, then only two additional children tests are inserted into $P_{\lambda}(z)$. After variation, fitness evaluation of new tests, population selection, and self-adaptation of variation weights is performed as described in Chapter 4. This evolutionary cycle repeats until the termination condition is triggered.

```
1   //  V = {Add, Sub, Rep, Mut, Recomb}   is the set of variation operators,
2   //  W = {ωAdd, ωSub, ωRep, ωMut, ωRecomb}   is the set of variation operator weightings,
3   //  M   is the maximum number of variation operations that can be applied to a test
4   //  (All other symbols, variables or definitions are as per Chapter 4)
5
6   z = 0   // Initialise evolution index
7
8   // Create the initial population of tests
9   for [ x in 1, …, μ ] {
10      t_x = CreateInitialTest      // Randomly create the initial test
11      f(t_x)                       // Evaluate the fitness of the test
12      P_μ(z) = P_μ(z) ∪ {t_x}      // Add test to the initial μ population of tests
13  }
14
15  // Begin the evolutionary cycles
16  while [ Terminate = false ] {
17
18      // Vary current P_μ(z) population to create at least λ new tests
19      for [ x in 1, …, λ ] {
20
21          // Select a test t_new from current P_μ(z) population to undergo variation, using k
22          // member tournament selection, where the participants of the tournament
23          // are randomly chosen from P_μ(z)
24          t_new = TourSel_k[ rand(P_μ(z))_1, …, rand(P_μ(z))_k ]
25
26          // Apply random number of variation operations on t_new, at most M number of
27          // variations are allowed
28          n = rand(1, …, M)   // Select random number of variations y to conduct
29
30          // Perform n variations
31          while [ y ≠ 0 ] {
32              y = y – 1
33
34              // Select which variation operation v from the set of operators V
35              // to conduct, based on variation weights in W
36              v = SelectVariation[V, W]
37
38              // Handle recombination variation specially because it creates two new tests
39              if [ v = Recomb ] {
40                  t_A = t_new      // Use t_new as the first parent test
41                  // Tournament elect the parent test in addition to t_new for recombination
42                  t_B = TourSel_k[ rand(P_μ(z))_1, …, rand(P_μ(z))_k ]
43
44                  // Perform recombination as defined in Chapter 4
45                  (t_AB1, t_AB2) = Recomb(t_A, t_B, Ax, Bx)
```

```
46                    // Randomly select one children test to be added into the new
47                    // child population Pλ(z), whilst the other children test will be used
48                    // to conduct further variations if y variations has not been carried out
49                    t_new = rand(t_AB1, t_AB2)      // Assign one children test to t_new for more
50                                                    // variations
51                    if [ t_new = t_AB1 ] {
52                        f(t_AB2)      // Evaluate the fitness of t_AB2
53                        Pλ(z) = Pλ(z) ∪ { t_AB2}        // Add t_AB2 to Pλ(z)
54                    } else {
55                        f(t_AB1)      // Evaluate the fitness of t_AB1
56                        Pλ(z) = Pλ(z) ∪ { t_AB1}        // Add t_AB1 to Pλ(z)
57                    }
58
59                } else {
60                    // Conduct other Add, Sub, Rep, or Mut variation held by v on t_new
61                    // as per usual, and assign newly varied test back to t_new,
62                    // for further variation if y ≠ 0
63
64                    // Select position of snippet in test to undergo variation, where n is the
65                    // number of snippets in t_new
66                    i = rand(1, …, n)
67
68                    if [v = Sub ∨ v = Mut ] {
69                        // Only i is needed for subtraction and mutate variation
70                        t_new = v(t_new, i)
71                    } else {
72                        // Require the snippet to insert for addition and mutation variation
73                        s = rand(S)      // Randomly select snippet from snippet library S
74                        t_new = v(t_new, s, i)
75                    }
76                }
77
78                // Insert varied test t_new into children population
79                f(t_new)      // Evaluate the fitness of t_new
80                Pλ(z) = Pλ(z) ∪ { t_new}        // Add t_new to Pλ(z)
81            }
82
83            // Variation complete, conduct population selection and adjust variation weights
84            // according to self-adaptation
85            Pμ(z+1) = Select(Pμ+λ(z), μ)    // Select from the combined parent and children
86                                            // test population Pμ+λ(z)
87            SelfAdaptation[V, W]
88
89            // Increment evolution index and repeat evolution if termination condition not
90            // satisfied
91            z = z + 1
92        }
```

Figure E.10 SAGETEG generalised pseudo code implementation

E.8.1 Overview of SAGETEG implementation

The complete SALVEM GEA test generation platform is shown in Figure E.11. As before, SALVEM GEA testing begins by establishing the snippets library to provide test building blocks to create test programs. To do this, the applications and use-cases of the target SoC are first identified. Common SoC operations including multi device and concurrent interacting SoC functions are extracted from these use-cases, and modularised into individual ANSI-C functions each representing a snippet. Next, the test generator tool SAGETEG, uses a combination of randomisation and GEA variation operators to create test programs in terms of snippets functions via the genetic evolutionary approach.

The resultant test programs are accompanied by a set of Verilog '.dat' files that provide the necessary memory data and I/O device stimulus during test execution. The test programs are then compiled and linked with the SoC device drivers, producing the executable test binary that is loaded into executable memory for simulation on the SoC. During test program simulation, coverage and other SoC test statistics are gathered. In particular, coverage data is fed back to SAGETEG as fitness results to the test generator to create the next evolution of tests. The verification cycle stops when the GEA test generation in terminated as described in Section 4.8 Chapter 4.

In SALVEM, snippets act as genes that make up the characteristics of each test individual. SAGETEG creates and output different test programs in terms of these snippet sequences. In SALVEM test programs, the sequence of snippets is realised by ANSI-C function calls to the snippet library implementation. Essentially, the actual functional test operations performed by the snippets are interfaced through the snippet library API. This enables more compact and faster test programs whereby short snippet function calls are chained together using GEA methods. SAGETEG simply chooses which snippet to call and the parameter settings for these snippets. The actual SoC test functions stimulated are implemented using SALVEM snippet library routines and their size remains the same. Therefore, the actual SALVEM test program itself grows and shrinks in terms of snippet function calls only, and can freely vary in size according to the GEA test generator as long as it does not exceed the SoC executable memory limits.

During test execution, fitness values are directly calculated from simulation coverage measurements. Simulation is carried out using the Synopsys VCS simulator. The simulator outputs raw coverage data at the conclusion of every test execution. A separate coverage metric tool, Synopsys cmView, is then used to collate and interpret the raw coverage data into sensible and human readable form. The coverage usually employed are line, toggle and conditional coverage. These coverage data are then transformed into a suitable format for input into SAGETEG, to influence the next evolution of test

creation. The fitness measurement process is automated by wrapper software that ties all the coverage measure and raw data interpretation operations together.

The main development effort focused on the GEA test generation tool SAGETEG. Once this was completed, SAGETEG was integrated into the existing SALVEM test platform for the Nios SoC. Our efforts were limited to modifying certain SALVEM platform components only and interfacing the new test generator. The SALVEM test platform was previously established to conduct verification of the Nios SoC using the random snippets test generator initially (Appendix D), and subsequently for prototyping genetic evolutionary test generation using the semi-adapted μGP tool (Appendix E.2). Integration simply involved replacing the previous random-only test generator tool with our GEA engine; and managing the interfaces to the rest of the platform as needed. Besides the SAGETEG test generation engine, test generation also requires the snippet library specification file and the external snippet creation functions' application programmers interface (API) library, which is described further below. These files are used to configure each test generation process.

The entire SAGETEG tool was implemented in approximately 20,000 lines of C++ code, 1000 lines of lex and yacc code for the snippets library parser, and around 600 lines of code for miscellaneous scripts such as the fitness evaluation and feedback scripts. The main GEA test generator engine was developed via C++ objected-oriented design. Dividing the design of the GEA engine into separate classes enables different forms of genetic evolutionary methods to be introduced and evaluated quickly. For example, new recombination techniques and test success evaluation can be easily implemented into the GEA engine by modifying the variation and fitness evaluation classes only, without affecting the remainder of the test generator. Thus, expanding and fine tuning the GEA engine with new and different genetic evolutionary techniques or components can be conducted more efficiently.

For manipulation with snippets, SAGETEG reads in the snippets library specification file, which details the available snippets for usage, using an internal lexical scanner and parser. Hence, the snippet library specification file must be set out in a particular format described in Section E.8.2. The external snippet configuration routines API (Section E.8.3) is provided in terms of a C++ object-oriented class file, and is created specifically for whichever library of snippets are utilised. The snippet configuration routine API library is compiled and linked with the test generator either statically or dynamically. If the test generator uses different snippet library versions often, then each different external API library is linked dynamically so the test generator can swap between different snippet libraries and call the corresponding API library easily at run time. Otherwise, the external snippet configuration routine API library is compiled and linked statically with the test generator binary.

Therefore, the snippets library along with its external snippet configuration classes can be created independently from the main test generator code base. Separating the library specification and implementation of the external snippet configuration class from the test generator, different snippet libraries for different SoCs can be developed in parallel independently. As long as the relevant snippet library specification file and external snippet configuration API classes are provided to SAGETEG, different sets of GEA created tests for different SoCs can be produced easily.

The entire test generator was developed and tested on a Linux RedHat platform, the same as the rest of the SALVEM architecture. In total, the test generator required 2 months of effort (of a single engineer) for development, and another month for testing and fine tuning. However, once completed, the test generator can be integrated into various verification platforms and create tests for different SoCs in a GEA manner. The SALVEM platform was previously developed as described in [CPL05a, CPL05b] but required more effort and resources which culminated in approximately 9 months of effort.

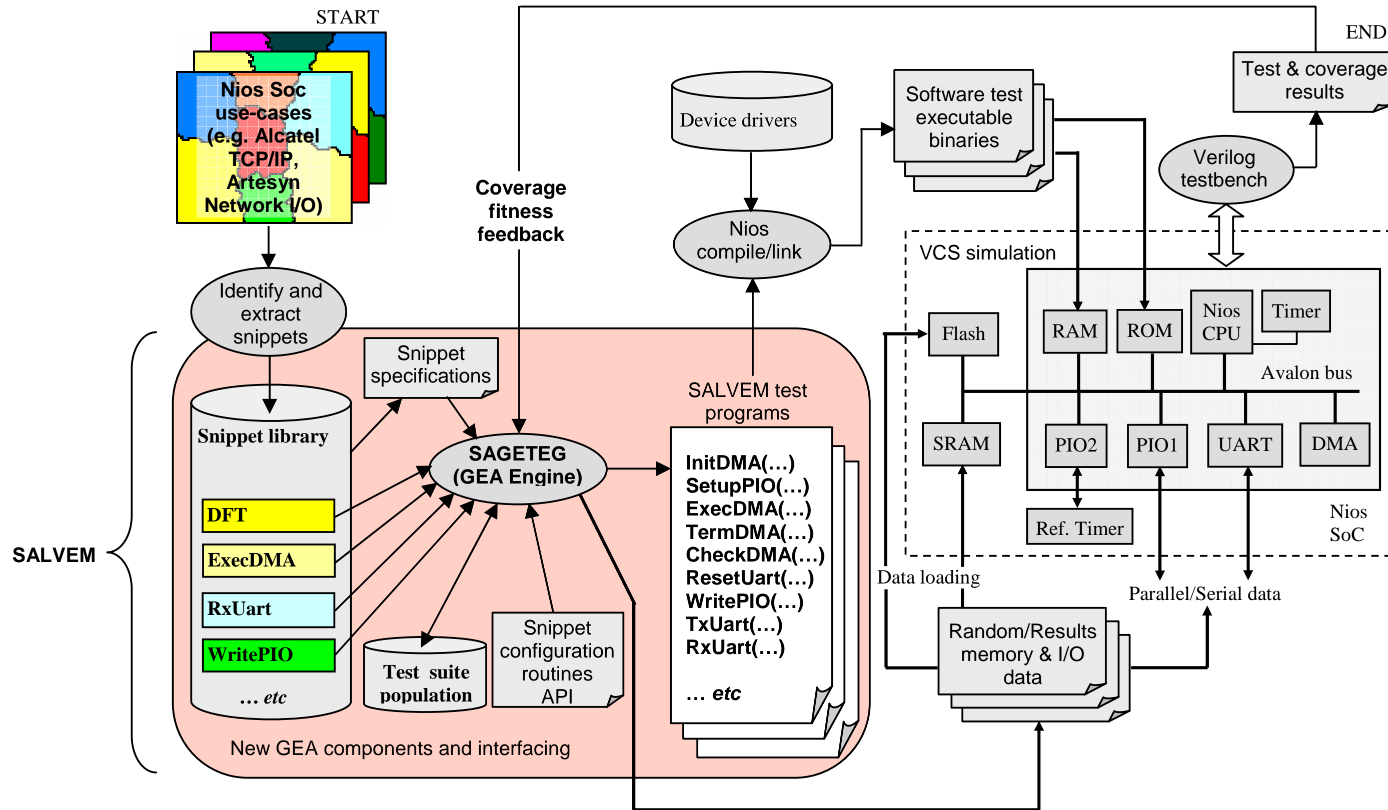


Figure E.11 Single-objective GEA SALVEM platform with SAGETEG

E.8.2 Snippets library specification file

The snippet library specification file specifies the snippets that are available to SAGETEG. In addition, for each snippet listed, additional snippet configuration is provided. Figure E.12 shows an example of a snippet specification for the RxTxUart snippet. The RxTxUart snippet invokes UART operations to test serial data transmit and receive functionalities. All snippets in the snippet library must be specified in a similar format to Figure E.12. The first portion of the snippet specification provides various snippet configuration and usage information for the test generator. The second part of the specification between keywords ‘snippet_vc:’ and ‘end_snippet’ specifies the actual ANSI-C snippet function call code, which will be inserted into the test program (lines 30 to 38). In some cases, such as the RxTxUart snippet, the function call code uses parameters which are resolved later by SAGETEG.

In the snippet specification, any parameter that needs to be resolve are pre-pended with a ‘$’ sign. The upper portion of the snippet specification (lines 17 to 26) states how that corresponding parameter is to be resolved using the ‘def_param’ parameter definition keyword. These parameter definitions specifies whether a parameter will be assigned a constant value, random value within a lower and upper range, value chosen from a set of discrete values, act as a counter, etc. Alternatively, the parameter may be assigned a value from the user-provided external snippet API function associated with the snippet. These external snippet configuration API functions are described in Section E.8.3 next. The external API functions are required for more complex snippets that require global data declarations, external I/O data streams, or parameters that need to be assigned with specific values depending on other selection decisions made for the snippet.

Prior to insertion into the test program, the test generator will look up the parameter specification and replace the corresponding parameter name with appropriate values. For example, in Figure E.12, the first parameter, ‘RxTxUart_mode’ (line 32) is resolved by the test generator choosing values from a discrete set of 0 or 1 (line 18). The remaining parameters are resolved using values chosen by the external snippet function. Using the ‘ext_snippet_fn’ keyword (line 15), an external snippet configuration routine is specified to assist the test generator to configure the RxTxUart snippet for inclusion in the test programs. The purpose and usage of the snippet configuration routine is described in greater detail in Section E.8.3.

```
1   snippet RxTxUart
2          # Tests the Transmit and Receive functions of UART device transfers
3
4          snippet_weight 20
5
6          dependency RxTxUart pre_non_strict ResetUart
7
8          # RxTxUart uses an external snippet function to assist the test generator to
9          # configure the snippet,
10         # e.g. 1) resolve snippet parameters,
11         #      2) declare global data structures (i.e. the TxUart_Buffer data to transmit)
12         #      3) generate external data stream (i.e. the external data stream the UART
13         #         receives)
14         # External snippet function pointers table index : 1
15         ext_snippet_fn 1
16
17         # RxTxUart snippet function call parameters to be resolved by the test generator
18         def_param discrete RxTxUart_mode 0 1
19         # RxTxUart snippet function call parameters to be resolved with assistance
20         # from external snippet function
21         # (All parameter values are returned by external snippet function as type string
22         #  corresponding to their given index in the table of returned parameter values)
23         def_param ext_param RxTxUart_length string 0
24         def_param ext_param RxTxUart_useEop string 1
25         def_param ext_param RxTxUart_Eop string 2
26         def_param ext_param TxUart_buffer string 3
27
28         snippet_vc:
29
30                 // UART RxTx Snippet
31                 TestBegin($testBegin_id);
32                 RxTxUart($RxTxUart_mode,  // Transfer mode (interruptible?)
33                          $RxTxUart_length,    // Transfer size
34                          $RxTxUart_useEop,  //  Transfer terminate by end-of-packet?
35                          $RxTxUart_Eop,       //  end-of-packet byte, if used
36                          $TxUart_buffer  // buffer of data stream to transmit
37                         );
38                 TestEnd($testEnd_id, 1);
39
40    end_snippet
```

Figure E.12 Example of a snippet definition in the snippets library specification file

In Figure E.12 at lines 31 and 38, the ‘TestBegin’ and ‘TestEnd’ function call is used to signal the start and end of a snippet operation, in this case, a RxTxSnippet snippet. These functions are used purely for logging and test statistical purposes. The ‘testBegin_id’ and ‘testEnd_id’ (lines 31 and 38) are global test program parameters defined at the beginning of the specification file to keep track of the snippets executed.

Each snippet specification may also include dependency and constraint rules that must be adhered to by the snippet. For example, the RxTxUart snippet has a pre non-strict dependency on the ResetUart snippet (line 6). This dependency states that the UART must be initialised at least once prior to any UART transfers conducted by the RxTxUart snippet. These dependencies were formally described in Section 4.5.2 at Chapter 4.

Another feature of the snippet specification library file is the use of selection probability weightings for each snippet. During test generation, when the GEA process creates initial test individuals or snippets are added using variation, selection weights influence which snippets are more or less likely to be chosen for inclusion into the test program. This enables user-directed testing initially, to drive the test generator to create tests that focus on specific SoC devices or operations. For example, at line 4, the RxTxUart snippet is assigned an initial snippet weight of 20 using the 'snippet_weight' keyword. All snippets in the snippet library specification file are assigned an initial snippet weight. During test generation, snippet selection weights are then automatically adjusted according to the self-adaptation strategy from Section 4.5.10 Chapter 4.

A similar biasing capability was also provided in the random-only SALVEM test generator to influence what snippets are chosen. However, providing selection probability weights in conjunction with a GEA process implements a guided random search technique. The test programs created will eventually evolve to an optimised state under the initial direction of user specified snippets and re-use of high coverage tests fed back from previous test generations.

The complete set of snippets in our snippets library for GEA testing of the Nios SoC is the same as that from Table E.4. Higher probabilities values are given to snippets that invoke SoC device transactions; in particular, snippets that perform operations concurrently with multiple devices such as the DMA snippets. Our snippets library includes the discrete Fourier transform (DFT) and inverse discrete Fourier transform (IDFT) snippets that perform DSP related operations to test the CPU core. These snippets were in fact created to test a digital signal processing (DSP) SoC previously [CLS+08], but could be reused for testing the Nios SoC here. Similarly, snippets such as the DMA snippets developed for the Nios SoC can also be used for DSP SoC testing as the DSP contains a DMA device as well (Section 4.12 Chapter 4 describes SALVEM testing of this DSP SoC). Despite being created to target specific SoCs, existing snippets can often be reused to help build up the snippet library for other different SoC testing. Thus, providing significant benefits for reducing engineering effort, whilst ensuring comprehensive snippets libraries can still be created.

Finally, besides the list of snippets and their configuration data, the snippets library specification file also contain global test program configuration information. Any parameter definitions, dependencies, or constraints not enclosed within a snippet specification (i.e., within keywords 'snippet' and 'end_snippet') is considered global, and applies to the entire test program during test creation. The specification file also specifies setup and cleanup code that must be inserted verbatim at the beginning and end of each test program. The setup code is needed to provide various device driver 'include' header file statements, the top-level test program 'int main (...)' routine opening, and other global declarations needed in the ANSI-C based SALVEM test program. Before the test program concludes, any remaining snippet checking code, and SoC device reset or termination code will be placed in the cleanup section.

E.8.3 External snippets API and snippets configuration routines

In the snippet library specification file, a snippet can specify its association to an external snippet configuration routine using the 'ext_snippet_fn' keyword followed by an integer index (line 15 in Figure E.12). The integer index is used to select from a list of pointers, the corresponding C++ snippet configuration routine function in the external snippet configuration class. The external snippet configuration class is a C++ class file that can be compiled into an API library and externally linked to the main test generator code. This enables the SAGETEG to call upon a snippet's associated external function when that snippet is selected for insertion into the test program.

The aim of external snippet configuration routines is to supplement the setup information of snippets listed in the snippet library specification file. A snippet's configuration routine will (1) select appropriate snippet parameter values, (2) create any additional test program data structures, and (3) generate external simulation data streams to be used by the snippet function during SoC operation.

For simple snippets, the snippet function call code can be simply inserted into the test program selecting various random parameter values independently on-the-fly. However, for more complex snippets, the snippet operation carried out requires the snippet function call parameters to be chosen carefully. The snippet could be restricted to operate under certain conditions only. If so, selection of snippet parameters may be constrained or depend on other snippet operation options chosen by the test generator earlier. Specifically, a snippet parameter value may depend on other previously chosen parameter values. For example, for UART snippets, if UART end-of-packet termination is chosen for the transfer termination parameter, the test generator must ensure the end-of-packet byte character is

assigned to the end-of-packet parameter. SAGETEG calls the snippet configuration routine to ensure these dependent parameters are assigned appropriate values.

Besides parameters, certain snippet operations may require additional configuration data to be generated. These may include global data structures at the start of the test program or external streams of byte data fed to the SoC I/O devices via Verilog '.dat' files. For example, many CPU matrix based snippets require array data structures in order to perform their required matrix operations. These array data structures are generated by the external snippet routine. They are usually declared at the global level at the start of the test program and used by the matrix snippets functions. The global arrays provide the inputs to the matrix snippets, and also the expected results to check against outputs from the matrix operations. For UART snippets, the receive and transmit data byte stream are also generated by the configuration routine. These data streams are also used as inputs and for expected results checking of UART snippets; to demonstrate serial transfers were conducted correctly.

Figure E.13 shows the relationships between the external snippet configuration routines class, snippet library specification file, test generator engine (SAGETEG), and the test program when a UART snippet is configured into the snippet sequence of a test. The dotted arrows indicate which modules in the test generation stage are responsible for certain components of the test. As described above, the actual UART snippet function call (excluding parameters that need to be resolved) is copied verbatim from the RxTxUart specification in the snippet library specification file, indicated by dotted arrow A.

To resolve parameters, the UART snippet function parameter values are provided by its corresponding configuration routine in the external snippet configuration routines class (dotted arrow B). The external snippet routine also defines the global data structure declaration as indicated by dotted arrow C. In this case, the global data declaration is an array of byte characters that the UART snippet function will transmit. Finally, dotted arrow D shows the external data stream of characters that will be used during SoC simulation for testing the UART receive function. Again, the external UART snippet routine is responsible for generating the data stream file.

In Figure E.13, the generation of global test program data structures and external data streams must also take into consideration various dependencies and constraints with the chosen snippet parameters, and vice versa. For example, if UART end-of-packet termination is to be used, the selected end-of-packet data byte assigned to the end-of-packet parameter must be generated as part of the global test program transmit array and UART receive external data stream. The aim of the external snippet configuration is to select a range of different snippet operations for the test programs each time; whilst

ensuring that the appropriate parameters, global test program data structures, and external data streams are generated by the test generator to satisfy any constraints or dependencies.

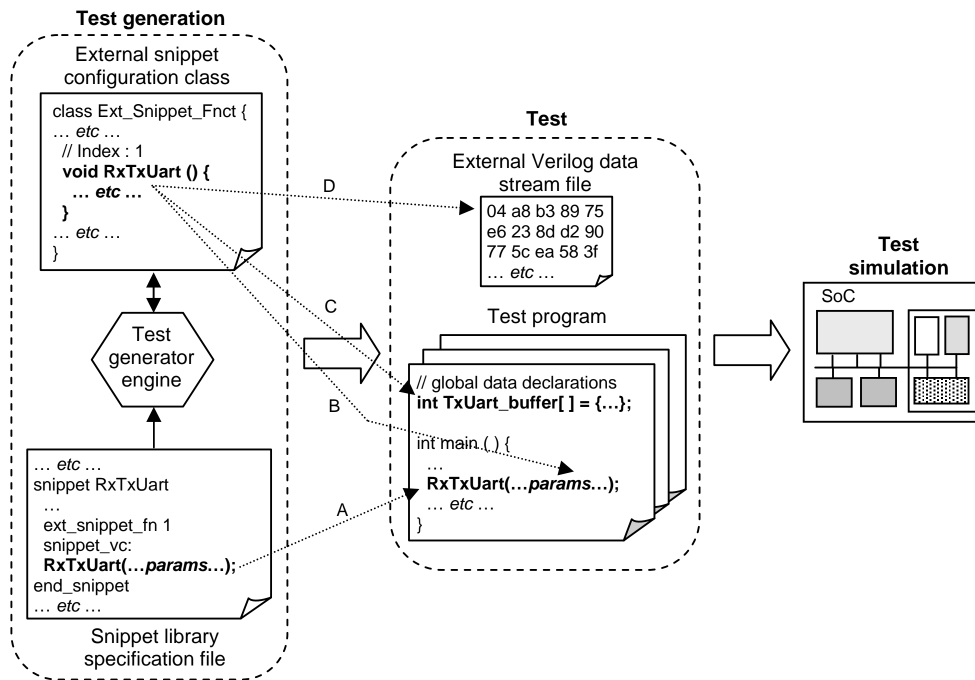


Figure E.13 Test generation outputs to facilitate test creation and simulation

The advantage in supplementing the test generator with external user-defined routines, is that any complexities in the configuration of snippets into test programs can be handled separately and specifically for each snippet; and independent of the test generator as well. The test generator can focus solely on the test program creation process itself. For verification of other SoCs, whenever more complex or different snippets for other SoCs are added into the library, the test generator does not need to be modified. Once the snippet API library functions and SoC device drivers are updated, only the snippet library specification file and a new external snippet configuration routine needs to be provided. This enables more straightforward upgrades of the snippets library with new snippets, and the test generator can be easily applied for different SoCs using different snippets library.

E.8.4 Other features of SAGETEG implementation

In addition to the test generation features above, other amendments were introduced into the genetic evolutionary process. These modifications are considered minor and can be controlled by the user before execution of the test generator. The aim is to try and enhance test generation by introducing slight deviations into traditional GEA flows and conventional parameter options. Firstly, the number

of variations applied to create a new test during each evolution is not fixed. SAGETEG may apply one or more different types of variation on an existing test from the current population to create the new test for the next evolutionary population. Prior to test generator execution, an upper limit is placed on the number of variations each test can undergo. Then during test evolution, each new test will undergo random variations each time as long as the number of variation operators applied does not exceed this limit. This mimics real-life evolutionary processes more closely, as new individuals from each evolutionary stage experiences far more complex interactions and modifications, rather than a single variation as was previously applied in earlier test generators [CCRS03a, CCRS03b, CCS03, CL07, CLS+08].

Furthermore, like proper evolutionary processes, individuals have a limited life span. Therefore, our test generator can be configured to impose limited life spans on tests. Previously, as long as any test does not fall into the bottom group of low coverage fitness, the test will be retained and continue to survive through future evolutions regardless. Hence, it is possible for tests to be created during the initial evolution and live till the end of the test generation process. Imposing a limited life span into test generation, any test regardless of its high fitness quality, will be removed from the test population whenever it has survived through a number of evolutions that exceeds its allowed life span. Imposing limited test life spans allows the test population to be refreshed with completely new tests from time to time. Otherwise after many evolutions, the test population may simply be over dominated with certain types of tests. Whilst these tests may provide the best coverage, no further improvement may be possible, and their unlimited life spans would prevent any significantly different tests from being introduced. New tests in future evolutions would simply be created by some small variant of these immortal tests.

There is one exception to enforcing limited lifespan. During test generation, if the best test that attains the highest coverage fitness thus far has outlived its life span, despite eliminating that test from the population, the test generator will still keep a copy of that test for future testing purposes. These best tests can then be analysed to identify what kind of SoC operations yielded the best coverage, and can even be reused in initial test population for future test generations. The test generation can be configured to keep track of the best x number of tests, for example, the top three tests.

For recombination variation, another modification introduced was the use of variable tournament selection sizes. Rather than be limited to rigid two individuals tournament selection previously [CCRS03a, CCRS03b], SAGETEG uses fluctuating number of individuals in our tournament selections. This provides more variation in the range of parents selected for recombination. Note that

the number of individuals used in tournament selection is also adjusted using the self-adaptation method from Section 4.5.10 Chapter 4. This allows automatic self-adjustment by the test generator to seek the best tournament selection size over the course of the test evolutionary process. In contrast, the original μGP approach was restricted to a fixed two member tournament selection scheme. Although, this was later extended by Sanchez [SSR+05] to incorporate an adaptive size similar to our approach above.

Finally, for the initial first evolution test population, SAGETEG can be specified to create these tests with different initial population sizes. Usually, the initial tests are created to contain a very low number of snippets, possibly even no snippets to represent an empty initial test. This allows the test generator to fully control the evolution of these tests. The test population will be *grown* entirely by variation to attain the maximum coverage efficiently, without using tests that are larger than necessary and wastefully taking up simulation time.

E.9 SAGETEG implementation comparisons with μGP

Given our implementation of GEA test creation techniques in SALVEM, a review of test generation features and comparison to other test generator such as μGP was conducted to highlight differences and identify any improvements that could be built into SAGETEG in the future. Firstly, the μGP uses instructions from a microprocessor's instruction set architecture as building blocks to create assembler instruction test programs. The μGP's instruction library is fixed and limited by the types of instructions that are defined for a processor. Each of their test building block instructions exercises units on the processor only. In our case, we employ a library of carefully crafted snippet functions. Despite having to create these snippet functions, our library of snippets is expandable and can invoke operations to test standalone devices or multiple system-wide interactions amongst different modules on the SoC. New snippets can be created or re-used from other similar SoC verification platforms for inclusion into the snippets library. The snippet library can be enhanced when needed and is not fixed.

Moreover, to add new snippets into our snippets library, each new snippet can be associated with an external snippet function that assists in the configuration and insertion of that snippet into test programs (Section E.8.3). This external snippet configuration routine is simply defined as part of an external C++ class that is complied and externally linked into the test or as an API library. For the μGP, any change in the instruction library would be limited to creating different shorter length sub-sequences of existing instructions that are captured in their *macros*. However, these macros are still

reliant on to the processor instruction set. If the processor architecture is not revised or the associated instruction set not updated, then the test capabilities of these macros are not modified at all. This is not surprising given the much simpler nature of the instruction test building blocks compared to our SALVEM snippets. Given the types of lower level processor operations conducted, the μGP only requires variation in instruction operands. Unlike SALVEM GEA test generation, other external data or complex parameter assignments are neither possible nor necessary.

In SAGETEG, well-defined dependency and constraint rules are specified to ensure test programs are created properly by variation. For the μGP, no clear dependency rules are defined. However, it is clear that under certain conditions, certain sequences of instructions must be executed. To enforce this, the μGP uses macros in their instruction library to define these instruction sequences and inserts them into tests. These macros facilitate similar pre and post strict dependencies in an ad-hoc manner. Like snippets, it is also obvious that some instructions do not need to be executed immediately before or after certain instructions. In SAGETEG, pre and post non-strict dependencies were used to complement existing strict dependencies whenever required snippets can be executed anywhere before or after the current dependent snippet in the test program. However, in the μGP, only strict dependencies can be enforced.

Similarly, in the μGP, no constraint rules are formally defined. The only constraint placed on μGP test creation is the prevention of critical branch or jump based loops being broken, or the formation of infinite loops by variation operators. In contrast, SAGETEG must follow a list of stringent rules when using variation or choosing snippet parameters. The lack of properly defined dependency and constraints rules in μGP is due to the simplistic nature of their instructions building blocks. The snippets and eventual SALVEM test programs are much more complex in the types of SoC processes it initiates. The operations invoked must be able to test a range of legal, unexpected and erroneous scenarios, whilst ensuring the test program can be executable and do not result in simulation dead-locks.

Both SAGETEG and μGP employ self-adaptation for various test creation parameters such as selection of variation operators during the test creation process. In both test generators, the self-adaptation technique was initially based on Rechenberg’s rule. However, unlike μGP, in addition to variation operators, SAGETEG also applies automatic self-adaptation to other test creation parameters such as snippet selection probabilities and recombination variation tournament selection sizes. In addition, for SALVEM GEA test generation, we revised our own self-adaptation method specifically to match the types of test population and test creation processes that provide better adaptation of test parameters.

For the μGP, their self-adaptation method also differs slightly. μGP employs 'inertia' in its test variable adjustment scheme. Even if test variable changes are suggested by Rechenberg's rule, μGP will not modify any variable too greatly due to their inertia factor.

In terms of test variation, both test generators employ similar operators as per any conventional GEA process. However, SAGETEG allows for greater flexibility in the way the variation operations are carried out internally. For example, recombination variation uses parents selected from tournament selection with changing sample sizes. Also, instead of single variation, a SALVEM test may undergo multiple variations during each evolutionary stage. And finally, unlike μGP, SAGETEG can be configured to select which individual variations are more or less likely to be employed initially. In μGP, only a single probability weighting can be selected, and all variation operators will be assigned this same value. As explained in Section 4.5.8 Chapter 4, to ensure the test population evolves in the best manner, the initial selection probability weightings of each variation operators must be specified.

In fitness evaluation, both μGP and SAGETEG may use multiple fitness measures for evaluation of test effectiveness during the GEA process. Specifically for test generation, both methods use various coverage metrics such as line, toggle or conditional coverage as test fitness quantifiers. In the μGP, multiple coverage fitness values can also be used simultaneously for the same single test generation. The μGP uses a ranking system whereby certain coverage metrics are given higher priority. When evaluating and comparing fitness amongst tests, higher priority coverage metric will be used first before examining lower priority ones. For SAGETEG, each of these coverage metrics can be used independently to drive each test generation process, and this is the approach taken for experimental purposes in Chapter 4. Our aim was to maximise each coverage metric. This is best achieved by using a single coverage metric as the sole fitness quantifier and running the test generation independently. A test generator that is driven by a single coverage metric is much more effective at maximising that metric compared to creating tests that needs meet multiple coverage criteria. To assess the quality of the test suite, μGP also uses methods based on De Jong's on and off line performance measuring. In SAGETEG, a more simplistic averaging technique is used to calculate test suite quality instead. Rather than simple ranking, to truly consider multiple fitness for driving test generation, a proper multi-objective GEA process such as that described in Chapter 6 is much more effective.

For test evolutionary termination, SAGETEG implements an additional termination criterion compared with μGP. Whenever the resultant fitness in the test population does not improve for certain consecutive number of evolutions, the test generator process will end. This additional termination check helps ensure the test generation process does not continue needlessly without the test population

actually enhancing test coverage. The termination condition indicates when the test population must be refreshed again and the test generation process should be restarted. Using a new initial test suite, the test generator may be led to other unverified portions of the test space previously not possible with current stagnated test populations.

Other significant differences between the μGP and SAGETEG are as follows. μGP provides a test population save and restore feature, which allows the test population from any evolutionary stage to be saved. Then later, test populations can be restored and used as the initial population for future test generation runs. The main advantage from this is when μGP employs a test suite assimilation tool as well. Sanchez et al. uses the save and restore feature to propose a method whereby old tests from previous design and verifications phases can be reused to refine and create new tests for future design and verification phases [SRS05]. Alternatively, the save and restore feature can be used to merge new tests with regression tests. When new tests are created for newly implemented hardware design features, these new tests can be assimilated into the previous test suite population. The combined test population can then be used as the initial population to begin verification of the new hardware feature. In this way, new design features can be tested better with regression testing. In our case, such save and restore features or assimilation of test suite between verification phases is beyond the scope of this thesis currently. However, a save and restore capability can be easily implemented as each test is internally represented using a simple data structure; which can be simply transferred via simple text files.

μGP also enables parallel test generation of evolving test suites using multiple simultaneous fitness evaluation and test population save and restore capabilities. However, for SAGETEG, such parallel test runs was not deem important given the sufficiently fast simulation speeds that is available, even from moderately powerful computers today. Additionally, snippets tests are not overly large and can be simulated quickly. At this stage, adding parallel test generation capabilities would not add much value into our test generator.

Finally, unlike the μGP, SAGETEG is able to revert back solely to a pseudo random test generator similar to many other random test creation tools proposed previously. In this configuration, all test creation decisions are purely random without any influence from GEA. No tests are retained or discarded, and the test generator simply creates all tests randomly to make up the specified number of tests needed to fill the test suite. The aim of providing pure random-only test generation capability is to gauge the original SoC coverage using a basic test suite without any GEA influence. Once an initial estimate of SoC coverage is obtained, the effectiveness of SAGETEG and coverage improvement

attained from the GEA process can be better measured. This also assists in the fine tuning process to check whether GEA enhancements add value to the test creation strategy, which also allows for more complete comparisons for experimental purposes.

The aim of the above SAGETEG usage and operational features are to provide greater control in how the GEA test generation process is conducted for SALVEM. This enables a wider range of fine tuning options to maximise verification effectiveness of SAGETEG to suit different SoCs.

E.10 Snippets employed for SAGETEG test generation for the Nios SoC

Table E.4 lists the snippets and their selection weightings for verification of the Nios SoC using SAGETEG. Selection weightings influence which snippets are chosen into test programs. Higher weights are given to snippets that invoke concurrent SoC operations or execute intensive SoC functions, such as CPU or DMA device snippets. Lower weights are given to reconfiguration or monitor snippets that setup devices for functional executions or check expected results. The snippets in Table E.4 represent an adequate set of SoC functions for stressing the Nios SoC. Each snippet can be mutated with many different input parameters to invoke a wide range of SoC functions to test. Indeed, the benefit of the snippets approach is that the number and types of snippets that make up the snippets library should not be more than necessary. This enables optimal test program sizes and efficient test simulations. Note that unlike the μGP feasibility study test generation in Appendix E.2, SAGETEG is able to use the entire set of snippets as is, without any needless modification or mapping of the snippets library.

Table E.4 Snippets employed for SAGETEG test generation

Snippet	Weight	Function
InitDMA	15	Configures DMA for transfer
ExecDMA	20	Executes and monitors DMA transfer
TermDMA	10	Terminates DMA transfer
CheckDMA	10	Validate DMA transfer success
ResetUart	10	Initialise UART device
TxUart	20	Transmit serial data
RxUart	20	Receives serial data
RxTxUart	20	Duplex serial data transfer
RxTxNDupUart	20	Non-duplex serial data transfer
GenRandNumSeq	20	Generates random number sequences on CPU
MatrixMultiply	20	Performs fast matrix multiply on CPU
MatrixInverse	20	Performs matrix inverse on CPU
Search	20	Performs large intensive search algorithms on CPU
Sort	20	Performs fast binary, bubble, etc sorts on CPU
Convolve	20	Performs DSP signal convolution function
DFT	20	Performs DSP discrete Fourier transforms
InvDFT	20	Performs DSP inverse discrete Fourier transforms
WriteMemory	20	Executes data units writing to on/off chip memories
ReadMemory	20	Executes data units reads from on/off chip memories
WriteReadMemory	20	Executes writes/reads of on/off chip memories
TestMemoryLogic	20	Checks correct operations of memories control logic
WriteROM	20	Specialised on-chip ROM testing
MissAlignedAddr	20	Invokes misc. memory error conditions handling
RestartTimer	20	Initialises the timer
ReadTimer	20	Checks correct timer operation and accuracy
SetTimerPeriodAutoRestart	20	Reconfigures timer parameters to restart timer ops
SetTimerPeriodNoAuto-Restart	20	Reconfigures timer parameters and delay timer ops
StopTimer	20	Terminates timer operations
SetupPIO	10	Initialise or clears PIO pins
ConfigPIODir	10	Re-configures PIO data directional pins
WritePIO	10	Transfers parallel data

E.11 Coverage results for Nios SoC devices

Table E.5 reports the coverage results for individual devices on the Nios SoC from experimentations in Section 4.11 Chapter 4.

Table E.5 Coverage result for individual devices on Nios SoC

Line coverage %	**CPU**	**DMA**	**Memory**	**Misc. SoC units**	**PIO**	**Timer**	**UART**	**SoC**
SAGETEG	99.8	100.0	100.0	96.7	96.2	100.0	96.7	98.9
µGP	98.2	100.0	100.0	93.0	98.7	79.2	94.9	97.5
Randomised	98.6	99.4	67.6	50.5	98.3	97.6	96.0	89.9
Manual application	61.4	98.5	35.8	75.4	82.0	100.0	95.0	66.6
Toggle coverage %	**CPU**	**DMA**	**Memory**	**Misc. SoC units**	**PIO**	**Timer**	**UART**	**SoC**
SAGETEG	96.3	97.5	96.1	92.3	89.5	54.6	71.3	93.7
µGP	89.6	95.6	94.2	86.6	89.5	14.6	58.6	87.1
Randomised	75.6	97.0	82.3	90.5	89.5	61.7	58.4	79.2
Manual application	49.2	78.9	21.8	77.5	49.5	14.6	57.6	48.8
Conditional coverage %	**CPU**	**DMA**	**Memory**	**Misc. SoC units**	**PIO**	**Timer**	**UART**	**SoC**
SAGETEG	82.5	98.5	N/A	89.2	75.0	88.5	85.3	83.0
µGP	72.1	93.9	N/A	88.6	75.0	28.8	72.3	72.4
Randomised	68.5	87.9	N/A	90.8	75.0	88.5	74.5	69.3
Manual application	65.3	80.3	N/A	77.3	37.5	82.7	66.3	65.6

Note : Memory units do not contain any conditional design code descriptors, hence are not applicable for conditional coverage measurement..

E.12 Coverage attainment above 80%

In Chapter 4, Section 4.11.2 described experimental results for SAGETEG whereby full coverage was not achieved. Whilst the SAGETEG gain over other test generation methods may not appear outstanding, especially for line coverage, one must consider these coverage results from a general verification perspective, taking into account the coverage levels already achieved.

To put results into proper context, for verification in general, attaining the final 5 to 10% coverage is exponentially difficult for any hardware design, let alone SoCs. Therefore, even a small percentage coverage gain above 85 to 90% is considered extremely valuable when compared to other methods. The fundamental impediment against coverage gain is that coverage increase is non-linear. It becomes exponentially difficult and displays a exponential or saturation characteristics as the coverage gets closer to 100%. Therefore, the coverage gain by SAGETEG over other methods is in fact highly beneficial.

From a genetic evolutionary algorithm perspective, SAGETEG GEA variation seeks out a range of different test regions, some of which are usually disjoint and independent. Some test regions may yield higher or lower coverage, hence account for variation spikes in the raw coverage progress graphs (Figures 4.12 to 4.14, Chapter 4). But the goal is to seek out unexplored test regions to creep closer toward full coverage. Therefore, the GEA process explores one region to another trying to seek out local maximums; if any are uncovered, variation weights still allow for test generation to seek out other regions. Over the GEA process, the aim is to sort out and only examine the more fruitful regions, to search for those local maximum and eventually attain the overall maximum for the current GEA process. At this point, the coverage is likely to saturate, and a new GEA process (test generation) must be restarted. The purpose of SAGETEG (and the GEA process) is allow for repeated executions so that further gains toward full coverage can be facilitated; even if each execution may yield low or non-existent gains because of the exponential difficulty with conquering the remaining 5 to 10% coverage. Eventually, with new snippets libraries and initial test populations, full coverage should be possible.

Also, our use of GEA is slightly different from conventional usages of GEA in problem solving or optimisation problems. Even though we define a definitive function or problem formulated objective equation to solve, our aim is not solely restricted to finding the one best solution that optimises or solves the particular function problem. We also make use of the GEA process to help enhance and seek out best coverage of the overall test suite (i.e., seek out as many fruitful high coverage yielding test regions). By targeting such a goal as well, at the end of test generation, the test suite will contain greater majority of high coverage tests that collectively provides greater overall cumulative coverage of the SoC.

E.13 Supplemental coverage progress graphs for SAGETEG

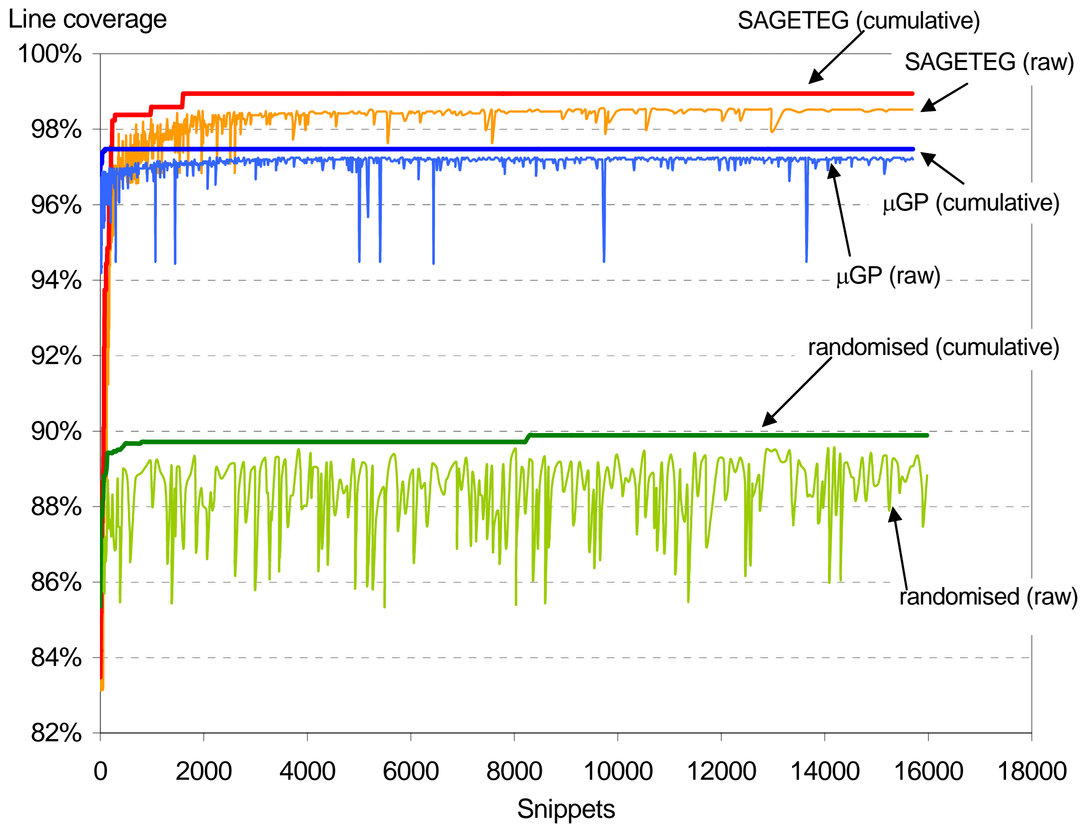


Figure E.14 Line coverage versus snippets

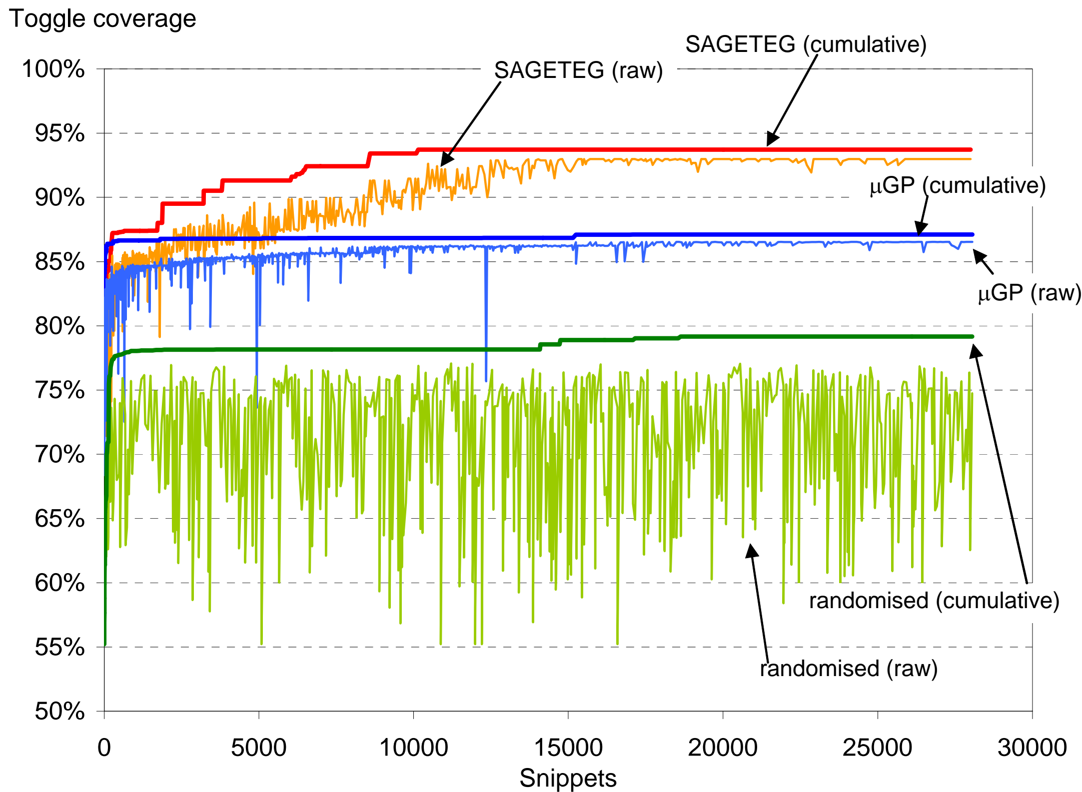


Figure E.15 Toggle coverage versus snippets

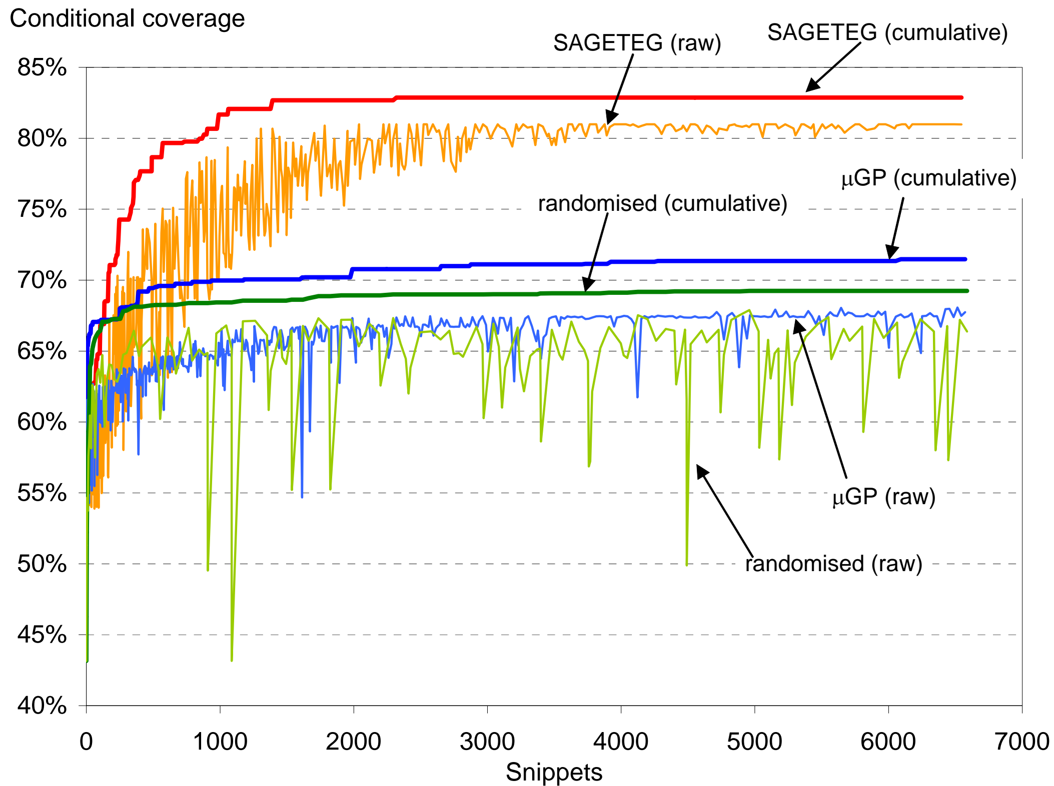


Figure E.16 Conditional coverage versus snippets

E.14 Genetic evolutionary effects on test generation efficiency

Despite obvious advantages using a genetic evolutionary method for test generation by SAGETEG, some penalty is incurred for conducting GEA operations. Attaining a higher (or equivalent) coverage using an optimised or smaller set of tests enables efficient test execution, but this is offset partly by resources required to carry out more complex evolutionary test create operations. In Appendix D, a random-only test approach simply selects random snippets and snippet parameters to form test programs. An evolutionary scheme however, requires consideration of preceding test generations and the types of tests created previously. Previous test programs are analysed to influence the types of snippets chosen for composing new tests. For example, snippet sequences from previous higher coverage yielding test programs are reused more often. Managing the population of prior tests and conducting snippet sequence analysis requires greater compute resource requirements.

Furthermore, besides simple random and mutating operations, other more complex variations such as snippet addition, removal or recombination are employed by SAGETEG. To form test programs using these operators, the test generator must ensure any modifications to existing snippet sequences do not violate any constraints or dependencies, i.e., the new test must still be executable. For instance,

recombination of two parent tests to form child test programs is complex. A suitable cross over point must be selected from both parents and checked for validity to ensure legal child test programs are created. Additionally, snippet variation dependency and self-adaptation considerations must be processed (Sections 4.5.8 to 4.5.10, Chapter 4). The processing time and resources needed to conduct these operations are greater than simply choosing test entities randomly. Maintaining and manipulating the population of tests including fitness values, and performing more complex variation and population selection operations implies the overall GEA test generation would take longer. However, these perceived shortcomings are considered acceptable trade-offs given the enhanced coverage that can be achieved.

E.15 Test generation effectiveness factor

To objectively demonstrate the usefulness of SAGETEG, we consider its coverage, time and memory usage results together by defining and evaluating an effectiveness factor metric. The goal of evaluating this effectiveness factor is to examine how much coverage improvement a test generation technique achieves, taking into consideration time and size factors. The factor is defined in Definition E.3. It considers the number of tests that achieved coverage improvement, and for each such test, the factor is directly proportional to coverage gain but inversely proportional to time and size usage.

Definition E.3 : Test generation effectiveness factor

Let n be the number of tests that achieved coverage improvement from the particular test generation method, the effectiveness factor is given by,

$$\sum_{i=0}^{n}(g_i + \frac{1}{t_i} + \frac{1}{s_i})$$

where g_i is the gain achieved by the ith test,
t_i is the test execution and test generation time required by the ith test, and
s_i is the memory size usage of the ith test.

□

The effectiveness factor examines the coverage gain achieved by each test, and weighs up this enhancement against the time and size trade-offs required to attain such gain.

The effectiveness factor calculated for all results from SAGETEG, μGP, and randomised testing are 0.31, 0.06, and 0.01 respectively. This supports the notion that SAGETEG performs most effectively to achieve best coverage given trade-offs in time requirements and memory size usage. The low effectiveness of μGP and randomised is largely attributed to their lower coverage results, and the excessive number of tests and snippets employed to gain this coverage.

E.16 Applying SAGETEG to a DSP SoC – supplementary details of the case study

This appendix provides additional details for the case study in Section 4.12 Chapter 4, whereby the SAGETEG test generator is employed for verifying a DSP SoC.

E.16.1 Introduction

A digital signal processor (DSP) system-on-chip (SoC) can be designed using a variety of architectures and techniques. This often presents different verification challenges compared to conventional SoC or processor designs. Verification of such designs should take into account the goals and applications of the DSP, and how they are eventually used. The case study proposes an application based verification methodology (SALVEM) along with a genetic evolutionary test generator (SAGETEG) to demonstrate this technique on a real-life DSP SoC design.

Designing digital signal processor (DSP) SoCs also create other verification complexities given the range of applications and end products DSPs are employed within. In order to test a DSP design more effectively, consideration must be given to how the DSP will be used and their intended applications. The eventual real-life usages of a DSP determine the particular design features and functions that are needed in the DSP. It is these design functionalities and their complexities that must be verified in-depth. Therefore, any effective verification strategy must incorporate extensive testing with application functions.

To demonstrate this, the software application level verification methodology (SALVEM) is employed to test the Tsinghua University Application Specific DSP (THUASDSP2004) [ZHZ+06]. The SALVEM technique was successfully used on the Nios SoC previously. The aim of this case study was to describe the application of SALVEM on a real world DSP SoC; thus demonstrate its feasibility

and usefulness for DSP testing. Furthermore, for verification of the DSP, SALVEM is enhanced by an automated test generator (SAGETEG) that uses genetic evolutionary methods to create tests.

The THUASDSP2004 DSP is an ideal candidate for SALVEM. It was designed specifically for multimedia applications and contains common DSP function blocks such as high performance mathematical and fast data transfer units, along with other specialised modules. These DSP architectural features are to be tested by SALVEM to enhance the design and verification quality of the DSP SoC.

Previously, certain DSP design and modelling environments like Matlab or Simulink do not describe the true hardware design implementation that will be eventually tape-out. They focus on high level DSP algorithmic validation, but the actual hardware design is not tested directly [Dau05]. Another DSP verification solution from Coware [CoW] allows for hardware design testing. However, to use Coware, designs described in Matlab must be synthesised to an equivalent hardware description using the AccelChip synthesis tool. These DSP test solutions are not suitable for all designs. Our THUASDSP2004 SoC contains different architectural features from conventional DSPs, and does not use the AccelChip synthesis flow.

Our approach is to employ the SALVEM technique with an inbuilt genetic evolutionary test generator. The aim of SALVEM is to create tests based on the application use cases of the SoC. Hence, important functionalities critical to the real-life operations of the SoC are guaranteed to be tested and verified.

E.16.2 Description of the DSP SoC

The Tsinghua University Application Specific DSP (THUASDSP2004) SoC [DHZ+05, ZHZ+06] is made up of a clustered very-long-instruction-word (VLIW) processor core for conducting DSP operations, memories, I/O modules such as interrupt and memory controllers, and a DMA for transferring large signal data. Figure E.17 shows the SoC architecture.

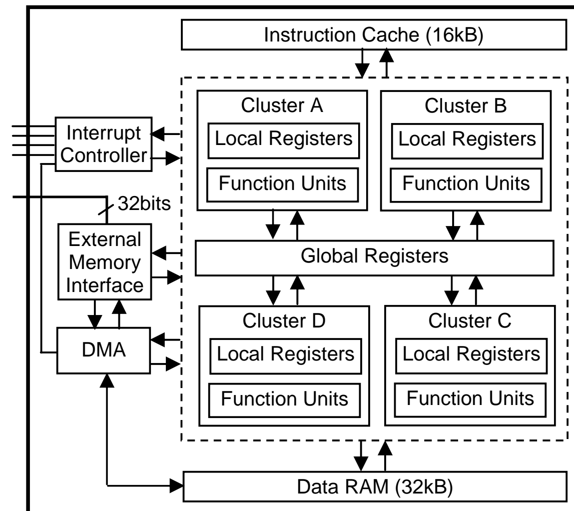


Figure E.17 THUASDSP2004 DSP SoC architecture

The THUASDSP2004 is designed to be configurable for use in various multimedia applications similar to a field programmable gate array (FPGA) based design, whilst delivering high performance matching that of an application-specific integrated circuit (ASIC) DSP [DHZ+05, ZHZ+06]. To achieve this, the DSP is built upon a clustered VLIW platform whereby different SoC functions can be grouped into different clusters or sub-divided into different function units. Adopting such an approach enables scalability and flexibility. Depending on the intended usages and requirements of the DSP, the design can cater for different numbers and configurations of clusters or function units [ZHZ+06]. Hence, resources such as instructions and register usage can be distributed across function units or register files. Instruction level parallelism and optimal performance is also achieved for the range of intensive multimedia applications.

Different clustered architectures will require different test conditions; therefore, any viable DSP verification approach must be able to accommodate these configurations. The SALVEM approach uses parameterisation in the snippet test building blocks to cater for various configurations, and adds new snippet test building blocks when clusters or function units are added or modified. Different snippet blocks and different snippet parameters creates new test programs each time the DSP architecture changes.

Signal processing requires intensive numerical calculations and large data array storage and transfers. Therefore, the THUASDSP2004 DSP contains specialised arithmetic logic (AL), multiply (ML), address branch (AB), and load/store (LS) function units specially designed for such DSP operations.

Furthermore, the THUASDSP2004 DSP also implements a unique register file based inter-cluster communication system [ZHZ+06]. In the DSP, a global register file is used to network clusters, and facilitate data transfers between function units. Whenever data produced by one cluster is needed by another, the data will be duplicated in the global register file and can be gathered by the other cluster. The advantage from such an implementation is that all data transfer delays and conventional bus cycle latencies are eliminated. Using this communication scheme amongst function units speeds up software pipelining, removes loop anti-dependencies, and enhances instruction level parallelism [ZHZ+06]. Furthermore, different configurations of clusters are connected easily with this register file.

Given these unique DSP features, SALVEM makes use of individual snippet building blocks to test specific operations of the DSP's function units, and also the global register file inter-cluster bus system.

E.16.3 Snippets library for DSP SoC verification

Snippets test building blocks must be carefully designed keeping in mind the type of SoC under verification. A DSP SoC must perform high intensive arithmetic operations at sufficiently precise fix or floating point level. Data handling mechanisms such as address generation, array handling and data transfers are also important given the large amounts of signal data that must be manipulated efficiently. For example, a DSP often requires many registers to hold temporary or intermediate data during processing. To ensure correctness of these DSP functions, a library of snippets for the SoC was created. Most of these snippets are self-checking, flagging test failure if snippet operations did not perform as expected. We describe the main snippets in the remainder of this section.

The THUASDSP2004 DSP consists of a DMA to handle high throughput transfers of signal data between various SoC on-chip and external memories. Hence, to test the DMA functionality, snippets were created to initialise, execute and check for correctness of DMA transfers. Furthermore, DMA snippets provide parameters to control transfer of different transaction amounts between different memory addresses each time the snippet is selected in a test program.

To initiate SoC operations, snippets rely on low level device drivers to access various SoC configuration registers. For example, reusing the Nios SoC DMA snippets from Chapter 3, to initialise a DMA transfer requires the DMA source and destination address registers, and transfer size registers to be configured. The InitDMA snippet uses device drivers to initialise these registers and other

snippet parameters to test the DMA differently each time. Other DMA snippets also access various on-chip registers to configure, monitor and validate DMA transfers, e.g. ExecDMA, TermDMA snippets.

In order to test various arithmetic processing capabilities of the SoC, common DSP operations such as discrete Fourier or cosine transforms, and filter functions should be applied. These are the types of applications that use DSP mathematical units and are best suited for testing them. Hence, snippet functions were created for the discrete Fast Fourier Transform (FFT), making use of snippet parameters to vary the type, range, precision and error tolerance of signal data operations carried out. The FFT snippets also employ cosine, sine and factorial functions to calculate a range of *n* point FFTs that mimic stress-testing of repetitive and high intensive signal processing operations.

Other specific features of the THUASDSP2004 DSP are also verified by SALVEM snippets. For example, the DSP implements a unique global register file to facilitate inter cluster communication. Each local register in a cluster has a corresponding associate register in the global register file, and vice versa. Whenever data from one cluster is needed by another cluster, the result from one cluster is written to both the local and associate global register. In this way, an external cluster may gather the desired data from the global register file. Using this double associate register writing scheme, communication between clusters is achieved.

The snippets employed to test this specialised register bus system should invoke repetitive data transfers and resolve numerous register address selections. To this end, the TestRegBus snippet we developed is based on a token ring transfer operation (Figure E.18). Initially, data from any arbitrary cluster is written to both its local and global registers. The data is then consumed by another cluster before it is passed back onto the global register *bus* and transferred to other clusters; until finally, the data is checked at the termination cluster. Figure E.18 shows the first three data transfers. The non-circle enclosed numbers show the global register file read/writes, while the circle enclosed numbers indicate the sequence of inter-cluster transfers.

The parameters of the TestRegBus snippet are the type and size of data to be transferred, the type and number of clusters involved in the transfers, and the transfer start and termination points.

The TestInt interrupt snippet tests the DSP’s interrupt handling and priority mechanism. The parameters to this snippet specify which interrupts are enabled and their priorities. Each time the snippet is called into a test program, different interrupts and priorities will be chosen to test the interrupt unit differently.

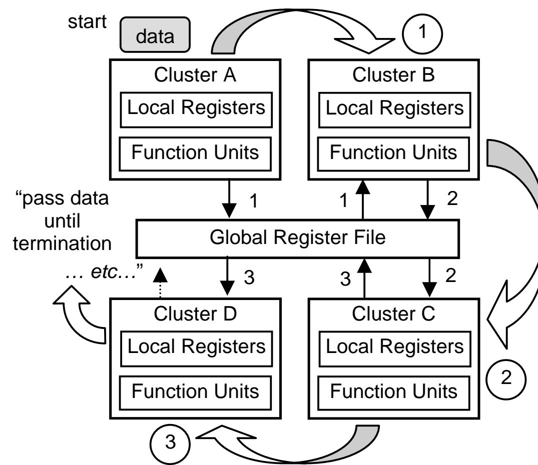


Figure E.18 TestRegBus snippet operation

E.16.4 SAGETEG test generation for DSP SoC verification – an overview

During test generation, SAGETEG selects a sequence of snippets and snippet parameters to form test programs in a genetic evolutionary influenced manner. Depending on the sequence of snippets chosen, a variety of sequential and concurrent operations on the SoC will be executed as was the case with Nios SoC verification. For example, if DMA and FFT snippets are chosen one after another, Fourier transforms of signal data can be concurrently tested whilst data is being shifted between memories.

To create effective and efficient tests, SALVEM employs genetic algorithms and evolutionary strategies to select the snippet sequence and parameters according to previous tests and their coverage information. This facilitates our coverage driven strategy. The GEA process, SAGETEG implementation, and application for DSP SoC verification was described previously in Chapter 4, and is employed in the same manner for this case study.

The GEA SALVEM test generation process was summarised in Figure 4.11 Chapter 4. In the beginning, test individuals are created to fill an initial population of μ number of tests. These initial tests may contain any number of snippets and relies on the evolution process to vary the test individuals further.

The experiments and results for the DSP SoC verification case study were described previously in Section 4.12.2 Chapter 4.

APPENDIX F. Markov Modelling for Test Parameter Selections

This appendix provides supplemental information for the analytical Markov based test generation parameter selection research in Chapter 5.

F.1 Transition probabilities for Markov modelling variation weight characteristics

This appendix section describes the derivation of Markov chain transition probabilities for modelling of GEA variation weight characteristics from Section 5.5 Chapter 5. Using information from the Markov chain from Section 5.4 Chapter 5, the assumptions, observations, and deductions justifying the transition probabilities of this Markov chain are as follows.

(i) For transitions between $E \rightarrow E$ or $H \rightarrow H$:

These transitions are possible whenever variation usage undergoes either incrementation or decrementation, so that variation weights are adjusted but remains with the same E or H intermediate states. The variation usage changes correspond to any of the $A \rightarrow A$ and $A \rightarrow D$, and, $D \rightarrow D$ and $D \rightarrow A$ transitions from the Markov chain of Section 5.4 Chapter 5 to maintain variation within intermediate states. Based on the transition mappings between A and D using c (variation continual change) and s (variation change switching) from Section 5.5 Chapter 5, the $E \rightarrow E$ and $H \rightarrow H$ transition probabilities is $2(c + s)$.

(ii) For transitions between $E \rightarrow H$ or $H \rightarrow E$:

These transitions occur when there is a continual change of variation usage such that increment or decrement of variation is followed on in consecutive evolutions; so that variation weights are increased from lower valued intermediate state E to higher intermediate state H , and vice versa. These changes are equivalent to $A \rightarrow A$ and $D \rightarrow D$ transitions, thus the $E \rightarrow H$ or $H \rightarrow E$ transitions have probability c each.

(iii) For transitions between $E \rightarrow L$ or $H \rightarrow U$:

Regardless of prior variation usage adjustment, the transition from E to the low limit L state requires next variation change to be a decrement, which corresponds to $D \rightarrow D$ and $A \rightarrow D$ transitions. Hence, the

$E \rightarrow L$ transition probability is $(c + s)$. Similarly, a transition from H to the upper limit U state requires increment variation adjustments $A \rightarrow A$ and $D \rightarrow A$, which results in $(c + s)$ as well.

(iv) For transitions between $E \rightarrow G$ or $H \rightarrow G$:
To fine tune variation variable to achieve desired goal state G requires a change in variation usage that is opposite to what was applied in previously. This fine tunning goal attainment is possible when variation adjustment switching changes between A and D, which is different to the prior type change. This corresponds to $A \rightarrow D$ or $D \rightarrow A$ transitions and so the transition probabilities for $E \rightarrow G$ or $H \rightarrow G$ is s. If variation usage adjustment continued to apply the same increment or decrement change, then overshooting past the goal state G would occur.

The remaining transitions are either not possible or reflect the absorptive state transitions within G, U and L. For example, non-realisable transitions such as $L \rightarrow G$ are not possible because a transition to intermediate state E is required before absorption into G. Variation weight values cannot simply skip states because large jump in values are not possible.

F.2 Derivation of test program composition Markov model and transition probabilities

Recall from Section 5.6 Chapter 5 that individual snippets from the snippets library could not be employed as Markov states for modelling the composition of test programs. Modelling genetic evolutionary algorithm (GEA) snippets usage with a Markov chain whereby each state element represents a snippet would result in unmanageable Markov chain. For instance, the Nios SoC's snippet library currently contain up to 30 snippets. Therefore, we group snippets into subsets. Our first phase of condensing snippets into groups is based on the SoC devices for which snippets cater for. For example, snippet groups for the DMA, UART, CPU, timer, and etc snippets that were created to test these devices explicitly. This reduces the number of state elements to model for a Markov chain from 30 down to 7.

The equivalent transition probability matrix for a Markov chain with such a set of 7 snippet type state elements is shown in Figure F.1. The rows and columns each represent snippets testing the DMA, UART, PIO, Timer, CPU, and memories devices; including a state in the Markov chain that represents when no snippets are included into the test program for the current evolution (i.e., first row and column of the matrix in Figure F.1).

		No change (F)	Concurrency (C)		Single threaded (O)			
		No snippet selection	DMA	UART	PIO	CPU	Timer	Memories
F	No snippet selection	0	1/6	1/6	1/6	1/6	1/6	1/6
C	DMA	$(1-p)(1/\sigma)^s$	$p(1/\sigma)^c$	$p(1/\sigma)$	$p(1/\sigma)$	p	p	p
	UART	$(1-p)(1/\sigma)^s$	$p(1/\sigma)$	$p(1/\sigma)^c$	$p(1/\sigma)$	p	p	p
O	PIO	$(1-p)(1/\sigma)^s$	p	p	$p(1/\sigma)^c$	p	p	p
	CPU	$(1-p)(1/\sigma)^s$	p	p	p	$p(1/\sigma)^c$	p	p
	Timer	$(1-p)(1/\sigma)^s$	p	p	p	p	$p(1/\sigma)^c$	p
	Memories	$(1-p)(1/\sigma)^s$	p	p	p	p	p	$p(1/\sigma)^c$

Figure F.1 Test composition Markov transition matrix based on SoC devices and types of device snippets

From Sections 5.4 to 5.6 in Chapter 5, the variables used in the transition probabilities of various Markov chain were explained. Some of these variables are GEA test parameters for which we are designing and analysing Markov chains to select values for. For instance, p is the probability (weight value) of a type of snippet selection. σ is the probability change factor of the variations usage adjustments applied to vary tests and include different types of snippets. And c and s are the amount of increment and decrement variation change adjustments.

The transition probability values are assigned based on certain observations and justifications similar to those given Section 5.6 Chapter 5 already. Briefly, they are as follow. If the current state was such that no snippet was added, it is not possible for no snippet to be included into the test because this would imply stagnation which the test generation is designed to avoid. Hence, the 'No snippet'→'No snippet' transition has probability zero. Instead, the snippets that can be chosen next are all given equal probability of selection. Amongst the remaining six types of snippets, the state transitions from 'No snippet' state to snippet inclusion each contain probability 1/6. For transition probabilities from one snippet type to another snippet type, these transitions contain at least the probability p representing snippet selection. For transitions whereby consecutive selections of the same snippets are conducted (i.e., $p(1/\sigma)^c$), an additional factor $(1/\sigma)^c$ is included. This is to take into account the greater likelihood of selecting the same type of snippets to continue further testing of the same devices catered by the currently selected snippet. This is indicated by the increase in variation usage and weight adjustments.

Except the non snippet selection to non snippet selection transition, the diagonal elements of the matrix whereby the same snippets are selected in both the current and next state, contain this transition probability.

For transition between current state of snippet selection to ‘No snippet’ selection state, the transition probability is $(1-p)(1/\sigma)^s$ because $(1-p)$ implies no snippet selection, given p is the probability of selection. The factor $(1/\sigma)^s$ is included to represent the increased effect of variation adjustment that reduces snippet inclusion likelihood when the variation switches between increment or decrement states and vice versa (to indicate the variation will no longer continue in a state of increment or decrement, and hence variation is less likely to select snippets).

Snippet dependencies also play a role in the transition probabilities. If in the current state a snippet is included into the test such that it depends on other dependent snippets, then these dependent snippets will be included as well in the next state with greater probability. For example, I/O interacting snippets usually have dependencies amongst one another. A DMA snippet may need to be followed up with UART or PIO snippets if the DMA transactions invoked by the DMA snippet involve transfers between UART or PIO devices. In this case, a factor of $(1/\sigma)$ representing higher variation usage for snippet inclusion is employed to represent the greater likelihood of such snippet selection in the next state. This effect is most pronounced especially for addition variation which recursively adds dependent snippets and further dependent snippets of dependent snippets (Section 4.5.8 Chapter 4).

Based on the probability matrix in Figure F.1, a smaller Markov chain with smaller number of elements transition matrix is created by grouping together common snippets and reducing down these snippets further into smaller number of snippet types to represent as states in the eventual final Markov chain used for analysis. The final grouping of snippets are indicated by the dashed lines in Figure F.1.

For actual GEA parameter selection analysis, transition probabilities of the smaller final Markov chain in Section 5.6 Chapter 5 are derived from the more detailed and lower level transition probability matrix in Figure F.1 (i.e., the smaller 3×3 matrix that can be extracted by grouping snippets partitioned by dashed lines in Figure F.1). This smaller matrix (and Markov chain) uses less state elements to represent the types of snippets based on what devices they explicitly test for and what kinds of operations are used for testing. For instance, snippets from DMA and UART can be further grouped together into a common type of snippets: concurrency type snippets that invoke or involve other SoC devices and snippets. Based on the dependencies of these types of snippets, one can observe that their transition probabilities in Figure F.1 is identical, and hence they can be grouped together. For the other remaining snippet state that can be transitioned to (i.e., snippets that invoke single threaded functions),

these snippets can be grouped together by their identical transition probabilities again to form the snippet state transition probability for single threaded snippets. In most cases, grouping snippets together further simply involves combining their transition probabilities. The transition probabilities for no snippet inclusion remain as before.

Therefore, the three type of snippets state elements used for the Markov chain in Section 5.6 Chapter 5 are the *C* state representing concurrency snippets that invoke multiple threads of SoC executions, *O* state representing snippets that invoke one single threaded type of SoC executions, and the *F* state representing no further snippet inclusions and a fixed snippet composition in tests. The transition probabilities for this final Markov chain and smaller transition matrix were discussed previously in Section 5.6 Chapter 5.

F.3 Transition probabilities for Markov modelling test population compositions

The transition probabilities from Section 5.7 Chapter 5 for Markov modelling the test population composition are summarised as follows.

(i) For transitions $V \to V$ and $X \to V$: $(2\lambda/\mu)(0.9u)(3w_F + 2w_C + 2w_O)$:

The test population selects only children tests, hence the transition probability is proportional to a population ratio of $2\lambda/\mu$. As the transition is to a destination state of *V* whereby more children tests are selected, this implies greater variation to create these higher coverage yielding children tests as well. Many new tests are varied that are successful for SoC testing. This accounts for the $0.9u$ factor where 0.9 is the highest possible variation usage weight assumed, and $u = 0.54$ was the probability of attaining that value (from (5.5.4) in Section 5.5 Chapter 5), and assuming the initial state was *E*. For snippet selection, the w_C and w_O factor (Section 5.6 Chapter 5) is multiplied by two for each of the addition and recombination variation that inserts concurrent and single threaded SoC operation snippets. The w_F factor is multiplied by three for each sub, replace and mutate variations, which do not change the snippet compositions within tests in any way.

(ii) For transitions $V \to X$, $X \to X$ and $Y \to X$: $(2\lambda/\mu)(0.5g)(3w_F + w_C + w_O)$:

Like the previous probability transitions, greater children tests are selected, hence the $2\lambda/\mu$ ratio is included. Whilst greater children than parent tests are selected in the destination state *X*, the variation variable is within range of the 1/5 variation success and stability goal state, and further from the boundary *U* or *L* states. Hence, the $0.5g$ factor (from (5.5.4) in Section 5.5 Chapter 5) is included,

where $g = 0.31$ and 0.5 is taken as the median variation usage weight between minimum and maximum range of 0.1 to 0.9 respectively. Because not all children tests are selected, the snippet selection factors w_F, w_C and w_O are lower.

(iii) For transitions $X \rightarrow Y$ and $Y \rightarrow Y$: $(\lambda/\mu)(0.5g)(3w_F + 2w_O)$:
Since more parents tests are selected compared with children tests, the population ratio is reduced to λ/μ. Less parent tests selection implies less successful variation, so the influence of the w_F, w_C and w_O factors are reduced further. Lower success variation implies less effective snippets such that concurrency snippets governed by w_C are eliminated from the transition probability. The Y state includes the $0.5g$ factor like the previous probability transitions because some variation under self-adaptation influence still applies.

(iv) For transitions $Y \rightarrow J$: $(\lambda/\mu)(0.1l)(3w_F)$:
Much greater parents tests are selected and little or no children tests are included in the test population, hence the population ratio is λ/μ again. The J state also implies mostly unsuccessful variation, therefore the variation factor is $0.1l$ from (from (5.5.4) in Section 5.5 Chapter 5) whereby 0.1 is the minimum variation weight and $l = 0.15$ is the corresponding low probability of 0.1 being applied. Also, the snippet selection only includes the w_F factor for subtract, replace and mutate variation because the snippet composition does not change significantly, resulting in largely unsuccessful variation and subsequently little or no new children tests are selected.

Note that the $J \rightarrow J$ transition is an absorptive state transition whereby the GEA process only selects parent tests for the entire test population, before terminating.

APPENDIX G. Multi-Objective Genetic Evolutionary Test Generation

This appendix provides supplemental information for the multi-objective test generation research in Chapter 6.

G.1 Impartial remainder selection policy

Recall from Section 6.5.2 Chapter 6, multi-objective GEA test selection employs a round robin scheme between objective bins so that an even distribution of tests that cover all conflicting objective subsets is retained. Ideally, the test population for subsequent evolutions should contain equal number of tests from each bin. However, depending on the number of objective subset bins and the predefined test population size, this fairness criteria is not always possible. Specifically, if the predefined test population size is not a multiple of the number of bins, equal number of tests from each bin cannot be chosen.

The maximum number of tests that can be selected evenly from all bins is the highest common multiple of the number of bins that can fit within the test population size. The remaining tests for the population will be chosen by performing tournament selections, based on the impartial remainder selection policy in this section.

For each tournament, a test from each of bin will be chosen to be participants. The chosen test from each bin will be the highest Pareto and Aggregate ranked test each time. After a test has participated in a tournament, it is removed from the originating bin it was chosen from. For the first remaining test, the tournament selection choses the highest ranked test immediately after the round robin selections stage from Section 6.5.2 Chapter 6.

The tournament selection is conducted using a combined fitness metric similar to that used in Section 6.5.2 of Chapter 6 for Aggregate ranking. This merged metric combines all the objectives of the GEA application domain, not just conflicting objectives from each objective subset only. The winner of the tournament according to the merged metric is then inserted into the next population. The tournament selection for choosing remaining tests corresponds to step (9) in Figure 6.6 of Section 6.5.2 Chapter 6, and is defined as follows.

Definition G.1 : Impartial remainder test selection

(i) Let μ be the number of tests to be selected into the test population, b is the number of object subset bins, and r is the number of remaining tests that cannot be selected evenly from each of the bins, such that $r = \mu \bmod b$.

Each of the remaining tests y_i are selected as follows,

$y_i = TourSel(x_1, x_2, \ldots, x_b)$ for $i = 1, \ldots, r$

where *TourSel* is the tournament selection function defined in (ii) below, and $x_1, x_2, \ldots, x_b$ are the best aggregate ranked tests selected from the current highest Pareto front of each objective subset bin.

Like the round robin test selection of steps (2) to (7) in Phase 3 of Section 6.5.2 Chapter 6, the tests x_1, $x_2, \ldots, x_b$ are also removed from their respective bins after they are chosen as participants in the tournament selection.

(ii) Let $TourSel : X^b \rightarrow X$ be the tournament selection function that takes in a test from each bin, and returns the test with highest aggregated fitness, where X is the set of possible tests capturing the current test population. *TourSel* is defined as follows,

$\mathrm{y}_i = TourSel(x_1, x_2, \ldots, x_b)$

such that $\forall\, x_j,\ f_m(\mathrm{y}_i) > f_m(x_j) \wedge x_i \neq \mathrm{y}_j$ for $i = 1, \ldots, r$ and $j = 1, 2, \ldots, b$

f_m is the fitness merging function that combines the fitness of all the multiple objectives of a test together. f_m is defined next in (iii).

(iii) The fitness merge function $f_m : X \rightarrow X$ takes in a test and combines all the objective fitness functions to produce an overall fitness measure. f_m is defined as,

$$f_m(x) = \frac{\left(f_1^{\max}(x) + \ldots + f_u^{\max}(x) + \dfrac{\mathrm{M}_1 - f_1^{\min}(x)}{\mathrm{M}_1} + \ldots + \dfrac{\mathrm{M}_v - f_v^{\min}(x)}{\mathrm{M}_v} \right)}{u + v}$$

where f^{max} and f^{min} are objective fitness functions that are to be maximised and minimised respectively, u and v are the number of maximising and minimising objective functions respectively, and M_v is the maximum fitness values possible for the ith minimising objective function f_i^{min}.

□

For multi-objective GEA test generation, f_m is defined as follows.

$$f_m(x) = \frac{\left(f_l(x) + f_t(x) + f_c(x) + \frac{\mathrm{M} - f_s(x)}{\mathrm{M}} \right)}{4}$$

where f_l, f_t, f_c, and f_s are the fitness functions for the line, toggle, conditional coverage, and test size objectives respectively, and M is the maximum test size for the test platform.

Whilst round robin selection inserts equivalent number of tests from each bin into the new test population, this method is not always applicable. For the remaining number of tests that cannot be chosen equally from these bins, the test selection strategy in Definition G.1 ensures all multiple objectives are given equal priority. This prevents the GEA process from favouring optimisation of certain objectives over others as more evolutions are conducted. It is essential the remaining tests that make up the rest of the population are selected based on how they optimise all objectives overall.

G.2 Diversity and fitness distance of multi-objective GEA test selection

Our GEA process aims to promote test diversity by selecting tests that display fitness variation amongst each other. The concept of fitness distances in the objective test space is to measure the difference in fitness of tests for particular objectives; and use this information for selecting tests. The selected tests must be widely separated from each other, and for each objective, the fitness distance should be as large as possible.

The aim is to avoid selection of tests within the same crowded region of the objectives' test space. In this way, the tests will be highly diverse and more effective in terms of the objectives it optimises. The resulting fitness attained from the population for each objective will be wide-ranging, but there will be at least one test in the population that achieves best optimal fitness for each objective. The overall goal is to make use of these diverse tests for further evolutions and optimise all objectives concurrently.

G.3 Pareto front drifting

This appendix section explains further the reasons for Pareto front drifting during multi-objective GEA. First, whilst the GEA variation process tries to search for new objective test space, it may in fact produce tests that achieve similar or lower coverage. This is due to the inherently random nature of

GEA variation and because most beneficial test space have been uncovered already at later stages of the GEA process.

Second, by employing a divide and conquer approach to partition objectives into subsets (Section 6.5.1 Chapter 6), certain undesired behaviours in the GEA process could occur. The selection and intermixing of tests from different objective subsets imply certain tests were chosen because they perform better for certain objectives, but they may perform worst for other objective subsets. If so, the Pareto front will begin to flatten and coverage increase in y-axis will decrease noticeably for other objective subsets.

At the same time, despite not gaining coverage improvement, the addition and recombination variation will continue to increase test size. Therefore, the Pareto front will flatten out and stretch along the direction of the positive test size x-axis. Eventually, when the test population achieves the maximum possible coverage given its range of snippet genome, no further coverage improvement will be possible.

The Pareto front will then hone in to a fixed coverage level given by the best tests in the current population. The front will flatten out at this level for the range of test sizes in the population, which determines the extent of the front drifting.

G.4 Multi-objective GEA termination according to test selection duplication

Besides Pareto fronts, another possible GEA termination method is to examine the test population directly. Such a method monitors the number of duplicated tests in the test population at every evolution. When more than 75% of the population contains identical tests, the GEA process is terminated. The test duplication rate can also be used as a guide to how fast the test population is converging toward optimality for all the objectives. Greater test duplication with test population more than 75% identical tests indicates insufficient diversity of snippet genome to improve the GEA process further. Hence, termination should be invoked. In general, a duplication rate of around 50% is useful for promoting higher fitness performing tests whilst maintaining adequate test population diversity.

G.5 Comparison with less diverse population of GEA test generations

The significance of diversity for multi-objective GEA can be demonstrated by comparing against a GEA process without any test duplication prevention and selection mechanisms employed in our test generator in Chapter 6. Figure G.1 to Figure G.6 shows the Pareto front plots and results for such a test generation.

Compared to Figures 6.11 to 6.16 in Chapter 6, in Figure G.1 to Figure G.6, without duplication containment, the selection of tests for each subsequent evolution is not sufficiently diverse. Hence overall, the best Pareto fronts from each evolution do not expand as much, are shorter and is more levelled in curvature compared with those in Figures 6.11 to 6.13 (Chapter 6). Additionally, the Pareto fronts stray significantly away from the desired upper-left region of maximal coverage and minimal test size on the Pareto plot, even at early evolutions.

Similarly, Figure G.4 to Figure G.6 also show the equivalent Pareto front plots to Figures 6.14 to 6.16 (Chapter 6) previously, but without diversity enhancing measures. In Figure G.4 to Figure G.6, the span between the best and worst Pareto fronts are in general closer together than the equivalent plots at corresponding evolutions in Figures 6.14 to 6.16 (Chapter 6). This indicates the test population in Figure G.4 to Figure G.6 are less diverse. They are all too similar to effectively optimise all objectives, thus accounting for slightly lower coverage and higher test sizes compared to the test run of Figures 6.11 to 6.16 (Chapter 6).

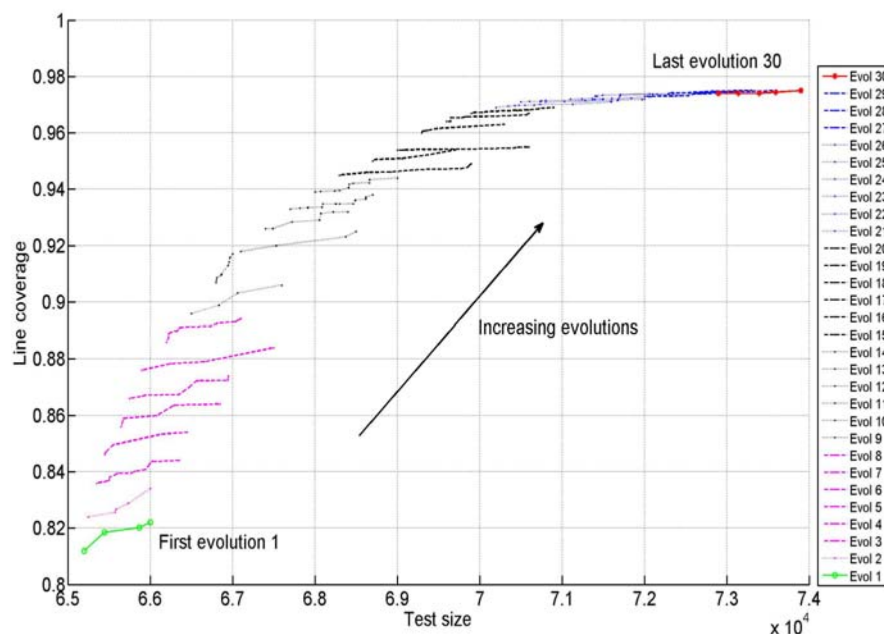


Figure G.1 Less diverse best Pareto front of each evolution (line coverage vs. test size)

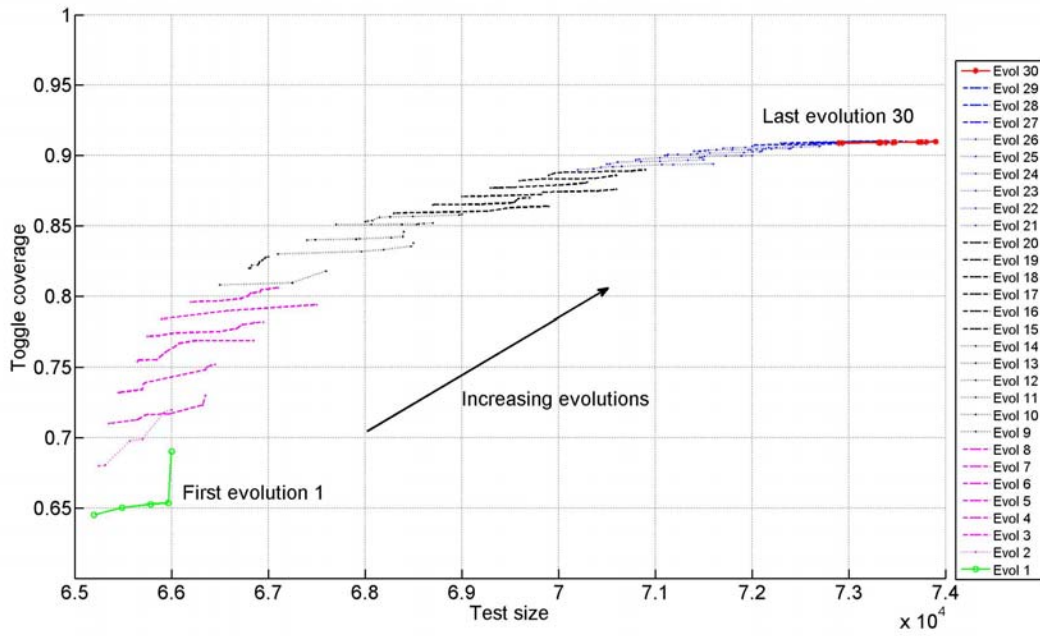


Figure G.2 Less diverse best Pareto front of each evolution (toggle coverage vs. test size)

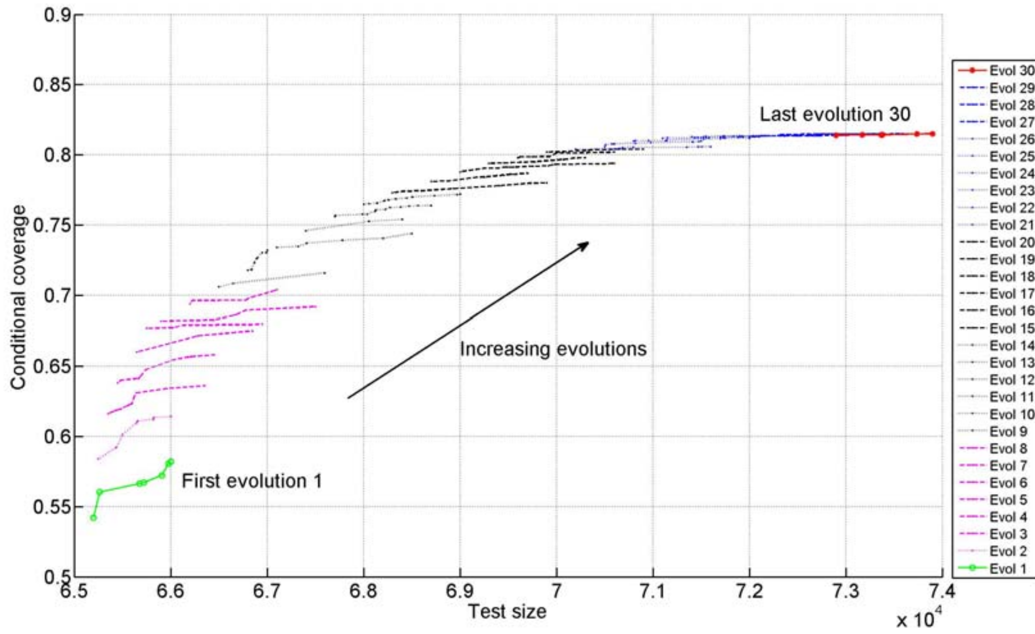


Figure G.3 Less diverse best Pareto front of each evolution (conditional coverage vs. test size)

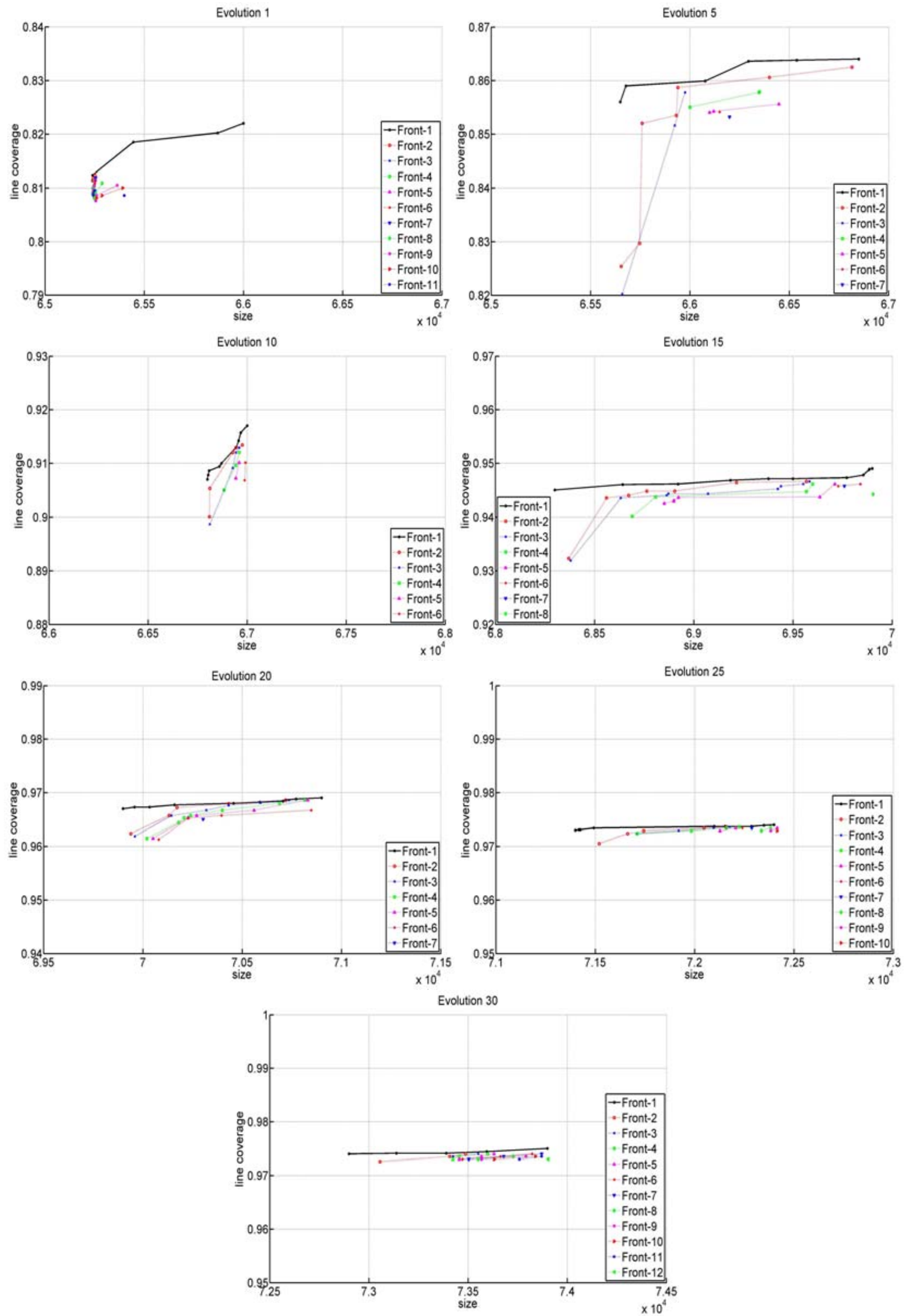


Figure G.4 Less diverse Pareto fronts at selected evolutions during GEA (line coverage vs. test size)

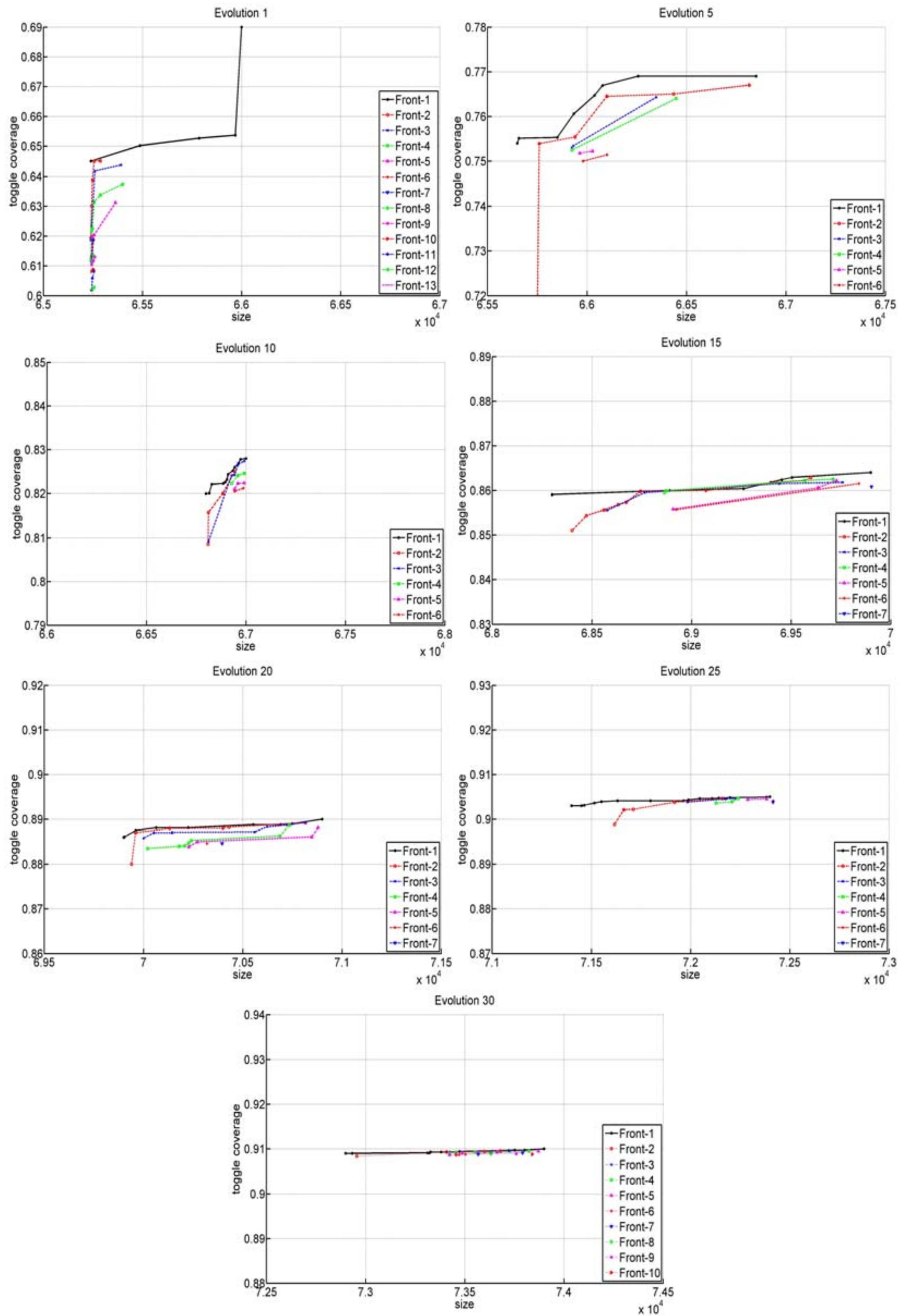


Figure G.5 Less diverse Pareto fronts at selected evolutions during GEA (toggle coverage vs. test size)

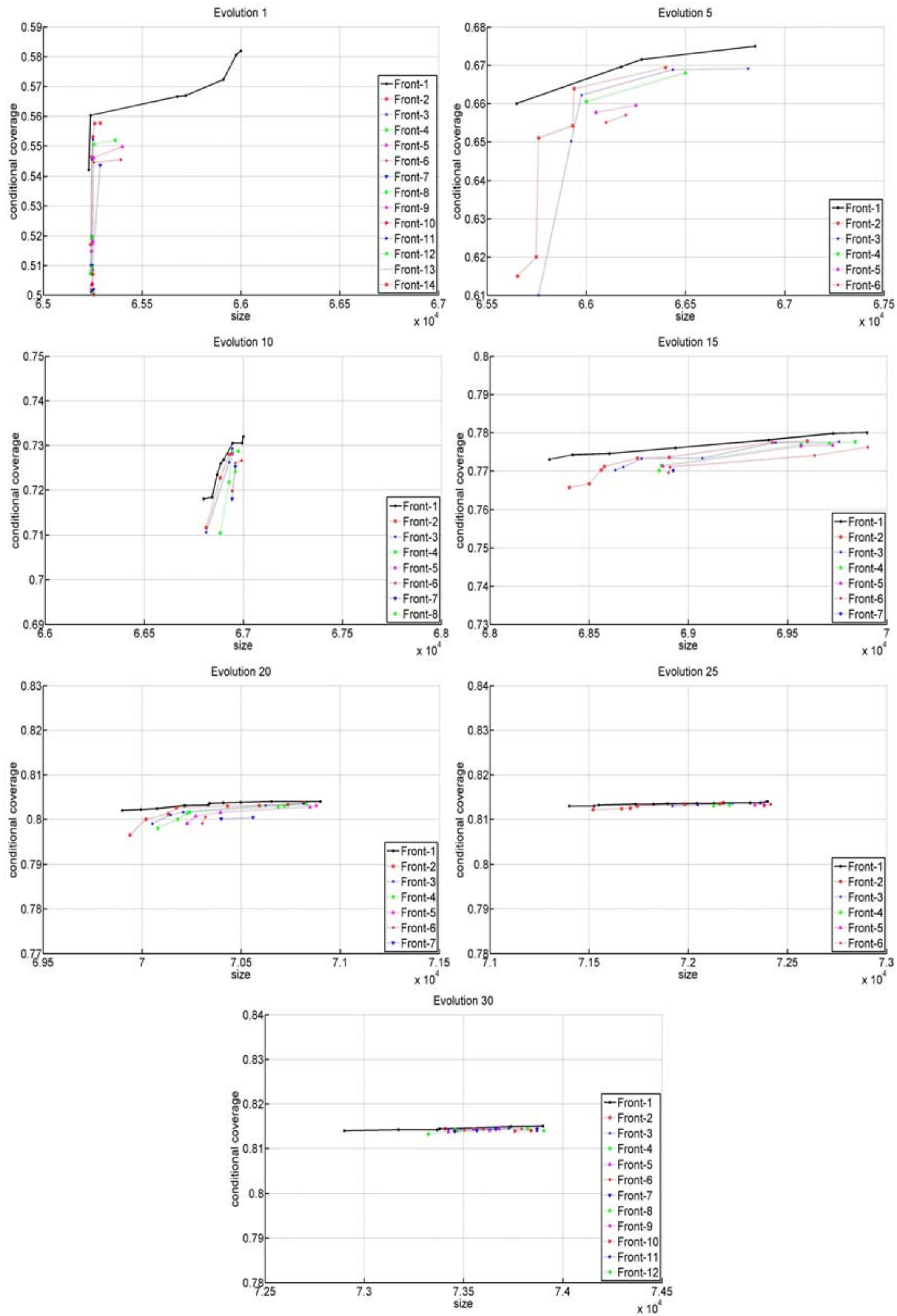


Figure G.6 Less diverse Pareto fronts at selected evolutions during GEA (conditional coverage vs. test size)

G.6 Multi-objective tri-axial plot characteristics

This appendix section expands on the tri-axial plot characteristics of the multi-objective GEA test generation process. The tri-axial plot from Figure 6.18 Section 6.9.3 of Chapter 6 is reproduced in Figure G.7 for discussions here.

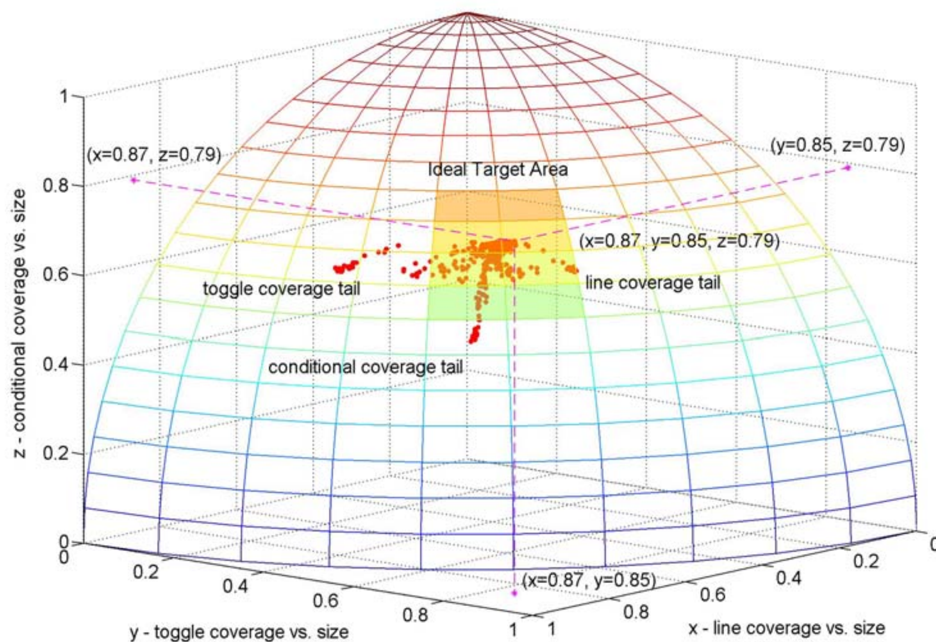


Figure G.7 Multiple objective GEA tri-axial graph

G.6.1 Test diversity and duplication with respect to tri-axial plot

Examining diversity on multi-objective tri-axial graph

Diversity affects the location of tests in the tri-axial graph, and their proximity to the target area. A less diverse test population reduces the range of tests to optimise all objectives. Hence, during variation (in particular, recombination), the GEA process is unable to intermix these limited tests as effectively. Newly created tests will contain less variety of snippet genome to simultaneously optimise objectives. The selection strategy employed from Section 6.5 Chapter 6 specifically addresses this issue, by selecting tests that cater for different objectives (i.e., tests from different objective subsets).

Without such a selection method, tests that are highly effective for specific objectives only but with low fitness for the multiple objectives overall, would not be retained. The evolution of these higher

performing tests for individual objectives would be lost. Instead, such high performing individual objective tests should be amalgamated with other tests that catered for other different objectives. This would result in much superior tests that provide higher fitness for all objectives simultaneously.

In Figure G.7, the tests are plotted within the vicinity of the ideal target test area due to the diversity and our multi-objective GEA selection policy.

Examining test duplication on multi-objective tri-axial graph

Test duplication affects how wide-spread tests are plotted across the target area. Greater test duplication causes tests to concentrate within a smaller area. Therefore, the GEA process controls the rate of duplication to ensure the test population are not too alike. Otherwise, the variation process would be restricted to using the same tests repeatedly, reducing objective fitness improvements. Besides explicitly measuring duplication rates in Figure 6.17 Chapter 6, the extent of duplication during the GEA process can be assessed by examining the distribution of tests on the tri-axial graph. In Figure G.7, rather than a concentration of tests, tests are spread across various regions of the ideal target test area. This confirms test duplication is not excessively high as was quantitatively revealed in Figure 6.17 Chapter 6.

G.6.2 Test generation effectiveness and characteristics from the tri-axial plot

In Section 6.9.3 Chapter 6, the notion of *test tails* plotted in the tri-axial graph were observed and described. A test tail can be mapped to one of the line, toggle, or conditional coverage versus test size objectives subset. The GEA process focuses on each of these objectives to gain fitness improvements, and at the same time, optimises all the objectives concurrently; by combining useful tests for each of the objective subsets together. As the tests from each objective subset tail are intermixed further, the test tails come closer together until the test population is fully combined with snippet genome specifically for optimisation of all the objective subsets.

At this point, using the intermixed test population, the GEA process continues to enhance all the objective fitness further by evolving the test population. Eventually, the combined test tails form a thicker plot line that increases outward from the target region in the direction of best fitness for all objectives, toward the ideal target test point.

Note that the width of the test tails lines and dispersion of tests across the target region depends on the diversity of the test population during GEA. The greater the diversity, the wider the test tails and more spread out the converged tests are throughout the test region. As described above, a greater diversity tests population implies better evolution and optimisation for each of the objective subsets. This is because a larger variety of tests are provided to seek out further improvements in the objective fitness space more effectively.

If the combined test population from convergence of test tails are more diverse and scattered, the GEA process would also be able to simultaneously enhance all the objectives further. The converged tests plot line will increase further away from the origin of the test tails toward the desired ideal target test point, gaining greater objective fitness. Also, the more optimised the test population is, the sharper the peak arises from the combined test tails plot.

G.6.3 Comparisons with tri-axial graphs of other test generations

In order to examine further the effectiveness our multi-objective GEA process using tri-axial graphs, we compare against other test generation variants and their tri-axial graphs in this section.

Figure G.8 shows an equivalent three dimensional tri-axial graph of tests for a GEA process with non diverse test populations. These tests were generated without the objective subset based selection and variation methods in Sections 6.5 and 6.6 Chapter 6. The tests plotted do not develop multiple test tails or originate from different regions. Instead, a single thickened test plot line concentrated along the same test region is shown; and increases outwards away from the ideal target test area.

The lack of multiple tails is because objectives subsets are no longer given higher priority for optimisation. Rather, the GEA process defaults to a selection scheme whereby tests are chosen based on the overall objectives fitness only. By doing so, it is difficult to manage and take on all objectives at the same time, especially if conflicting objectives exists. Hence, the level of objective fitness gained for the overall objectives set is lower compared to Figure G.7.

Unlike the GEA process in Figure G.8, by segregating the optimisation of objectives and recombining tests afterwards, the likelihood of successfully optimising the entire objectives set is higher in our GEA test generation. Also, compared to Figure G.7, the thickness of the combined test plot line is narrower in Figure G.8. This is directly attributed to lower test population diversity in the non diverse GEA process; which results in smaller range of coverage and test size fitness.

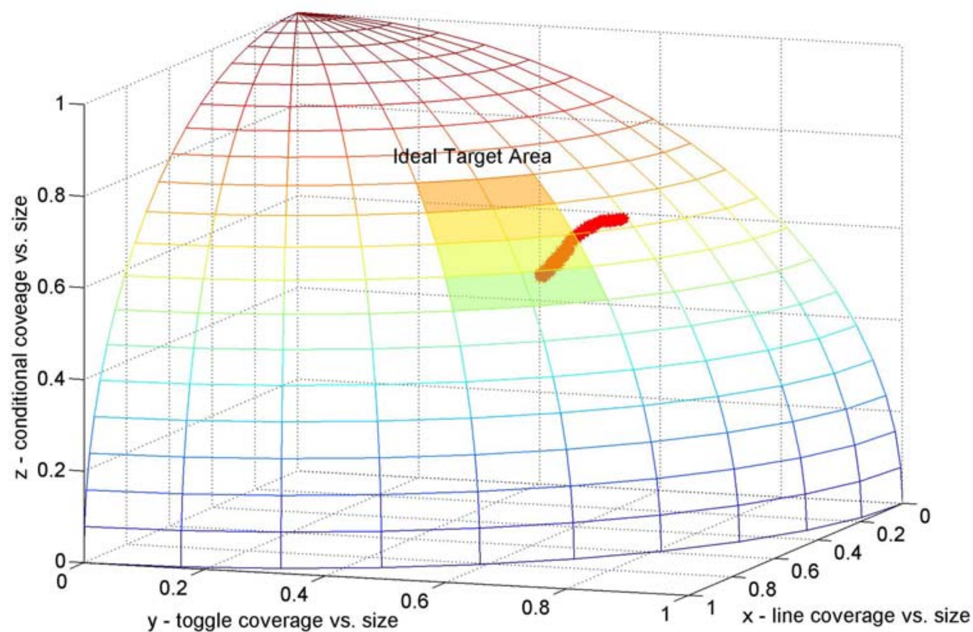


Figure G.8 Less diverse multiple objective GEA tri-axial graph

For another comparison, Figure G.9 shows the resultant tri-axial plot for a non multi-objective GEA test generation process. The tests plotted in Figure G.9 are obtained from individual and independent single objective GEA test runs; whereby each of the line, toggle and conditional coverage objectives are optimised separately by different GEA processes. These independently evolved tests are plotted at three distinct and separated regions. Each region reflects high objective fitness for one of the line, toggle or conditional coverage only.

Compared to Figure G.7, these tests do not lie close to the target region at all because none of the tests are evolved to concurrently enhance more than one objective. The three clusters of tests each reside closer toward the origin of two of the three axis for which their corresponding objectives are not optimised for. For example, the left-most cluster in Figure G.9 was optimised for line coverage only. Hence, it is located where fitness for line coverage is high but is below the midpoint of the toggle and condition coverage axial range. Similarly, the right-most cluster caters for toggle coverage and top cluster cater for conditional coverage only.

Unlike multi-objective GEA in Figure G.7, the best possible tests that can be attained must be chosen from one of the three clusters in Figure G.9. It is not possible to select a test that is favourable for all the objectives; a trade-off must be made based on which objective is deemed higher priority.

Essentially, the single objective GEA test generation are only focused on one objective. Greater time and resource are also needed to run each objective test run independently multiple times.

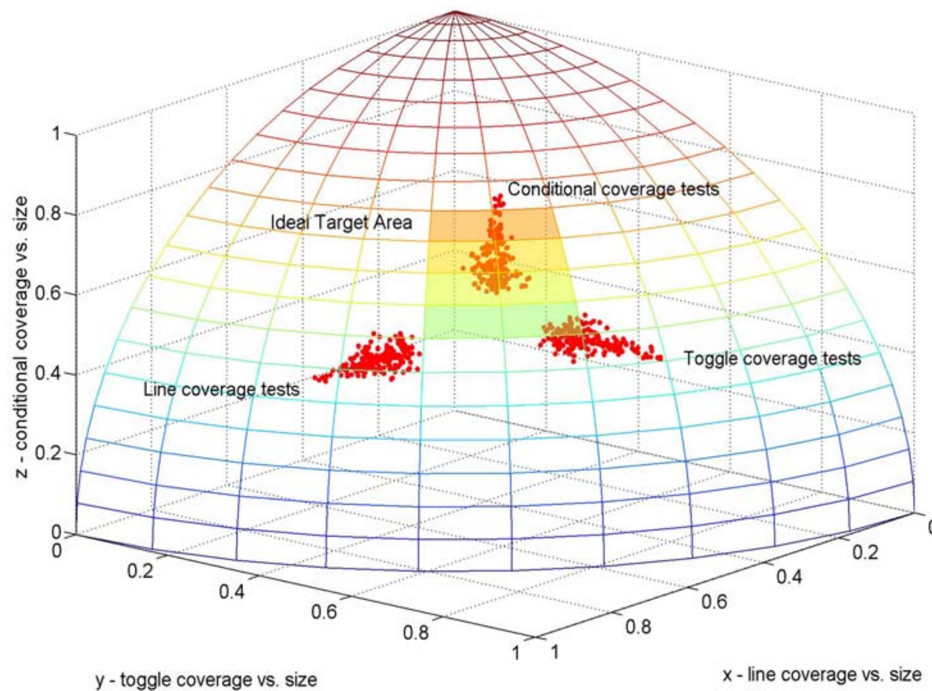


Figure G.9 Non multiple objective GEA tri-axial graph

G.7 Pareto front based GEA termination details

G.7.1 Identification and selection of Pareto front GEA termination threshold values

By repeatedly conducting preliminary multi-objective GEA test generation runs for a fixed number of evolutions, well beyond the point at which the Pareto front terminations would have been triggered, various values for the slope and gap distance threshold can be sampled.

Based on these empirical results, proper values for the slope and gap distance threshold values are chosen for retuning the test generation. These calibrated threshold values should trigger termination when it is clear further improvements in objective fitness are unlikely or extremely low. For example, examining similar Pareto front graphs to those in Figures 6.11 to 6.16 Chapter 6, the slope and gap distance values at which objective fitness optimisations has stalled can be identified at corresponding evolutions from graphs similar to Figures 6.19 and 6.20 in Chapter 6.

This process of threshold values identification are conducted at least ten times for the current GEA test generation configurations in Section 6.9 Chapter 6, and averaged out to provide a calibrated slope and gap distance threshold of 0.2 and 0.5 respectively, which was applied for actual test generation runs.

G.7.2 Narrowing of Pareto fronts at an evolution cycle

In Figures 6.14 to 6.16 in Chapter 6, from evolutions 10 to 30, the best and worst Pareto fronts gradually become closer compared to their proximity with each other at the start of the GEA process. This indicates test diversity is lowering and the best Pareto front does not expand as much as before. The reduced proximity is caused by the worst Pareto front simply closing in on the best Pareto front as the GEA process eliminates low performing tests from the population. Note that the best Pareto front and best attainable objective fitness does not actually improve as much as before. Rather, the best front is increasing at a lower rate whilst the worst front continues to increase and shift closer toward the best front. When this occurs, given the slowdown in coverage increments and enlargement of test sizes, this indicates the GEA process is proceeding toward termination conditions.

The reduced proximity and the closing in of the worst front toward the best front during evolutions is an expected behaviour of the test generation process. During GEA, low fitness performing tests are progressively removed from one evolution to the next. In Figures 6.14 to 6.16 in Chapter 6, this is clearly demonstrated by the elimination of outlier tests in the evolution 10 plot, that are no longer present in the corresponding plot at evolution 15 in Figures 6.14 to 6.16 Chapter 6. The outlier tests are shown as single dot points or straight lined Pareto fronts. By evolution 20 onwards, they are fully eliminated and the Pareto fronts shift towards the best Pareto front.

The removal of these under performing tests and replacement with higher fitness enhanced tests shows the GEA process optimising the population as expected. In fact, the changes in Pareto fronts from one evolution to another indicate fitness objective improvements. When the best tests reflected by the best Pareto front does not improve any further, and the tests of the worst Pareto front are simply allowed to close in on the best front, this indicates the GEA process has converged and the test population has stagnated. Further optimisations of the current test population would be extremely difficult and unlikely. The test population is no longer diverse to create further different tests, and hence termination can be invoked.

G.8 Test execution times during multi-objective GEA test generation

Figure G.10 plots the test execution time against each individual test as they are generated and executed during multi-objective GEA. Three distinct phases are apparent. At the start of GEA test generation between tests 1 to 200, the test executions times do not increase significantly, hovering around 200 seconds. This corresponds to initial stages of multi-objective GEA when test coverage is actively sought whilst maintaining test sizes at current levels. Hence, test execution times do not increase.

From approximately tests 200 to 400, test execution times begin to grow. This reflects the period when the GEA process deemed the previous pool of snippet genome to have achieved maximum coverage possible. Therefore, more intensive forms of GEA variation and many more snippets are added from the snippet genome pool to gain further coverage. The resultant tests induce more complex test functions and their size slowly increases, adding to test execution times as well.

Finally, toward the end of the GEA process from tests 400 onwards, the coverage levels begin to saturate. Any GEA variation using the existing snippet genome pool from the snippet library does not enhance the tests in any way, and test times do not vary much.

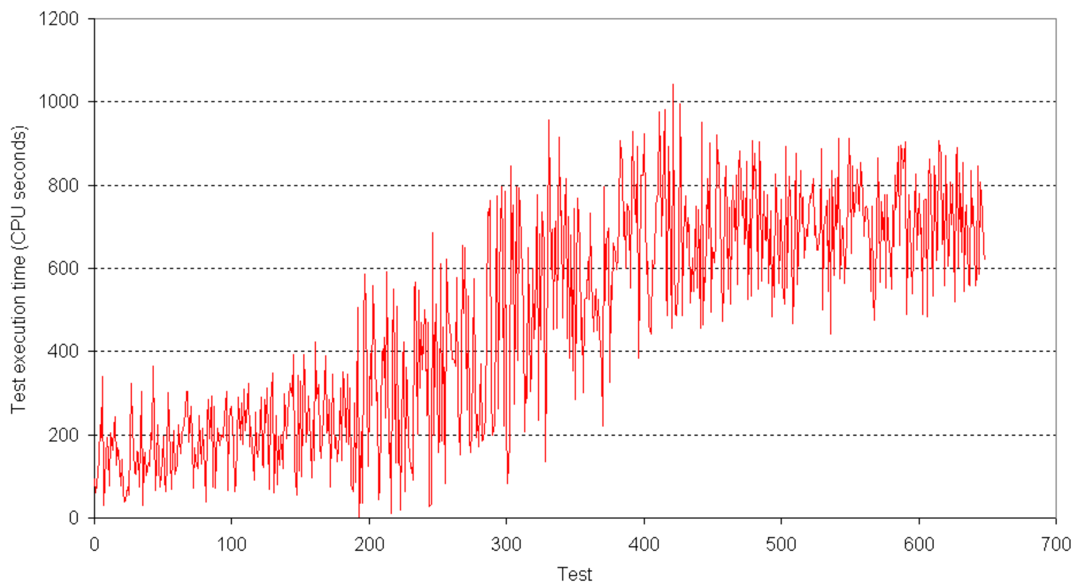


Figure G.10 Multi-objective GEA test execution time for each test

G.9 Summary review of multi-objective GEA test generation experimental analysis

Optimisation of multiple objectives examined by Pareto front plots

An important criteria for our test generation is to create tests that cater for multiple test objectives simultaneously. Specifically, one must ensure the test generator does not favour any one or more objectives over another. Otherwise this produces greater fitness for some objectives whilst other objectives fitness becomes extremely poor. A number of measures described in Section 6.5 Chapter 6 for our GEA population selection were put in place to ensure proper multi optimisation of all objectives. The capability of our GEA test generation method for multiple objectives was demonstrated by examining the Pareto fronts and other statistics gathered during the test generation.

One of the strategies to achieve desired multi-objective optimisation is to employ our test population selection policy (Section 6.5 Chapter 6) and recombination method (Section 6.6 Chapter 6). The effect of applying this selection and recombination method can be exposed from the objective fitness results. Figures 6.11 to 6.16 in Chapter 6 showed using this selection method within a typical GEA test generation provides higher coverage fitness for lower test size. In contrast, without proper selection, there is greater likelihood for the GEA process to favour certain objective subsets, and the overall objective fitness results can be lower (Figure G.1 to Figure G.6).

Inter Pareto front plots comparison

To attain additional evidence our multi-objective GEA process optimises all objectives concurrently, one can compare each of best Pareto front plots in Figures 6.11 to 6.13 in Chapter 6 with one another. The Pareto front plots for each of the line, toggle and conditional coverage versus test size objective subsets display similar characteristics and trends. For any one Pareto front plot, as coverage increases whilst containing test size, a similar pattern is shown by the Pareto front plots of the other two objectives subset. This shows that the GEA test generation is optimising all objectives simultaneously as required.

Despite objectives being optimised at different rates and the Pareto fronts saturating at different levels in Figures 6.11 to 6.13 in Chapter 6, the test generation process assigns equivalent priority to enhance each of the objectives. The scenario whereby some objectives are optimised at the expense and degradation of other objectives do not occur. This shows the GEA test generator is creating tests to find the best trade-off optimality amongst all the objectives.

Test generation performance from multi-objective tri-axial graph

The ability to optimise for multiple objectives was also demonstrated by examining the three dimensional tri-axial objective subset plot in Figure G.7. With our selection method, the GEA process aims to optimise all objective subsets. In Figure G.7, the tests are plotted originating from three different regions as the GEA process tries to optimise for three objectives subsets concurrently. Eventually, using on-going recombination variation, these tests are intermixed into a diverse test population that optimises all objectives. The converged tests plot in Figure G.7 demonstrates the GEA process allocates even priority to each objective and all objectives were optimised concurrently. Additionally, recombination ensures mating takes place with tests from different objective subset bins. Hence, the diversity of the snippet genome is preserved to tackle genetic drift.

Test population diversity and test duplication

An important facet of attaining high fitness for all objectives is to provide and maintain snippet genome diversity within the test population. Diversity depends on a number of factors, one of which is the level of test duplication in the test population. As outlined in Section 6.5 Chapter 6, whilst the selection policy employed aims to maintain fair optimisation amongst objectives, it also ensures tests that perform extremely well for any particular objectives are selected often. This enriches the population with greater snippet genome from these duplicated tests, in order to enhance fitness of these particular objectives.

However, the downside is too many duplication of these same tests may reduce diversity and degrade the overall fitness of all objectives. Therefore, during GEA, the level and rate of test duplication must be managed carefully using the mechanisms described in Section 6.5.3 Chapter 6.

In Figure 6.17 Chapter 6, the percentage of test duplication is shown during the entire GEA process. The amount of test duplication is maintained around 50%, with only certain evolutions exceeding this level significantly. However, such occurrences are brief. And if the duplication level could not be contained at a particular evolution, then within another few subsequent evolutions, the duplication rate is returned back toward 50%.

Elitism in GEA test generation

Another success indicator of multi-objective GEA is to observe the number of high fitness performing tests that must be added via elitism. Elitism is the process in GEA by which the best tests for an individual objective are ensured for selection to the next generation. If the test selection process did not select such tests explicitly, but had to rely on elitism to include these tests, this implies these tests performed extremely well for certain objectives but achieve low fitness for other objectives.

The greater the number of elitist tests that must be inserted independently via elitism, the higher likelihood that certain objectives are given preferential optimisation. Therefore, to determine if the GEA process is optimising multiple objectives fairly, one can examine the number of elitists test that are added every evolution; to monitor and check if it is kept low.

Throughout our multi-objective GEA testing, only two elitist tests required insertion into the next population explicitly. In contrast, the lower performing non-diverse GEA test generation in Appendix G.5 required 38 elitist tests to be added. This shows our multi-objective GEA selection process from Section 6.5 Chapter 6 is more effective at populating the next evolution with tests that can optimise all objectives simultaneously.

G.10 Individual SoC device coverage comparisons

Table G.1 breaks down the coverage results for prominent devices of the SoC design, attained from multi-objective GEA test generations and other comparative methods. For toggle coverage, using multi-objective GEA, the timer, PIO and UART devices have sub 90% coverage. This is because these devices have large range of data and counter state elements that demand many different signal values for propagation. In addition, the extent of exercising these values for all operating device conditions requires longer run length times, which is contrary to the test size minimisation objective.

A similar case could be put forth for the DMA device as well. However, DMA coverage is much higher because the DMA handles many more transactions from all other devices. Hence, a greater span of operations and associated signal values transit through the DMA device, pulling up its coverage. Regardless, the different coverage metric for many of the devices achieves improvement with the multi-objective GEA method.

For example, comparing against the individual device coverage of other methods, Table G.1 shows for all devices, multi-objective GEA testing exceeds or at least matches the best coverage attained by other

test generation methods. In general, multi-objective GEA performs better for at least two of the three coverage types, and achieves similar coverage for the remaining coverage metric.

Table G.1 Individual SoC device coverage results

Line coverage tests %	Multi-objective GEA	Single objective GEA (SAGETEG)	μGP	Random test generation	Manual application based tests
CPU	99.8	99.8	98.2	98.6	61.4
DMA	100	100	100	99.4	98.5
Memories	100	100	100	67.6	35.8
Misc. SoC units	99.2	96.7	93.0	50.5	75.4
PIO	96.2	96.2	98.7	98.3	82.0
Timer	100	100	79.2	97.6	100
UART	98.3	96.7	94.9	95.0	95.0
Toggle coverage tests %	Multi-objective GEA	Single objective GEA (SAGETEG)	μGP	Random test generation	Manual application based tests
CPU	96.5	96.3	89.6	75.6	49.2
DMA	97.5	97.5	95.6	97.0	78.9
Memories	97.1	96.1	94.1	82.3	21.8
Misc. SoC units	97.2	92.3	86.6	90.5	77.5
PIO	89.5	89.5	89.5	89.5	49.5
Timer	75.4	54.6	14.6	61.7	14.6
UART	78.9	71.3	58.6	58.4	57.6
Conditional coverage tests %	Multi-objective GEA	Single objective GEA (SAGETEG)	μGP	Random test generation	Manual application based tests
CPU	82.6	82.5	72.1	68.5	65.3
DMA	98.5	98.5	93.9	87.9	80.3
Memories	N/A†	N/A†	N/A†	N/A†	N/A†
Misc. SoC units	91.3	89.2	88.6	90.8	77.3
PIO	75.0	75.0	75.0	75.0	37.5
Timer	94.2	88.5	28.8	88.5	82.7
UART	86.4	85.3	72.3	74.5	66.3

† – No explicit conditions are present in the SoC memory units.

Specifically, the CPU, timer, UART, and miscellaneous portions of the SoC are covered more comprehensively using multi-objective GEA. For certain devices, like the memories, there is no applicable branching logic paths, hence conditional coverage measuring is not relevant. Other SoC design test space classified under the 'Misc. SoC units' category can be difficult to exercise because they do not fall under any well-defined types of test functionalities. Such examples include various SoC bus transactions and their conflict resolutions, or system wide error conditions. Rather than rely on individual SoC device oriented snippets, dedicated snippets interacting with other library snippets are employed. This demonstrates the versatility of multi-objective GEA – to combine useful snippet genome from multiple objectives driven tests in order to enhance overall coverage.

G.11 Complete time comparison results

Table G.2 shows the entire time comparison results for all test generation methods, including the individual line, toggle and conditional coverage test run times of single objective test generations that make up their combined times in the second and third rows.

Table G.2 Complete test 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[*†§]
Line coverage tests time	Multi-objective GEA	Single objective GEA (SAGETEG)	µGP	Random test generation	Manual application based tests
Test execution CPU time (s)	N/A[‡]	232,223	96,309	158,382	3,745[*]
Elapsed time (days)	N/A[‡]	2.7	1.3	2.0	4[*†]
Toggle coverage tests time	Multi-objective GEA	Single objective GEA (SAGETEG)	µGP	Random test generation	Manual application based tests
Test execution CPU time (s)	N/A[‡]	320,951	416,957	222,411	5,652[*]
Elapsed time (days)	N/A[‡]	3.8	5.2	2.8	4[*†]
Conditional coverage tests time	Multi-objective GEA	Single objective GEA (SAGETEG)	µGP	Random test generation	Manual application based tests
Test execution CPU time (s)	N/A[‡]	199,710	237,757	219,662	13,157[*]
Elapsed time (days)	N/A[‡]	2.3	2.9	3.3	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.
* – 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 automated methods, the same manually created tests were used for line, toggle and conditional coverage test runs, hence the combined total time remains the same as the individual test runs time.
‡ – Multi-objective GEA testing maximises line, toggle and conditional coverage in a single test run, not individually.

G.12 Effectiveness factor comparison

Another evaluation of multi-objective GEA testing against other methods is to measure the effectiveness factor. The effectiveness factor is a specially devised metric associated with coverage gains and was used and defined in Chapter 4 and Appendix E.15. It indicates the usefulness of an automated test generation technique, taking into account factors such as the test coverage achieved, and the time and test simulation resources required.

Given that the multi-objective GEA methods actively optimises for coverage and test size concurrently, the effectiveness factor was calculated to be 0.8, which is greater than average effectiveness factors of 0.3 for SAGETEG, 0.1 for µGP, and 0.01 for random methods. These lower efficiency factors are directly due to coverage being the only test goal of these previous test generation methods.

G.13 Additional coverage attainment progress graphs

To supplement the experimental coverage attainment and progress graph results of multi-objective GEA test generation (Section 6.10.5, Chapter 6) for comparison with other methods, this section provides further graph results in the form of coverage versus number of snippets, and coverage versus number of evolutions graphs (Figure G.11 to Figure G.16).

Note that the x-axis in Figure G.11 to Figure G.13 indicate the cumulative number of snippets that has been employed to exercise the SoC. Also, the random test method does not conform to any GEA related technique, hence cannot be plotted against the number of evolutions in Figure G.14 to Figure G.16.

The existing coverage progress graphs from Section 6.10.5 Chapter 6 are also reproduced in Figure G.17 to Figure G.22 here for further discussion and comparisons in Appendix G.13.1.

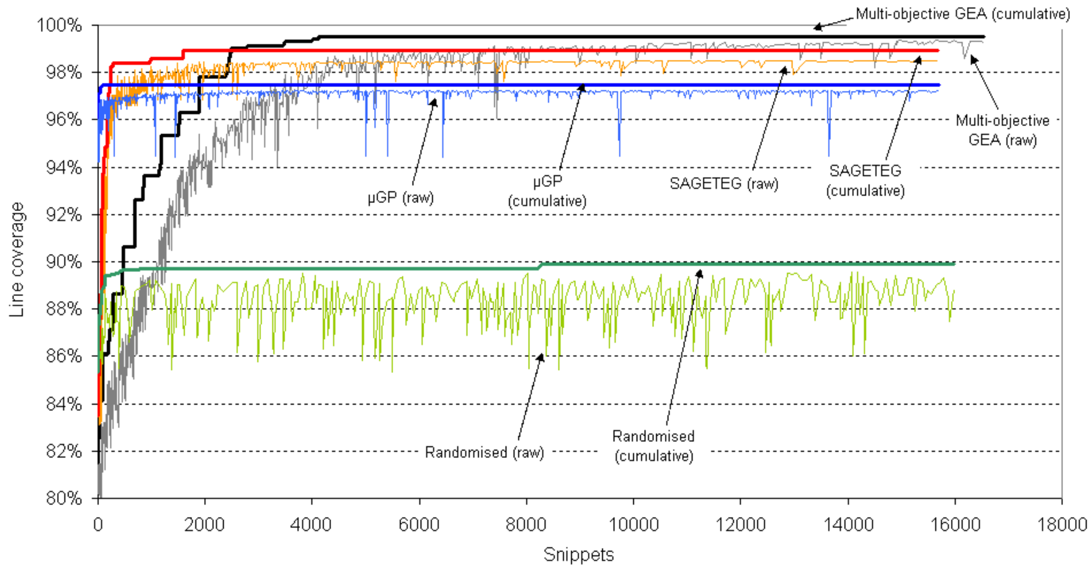


Figure G.11 Line coverage vs. number of snippets

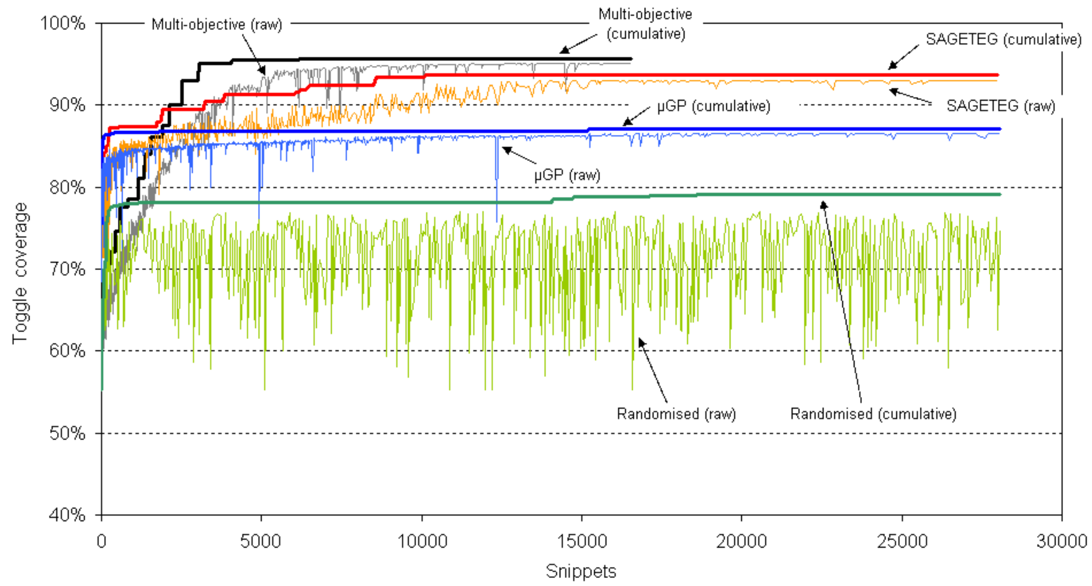


Figure G.12 Toggle coverage vs. number of snippets

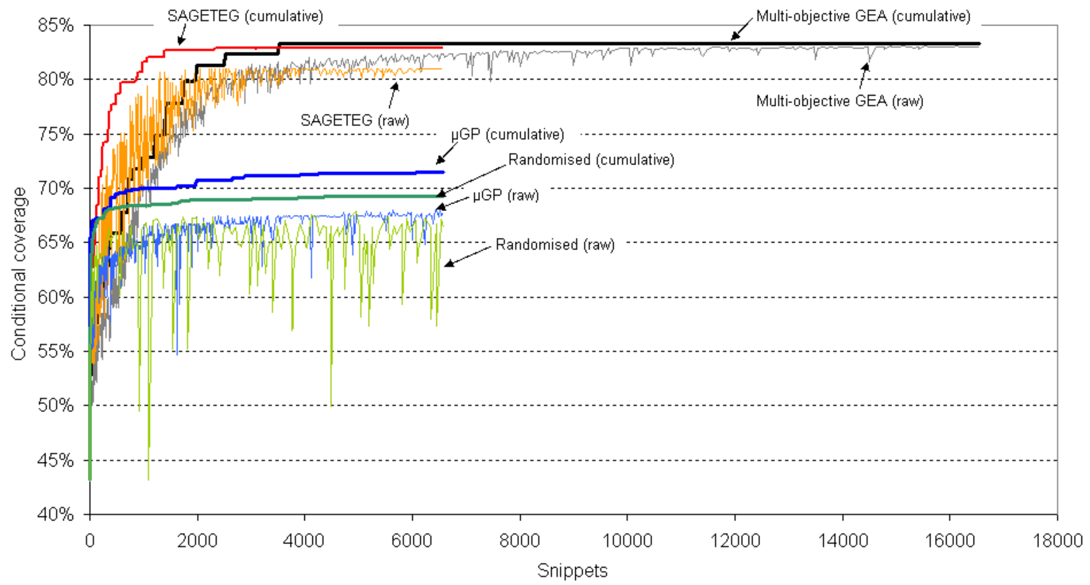


Figure G.13 Conditional coverage vs. number of snippets

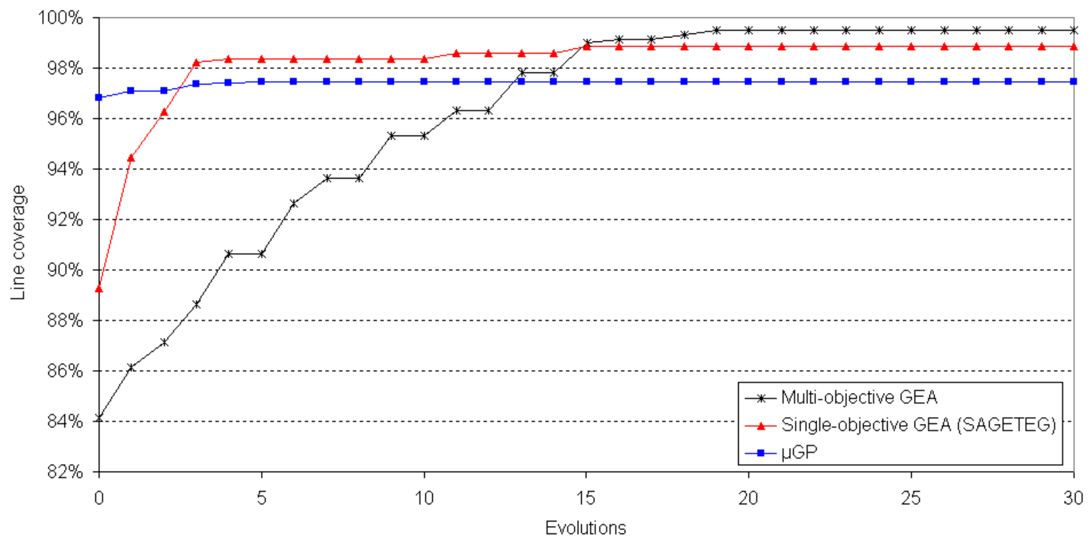


Figure G.14 Line coverage vs. number of evolutions

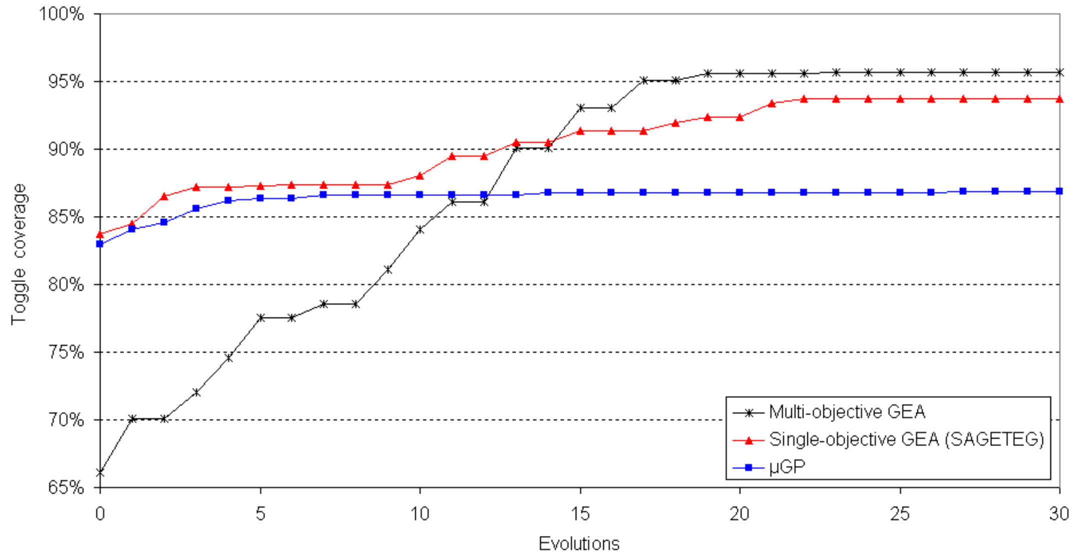


Figure G.15 Toggle coverage vs. number of evolutions

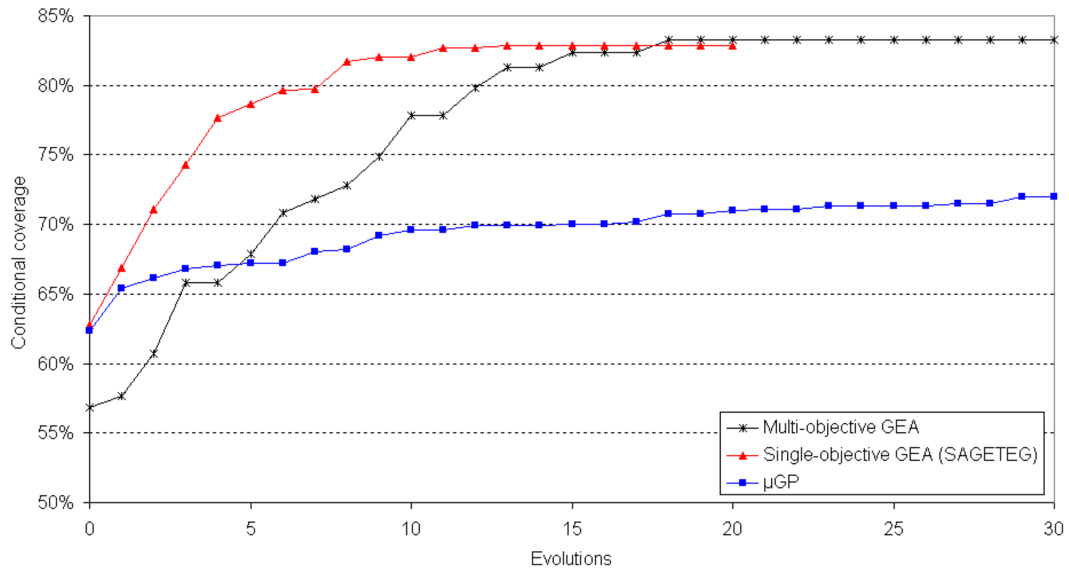


Figure G.16 Conditional coverage vs. number of evolutions

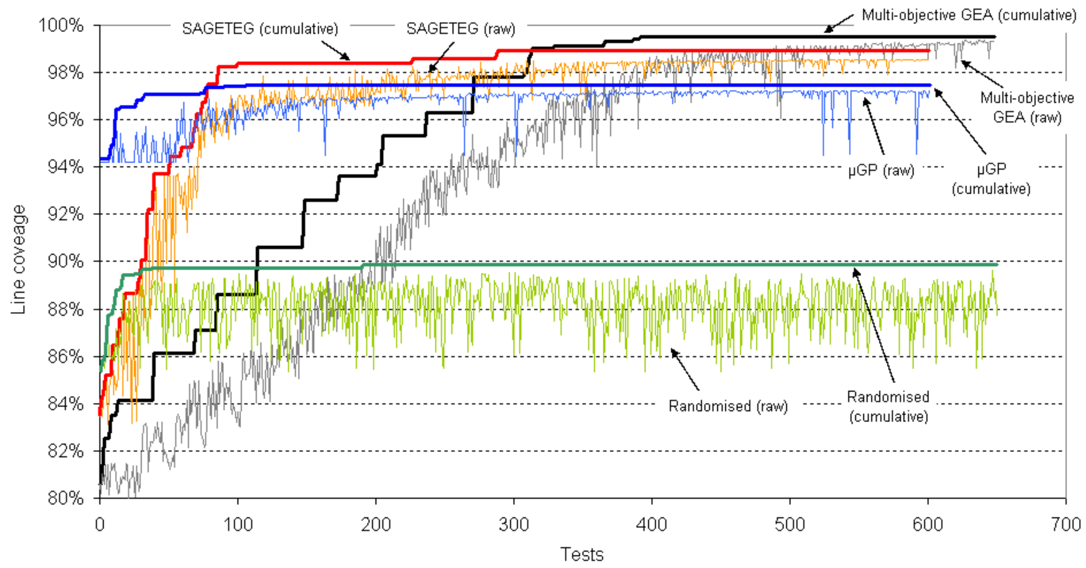


Figure G.17 Line coverage vs. number of tests

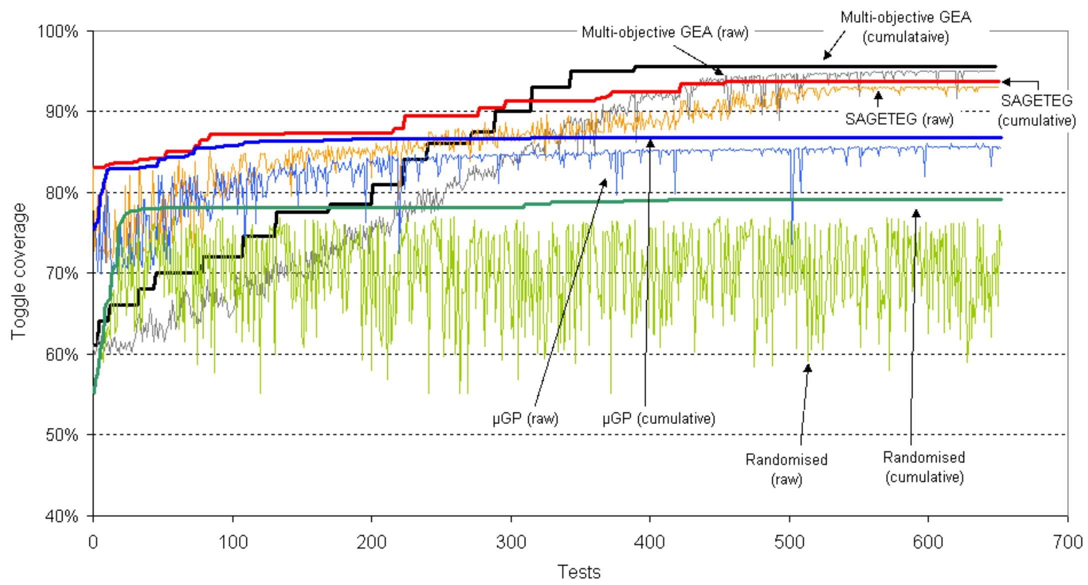


Figure G.18 Toggle coverage vs. number of tests

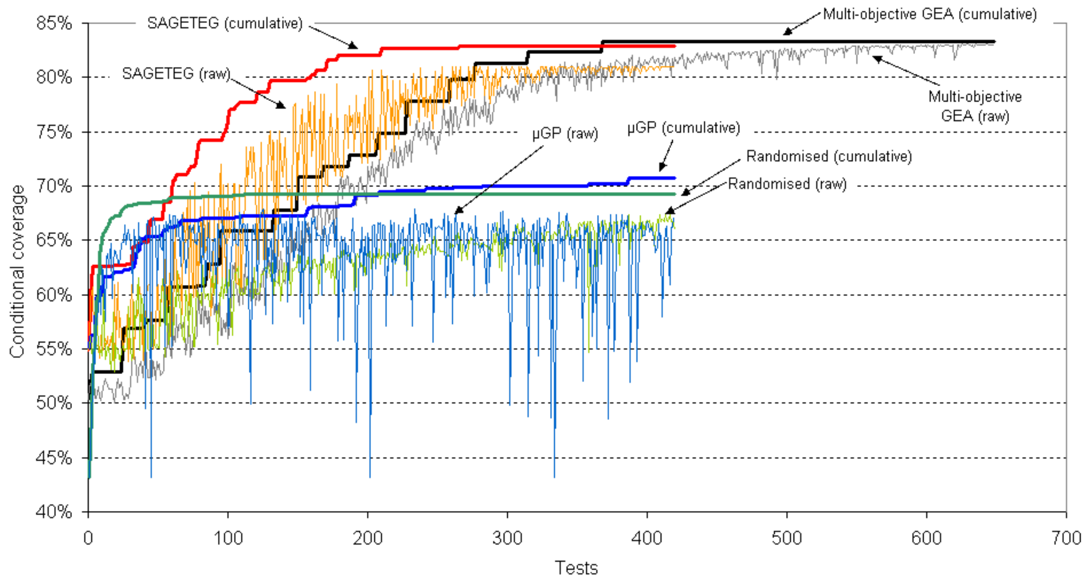


Figure G.19 Conditional coverage vs. number of tests

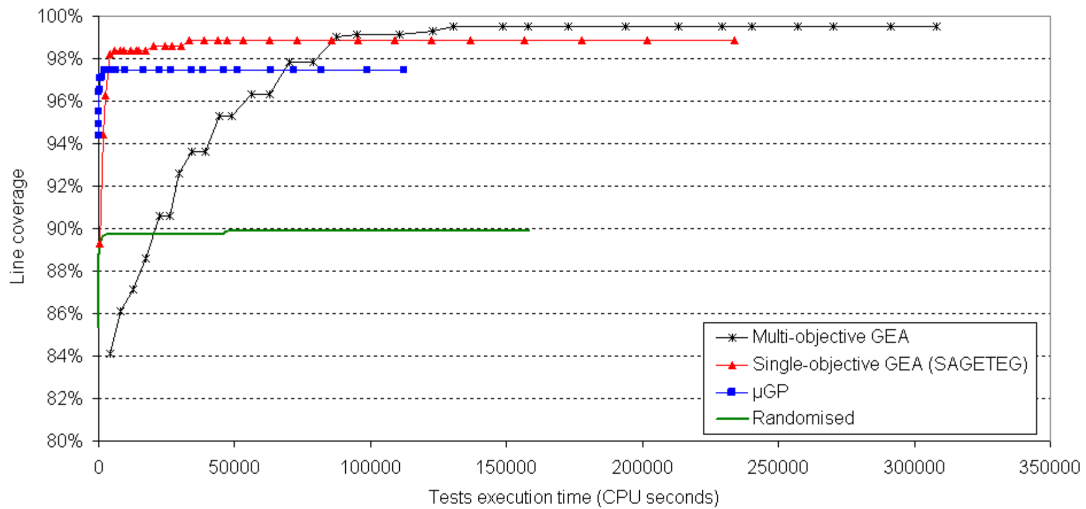


Figure G.20 Line coverage vs. time

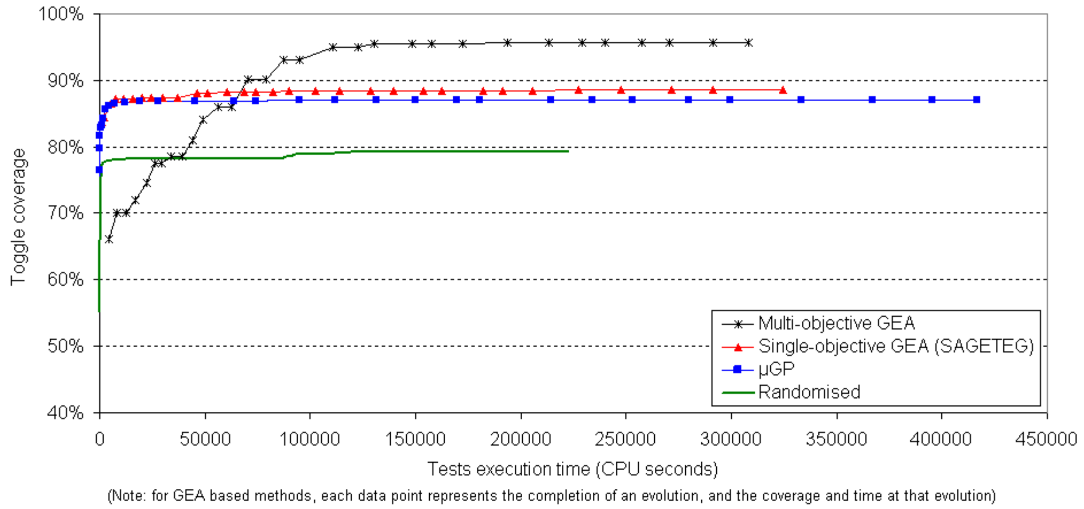


Figure G.21 Toggle coverage vs. time

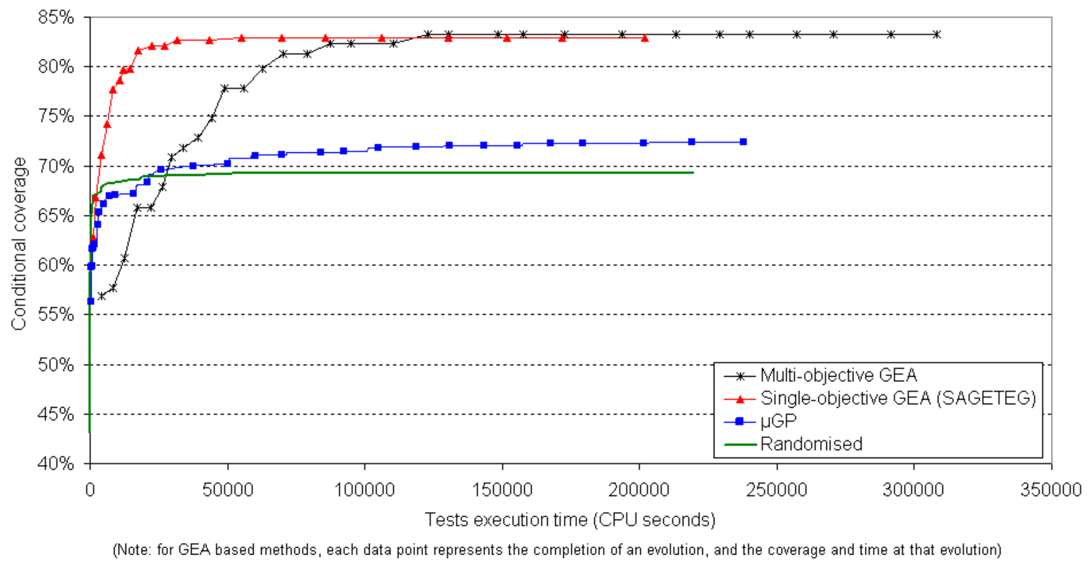


Figure G.22 Conditional coverage vs. time

G.13.1 Additional coverage progress graph observations and comparisons

This subsection describes further observations and comparisons of the multi-objective GEA test generation with other methods, it is a supplement to the discussions on coverage attainment graphs in Section 6.10.5 Chapter 6.

Oscillating raw coverage graph characteristics

The coverage graph lines in Figure G.17 to Figure G.19 reveals interesting characteristics for multi-objective GEA. First, we examine multi-objective GEA raw coverage lines in Figure G.17 to Figure G.19. Multi-objective GEA shows less variation in the range of raw coverage levels. The upper peaks and lower troughs of coverage levels from individual test data points on the graphs are smaller and closer together, compared to the raw coverage graphs lines of other test generations. One reason for this could be due to the more stringent round-robin multi-objective orientated GEA selection process.

For each objective, the number of available test slots for test selection into the next evolution population is less. The tests to be selected are spread across a number of objectives. This implies the number of tests selected specifically for each coverage objective is lower, and hence, only the higher coverage performing tests for a particular objective are selected. Low coverage performing tests will not be chosen at all. Therefore, the gap between coverage levels from consecutive tests and evolutions will be smaller. This accounts for the smaller variation in raw multi-objective GEA coverage graphs.

In contrast, for single objective GEA methods, the entire population for the next evolution is dedicated to tests selected by a single coverage objective only. Hence, there are more test slots, and greater number of low coverage performing tests will be included into the population; after the best performing tests have been chosen. This translates into a raw coverage graph line that displays larger coverage variation in coverage levels.

Next, we observe that multi-objective GEA relies heavily on the re-use and recombination variation of snippet genome from tests that were optimised for multiple coverage objectives. With this approach, there is a possibility for the current population snippet diversity to become overly focused on these set of snippet genome only. Hence, during the GEA process, exploration of test space may be limited if insufficient addition or mutation variation is applied. For example, the multi-objective GEA employs 324 recombination operations compared to 206, 181, 201 and 170 operations for add, subtract, mutation, and replace operations respectively. Also, SAGETEG only employs an average of 257 recombinations throughout its GEA process.

Another observation from the multi-objective graph lines is the occurrence of more low coverage troughs (low coverage downward spikes) toward the end of test evolutions; when compared to SAGETEG. The reason for this is because there may be remaining tests toward the end of the multi-objective GEA process that were selected due to its high fitness for one particular coverage objective type, but lower coverage for other objectives. Despite such tests being unable to provide high fitness

for all objectives overall, they were still selected because of their usefulness for other objectives. For graphs whose coverage objective fitness attained by tests were low, these tests appear as spikes in those graphs.

A fully successful multi-objective GEA process would be able to prevent such lower spikes and return tests that are completely optimal. This is an area in which the multi-objective test generation could be researched further and improved upon.

Conditional coverage test runs and their coverage progress comparison

We also comment on the shorter test runs for conditional coverage by the other test generation methods. As can be observed in Figure G.19, Figure G.13 and Figure G.16, a number of previous conditional coverage test generations were conducted for only 20 evolutions, or approximately 420 tests and 6500 snippets, which is much less than the multi-objective test run. The conditional coverage run was terminated because conditional coverage requires significant amount of CPU simulation and memory resources, and can be exponentially longer to run as individual test size grows.

Unlike multi-objective GEA, the individual test sizes are not constrained and cannot be handled by the memory and processing capabilities of the simulation platform. Due to this, the larger complexity in measuring conditional coverage prompted these previous test generation runs to be halted earlier. Regardless, it can be seen the conditional coverage is well and truly saturated by evolution 20, when similar number of tests and snippets are executed at this evolutionary stage. Furthermore, extending the test run length simply adds to the test execution time without any coverage gain.

For multi-objective GEA however, conditional coverage has only just saturated at evolution 20. Therefore, given that test sizes will be actively constrained where possible, the multi-objective run is extended longer. This ensures the coverage is fully saturated and that any further possible coverage improvements are given opportunity to be realised.

Other comparisons and observations

Besides accounting for shorter conditional coverage test runs, other observations regarding the line and toggle coverage test runs of previous methods can be made. For line coverage, in Figure G.20, given all test generation methods execute similar number of tests, snippets and evolutions, the longer

execution time of multi-objective GEA is directly due to the additional processing required to simultaneously optimise multiple objectives.

In Figure G.21, for toggle coverage test runs, the longer test run times of SAGETEG and µGP is due to the greater number of snippets being executed by these two test generation methods to achieve their coverage levels. Whilst random toggle coverage test runs also executes greater number of snippets, it is peculiar that the test execution time is less. This could be because the snippets are simply chained together randomly, and hence their interactions with one another are not as complicated compared with composing snippets tests via GEA variation and driven by coverage goals. This behaviour requires further investigation.

In Figure G.20 to Figure G.22, note that the first data point for all test generation graph lines represents the time taken to create the initial population of tests. The first data point is further along the time axis for multi-objective GEA because this test population must cater for multiple objectives. Hence, the generation and execution times of multi-objective GEA tests is longer. Because previous test generation methods only require creation and execution with regards to one coverage objective, their initial test population is created and runs much quicker. For example, their first data points in Figure G.20 to Figure G.22 are much earlier closer to zero.

The above argument also applies to the completion of GEA evolutions during testing. In Figure G.20 to Figure G.22, the data points of SAGETEG and µGP graph lines are closer together during early to middle stages of testing. This indicates each of their evolutions complete in shorter durations. As testing continues toward later stages, the data point gaps start to widen, indicating later evolutions require longer time to complete. This is expected because tests will become more complex and larger with more evolutions, in an attempt to gain further coverage. Hence, require longer test execution and evolution completion times.

For multi-objective GEA, the data point gaps and duration for each evolution are not as close together at the start of testing, and slowly increases as testing continues. The change in evolution times and data point gaps do not vary as significantly. This is because multi-objective GEA testing optimises and measures multiple objectives at one time, for all the tests. Hence, the effect of longer and more complicated tests as evolution proceeds further is not as significant. The complexity and additional processes by multi-objective optimisation negates this effect to some extent. Whereas for previous GEA based test generations, early evolutions can be performed quickly for earlier GEA simplistic tests because only one coverage goal needs to be maximised and measured. The effect of large or more complex tests in single-objective based GEA is much greater as evolutions proceed further.

APPENDIX H. Attribute Combinations Coverage

This appendix provides supplemental data for attribute combinations coverage research in Chapter 7.

H.1 Attributes and domains of the Nios SoC attribute combinations coverage

Table H.1 lists the domains employed for attribute combinatorial coverage measuring of the Nios SoC. Table H.2 list the attributes and subset of certain domains and their example domain values for these attributes. Table H.2 shows only a partial list of the example attributes and domain values for example demonstrative purposes for this thesis. The entire list is too large for inclusion into this thesis, and is specified fully in the implementation code of the coverage measuring tool. The T and X domains are not shown in Table H.2. These domains are similar for all attributes, with identical meanings to capture specific or a wide set of common values for the attribute. The domains are used to build up the targ_comb attribute combinatorial set specification goals for each snippet in the control graph.

Table H.1 Domains employed for Nios SoC coverage measuring

Domains	Symbol	Intention
Top level terminal	T	Indicates the attribute contains a single specific value, which can be any attribute value from any domains beneath the T domain. The T domain completes the partial order structure.
Boundary	B	Captures attributes values that are at the boundaries of legal ranges of values for the attribute. Values greater or less than legal ranges are considered to belong to the Illegal domain.
Non-boundary	NB	Refers to attribute values that are within legal operating ranges of an attribute, but is not part of the Boundary domain. Examples include reset or default register attribute values, or specific source or destination I/O port addresses.
Intentional error	InE	Represents attribute values that are asserted due to erroneous SoC conditions, which are explicitly triggered by testing or can occur during normal operation.
Unintentional error	UnE	Represents erroneous, undefined, or unknown attribute values that occur unexpectedly without explicit initiation from testing; or cannot be easily triggered during normal SoC operation. Examples include attribute values that are strictly not allowed by the SoC, or meaningless values that do not correspond to any settings or status of the SoC.
Legal	L	Encapsulates attribute values during normal and error-free SoC operation.
Illegal	IL	Indicates any attribute values when an error or other unknown abnormal illegal condition occurs during SoC operation.
Don’t-care ignore	X	Represents multiple attribute domain values concurrently, it covers all attribute values of other domains. This implies such attribute values can be ignored because they are already exercised and covered by higher domains; or they are irrelevant to an attribute combinatorial set and do not contribute to coverage.

Table H.2 Attributes and example domain values for coverage measuring of Nios SoC

Attributes	Boundary	Non-boundary	Intentional error	Unintentional error	Legal	Illegal
DMA status register	3'b000: idle	3'b101: transferring, 3'b001: transfer just started		3'b110, 3'b111: undefined behaviour	3'b011: transfer completed	
DMA trigger transfer register		0: idle, 1: trigger transfer				
DMA termination status register	3'b000: no termination	3'b001: number of packets transferred 3'b010: read end-of-packet 3'b100: write end-of-packet		3'b111, 3'b011, 3'b101, 3'b110: multiple termination triggered		
DMA incrementation mode register	2'00: no address increment, 2'11: read and write address increment	2'01: read address increment, 2'b10: write address increment				
DMA termination mode register	3'b000: no termination, 3'b111: all termination	3'b001: number of packets transferred 3'b010: read end-of-packet 3'b100: write end-of-packet			3'b011, 3'b101, 3'b110: multiple termination	
DMA read/receive address register	Max and min RAM, ROM, SRAM, Flash memory boundary addresses, odd valued memory address	8'h00940800: UART receive port address, 8'h0094086: PIO port address, Max memory boundary sizes to min memory boundary sizes inclusive address	Max RAM, ROM, SRAM, Flash memory boundary addresses + 2, Min RAM, ROM, SRAM, Flash memory boundary addresses − 2, UART transmit port address,	Max memory boundary size + 1 address, Min memory boundary size – 1 address		

			UART receive port and PIO port address ± 1			
DMA write/transmit address register	Max and min RAM, SRAM, Flash memory boundary addresses, odd memory address	8’h00940804: UART transmit port address, 8’h0094086: PIO port address, Max memory boundary sizes to min memory boundary sizes inclusive address	Max RAM, ROM, SRAM, Flash memory boundary addresses + 2, Min RAM, ROM, SRAM, Flash memory boundary addresses − 2, UART transmit port address, UART receive port and PIO port address ± 1	Max memory boundary size + 1 address, Min memory boundary size − 1 address		
DMA length transfer size register	Zero transfer size, Large transfer size (e.g., 2^8 bytes)	Byte, halfword, word multiple size (eg. 8, 16, 32, 64)		Negative size value	Single value between zero and max DMA length register size	
DMA interrupt enable/disable pin		0: enable 1: disable				
UART receive data register	8’h0: all zeros byte 8’hff: all ones byte	8’h55, 8’hAA: alternating ones and zeros bytes 8’h0F, 8’hF0: lower and upper ones byte				
UART transmit data register	8’h0: all zeros byte 8’hFF: all ones byte	8’h55, 8’hAA: alternating ones and zeros bytes 8’h0F, 8’hF0: lower and upper ones byte				
UART end-of-packet register	8’h0: all zeros byte 8’hff: all ones byte	8’h55, 8’hAA: alternating ones and zeros bytes 8’h0F, 8’hF0: lower and upper ones byte				

UART transfer divisor rate register	3: min rate 10: max rate		0, 1, 2: below min rate	11, 12: above max rate	4, 5, 6, 7, 8, 9: normal rate	
UART transfer complete interrupt register	0: no interrupt 5: receive and transmit interrupt	1: receive interrupt 4: transmit interrupt				
UART end-of-packet interrupt pin					0: enable 1: disable	
UART error interrupt pin					0: enable 1: disable	
UART transfer interrupt pin					0: enable 1: disable	
UART transmit on error interrupt pin					0: enable 1: disable	
UART receive on error interrupt pin					0: enable 1: disable	
UART transfer status register	0: idle 5: receive and transmit	1: receive 4: transmit				
UART end-of-packet setting register					0: no end-of-packet 1: end-of-packet	
UART error setting register					0: no error	1: error triggered
UART transmit on error register					0: no error	1: error triggered
UART receive on error register					0: no error	1: error triggered
PIO data register	8'h0: all zeros byte 8'hff: all ones byte	8'h55, 8'hAA: alternating ones and zeros bytes 8'h0F, 8'hF0: lower and upper ones byte				

PIO direction setting register	8'h0: all zeros byte 8'hff: all ones byte	8'h55, 8'hAA: alternating ones and zeros bytes 8'h0F, 8'hF0: lower and upper ones byte				
PIO interrupt mask setting pins	8'h0: all zeros byte 8'hff: all ones byte					
PIO edge capture signals	8'h0: all zeros byte 8'hff: all ones byte					
CPU global/local registers	Largest positive and negative value, Smallest positive and negative value				Binary multiples of bytes value	
CPU interrupt setting and priority signals	No priority and interrupts set. All priority and interrupts set at same level, No maskable interrupts, All interrupts maskable				Individual priority for each interrupt, Same priority for multiple interrupts	Conflicting priority and interrupt at same level
CPU overflow error condition register	No overflow, all overflow triggered			Non-recoverable overflow error	Single overflow: Arithmetic overflow, register window overflow	
CPU registers/stack depth status	Zero depth, maximum depth				Single non-zero depth above maximum	
Memory configuration register	Memory address range cascaded sequentially	Non sequential memory address range	Overlapping memory address range		SRAM row/column maximum and minimum setting	
Memory access address register	Write/read to multiple memories, parallel access of multiple memories		Write access to ROM or illegal program space		Write/read to RAM, SRAM, Flash memory	

Memory access size pin signal			Non even, byte, halfword, word		Byte, halfword, word size	
Timer control register	4’b0000: idle, 4’b0111: interrupt enabled, continuous mode and start timer		4’b0011: conflicting start and stop timer	4’b0001: enable interrupt without running, 4’b0010: continuous mode without running	4’b0101: interrupt enabled and start timer, 4’b0110: continuous mode and start timer	
Time status register	2’b00: Idle, 2’b10: Running			2’b11: timed out and running state		2’b01: Timed out
Timer upper period setting	8’h0: all zeros byte 8’hff: all ones byte	8’h55, 8’hAA: alternating ones and zeros bytes 8’h0F, 8’hF0: lower and upper ones byte			Single value between 0 and 256	
Timer lower period setting	8’h0: all zeros byte 8’hff: all ones byte	8’h55, 8’hAA: alternating ones and zeros bytes 8’h0F, 8’hF0: lower and upper ones byte				
Timer snapshot counter register	0: minimum counter value 2^32: maximum counter value				Single binary byte multiple value	

Attribute values are provided either quantitatively with associated meanings (e.g., 3’b000: idle), or qualitatively with descriptive meanings (e.g., Max memory sizes).

Quantitative attribute value entry format: *S’T⟨value⟩ : meaning of value*
S is the size and number of digits of the value
T is the type of the value, b for binary, h for hexadecimal
<*value*> is the attribute value in binary or hexadecimal digits.

H.2 Control graph definitions for coverage measuring

In addition to partially ordered domains and its hierarchical structure, the degree of coverage attributes measuring can be managed further by employing control graphs like those from symbolic trajectory evaluation (STE). The control graphs capture all possible snippet execute sequences that can make up a SALVEM test. Using such a graph based coverage model, additional efficiency is provided because coverage measuring is only performed on those attribute combinations that are realisable from these snippet sequences. Other attribute combinations that are not possible from SALVEM testing are not captured by the control graph and do not need to be measured, thus reducing coverage effort.

The control graph is constructed as follows. First, the control graph maps each snippet to a graph node. By doing so, each node represent a major function or operation that the SoC can perform. This allows the coverage method to identify and measure these primary SoC functionalities according to the different attribute combinatorial set at each snippet node. The certification or lack of SoC functionalities exercised by snippets can then be easily measured and examined. Each snippet node specifies a coverage goal, outlining the SoC operations to perform for testing, and the associated attributes combinations to exercise.

Graph edges designate the flow of snippets that can be composed within SALVEM tests. Edges capture the possible sequences of snippets that can be composed under SALVEM test generation. The outgoing edges from each graph node point to the next possible snippet and associated SoC operations that can be initiated during the course of SALVEM testing. Hence, coverage measuring can narrow down the possible attribute combinatorial set of attribute domain values which can be measured at each next snippet node. This allows exercised attribute combinations during testing to be processed more efficiently as the expected attribute combinatorial set can be used immediately for the measurement. The traversal of the control graph during coverage measuring is also facilitated by these edges.

For SALVEM testing, the control graph is defined as follows.

Definition H.1 : Control graph definition

A control graph g is defined as a tuple of the form,

$g = \langle N, E, S, F, A, C \rangle$, where

- N is the set of nodes in the control graph, as defined in Definition H.2, and each node corresponds to a snippet in the snippet library;
- E is the set of unidirectional and unlabelled edges in the control graph;
  $E = \{\langle n_1, n_2 \rangle$: snippet node n_2 is executed by the SoC after snippet node n_1; $n_1, n_2 \in N\}$;

- S is the set of snippet nodes that are at the beginning of a SALVEM test, at which graph traversal can start, $S \subset N$;
- F is the set of snippet nodes at the end of a SALVEM test, at which graph traversal finishes, $F \subset N$;
- A is the set of antecedent attribute specifications that can be assigned to a node, and is described in Appendix H.4;
- C is the set of consequent attribute specifications that can be assigned to a node, and is described in Appendix H.4;

□

Except the set S of start and F of final snippet nodes, any node can have any number of incoming and outgoing edges, allowing for divergent coverage graph measuring. For a control graph, the set of antecedents A and consequents C are building block predicates that formally describe what attribute domain values are to be exercised by testing, and measured for coverage. They are employed to construct formalisations of the attribute combinatorial specifications at each snippet node in Appendix H.4. This restricts the possible attribute combinations and domain values that need to be measured at each node.

The definition of the control graph node is given in Definition H.2.

Definition H.2 : Control graph node definition

A control graph node n is defined as a tuple of the form,

$n = \langle s, I, O, an, cq, cmb \rangle$, where

- s is the designated snippet of the node, $s \in$ snippet library;
- I is the set of incoming edges into the node, $I = \{\langle i, n \rangle$: snippet node n is executed by the SoC after snippet node i; $i, n \in N\}$;
- O is the set of outgoing edges from the node, $O = \{\langle n, o \rangle$: snippet node o is executed by the SoC after snippet node n; $n, o \in N\}$;
- an is the antecedent attribute specification for the node, $an \in A$;
- cq is the consequent attribute specification for the node, $cq \in C$;
- cmb is the informal attribute combinatorial set specification of relevant attributes, it specifies attribute domain values that must be exercised and measured in combination with each other for coverage purposes. The cmb is also equivalent to the target combinatorial set (targ_comb) at a snippet node, and is described in Appendix H.7.

□

Diagrammatically, an example of the control graph node is shown in Figure H.1.

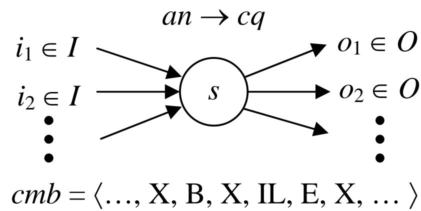


Figure H.1 Control graph node

The attribute combinatorial set *cmb* at each node describes the relevant attributes and desired combinations of partially ordered domain values to evaluate the coverage metric. This was described in greater detail in Section 7.9 Chapter 7. Even though the focus for coverage is to measure the amount of attribute combinatorial values exercised by testing, complementary measure of test coverage effectiveness could also be deduced from how widespread the control graph has been exercised. The control graph implicitly captures all possible SoC behaviours that can be exercised by SALVEM testing, hence indicates how effective the tests has been exploited to test the SoC. Currently, this avenue of investigation is beyond the scope of the coverage research.

Using the test execution trace of attribute values and by matching these exercised attribute combinations against the information described at each snippet node, the coverage graph is traversed according to the operations of the SALVEM test program. For the Nios SoC, the coverage attribute control graph employed is shown in Appendix H.3.

From Figure H.1 (and the Nios SoC control graphs of Appendix H.3), a snippet node in the control graph can be traversed from any number of prior nodes, and can traverse to other multiple nodes (including itself). The multiple input and outgoing graph edges imply graph traversal can follow numerous paths, which must be resolved effectively. By formally specifying antecedents and consequents, the graph traversal can be conducted using techniques from STE to only check and match minimal attributes that are required by the snippet node to determine if the node is to be traversed. Such a process in STE is termed trajectory checking, and is adapted for use along with other abstraction mechanisms for our coverage process as described in Section 7.8 of Chapter 7.

H.3 Control graphs used for attribute combinations coverage

Figure H.2, Figure H.3, Figure H.4, Figure H.5, Figure H.6, and Figure H.7 show the snippets based control graphs used for attribute combinatorial coverage measuring. Whilst the figures are shown individually, in fact, they are combined together to form the entire overall control graph for coverage measuring as a whole.

Given the size and makeup of the control graph, for simplicity and better representation, the graph and snippet nodes are divided and grouped into individual dashed boxes according to the type of device the snippet is primarily intended for. Each dashed box of snippets shows the traversal of snippet nodes possible within each group of snippets. For example, for DMA snippets, if the current traversed snippet node is the CheckDMA, then the next possible snippets that can be traversed to within the DMA snippet group are the InitDMA and ExecDMA snippet nodes. Alternatively, the traversal can proceed to a snippet of another snippet group.

There are incoming IN and outgoing OUT arrows for each group of snippets, which represents the traversal of snippets from and to other groups of snippets. That is, a snippet with a connection from the IN incoming arrow can be traversed into from another snippet of some other group of snippets. Similarly, a snippet with a connection to the OUT outgoing arrow can traversed to another snippet of some other group of snippets. Via the IN and OUT arrow connections, applicable snippets from any group can traverse to applicable snippets of other groups. All snippet groups are connected bi-directionally with one another. If the graph connection of the snippet groups was created, the snippet groups would be connected in a mesh configuration.

The traversal of the control graph can begin at any of the snippets connected to the IN arrow of any group of snippets. Graph traversal can terminate after any snippet node whenever the next exercised combinations do not match any of the possible next nodes that can be traversed to, or when all exercised combinations have been processed.

Note that in some snippet group's control graph, a snippet that initialises or resets the particular device for the snippet group is not required to perform any operations immediately on the device. Therefore, the control graph reflects this by allowing for a direct connection leading out from the reset/initialisation snippet to the OUT arrow of the snippet group's control graph. At a later stage, when operations are tested on the device, the control graph allows for direct connection into the snippet group's control graph again to the relevant snippet.

Finally, the design and usage of the control graph could have been implemented at a lower level than currently employed. The control graph hierarchy of nodes can go beyond the snippet node level to examine the attribute combinations exercised by operations conducted by the snippet. This was the original intention of the control graph design and coverage measuring, and would have provided for greater control and fine-grained measuring of the functional coverage information that can be extracted. However, conducting coverage measurements at such lower level would result in too complex and more resource intensive processing. The current level of control graph and function coverage measure data that can be extracted is deemed adequate, and does not overly exert the processing required.

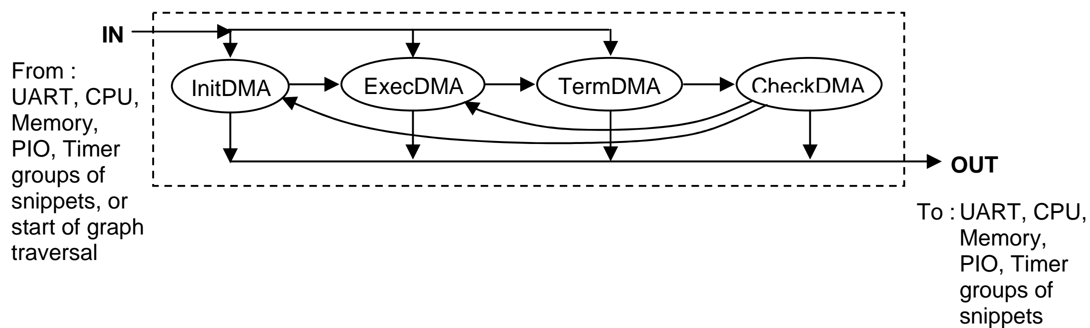


Figure H.2 Control graph portion for group of DMA snippets

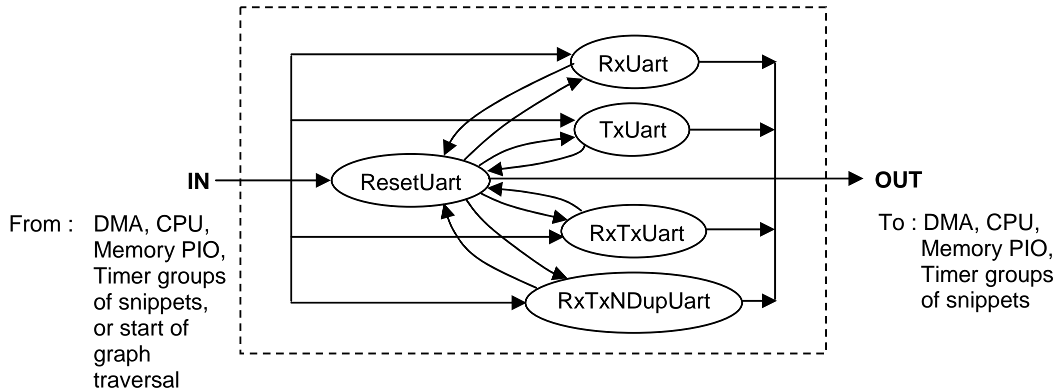


Figure H.3 Control graph portion for group of UART snippets

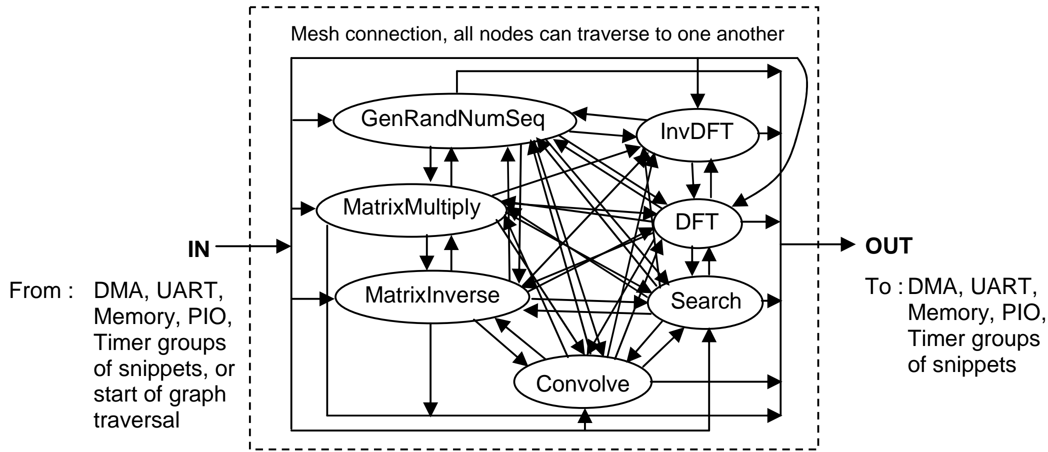


Figure H.4 Control graph portion for group of CPU snippets

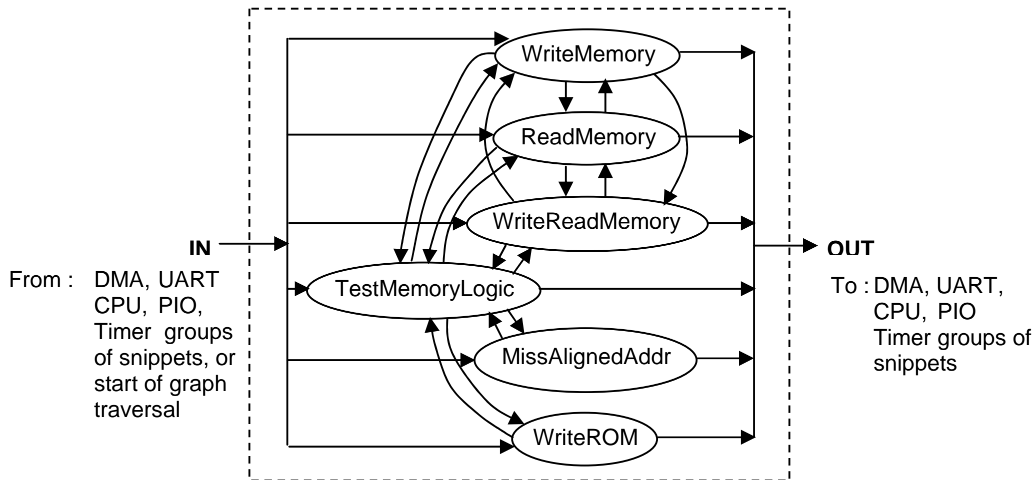


Figure H.5 Control graph portion for group of Memory snippets

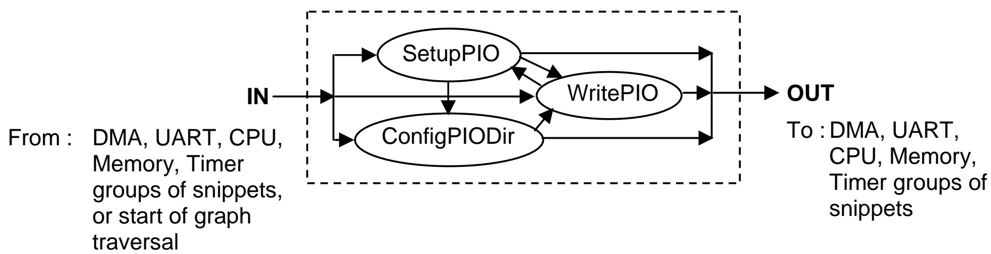


Figure H.6 Control graph portion for group of PIO snippets

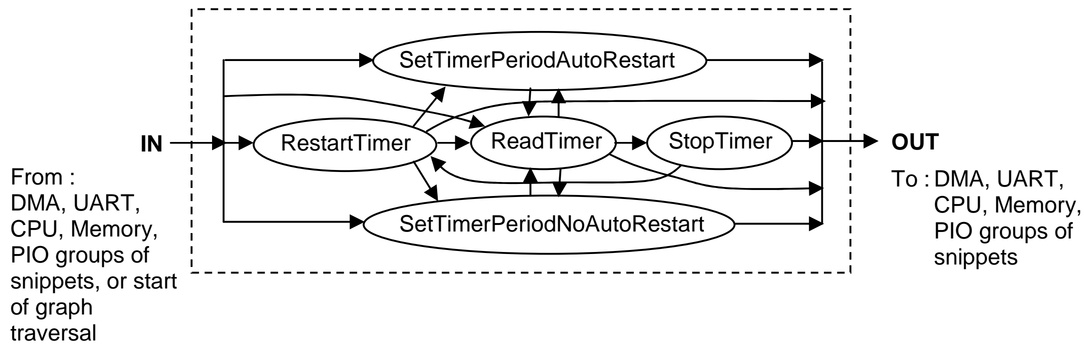


Figure H.7 Control graph portion for group of Timers snippets

H.4 Coverage goal specifications with antecedents and consequents

Antecedents and consequents are formalised specifications of attributes and partially ordered domain value combinations; which are realised at each snippet node during test and coverage measuring. They allow STE based methods to perform control graph traversal for coverage measuring. First, a brief background of the rationale behind STE antecedent and consequent specifications for describing attributes is presented.

The adaptation of STE for coverage measuring, specifically its control graphs and antecedent consequent specifications, was motivated by similarities between STE circuit checking states and coverage attribute combinations. An attribute combination exercised by a snippet is similar to a circuit state exercised by the hardware circuit in STE. As tests are executed, the coverage attributes advances from one combination to the next, not unlike the transition of circuit states during hardware operation.

In classical STE, antecedents set up desired circuit states that correspond to some function of the circuit. Subsequently, at the next node, consequents specify the expected circuit response to this function. This *set up - check response* pattern is made possible by specifying inputs and storage elements in antecedents, and outputs and storage elements in consequents.

In the coverage domain, a snippet *set up – check response* pattern can also be established. This enables modified STE algorithms to check for realisation of desired and expected coverage combinations. Antecedents predominately specify attributes of write and read-write registers, whilst consequent denote read and read-write registers.

The attribute antecedent and consequent specifications are composed of two basic elements, which are used recursively to describe the attribute combinatorial domains to exercise. Definition H.3 defines the simple predicate and conjunction elements used within antecedent consequent specifications.

Definition H.3 : Simple predicate and conjunction elements

Let f be the antecedent or consequent specification, which can be expressed using the following simple predicate and conjunction elements defined as follows.

(i) Simple predicate : $f = (a \text{ is } p)$ where $p \in P$ and $P = \{\text{T, B, NB, InE, UnE, L, IL, X}\}$

A simple predicate f expresses an attribute a must exercise a specific partially ordered domain value.

(ii) Conjunction : $f = (f_1 \wedge f_2)$ where f_1 and f_2 are either simple predicates or further conjunctions of simple predicates.

The conjunction element allows multiple simple predicates to be recursively specified and chained together. For example, the conjunction components f_1 and f_2 can themselves be conjunctions of other conjunction components, which eventually are broken down to single simple predicates.

□

The notations in Definition H.3 are reused primarily to maintain consistency with traditional STE circuit checking antecedent and consequent descriptors. The use of simple predicates and conjunctions ensure only the desired and expected attribute partially ordered domain combinations need to be exercised by testing, and measured as part of coverage. Employing these building block elements for antecedents and consequents allows for the minimal set of attributes to be processed; to check for consistency with snippet behaviours according to the control graph. This reduces the amount of coverage measuring. Definition H.4 defines the antecedent and consequent specifications.

Definition H.4 : Antecedent and consequent specification

Let an be the antecedent, composed recursively of simple predicate and conjunction elements from Definition H.3.

$an = (a \text{ is } p) \wedge f$

where f contains further simple predicates and conjunction of predicates, or is an empty function otherwise.

a and attributes in f are SoC configuration registers that assert and initiate SoC operations, for example write access control registers.

Let *cq* be the consequent, composed recursively of simple predicate and conjunction elements from Definition H.3.

$cq = (a \text{ is } p) \wedge f$

where f contains further simple predicates and conjunction of predicates, or is an empty function otherwise.
a and attributes in f are SoC configuration registers that indicate and monitors SoC operations, for example read access status registers.

□

H.4.1 Least upper bound (LUB) operation for specified domains

This section describes the specific outcomes of least upper bound operations between domains specified in Figure 7.3 of Chapter 7. In addition to Definition H.3, note that a special case of the LUB exists whereby the least upper bound of two domains are also given as a disjunction ∪ if a legal domain or legally derived domains (i.e. B or NB) is LUB'ed with domains from the illegal derived domains, or vice versa. Given that the boundary and non-boundary, and intentional and unintentional error domains are derived from the disjoint legal and illegal domains, it is important to ensure that the attribute values of these disjoint domains are exercised and measured individually rather than being assigned the X domain immediately.

The LUB operation for domains in the partially ordered structure of Figure 7.3 Chapter 7 is best expressed in the format of a truth table in Table H.3.

Table H.3 LUB operation truth table

LUB	T	B	NB	InE	UnE	L	IL	X
T	T	B	NB	InE	UnE	L	IL	X
B	B	B	L	B ∪ InE	B ∪ UnE	L	B ∪ IL	X
NB	NB	L	NB	NB ∪ InE	NB ∪ UnE	L	NB ∪ IL	X
InE	InE	B ∪ InE	NB ∪ InE	InE	IL	L ∪ InE	IL	X
UnE	UnE	B ∪ UnE	NB ∪ UnE	IL	UnE	L ∪ UnE	IL	X
L	L	L	L	L ∪ InE	L ∪ UnE	L	X	X
IL	IL	B ∪ IL	NB ∪ IL	IL	IL	X	IL	X
X	X	X	X	X	X	X	X	X

The least upper bound operation is similar to that employed for STE circuit checking between the signal logical states of {0, 1, X}. Taking the LUB between these states allows for the don't-care state X to be employed more often during circuit evaluation; and hence reduces the amount of states that need to be explicitly checked. The LUB operation and its benefits in STE are shown in the example in [BBS91, Che03]. Similarly, taking LUB of attribute domains to produce lower level domains for use in coverage measuring reduces the coverage effort required.

H.4.2 Conversion of antecedent and consequent specifications to attribute combinatorial set

Given the definition of LUB, the conversion of formalised antecedent consequent specifications to an informal attribute combinatorial set is described in Definition H.5.

Definition H.5 : Converting antecedent and consequent specifications to attribute combinatorial set

Let $\delta : f \rightarrow CMB$ be a function that takes in an antecedent or consequent specification and converts it to an equivalent attribute combinatorial set; where f is the antecedent or consequent composition of simple predicates and conjunction from Definition H.3, and *CMB* is the attribute combinatorial set from Definition 7.2 of Chapter 7.

The function δ can be recursively applied to the simple predicates or conjunctions within antecedents and consequents to form the overall attribute combinatorial set.

(i)

For a simple predicate (a_i is p), from Definition H.3 (i), let Z be the set of attributes for the snippet node.

$$\delta(a_i \text{ is } p) = \langle \mathrm{X}, \ldots, \mathrm{X}, P, \mathrm{X}, \ldots, \mathrm{X} \rangle,\ \forall a_j \in Z \text{ and } j \neq i,\ D_j = \mathrm{X}$$

where attribute a_i is the ith attribute in the combinatorial set tuple that holds partial domain p, and all other attributes are assigned the X domain.

The δ function constrains only the attribute in the simple predicate to the specified partial order domain. All other attributes are assigned X so that they can assume any value and do not need to be explicitly examined for coverage purposes.

(ii)

For conjunctions ($f_1 \wedge f_2$), from Definition H.3 (ii),

$\delta(f_1 \wedge f_2) = \delta(f_1)$ LUB $\delta(f_2)$ where LUB is the least upper bound operator, operating on the partially ordered domain structure in Figure 7.3 of Chapter 7.

The δ function recursively breaks down conjunctions to eventually operate on simple predicates producing the appropriate attribute combinatorial set in (i). Using the LUB operator, the attribute combinatorial set of these conjuncted simple predicates are combined together to form the overall attribute combinatorial set from the antecedent or consequent.

The use of δ as the antecedent consequent conversion function symbol is to maintain consistency with the corresponding trajectory state building operation in STE.

□

The antecedent and consequent, along with the converted attribute combinatorial set describes the coverage goals at each snippet node, and collectively the verification goals of SALVEM testing. The antecedents and consequents at a node capture the range of application behaviours each snippet must verify. The antecedent specifies domains for attributes that are being driven by SALVEM test programs and the SoC. The consequent specifies attribute domains to ensure correct and required attribute values are exercised by the test programs.

From antecedents and consequents, the adaptation of attribute combination merging in Definition H.5 is similar to STE's process of evaluating defining trajectory sequences of circuit states during each time step of circuit operation. STE builds up minimal circuit states to assert and check using antecedents and consequents. Similarly, our coverage method constructs attribute combinatorial sets that prescribe only the necessary attributes and domain values to exercise by testing and coverage measure. The antecedents and consequents define the minimal sequence of attribute values that must be realised, in order to capture the range of attribute combinations and associated SoC functionalities at each node. By doing so, adapted STE techniques can be reused to allow efficient node matching.

H.5 Coverage measuring graph traversal

Based on the control graph definitions in Definition H.1 and Definition H.4, the graph traversal method is depicted with pseudo code in Figure H.8. Given a graph *g*, the graph traversal is initiated by executing Traverse[*g*, *S*] where *S* is the set of snippet nodes from where graph traversal may

commence. The traversal function is called recursively, and can be terminated when either the entire trace of exercised attributes combinations is processed or a terminating snippet of the control graph is encountered (i.e. a snippet from the set F of snippet nodes at the end of SALVEM tests).

```
1   Traverse [g, N] {
2
3       foreach n in N {
4
5           // From Definition H.5, attain the simplified attribute combinatorial set cmb from
6           // the antecedent an and consequent cq specification f_an and f_cq respectively.
7           cmb = δ(f_an) LUB δ(f_cq)
8
9           // Determine if the test exercised attribute combination ex matches node n
10          if [Node_Match(ex, cmb) = true] {
11
12              // Further evaluate the coverage metric, Section 7.9 Chapter 7
13              Evaluate_Coverage_Metric(n, ex)
14
15              // Attain the next exercised attribute combination during testing of test t
16              ex = Get_Next_Exercise_Combination(t)
17
18              // Terminate graph traversal if there is no exercised combination
19              if [ex = ∅] {
20                  exit_Traverse_function
21              }
22
23              // Terminate graph traversal if n is part of the set F which are the last
24              // nodes in the control graph and at which traversal terminates
25              if [n ∈ F] {
26                  exit_Traverse_function
27              }
28
29              // Attain the set of next potential nodes that can be traversed to from
30              // the current node n, from the set of outgoing edges O of n
31              N = Next_Outgoing_Nodes(O)
32
33              // Recursively invoke Traverse function to continue graph traversal
34              Traverse [g, N]
35
36              // Only one node is guaranteed to match the exercised attribute
37              // combination,
38              exit_loop
39
40          }
41      }
42  }
```

Note : comments are pre-pended by '//'

Figure H.8 Control graph traversal method pseudo code

In Figure H.8, whenever the snippet node matches, the coverage evaluation carried out at line 13 involves updating the coverage metric. Briefly, the quantitative coverage metric is increased if the exercised attribute combination has not been triggered previously, and the snippet exposes new test space that contributes to test coverage. To ensure efficient and manageable attribute combinatorial processing, this coverage metric evaluation operation performs other attribute domain and combinatorial manipulation; Section 7.9 in Chapter 7 outlined this process in detail.

H.6 Control graph traversal snippet node matching pseudo code implementation

The pseudo code implementation of the snippet node matching operation conducted for control graph traversal is given in Figure H.9. The formalised definition of the snippet node matching was given in Definition 7.5 in Chapter 7.

```
1   Node_Match [ex, cmb] {
2
3     // For the current snippet node under examination for graph traversal,
4     // ex : is the set of exercised combination of attribute values from testing.
5     // cmb : is the set of domains specified for the targ_comb attribute combinatorial set of
6     //         the current snippet node.
7
8     i = 0
9     foreach [ei ∈ ex] ∧ [di ∈ cmb] {
10        if [ei ≤ di] {
11            i = i + 1  // increment until i is the last index in the tuple of attributes in the
12                       // combinatorial set for the current snippet
13        } else {
14            return false    // snippet node does not match
15        }
16    }
17    return true     // all exercised attribute values are covered by the snippet node
18                    // targ_comb, snippet node matches
19
20 }
```

Figure H.9 Pseudo code implementation for snippet node matching operation

H.7 Target combinatorial set (targ_comb)

Motivations of the target combination

The need for target combinations arises from graph traversal and coverage metric evaluation. Referring to Figure H.8 again, during graph traversal, the snippet node matching operation, *Node_Match* at line 10 is the primary facilitator for advancing through the control graph. The goal of the node matching process is to match the exercised attribute combination to the relevant snippet for which the attribute combination was triggered by; thus measuring the coverage of the attribute combination to the appropriate snippet. Rather than manipulate formalised antecedent and consequent specifications directly, the node matching operation examines the corresponding attribute combinatorial set of the antecedent and consequent; which was simplified from the antecedent and consequent as described in Definition H.5 using the δ operator.

Description of the target combination

Recall from H.4 that the antecedent and consequent derived attribute combinatorial set is in fact equivalent to a description of the coverage goals at the snippet node. This coverage goal based combinatorial set is denoted as the target combination (targ_comb) of the snippet node. That is, targ_comb $\in CMB$ from Definition 7.2 Chapter 7.

The targ_comb captures the attributes combinations that must be exercised during testing for coverage measurement. At each snippet node, the antecedent consequent simplified targ_comb specifies the type and range of attribute combinations that must be realised by SALVEM tests to achieve full coverage for that particular snippet node; and collectively, full coverage for the SoC when undertaking SALVEM verification.

The targ_comb restricts the range of attribute values and combinations of these values using partially ordered domains as described in Section 7.5 of Chapter 7. This constrains the required coverage space to exercise into smaller regions, as depicted in Figure 7.4 Chapter 7. Hence, the goals for coverage measurement at each snippet node (and the overall SoC coverage goal) are essentially a restricted sub portion of the test space, corresponding to the important and interesting SoC functions that require explicit testing.

The identification and specification of what SoC functionalities are critical or require special test focus depends on the verification team. The antecedents, consequents, and eventually the targ_comb

provides a means of describing and narrowing down these test scenarios that must be verified, which effectively are the coverage and verification goals.

During coverage measurement, the targ_combs serves two purposes. In Section 7.8 Chapter 7, the targ_combs are also used for matching against the exercised attribute combination test trace, to ensure the exercised attribute combination is evaluated for the correct and applicable snippet node. This facilitates snippet node matching and control graph traversal. Compared to strictly formal antecedents and consequents, using targ_combs for snippet node matching is less complex and more efficient. The control graph snippet node matching was described in Section 7.8 Chapter 7.

The targ_comb are used for coverage metric evaluation to assess exercised attribute combinations against the set of combinations required by testing. That is, how much of the attribute combinations governed by the targ_comb coverage goals have been realised, so as to evaluate the quantitative coverage metric and reveal progress of verification. The use of targ_comb for coverage metric measurements is described in Section 7.9 Chapter 7.

H.8 Cumulative combinatorial set (cumu_comb)

Example of cumu_comb usage

The strategy in employing the cumu_comb to keep track of exercised attribute combinations using partial order domains is illustrated diagrammatically in Figure H.10. It depicts an example of the overall coverage measure process. In the example, the initial domain assigned to attributes in the cumu_comb is T, which is the single valued domain at the peak of the partial order structure (Figure 7.3 Chapter 7). In the beginning, the T domain signifies any single attribute value can be exercised. Until such time when sufficient attribute values have been exercised such that all values from a partial order domain are fulfilled, the attribute domain in the cumu_comb shall remain as T. As more attribute combinations are realised, these attributes get replaced with other domains lower in the partial order structure.

For example, the progression of the DMA termination attribute (DMATerm) commences from T to the boundary (B), then also the non-boundary (NB), and eventually to the legal (L) domain. This indicates during testing, values from the DMA termination boundary domains were exercised, until all values from this domain were covered and hence the attribute domain transitioned to B. Eventually, as other attribute values are realised, the entire set of legal domain values become satisfied, and the attribute

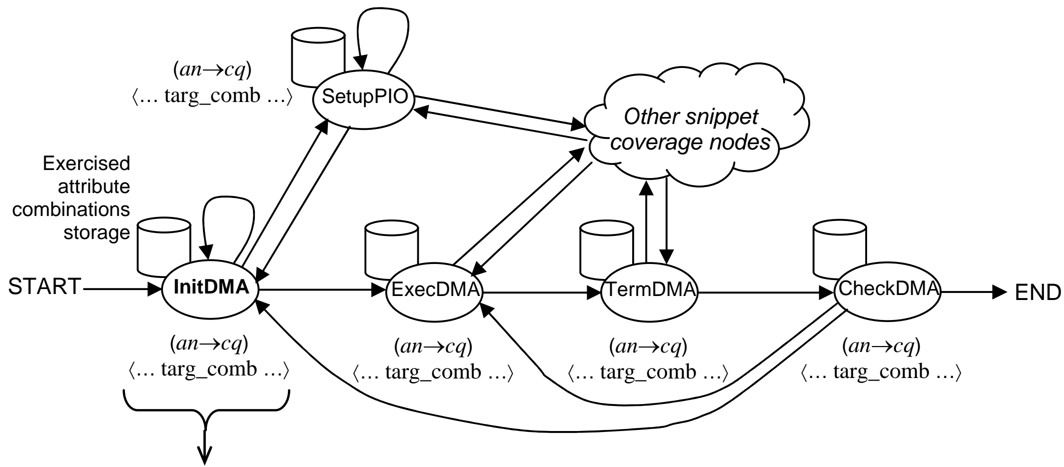
domain is replaced with the legal L domain. At this stage, given the targ_comb specifies the L domain for the DMATerm attribute as well, the coverage goal for this attribute can be considered met. As testing continues, more combinations of attribute values are exercised, and other attributes' domain in the cumu_comb eventually achieve their corresponding target domains in the targ_comb. This signifies the coverage goal for that snippet is achieved.

The example of the T domain transitioning to the target legal domain is based on a scheme whereby attribute values are exercised with the aim of replacing attribute domains in the cumu_comb with lower ordered domains. Recall that the partial order structure in Figure 7.3 Chapter 7 is ordered by information content, and that the lower ordered domains represent greater coverage of attribute values. The goal is to direct testing to produce attribute values toward these target domains, and by doing so, continually *reduce* the cumu_comb domains of these attributes lower down the partial order hierarchy. This indicates the desired attribute values of the target domain are indeed captured.

The concept of directing attribute value testing toward lower ordered domains is beneficial because it restricts testing to only focus on the essential attribute values, out of all the possible attribute combinations. Besides reducing the state size of what must be exercised, this approach facilitates more efficient coverage measuring, as attribute values are managed in terms of a selected set of domains, rather than the entire set of possible raw values. A full treatment of this attribute domain reduction strategy was outlined in greater detail in Section 7.9.3 Chapter 7.

Finally, in Figure H.10, note that the X domain was also used to initialise one of the attribute in the cumu_comb. Such an initialised attribute implies no particular attribute values need to be specifically exercised or targeted for this attribute. This attribute can be considered as don't care during test and coverage measuring. In fact, the meaning of X domain in the partial order structure signifies all attribute domain values in the partial order, given that X is the lowest domain. Hence, assigning X initially signifies all values of the attribute and any potential attribute value goals are covered already, and the attribute can be ignored.

This example in Figure H.10 demonstrates one of the main 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. Appendix H.14 details more comprehensive examples of the cumu_comb attribute update operations, and evolution process of cumu_comb attribute domains during coverage measure. Besides guiding the test and coverage measure process, the cumu_comb is also essential to evaluating the quantitative coverage metric.

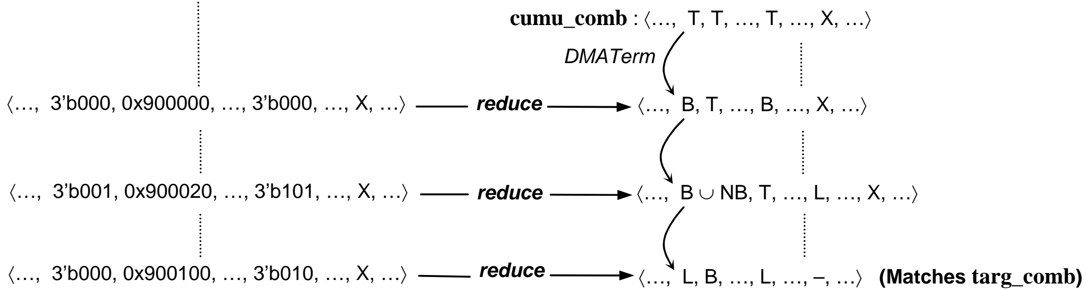


InitDMA attribute combinatorial set : ⟨…, *DMATerm*, *DMAReadAddr*, …, *DMAStatus*, …, *UARTEop*, …⟩
Antecedent→Consequent specification : ((***DMATerm* is L** & *DMAReadAddr* is B) → (*DMAStatus* is L))
InitDMA target combination targ_comb : ⟨…, **L**, B, …, L, …, –, …⟩

Processing of exercised combinations, update and reduction of cumu_comb at InitDMA snippet node during testing

Exercised combinations matching InitDMA node targ_comb during testing :

Reductions at cumu_comb of InitDMA node :



other exercised combinations matching InitDMA targ_comb during testing

For *DMATerm* attribute, combinations exercised during testing such that cumu_comb has been updated and reduced from T to the legal domain, which is the target domain specified by the targ_comb coverage goal.

Attributes:
DMATerm : DMA termination mode attribute
DMAReadAddr : DMA source read device address
DMAStatus : DMA transfer active/inactive status
UARTEop : UART transfer termination end-of-packet

Figure H.10 Example usage of cumu_comb for coverage measuring, and update and reduction

H.9 Conventional combinations based coverage evaluation

To ensure accurate coverage measuring and unique combinations, the basic concept in measuring attributes combinations coverage is to keep track of all the exercised combinations, and only increment the coverage metric if new combinations not previously exercised are realised. Therefore, the intuitive and simplistic approach is to compare each exercised combination against the entire set of combinations previously realised. Additionally, any unique combination exercised will also need to be stored for future evaluation of other exercised combinations. This basic coverage metric evaluation concept is summarised in Figure H.11.

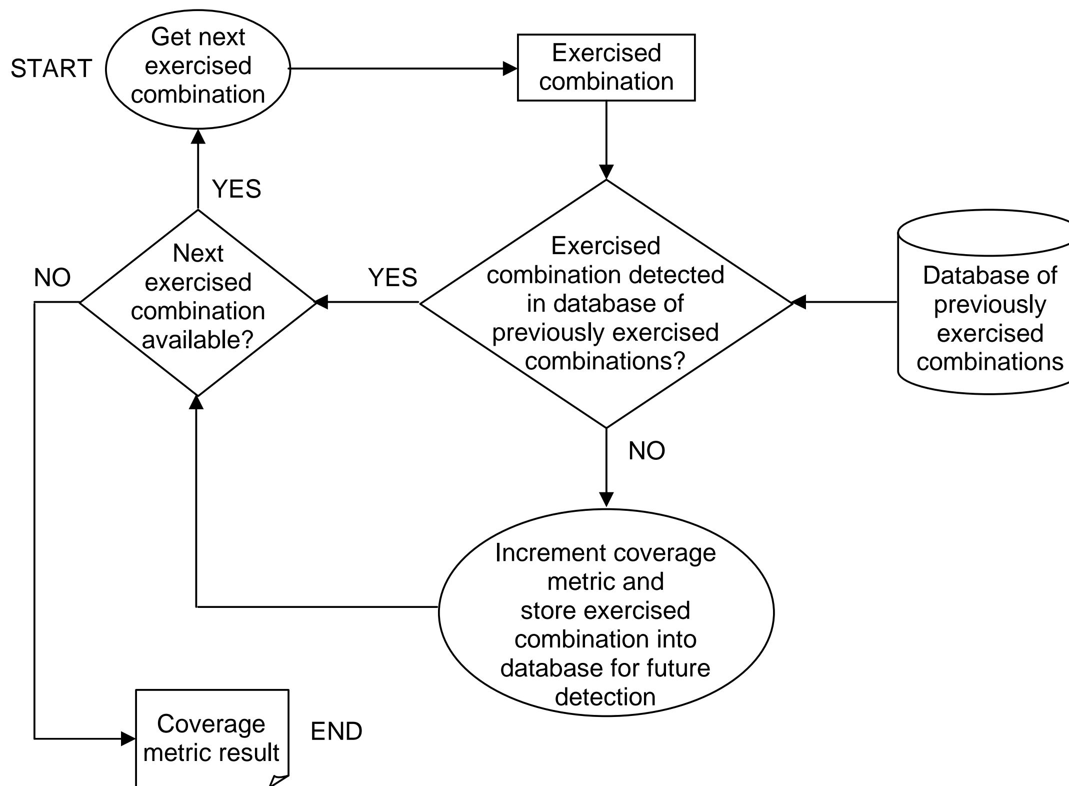


Figure H.11 Conventional attribute combinations based coverage metric evaluation

Simply employing a strategy such as in Figure H.11 creates a number of issues. First, every exercised combination must be compared against previous combinations. If the exercised combination is unique, then it must be compared individually against all previous combinations to prove it has never been realised before. Only then can the coverage contribution from this attribute combination be included in the coverage metric.

Despite the existence of efficient sorting or search methods, individual comparisons of all combinations are still an expensive exercise. The comparison effort also increases with every realisation of a unique combination. The addition of each unique combination into the exercised

combinations database accumulates the number of comparisons that must be performed for subsequent exercised combinations. Eventually, this imposes a hard limit on the combinations that can be stored and processed, and prevents further coverage measuring.

H.10 BCM and WCM evaluation

This section of the appendix provides supplemental details regarding the Best Combinations Metric (BCM) and Worst Combinations Metric (WCM) evaluation.

General observations of BCM and WCM

An important concept of the metrics calculation is to identify and measure coverage contribution of unique combinations; so that duplicated combinations' coverage and over-inflated coverage result do not eventuate. To do this, previously exercised and measured combinations are recorded for comparisons against future combinations that are realized. However, for practicality and effectiveness of our coverage method, only the most recent set of combinations exercised are retained, not all prior combinations are available for subsequent coverage measuring. The WCM and BCM metric definition differ primarily in how they are calculated with respect to estimating coverage of unique combinations.

WCM measures the unique attribute values and combinations accumulated during coverage measure; repeated attribute combinations are excluded. BCM is a count of the exercised combinations based on combinations from within the subset of most recent prior combinations only, not the entire series of previously exercised combinations. Therefore, the BCM may include attribute combinations from much earlier previous testing. The WCM is a conservative count of the worst case functional coverage percentage attained by the test suite throughout the entire measurement across all stages of recently captured combinations. The BCM provides the optimistic assessment. It is the best case count of all functional behaviours tested from each stage of testing (as defined by every accumulated set of most recently exercised prior combinations). The BCM may at times ignore erroneously repeated measurements of identical functions tested throughout the evaluation process.

The level of accuracy in the WCM and BCM is directly influenced by the chosen capacity to store recently exercised combinations. We aim to determine the best storage size to reduce susceptibility to errors (based upon experiments in Section 7.12.2 Chapter 7). The prior combinations exercised that are no longer retained are continually captured in an abstract manner using partial order domains. These

abstracted coverage information is used extensively for coverage measure and metric evaluation. The storage of prior combinations was described as windowing (Section 7.10 Chapter 7), and abstractive capture of their combinations information is denoted update and reduction respectively (Section 7.9.3 Chapter 7).

BCM specific observations

Recall that the cumu_comb captures only the attribute domain values independently, and abstracts away their combinatorial relationship with each other. By relying on the cumu_comb, the exercised combination could have been exercised much earlier, skewing the BCM slightly. However, we reason that such distortion of the BCM is uncommon because similar combinations are more likely to occur in groups, based on the same flow of SoC functions conducted by the snippet at any one time.

For example, in a DMA transfer snippet operation, the combinations exercised will be very similar. The attributes for DMA device settings are constant, whilst only the DMA addresses and transfer count changes with each data transaction. Therefore, these combinations, which are closely exercised with respect to one another, will be stored together in the database to ensure accurate BCM. Combinations that are not recently exercised within each other to be stored in the database together would be too dissimilar, and originate from completely different snippet induced SoC operations. Hence, it is unlikely that an exercised combination that was not already included in the current most recently exercised combinations database would have been exercised earlier.

The reasoning behind similar groups of combinations occurring relative to one another is based on the similar principle of temporal locality in microarchitectural caches. The motivation for caches is based upon the understanding whereby a piece of data retrieved for use from memory will very likely be needed again in the near future. Similarly, if an exercised attribute combination is to be realised more than once, it will most likely be exercised in relatively close succession from one another. Hence, such repeated combinations will be detected within the same stored combinations database, enhancing the uniqueness of combinations calculated by the BCM. The BCM is a maximal count because it is more likely to include an exercised combination as long as it is not already in the most recent set of exercised combinations.

BCM and WCM calculation procedure

Definition H.6 defines the conditions under which the BCM and WCM metrics are updated.

Definition H.6 : BCM and WCM evaluation conditions

Let R be the sub set of most recently exercised attribute combinations that are possible from the entire attribute combinatorial set, $R \subseteq CMB$ where CMB was defined in Definition 7.2 Chapter 7.

For a snippet node with m attributes in its combinatorial set $\langle a_1, a_2, \dots, a_m \rangle$, let the exercised combination of attribute values be a tuple $ex = \langle e_1, e_2, \dots, e_m \rangle$, and let cumu_comb = $\langle c_1, c_2, \dots, c_m \rangle$ where e_i and c_i are the corresponding attribute domain of attribute a_i $\forall\ i \in \{1, \dots, m\}$.

The BCM and WCM are updated by incrementation of one based upon the exercised combination ex if the following conditions occur.

(1) BCM = BCM + 1 if $\forall\ r \in R, r \neq ex$

That is, there is no previously exercised combination r that is the same as the exercised combination ex.

(2) BCM = BCM + 1and WCM = WCM + 1

if $\exists\ a_i : (e_i \leq c_i$ is false$) \wedge (e_i$ was not previously realised$)$,

where $i = 1, .., m$ and $\leq$ is the partial order relation operator defined in Definition 7.3 Chapter 7.

That is, there exists at least one attribute in which the domain value of the exercised combination is not covered by the existing domain of the corresponding attribute in the cumu_comb, such that the attribute domain value exercised was never previously realised. The domain of the exercised attribute e_i was never realised previously, hence was not updated in the cumu_comb and is not within the range of values captured by any of the existing domains in the cumu_comb domain c_i.

□

The BCM and WCM provide preliminary assessment of the test and coverage measure whilst minimising the processing and resource usage to facilitate efficient coverage measure. The final exercised number of combinations (as used in the metric calculation formula in Section 7.3 Chapter 7) is used as the eventual quantitative coverage metric, and is based on the BCM. The BCM provides an initial estimation of the quality of a test suite. Once this primary coverage metric has been maximised,

more effort can be directed to create tests for the secondary WCM metric; and ensure all combinations have been uniquely exercised. Focusing test effort toward the BCM enables faster and more efficient test suite application and coverage measurement. Once an adequate test suite has been created according to the BCM, less effort will be required to satisfy the WCM. When both BCM and WCM are sufficiently satisfied, the final coverage metric is estimated between the WCM and BCM – which is usually the median between these two figures. As described later, our experimental results will be based on the primary BCM measurements.

H.11 Attribute combinations coverage metric evaluation pseudo code implementation

The pseudo code implementation of attribute combinatorial coverage metric evaluation is given in Figure H.12. The coverage metric evaluation was described in Section 7.9.2, Chapter 7.

The coverage metric evaluation process is efficient by making use of partially ordered domains and domain operators to check for newly realised attribute values against the cumu_comb. Whenever a new previously exercised attribute domain value is detected, this implies immediately the exercised combination is unique. This check is performed for every attribute in the snippet node's attribute combinatorial set (lines 13 to 28 in Figure H.12). If the uniqueness of the exercised combination cannot be determined in this way, the exercised combination is compared against previously exercised combinations in the current window phase (lines 30 to 41 in Figure H.12). Checking of domains using partial order operations to detect unique exercised combinations first facilitates efficient coverage metric evaluation because the number of attributes to check during coverage measure is constant, and the partial order comparison check is quick. Comparing against previous combinations depends on the accumulating number of previously exercised combinations stored, hence is not as efficient; and is only employed if uniqueness of the exercised combination cannot be detected using partial order attribute domain checking.

At the end of the coverage metric evaluation (lines 43 to 49 in Figure H.12), if the window of previously exercised combinations has reached the exercised combinations window size limit, the reduction process in Appendix H.13 is triggered. Information from these currently stored combinations are abstracted into the cumu_comb, and the previously exercised combinations storage is cleared to begin another window phase so newly exercised combinations can continue to be stored for this new window phase.

```
1   Evaluate_Coverage_Metric [n, ex] {
2
3     // n : is the snippet node currently traversed and matched.
4     // ex : is the set of exercised combination of attribute values from testing.
5
6     // Get the set R of most recent previously exercised combinations for the
7     // snippet node n of the current window phase
8     R = Get_Previous_Exercised_Combs(n)
9     cumu_comb = Get_Cumu_Comb(n)    // Get the cumu_comb of n
10
11    unique_combination = false
12    i = 0
13    foreach [e_i ∈ ex] ∧ [c_i ∈ cumu_comb] {
14        if [e_i ≤ c_i is false] ∧ [e_i was never previously realised] {
15            // At least one new attribute value is detected, ex is a uniquely
16            // exercised combination, increment coverage metric
17            unique_combination = true
18            BCM = BCM + 1
19            WCM = WCM + 1
20            R = {ex} ∪ R    // add the exercised combination to recently exercised set
21            Update_cumu_comb   // Update the cumu_comb with exercised combination
22                               // by using the Update operation in Appendix H.13.
23                               // The Update function in Figure H.14 is called for
24                               // every attribute for the update process.
25            exit_loop
26        }
27        i = i + 1  // increment until i is the last index in the tuple of attributes
28    }
29
30    if [unique_combination is false] {
31        // no new attribute value was detected to indicate the uniqueness of the
32        // exercised combination, check if the exercised combination exists in
33        // the most recent set of previously exercised combinations
34        if [ex ∉ R] {
35            // exercised combinations was not previously exercised in current window
36            // reduction phase, increment BCM
37            BCM = BCM + 1
38            R = {ex} ∪ R
39            Update_cumu_comb
40        }
41    }
42
43    // Check the size of the stored combinations, and perform reduction if required
44    If [ |R| > window_limit] {
45        Reduce_cumu_comb  // Reduce the cumu_comb with combinations from R
46                          // by using Reduce operation in Appendix H.13. Reduce
47                          // function in Figure H.15 is called for every attribute.
48        R = ∅             // Clear R to store new combinations in next window.
49    }
50 }
```

Figure H.12 Attribute combinatorial coverage metric evaluation pseudo code

H.12 cumu_comb reduction per partial order level

Domain reduction of attributes in the cumu_comb is conducted by reducing domains one level at a time. That is, a higher level parent domain can only be replaced with a lower child domain in one reduction stage if it has a direct path to the lower level child domain. For example, according to the partial order structure in Figure 7.3 Chapter 7, a Boundary domain can only be reduced to the Legal domain at one time. A domain cannot be reduced to lower level domains by skipping intermediate domains. Intermediate domains must also be examined for reduction before reducing to lower domains.

The condition for reduction is best explained using Figure H.13, which shows an artificial but simplistic partial order structure. In Figure H.13, the lower level child domain *r* is the root domain of multiple parent domains one level above it. A domain d_1 can be reduced to its root domain *r* if and only if all attribute values of domain d_1 and all attribute values of other domains sharing the same root domains have been realised. Only when a reduction to the next domain has occurred, further analysis of the newly reduced domain can be conducted with other existing or subsequent domains values to perform further reduction. For each potential reduction opportunity, the root child domain from which a higher level parent domain can be reduced to can be identified elegantly by taking the least upper bounds of the current domains assigned to an attribute in the cumu_comb.

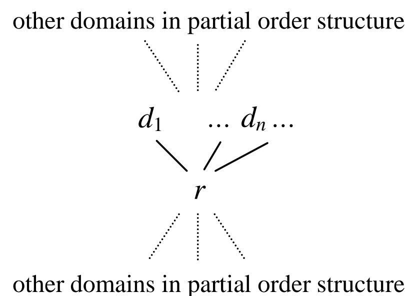


Figure H.13 Example partial order domain for reduction

H.13 Cumulative combination update and reduce pseudo code

The operation of the cumu_comb update and reduction process as defined in Section 7.9.3 of Chapter 7 can be further illustrated by their simplified pseudo code implementation for coverage measure. Figure H.14 shows the simplified pseudo code of the first stage cumu_comb updating procedure.

```
1   Update [v, e, d, D, E] {
2
3     // For the current attribute in the combinatorial set of a snippet node,
4     // v : is the exercised attribute value of the exercised combination under examination
5     // e : is the exercised attribute domain value of the exercised combination
6     // d : is the existing attribute domain value currently assigned in the cumu_comb
7     // D : is the set of attribute domain values currently represented by d in cumu_comb
8     // E ;  is the set of attribute values previously exercised and captured by domains in
9     //       cumu_comb
10
11    if [ (cumu_comb ≠ targ_comb) or (e ≠ d) ] {
12        // Update (and reduction) possible because :
13        // (i)   exercised combinations and attribute values in cumu_comb thus far
14        //       has not achieved coverage goal of targ_comb
15        // (ii)  exercised domain value has not been previously exercised and reduced to
16        //       be same as currently in cumu_comb
17
18        if [e ≤ d] {
19            // Exercised domain captured by existing lower ordered domain in cumu_comb
20            if [e ∈ D] {
21
22                if [v ∈ E] {
23                    // Exercised attribute value was previously exercised
24                    exit   // Examine the next attribute
25                } else {
26                    // Update the cumu_comb by storing the exercised attribute value
27                    E = E ∪ {v}
28                    if [all attribute values captured by (d ∪ e) has been exercised] {
29                        Reduce [v, e, d, D, E]
30                    }
31                }
32
33            } else {
34                // Exercised attribute domain was already updated and reduced to a lower
35                // domain previously, and all attribute values encapsulated by e were
36                // already exercised
37                exit   // Examine the next attribute
38            }
39
40        } else {
41
42            // e is a new domain previously unrealised
43            // Update the cumu_comb with the exercised attribute domain value
44            d = (d ∪ e)
45            if (all attribute values captured by (d ∪ e) has been exercised) {
46                Reduce [v, e, d, D, E]
47            }
48
49        }
50  }
```

Figure H.14 cumu_comb update pseudo code implementation

In the first stage, the update process identifies whether the coverage goals specified by the targ_comb has been achieved or whether the exercised domain is already the same as that currently in the cumu_comb (line 11). If either conditions are true, this implies the exercised combination cannot provide further coverage contribution, and no further update (or reduction) of the cumu_comb is required.

Otherwise, the update process performs an efficient and quick assessment of the domain of the exercised attribute value. On line 18, using the partial order operator $\leq$ in the operation defined in Definition 7.6 Chapter 7, the exercised attribute domain can be easily identified to check if it is captured by any existing domains in the cumu_comb already. If the exercised domain is not covered by the domain in cumu_comb, then update can proceed as per Definition 7.12 and the second stage reduction is triggered.

If the condition on line 18 is true, and the exercised domain is covered by the existing cumu_comb domain, then further examination of the attribute exercised domain value is required. Firstly, a check is conducted to determine if the exercised domain is not in the set of domains currently assigned in the cumu_comb (line 20). If the exercised domain does not exist, then given it is covered by the cumu_comb domain d from the condition in line 18, this implies the exercised domain was previously exercised and is already reduced to d. Hence, no update (and reduction) is needed (line 37).

If the condition on line 20 is true, and the exercised domain exists in the set of cumu_comb domain for that attribute, then the specific attribute domain exercised is examined (lines 22 to 31). The examination is simple and consists of comparing with previous exercised values to determine if the attribute was has been exercised before. If not previously exercised, then update of just the attribute value (and possible reduction of domains contributed by this exercised value) is carried out.

The approach of the update process in Figure H.14 above is to conduct minimal and efficient processing and comparisons using domains and partial order operations at the abstract combinatorial attribute information level; before moving on to examine and manipulate lower level and detailed data such as the actual exercised value. In this way, the update process contributes to faster and inexpensive overall coverage measuring.

The function Reduce[v, e, d, D, E] in Figure H.14 above for the second stage cumu_comb reduction is described in Figure H.15 below.

```
1   Reduce [v, e, d, D, E] {
2
3     // For the current attribute in the combinatorial set of a snippet node,
4     // v : is the exercised attribute value of the exercised combination under examination
5     // e : is the exercised attribute domain value of the exercised combination
6     // d : is the existing attribute domain value currently assigned in the cumu_comb
7     // D : is the set of attribute domain values currently represented by d in cumu_comb
8     // E ;  is the set of attribute values previously exercised and captured by domains in
9     //      cumu_comb
10
11    L1 : for e in set_of_exercised_domains {
12          if (d = X) {
13               // No reduction of lower root domain possible
14               exit_Reduce_function
15          }
16          // Identify possible root domain r for current exercised attribute domain e
17          for d ∈ D {
18               r = e LUB d
19          }
20
21          // Check that the set U of upper level domains immediately one level above the
22          // root domain have all been previously realised and captured by the new
23          // exercised domain value e or existing domains d in the cumu_comb
24          U = Get_One_Level_Higher_Parent_Domain [r]
25          L2: for u ∈ D {
26                if (u ≤ e) {
27                     next _loop_L2   // upper domain captured by e
28                }
29                for d ∈ D {
30                     if (u ≤ d) {
31                          next_loop_L2    // upper domain captured by d
32                     }
33                }
34                next_loop_L1 // Attempt reduction for any next exercised domain value e
35          }
36
37          // Perform reduction: replace domain in cumu_comb with root domain
38          //                              if safe to do so
39          If (Safe_To_Replace [r, e, d, D, E] = true) {
40               d = r
41               D = {d}
42               // Check if further reduction to lower root domain is possible
43               Reduce [v, e, d, D, E]
44          }
45
46     }
```

Figure H.15 cumu_comb reduction pseudo code implementation

In the second stage, which is the reduction process, a check at line 12 (in Figure H.15) is first conducted on the existing cumu_comb domain. If the cumu_comb domain d is already reduced to the lowest possible domain X, then no further reduction is possible and the reduction process terminates.

If there is potential for reduction, all the possible exercised domain *e* is examined with existing domain *d* in the cumu_comb to identify a root domain *r*. The root domain is to replace the existing cumu_comb domain and represents a greater number of attribute values being exercised and captured. The identification of the root domain on line 18 is conducted using the least upper bound operation as described in Definition 7.13.

Once the root domain is determined, all the parent domains one level immediately above the root domain are identified using the *Get_One_Level_Higher_Parent_Domain*[*r*] function on line 24. The *Get_One_Level_Higher_Parent_Domain*[*r*] function simply maps a domain to the set of domains one level above it. Given this set of parent domain one level above, the exercised domain *e* and existing cumu_comb domain *d* are examined to ensure that they capture all appropriate attribute values specified by the set of parent domains. For every upper level parent domain, if the parent domain is captured by either the exercised domain or cumu_comb domain, then the function skips to repeat the check for the next parent domain at lines 27 and 31. If any of the parent domain is not captured by either the exercised or cumu_comb domain, then no reduction to this root domain is possible. The process will move on with examination for reduction of the next exercised domain at line 34.

If reduction is possible, then before proceeding to replace the current cumu_comb domain with the root domain *r*, a check is conducted to ensure that the replacement will not adversely affect any other possible reduction to other root domains. The check is to ensure there is no other lower level root domain r_{other} that captures existing cumu_comb domain *d* (i.e. $d \leq r_{other}$) such that (i) r_{other} can be reduced to the corresponding attribute targ_comb domain, and (ii) r_{other} is not a domain already reduced to *d* of the current cumu_comb. If these conditions exist, and reduction is conducted to replace the cumu_comb domain with the root domain *r*, reduction to r_{other} and satisfaction of targ_comb at a later stages is not possible. This scenario is represented diagrammatically in Figure H.16.

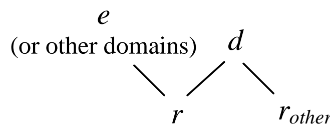


Figure H.16 Pre-reduction check partial order structure condition

This check that ensures reduction is safe to replace existing cumu_comb domain with *r* is captured in the *Safe_To_Replace* [*r*, *e*, *d*, *D*, *E*] function at line 39. In our partial order structure, comparing the structure in Figure H.16 to the partial order structure in Figure 7.3 of Chapter 7, this condition is unlikely to occur unless reduction from T domain is performed. For this case, given T domain is the single upper most domain, this check and reduction restriction is specially exempted to speed the reduction process. The inclusion of the *Safe_To_Replace* [*r*, *e*, *d*, *D*, *E*] function check in Figure H.15

is included for cases when the reduction pseudo code is applied to other partial orders and coverage measuring.

If reduction is allowed to proceed, then lines 40 to 43 replaces the existing cumu_comb domain with the root domain, and the reduction process is called recursively to potentially reduce this newly reduced cumu_comb domain further to even lower ordered domains. Despite updating and reducing the cumu_comb with exercised domains one step and one level at a time, the recursive triggering of the reduction process ensures the cumu_comb domains are always reduced to the lowest possible level in the partial order.

H.14 Attribute combinations coverage detailed example

Figure H.17 and Figure H.18 show a detailed example of the coverage measuring operations that are conducted at a snippet node. Specifically, the example shows the processing of exercised combinations from either the same or different tests whenever graph traversal encounters and matches the snippet node. The example demonstrates a possible scenario whereby a trace of snippet node matched exercised combinations is examined against the snippet node's targ_comb, and the coverage metric evaluation, cumu_comb update and reduction operations are shown. The example is given for a typical I/O transfer based device snippet with the following attribute combinatorial set,

$$\langle \text{start, int_enable, termination, unit_size, read_address, write_address, transfer_size} \rangle$$

and corresponding attributes and domain attribute values in Table H.4.

Figure H.17 shows the coverage process for individual exercised combinations in sequence one after another, but for single exercised combination windows, so that reduction is conducted every time for example purposes only. The example is primarily to demonstrate domain replacement in the cumu_comb arising from reductions. Hence, the exercised combinations are contrived to show reductions with just about every exercised combination. Eventually, the cumu_comb is updated and reduced with exercised combinations that match the snippet node targ_comb; the coverage goals of the snippet node is satisfied. Figure H.17 focuses on the update and reduction progress of the cumu_comb.

Figure H.18 demonstrates the lifecycle of the coverage measuring at the snippet node for selected exercised combinations throughout multiple reduction stages, and of window size greater than one. Although in this figure, the window size is still artificially small for example purposes. Like Figure H.17, Figure H.18 details the operations conducted at each snippet node to evaluate the exercised

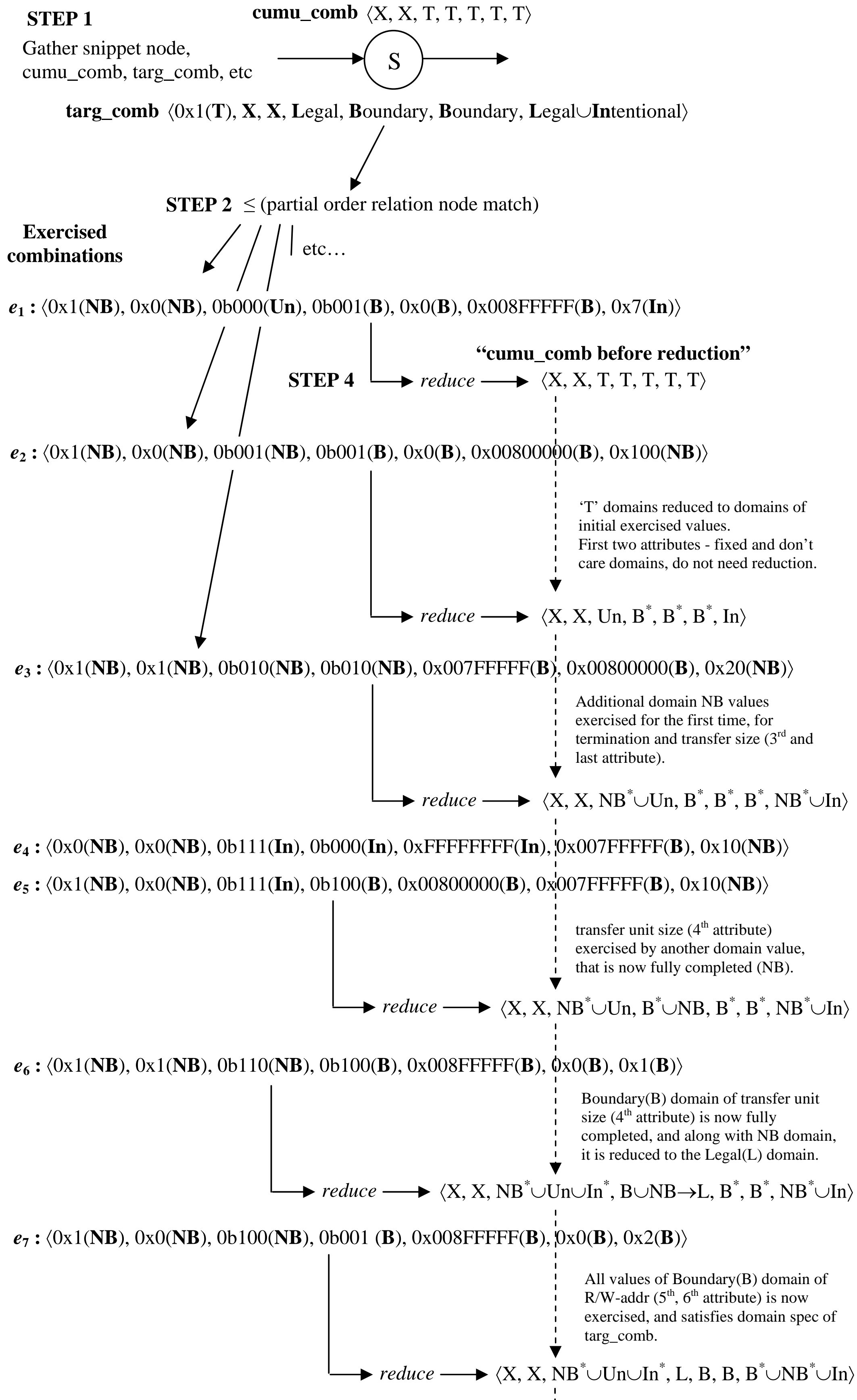
combination, and specifically demonstrates the update of various metric, and cumu_comb and targ_comb as well. The steps 1 to 5 in Figure H.17 and Figure H.18 correspond to the coverage measuring operational steps described in Figures 7.9 and 7.10 of Chapter 7.

Table H.4 Example attributes and domains for H.14 example

	Boundary	Non-Boundary	Intentional Error	Unintentional Error
start		0x0, 0x1		
int_enable		0x0, 0x1		
termination		0b001, 0b010, 0b100, 0b110	0b111, 0b101, 0b011	0b000
unit_size	0b001, 0b100	0b010	0b111, 0b011, 0b101, 0b110	0b000
read_address	0x0, 0x007FFFFF, 0x00800000, 0x008FFFFF	0x00900000, 0x00900840, 0x00001234	0xFFFFFFFF, 0x00900804	0x7, 0x101
write_address	0x0, 0x007FFFFF, 0x00800000, 0x008FFFFF	0x00900000, 0x00900840, 0x00001234	0xFFFFFFFF, 0x00900804	0x7, 0x101
transfer_size	0x1, 0x2, 0x4, 0x00800000	0x10, 0x20, 0x40, 0x100	0x7	0x0, 0xFFFFFFFF

In the example Figure H.18, the maxc and minc are used to monitor against progress of the coverage metric count of exercised combinations when compared against the possible combinations that can be realised at the snippet node. The maxc is described as the number of cross-product combinations encapsulated by an abstract combinatorial set such as the cumu_comb. It represents the maximum number of unique combinations that can be exercised to satisfy the cumu_comb at various stages during coverage measure. That is, the number of combinations whereby every attribute domain value of the cumu_comb is exercised with every other domain value.

On the other hand, the minc is the minimum number of combinations required to satisfy the abstract combinatorial set such as the cumu_comb. It represents the least number of combinations that will exercise each every attribute domain value of the cumu_comb at least once. That is, this minimum number of combinations is identified to be the count of combinations that exercises an unique attribute domain value each time. Essentially, minc is the largest number of domain values in the cumu_comb that can be exercised uniquely. The examples in this section provide greater details into various coverage measuring operations, and are an extension of the explanation and example diagram shown in Figure 7.9 in Chapter 7.


STEP 1
Gather snippet node, cumu_comb, targ_comb, etc
cumu_comb ⟨X, X, T, T, T, T, T⟩
S
targ_comb ⟨0x1(T), X, X, Legal, Boundary, Boundary, Legal∪Intentional⟩
STEP 2 ≤ (partial order relation node match)
Exercised combinations
etc…
e1 : ⟨0x1(NB), 0x0(NB), 0b000(Un), 0b001(B), 0x0(B), 0x008FFFFF(B), 0x7(In)⟩
"cumu_comb before reduction"
STEP 4
reduce
⟨X, X, T, T, T, T, T⟩
e2 : ⟨0x1(NB), 0x0(NB), 0b001(NB), 0b001(B), 0x0(B), 0x00800000(B), 0x100(NB)⟩
'T' domains reduced to domains of initial exercised values.
First two attributes - fixed and don't care domains, do not need reduction.
reduce
⟨X, X, Un, B*, B*, B*, In⟩
e3 : ⟨0x1(NB), 0x1(NB), 0b010(NB), 0b010(NB), 0x007FFFFF(B), 0x00800000(B), 0x20(NB)⟩
Additional domain NB values exercised for the first time, for termination and transfer size (3rd and last attribute).
reduce
⟨X, X, NB*∪Un, B*, B*, B*, NB*∪In⟩
e4 : ⟨0x0(NB), 0x0(NB), 0b111(In), 0b000(In), 0xFFFFFFFF(In), 0x007FFFFF(B), 0x10(NB)⟩
e5 : ⟨0x1(NB), 0x0(NB), 0b111(In), 0b100(B), 0x00800000(B), 0x007FFFFF(B), 0x10(NB)⟩
transfer unit size (4th attribute) exercised by another domain value, that is now fully completed (NB).
reduce
⟨X, X, NB*∪Un, B*∪NB, B*, B*, NB*∪In⟩
e6 : ⟨0x1(NB), 0x1(NB), 0b110(NB), 0b100(B), 0x008FFFFF(B), 0x0(B), 0x1(B)⟩
Boundary(B) domain of transfer unit size (4th attribute) is now fully completed, and along with NB domain, it is reduced to the Legal(L) domain.
reduce
⟨X, X, NB*∪Un∪In*, B∪NB→L, B*, B*, NB*∪In⟩
e7 : ⟨0x1(NB), 0x0(NB), 0b100(NB), 0b001 (B), 0x008FFFFF(B), 0x0(B), 0x2(B)⟩
All values of Boundary(B) domain of R/W-addr (5th, 6th attribute) is now exercised, and satisfies domain spec of targ_comb.
reduce
⟨X, X, NB*∪Un∪In*, L, B, B, B*∪NB*∪In⟩

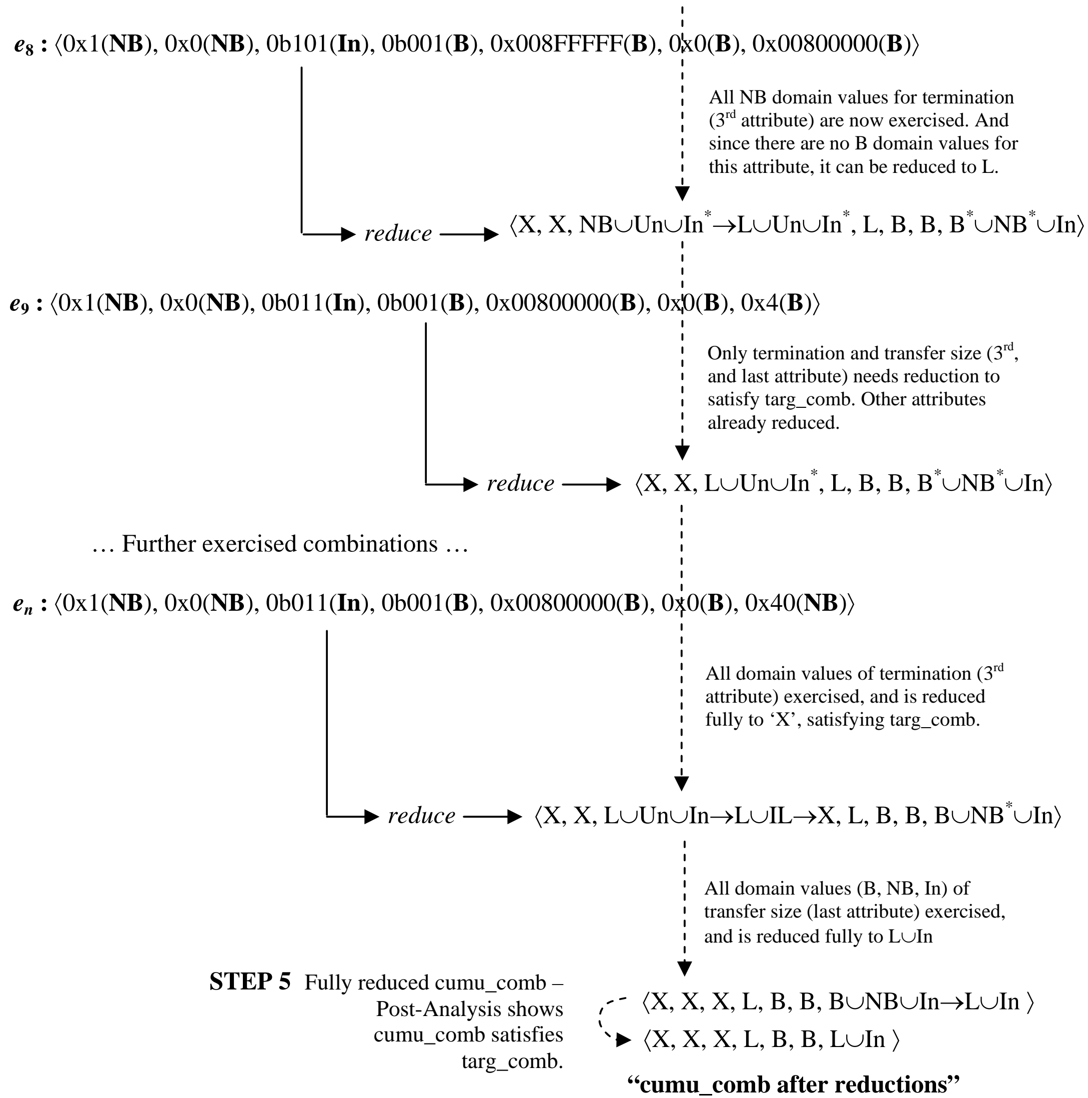
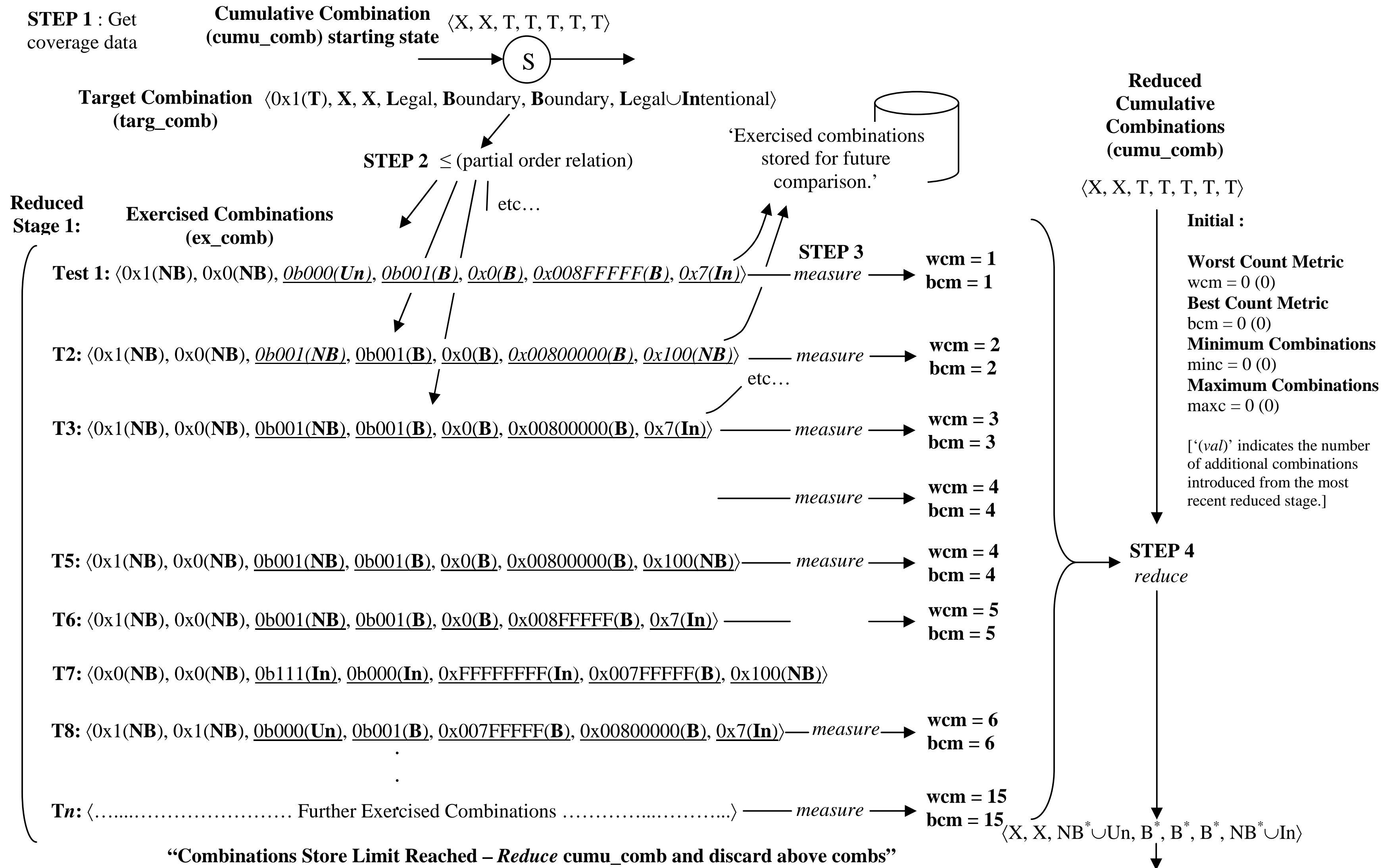
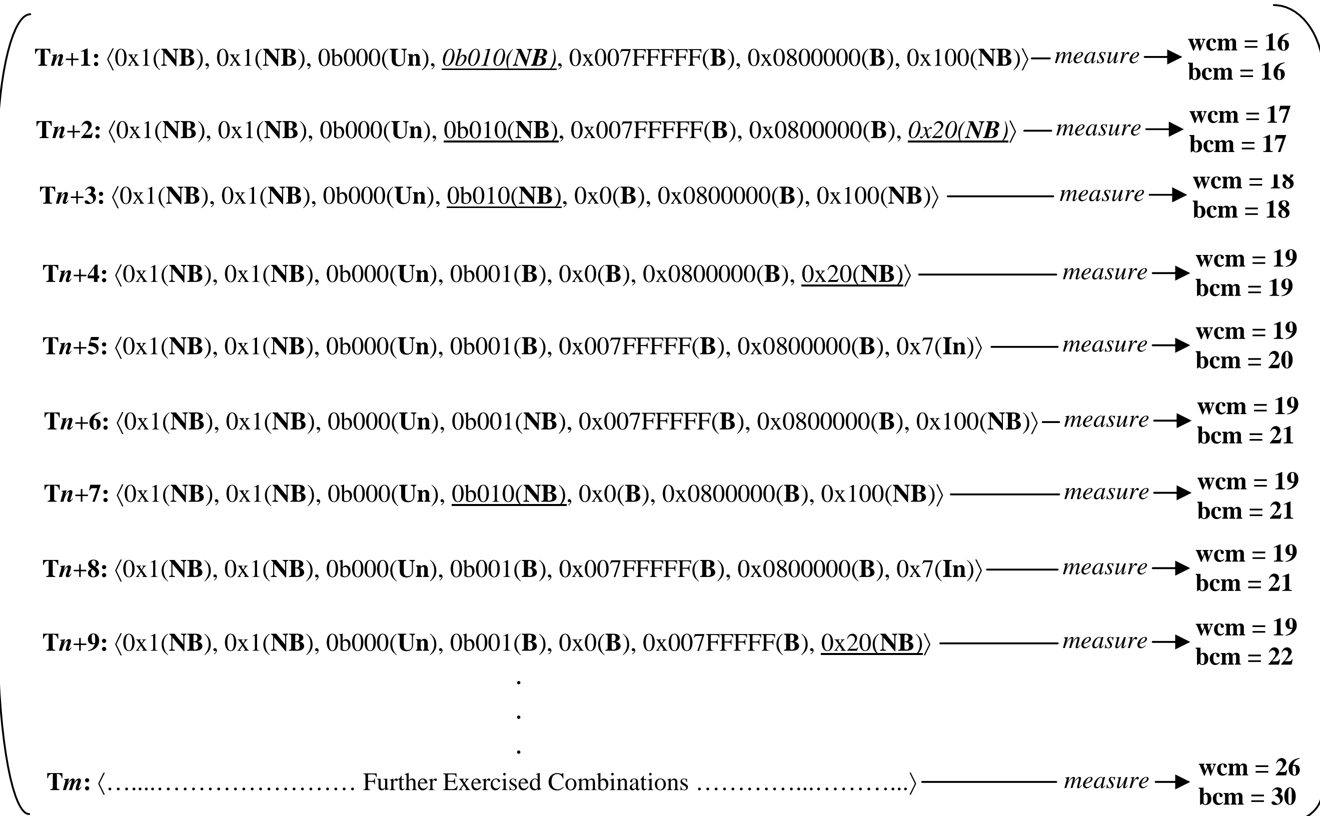
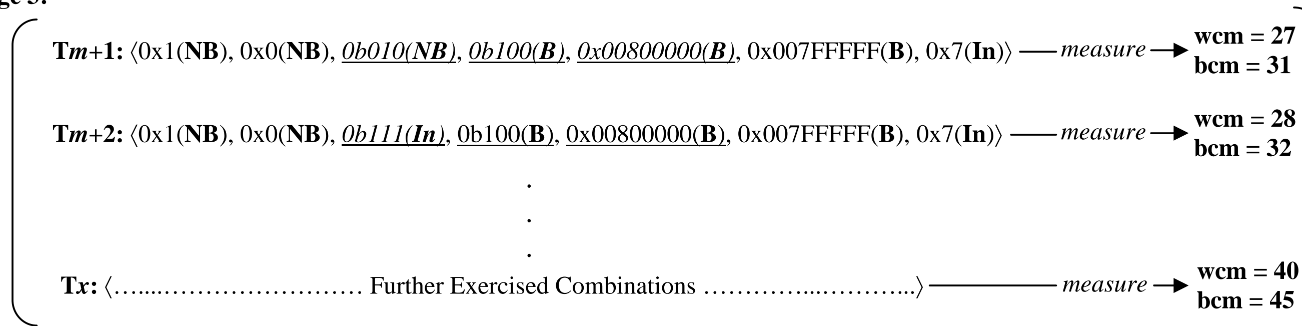


Figure H.17 Attribute combinations coverage measuring example at a snippet node for individually exercised combinations


STEP 1 : Get coverage data
Cumulative Combination (cumu_comb) starting state
⟨X, X, T, T, T, T, T⟩
S
Target Combination (targ_comb)
⟨0x1(T), X, X, Legal, Boundary, Boundary, Legal∪Intentional⟩
STEP 2 ≤ (partial order relation)
etc…
'Exercised combinations stored for future comparison.'
Reduced Cumulative Combinations (cumu_comb)
⟨X, X, T, T, T, T, T⟩
Initial :
Worst Count Metric
wcm = 0 (0)
Best Count Metric
bcm = 0 (0)
Minimum Combinations
minc = 0 (0)
Maximum Combinations
maxc = 0 (0)
['(val)' indicates the number of additional combinations introduced from the most recent reduced stage.]
Reduced Stage 1:
Exercised Combinations (ex_comb)
STEP 3
Test 1: ⟨0x1(NB), 0x0(NB), 0b000(Un), 0b001(B), 0x0(B), 0x008FFFFF(B), 0x7(In)⟩ measure
wcm = 1
bcm = 1
T2: ⟨0x1(NB), 0x0(NB), 0b001(NB), 0b001(B), 0x0(B), 0x00800000(B), 0x100(NB)⟩ measure
wcm = 2
bcm = 2
etc…
T3: ⟨0x1(NB), 0x0(NB), 0b001(NB), 0b001(B), 0x0(B), 0x00800000(B), 0x7(In)⟩ measure
wcm = 3
bcm = 3
measure
wcm = 4
bcm = 4
T5: ⟨0x1(NB), 0x0(NB), 0b001(NB), 0b001(B), 0x0(B), 0x00800000(B), 0x100(NB)⟩ measure
wcm = 4
bcm = 4
STEP 4
reduce
T6: ⟨0x1(NB), 0x0(NB), 0b001(NB), 0b001(B), 0x0(B), 0x008FFFFF(B), 0x7(In)⟩
wcm = 5
bcm = 5
T7: ⟨0x0(NB), 0x0(NB), 0b111(In), 0b000(In), 0xFFFFFFFF(In), 0x007FFFFF(B), 0x100(NB)⟩
T8: ⟨0x1(NB), 0x1(NB), 0b000(Un), 0b001(B), 0x007FFFFF(B), 0x00800000(B), 0x7(In)⟩ measure
wcm = 6
bcm = 6
Tn: ⟨…... Further Exercised Combinations ….....⟩ measure
wcm = 15
bcm = 15
⟨X, X, NB*∪Un, B*, B*, B*, NB*∪In⟩
"Combinations Store Limit Reached – Reduce cumu_comb and discard above combs"


Reduced
Stage 2:
Tn+1: ⟨0x1(NB), 0x1(NB), 0b000(Un), 0b010(NB), 0x007FFFFF(B), 0x0800000(B), 0x100(NB)⟩ measure
wcm = 16
bcm = 16
Tn+2: ⟨0x1(NB), 0x1(NB), 0b000(Un), 0b010(NB), 0x007FFFFF(B), 0x0800000(B), 0x20(NB)⟩ measure
wcm = 17
bcm = 17
Tn+3: ⟨0x1(NB), 0x1(NB), 0b000(Un), 0b010(NB), 0x0(B), 0x0800000(B), 0x100(NB)⟩ measure
wcm = 18
bcm = 18
Tn+4: ⟨0x1(NB), 0x1(NB), 0b000(Un), 0b001(B), 0x0(B), 0x0800000(B), 0x20(NB)⟩ measure
wcm = 19
bcm = 19
Tn+5: ⟨0x1(NB), 0x1(NB), 0b000(Un), 0b001(B), 0x007FFFFF(B), 0x0800000(B), 0x7(In)⟩ measure
wcm = 19
bcm = 20
Tn+6: ⟨0x1(NB), 0x1(NB), 0b000(Un), 0b001(NB), 0x007FFFFF(B), 0x0800000(B), 0x100(NB)⟩ measure
wcm = 19
bcm = 21
Tn+7: ⟨0x1(NB), 0x1(NB), 0b000(Un), 0b010(NB), 0x0(B), 0x0800000(B), 0x100(NB)⟩ measure
wcm = 19
bcm = 21
Tn+8: ⟨0x1(NB), 0x1(NB), 0b000(Un), 0b001(B), 0x007FFFFF(B), 0x0800000(B), 0x7(In)⟩ measure
wcm = 19
bcm = 21
Tn+9: ⟨0x1(NB), 0x1(NB), 0b000(Un), 0b001(B), 0x0(B), 0x007FFFFF(B), 0x20(NB)⟩ measure
wcm = 19
bcm = 22
Tm: ⟨…....…………………… Further Exercised Combinations …………...………...⟩ measure
wcm = 26
bcm = 30
reduce
After Reduced
Stage 1 :
wcm = 15 (15)
bcm = 15 (15)
minc = 2 (2)
maxc = 16 (16)
"Combinations Store Limit Reached – Reduce cumu_comb and discard above combs"
⟨X, X, NB*∪Un, B*∪NB, B*, B*, NB*∪In⟩

Reduced Stage 3:



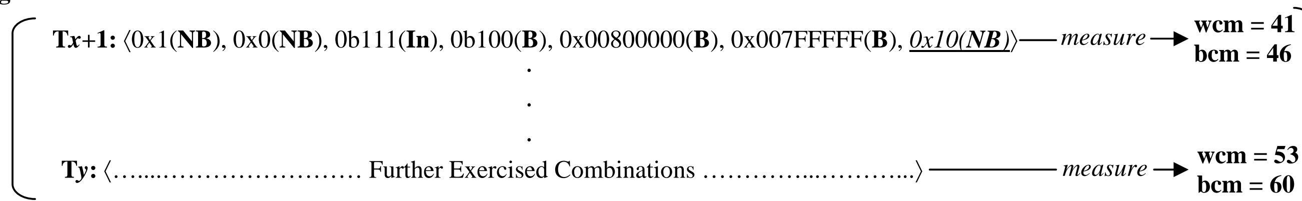
After Reduced Stage 2 :
wcm = 26 (11)
bcm = 30 (15)
minc = 3 (1)
maxc = 48 (32)

reduce

"Combinations Store Limit Reached – *Reduce* cumu_comb and discard above combs"

⟨X, X, $NB^* \cup Un \cup In^*$, $B \cup NB \rightarrow L$, B^*, B^*, $NB^* \cup In$⟩

Reduced Stage 4:

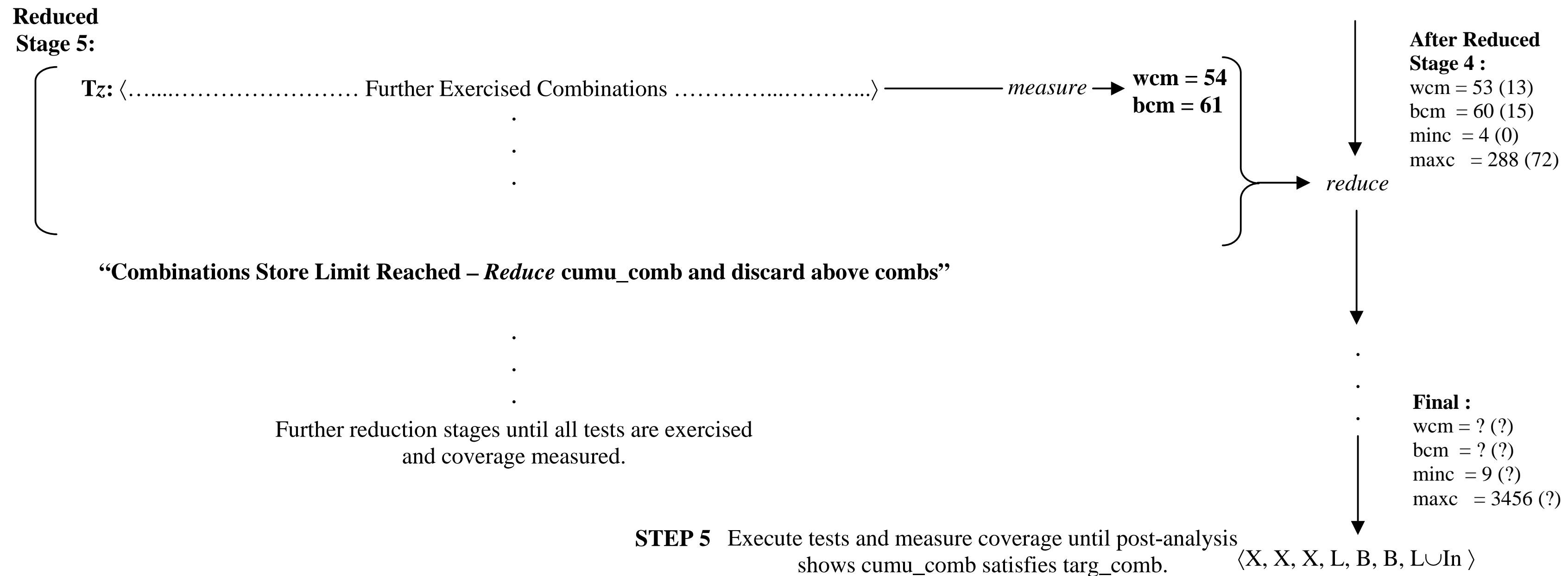


After Reduced Stage 3 :
wcm = 40 (14)
bcm = 45 (15)
minc = 4 (1)
maxc = 216 (168)

reduce

"Combinations Store Limit Reached – *Reduce* cumu_comb and discard above combs"

⟨X, X, $NB^* \cup Un \cup In^*$, L, B^*, B^*, $NB^* \cup In$⟩



Notes: The different kinds of combinations are handled as follows.

a) All combinations in reduced stage 1 (unless already exercised) will exercise new attribute values and combinations of these values (when compared to the initial cumu_comb). Hence, all combinations are stored and both wcm and bcm are incremented. E.g. combinations T1, T2, T3,…,T*n*.
b) A combination already exercised in the current reduced stage will be detected by comparing against the set of combinations stored previously; wcm and bcm are both unchanged. E.g. combination T5 has been exercised previously by combination T2; and combination T*n*+8 has been exercised previously by T*n*+5.
c) A combination that does not satisfy targ_comb of the snippet node is ignored. E.g. combination T7.
d) Combinations with attribute values which have not been exercised in previous stages (when compared to the most recently reduced cumu_comb), imply these combinations have never been exercised before. Hence, these combinations are stored and both wcm and bcm are incremented. E.g. combination T*n*+1, T*n*+2, T*n*+3, T*m*+1, T*m*+2, T*x*+1.
e) Combinations that have not been exercised in the current reduced stage, and do not exercise any new attributes (when compared to previous reduced stages) are stored but increments bcm only. It is uncertain whether such combinations have been exercised in previous reduced stages because the combinations from those stages have been discarded.
   E.g. combination T*n*+5 has been exercised previously by T8; whereas combination T*n*+6 has never been exercised – there is insufficient information to ascertain the uniqueness of combinations T*n*+5 and T*n*+6.

f) Combinations that exercises new attributes values (as explained in iv) but have been exercised previously in the current reduced stage are not stored; and wcm and bcm are both unchanged. E.g. combination T*n*+7 exercises new attributes values but was previously exercised by T*n*+3.
g) In this example, the reduce stages are shown to be very small before combinations are reduced and discarded (i.e. the combinations store limit is only up to 15). Therefore, from stage 3 onwards, the number of desirable cross-product combinations (due to newly exercised attribute values in the reduced stage) maxc is a lot greater than exercised combinations, wcm and bcm. In the actual system, the reduce stages will be a lot larger. Many more combinations can be exercised to improve the metric counts toward bcc in the reduced stages. However, how much improvement is uncertain.

Figure H.18 Attribute combinations coverage measuring example at a snippet node for selective combinations

H.15 Effects of coverage windowing size

Regardless of fixed or variable size, choosing an appropriate window size affects coverage measure effectiveness. The window size can be selected to provide better computational performance so coverage data is extracted and fed back to the test generator efficiently. However, the accuracy and usefulness of coverage data attained is also dependant on the window size, in particular, how many attribute combinations are available at the end of each windowing stage for analysis.

In particular, the effect of the window size on the WCM and BCM coverage metric is obvious. For a larger window size, this provides more opportunities to detect new domain values. Many more combinations that exercise new domain values can be processed within the same window stage before they are updated in the cumu_comb and allocated to the next window. This implies the WCM is higher for larger window sizes. By the same argument, a larger window implies many more exercised combinations can be stored. Therefore, the likelihood of detecting identical combinations is greater, reducing the count of BCM but providing a more accurate tally.

Furthermore, window size affects coverage measuring performance. Processing power is expended to execute various graph traversals and comparisons between attribute values or combinations. For example, to perform coverage operations such as check for realization of new domain values, detect identical combinations, or reduction of domains in the cumu_comb, as outlined in Section 7.9 Chapter 7. Therefore, a larger window size requires greater CPU time to perform these operations. A larger window size also implies more attribute values and combinations can be accumulated. Subsequently, more traversals through each graph node, and more comparisons between a greater number of stored values and combinations are needed during coverage measure. In effect, a larger window requires additional memory to provide the window capacity needed to hold the attribute values and combinations.

Despite gaining more favourable WCM and BCM coverage results using larger windows, a window too large may in fact result in highly inefficient coverage measurement and excessive bottlenecks in the test flow; or even worst, incomplete coverage measurements. These behaviours are confirmed by our experimental results, and are useful for configuring an efficient coverage evaluation in SALVEM. Particularly, careful consideration must be taken to find an optimized window size and understand its implications on coverage measure. These considerations were explored via experiments in Section 7.12.2 Chapter 7.

H.16 Summary of coverage metric evaluation

The coverage measurement process begins by attaining the trace of exercised attribute combinations from the next available test in order to commence control graph traversal. As per step 1, the target and cumulative combination (targ_comb and cumu_comb) of the next applicable snippet node (i.e. the first snippet node at the start of graph traversal) are identified to begin graph traversal with the exercised combinations.

Before a snippet node can be traversed, the cumu_comb is initialized to represent an empty coverage space. This is conducted only once for each snippet node at the start of coverage measuring, despite multiple traversals of the control graph from each test. Attributes in the cumu_comb are either assigned the top level empty terminal domain T, signifying the attribute has not exercised any values; or the lowest level X domain signifying the attribute is ignored for the functions tested for this targ_comb goal. The aim is to exercise attribute values and combinations during testing to update the cumu_comb with domains that match the targ_comb, thus satisfying the coverage goal.

Using the identified targ_comb of the current snippet node and the next exercised combination from the test, the traversal to this snippet node is facilitated by matching the exercised combination to the targ_comb. If the exercised combination is unable to satisfactory match the targ_comb, the targ_comb of the next possible snippet node is examined. Multiple snippet nodes can be available to attempt node match and act as graph traversal targets because the graph allows for nodes with multiple fan-out edges. However, only one snippet node is guaranteed to match a particular exercised combination.

The exercised combination to targ_comb node matching is conducted using the partial order $\leq$ operator. The node matching and graph traversal is the second step in the coverage measure process (in Figure H.19 and Figure 7.9 of Chapter 7), and was described in Section 7.8 Chapter 7. If no suitable snippet node can be matched, then graph traversal and coverage measuring of the current test terminates.

Otherwise, if graph traversal to the snippet node is successful, then every new combination of exercised attribute values is analysed against the cumu_comb to determine if it is unique and which metric, WCM or BCM it contributes to. Firstly, the combination is examined to detect if any attribute exercises a new domain value. If so, this implies the combination is unique, a new function was tested on the SoC, and the WCM and BCM is immediately incremented to reflect this.

On the other hand, if no new domain value was exercised, the combination is compared against previously stored combinations in the current window. If the exercised combination does not exist in

the current window, then only the BCM is incremented. The SoC function corresponding to this combination may have been tested previously, but not during the current window of functions processed already. When either WCM or BCM has been updated, the exercised combination is stored for the current window stage to detect any repeated combinations and SoC test functions.

The process of establishing whether any new domain values have been exercised, and whether to update coverage metric (BCM and WCM) constitute step 3, and was described in Section 7.9.2 Chapter 7.

Next, in step 4, the cumu_comb update and reduction process is performed. The update and reduction process is based on the operations in Section 7.9.3 and Figure 7.6 in Chapter 7. It is conducted for each attribute in the cumu_comb. If the exercised combination is found to have contributed to coverage by updating WCM or BCM, the cumu_comb is updated as described by the update process which is stage one in Section 7.9.3 Chapter 7. After a period of testing, if the number of combinations stored reaches window capacity, the cumu_comb undergoes reduction to further update the attributes and combinational information gathered during that window stage. The reduction process was described as stage two in Section 7.9.3 Chapter 7. Unlike Figure 7.6 Chapter 7, in Figure H.19 of the actual coverage method, the update process is conducted for every coverage contribution. Whereas reduction, is only carried out depending on the allocated exercised combinations storage capacity. The first update stage and second reduction stage corresponding to Figure 7.6 Chapter 7 are in fact segregated by the windowing process.

Briefly, the update and reduction process is as follows. For each attribute, the exercised combinational values during the window are examined. If all values from a particular domain were exercised, the attribute in the cumu_comb is updated with that domain to indicate the additional coverage combinations space that has been covered. The updated cumu_comb is then used in the next window stage to aid WCM and BCM measurements. If reduction is performed as well, the stored coverage combinations are discarded, freeing up capacity for the next set of exercised combinations in the next window stage.

During coverage measurement, the storing of exercised coverage combinations is required to check for duplicates and preserve accurate coverage results. But when the storage capacity at a node is filled according to the window process, a modified domain reduction technique abstracts the stored combinations to a single coverage set. Specifically, the cumu_comb is used to abstract and capture the attribute values and independent combinations information from each window stage.

Precise information from the attribute combinations are not retained, hence, some small inaccuracy is introduced into the BCM. An exercised combination may have been realized in previous window stages, however, we assume optimistically that this is uncommon and increment the BCM accordingly. Dependency information between attribute values is traded off to condense the number of combinations needed for storage. Our coverage method can still make use of independent attribute data in the cumu_comb to ensure coverage accountability and accurate results. Attribute domain reduction reduces the memory resource requirements.

Note that the cumu_comb is progressively updated for each window until the end of testing and coverage measure. The coverage space captured by testing and represented by the cumu_comb, is compared to that of the targ_comb to identify which combinations and which SoC functions have (or have not) been tested. The final WCM and BCM only provide the quantitative coverage effectiveness result.

Finally, when exercised combinations from a test are exhausted, the coverage measuring repeats steps 1 to 4 examining the trace of exercised combinations from the next test, and initiate a new graph traversal from the first snippet node again. Step 5 is the remaining operation in which the overall coverage metric are quantitatively collated to provide the final coverage result. Appendix H.17 outlines a post coverage measuring collation and analysis.

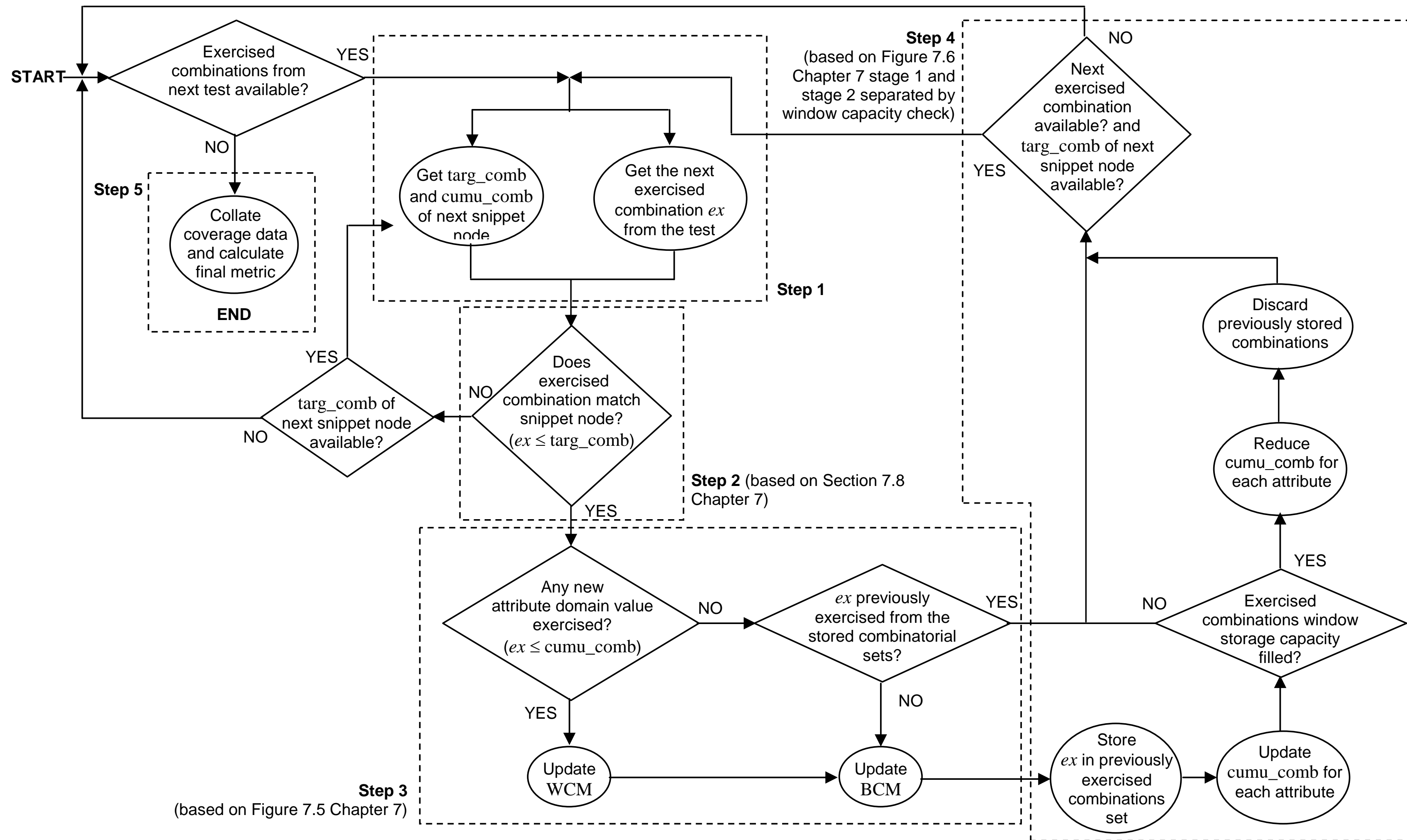


Figure H.19 Coverage measuring control flow process

H.17 Attribute combinations coverage post analysis

Whenever all exercised combinations from testing have been evaluated by attribute combinatorial coverage measuring, a post coverage analysis operation can be conducted to assess the effectiveness of the testing from the attribute combinatorial perspective. Besides the coverage metric that quantitatively reports the percentage of functional operations tested by snippets, it is beneficial to identify some form of functional coverage information that outlines what has or has not been exercised with respect to attribute combination measuring.

To this end, our coverage post analysis process identifies the *gap* in attribute combinatorial domain values between the domain values already exercised and the target domain goal values. That is, what are the missing attribute combinatorial domain values needed to be exercised in order for the cumu_comb to match the targ_comb, and for coverage goals to be met. For each attribute, the domains reported by the cumu_comb will be compared to the domains specified by the targ_comb. According to the partial order domain structure (Figure 7.3, Chapter 7), the domain values gap between the cumu_comb and targ_comb will be reported. These gaps of missing domain values are what are required to be exercised in order for the cumu_comb to match the targ_comb coverage goal. By reporting this information, testing and coverage measuring can be analysed to direct snippet testing to exercise for these missing domain values. By targeting these missing domain values, these maps to previously untested functional operations, and enhances SALVEM verification.

The coverage post analysis will first check whether the cumu_comb satisfies the targ_comb. That is, are the set of exercised combinatorial values fully covered by targ_comb, or are there attribute domain values that remain unexercised. If the cumu_comb is not yet reduced to match targ_comb from exercised attribute combinations, then (i) the attribute domain values exercised thus far needs to be established, and (ii) the remaining attribute domain values unexercised must be identified.

For (i), the attribute domains already exercised can be simply determined from the cumu_comb. To identify missing attribute domain values between the cumu_comb and targ_comb for (ii), according to the partial order structure, a traversal of the domains between those specified in the cumu_comb and targ_comb must be conducted. The domains traversed correspond to the missing unexercised domains that are required to satisfy targ_comb, and improve SALVEM testing.

The domain traversal between cumu_comb and targ_comb can be conducted either by a top-down traversal or bottom-up traversal. In a top-down traversal, the cumu_comb domains provide the starting point to commence traversal. Traversal from cumu_comb to targ_comb down the partial order records what domains are missing. In a bottom-up approach, the traversal starts at the targ_comb.

The coverage post analysis top-down strategy is shown in Figure H.20, and is invoked for each attribute of each snippet node. Firstly, the domains to start traversal from is determined (lines 7 to 19). On lines 23 to 26, if the cumu_comb and starting domain is T, then attribute domain values have not be exercised for that attribute. The traversal would have commenced from domains immediately beneath the T domains, but the missing unexercised domains are simply all attribute domain values covered by targ_comb.

During traversal at lines 29 to 50, unexercised missing domain values traversed are accumulated in *M* to be reported at the end of the analysis process. To identify further lower domains for traversal in each traversal loop (line 29), the current domain being traversed and under examination undergoes a greatest lower bound operation (GLB) with all other domains at the same partial order level as this domain (line 40 to 47). The GLB operation is inverse to the least upper bound operation defined in Definition 7.4 at Chapter 7. The resultant domain of the GLB operation is then checked to ascertain whether it can be reduced to and covered by the targ_comb. If so, then the resultant GLB domain is added to the set of domains to continue top-down traversal toward. This traversal loop repeats until the targ_comb is encountered, and finally the set of unexercised domains is reported.

In a bottom-up traversal approach, traversal begins at the targ_comb and continues up the partial order structure until the exercised attribute domain values are encountered. A bottom-up strategy may be considered more advantageous over the top-down approach. In the top-down approach, traversal begins from cumu_comb domains only. Therefore, if insufficient domains are exercised such that a lack starting point domains are available, there may be other paths down the partial order covered by the targ_comb that do not get traversed. For example, for a targ_comb X domain and cumu_comb L domain, identifical of missing IL, InE, and UnE domains may not get traversed and identified as missing because the starting domain may not be aware of them.

The bottom-up traversal approach is shown in Figure H.21. Traversal begins at targ_comb which acts as the traversal starting domains (line 9). In each traversal loop (lines 16 to 38), missing unexercised domain values are accumulated for reporting in *M* and traversal continues as long as the currently traversed domains still covers the domains in the cumu_comb (line 23). Traversal will stop when the current traversed domain no longer covers cumu_comb indicating the cumu_comb domain has been encountered. During the traversal loop, the next set of domains to continue upward traversal toward is simply attained from the parent domains of the current domains under traversal examination (lines 31 to 35). The traversal loop ensures that all missing unexercised domain values, including those that cannot be traversed from current domains in the cumu_comb are detected.

```
1 Top_Down_Post_Analysis [n, a] {
2
3     // For the current attribute in the combinatorial set of a snippet node,
4     // n : is snippet node under examination for coverage post analysis
5     // a : is attribute in the snippet node combinatorial set to perform top-down analysis for
6
7     targ_comb = Get_Targ_Comb(n, a)    // Get the targ_comb domains of n for a
8     cumu_comb = Get_Cumu_Comb(n, a)    // Get the cumu_comb of n for a
9     S = ∅     // S is the set of start domains from which top-down analysis will commence
10    // Gather the start domains by examining the domains in cumu_comb that have been
11    // exercised from testing, and include domains that are covered by the targ_comb but
12    // not in targ_comb
13    foreach [t ∈ targ_comb] {
14        foreach [c ∈ cumu_comb] {
15            if [c < t] ∧ [c ∉ S] {     // only include non-duplicate starting domains in S
16                S = {c} ∪ S
17            }
18        }
19    }
20    // M is the set of missing untested attribute domain values between
21    // cumu_comb and targ_comb
22    M = ∅
23    if [S = {T}] {
24        // cumu_comb starting domain is T, missing unexercised domain values are all
25        // domain attribute values above domains in the targ_comb to the top domain T
26        M = { all attribute domain values covered by targ_comb }
27    } else {
28        // Perform top-down traversal analysis
29        while [S ≠ ∅] {
30            // N is the next set of start domains to start top-down traversal analysis from S
31            N = ∅
32            foreach [s ∈ S] {
33                // Include the missing unexercised attribute domains at s if not in M already
34                if [s ∉ M] {
35                    M = { attribute domain values in s } ∪ M
36                }
37                // Determine any next domains to continue top-down traversal analysis
38                // from domains in s. Perform traversal if any domains at the same level
39                // shares common root domain as s that is covered by targ_comb
40                foreach [l ∈ set of domains at same level as s in partial order] {
41                    // Find common root domain using greatest upper bound operator GLB,
42                    // which is the inverse operation of LUB in Definition 7.4, Chapter 7
43                    common_root = GLB(l, s)
44                    if [common_root < domains in targ_comb] ∧ [l ∉ N] {
45                        N = {l} ∪ N     // Add to next domains to continue traversal
46                    }
47                }
48            }
49            S = N   // Set up next set of domains to continue top-down traversal analysis
50        }
51    }
52    return M
53 }
```

Figure H.20 Top down post analysis pseudo code

```
1 Bottom_Up _Post_Analysis [n, a] {
2
3     // For the current attribute in the combinatorial set of a snippet node,
4     // n : is snippet node under examination for coverage post analysis
5     // a : is attribute in the snippet node combinatorial set to perform bottom-up analysis for
6
7     targ_comb = Get_Targ_Comb(n, a)     // Get the targ_comb domains of n for a
8     cumu_comb = Get_Cumu_Comb(n, a)     // Get the cumu_comb of n for a
9     S = targ_comb    // S is the set of start domains from which bottom-up analysis will
10                     // commence
11
12    // M is the set of missing untested attribute domain values between
13    // cumu_comb and targ_comb
14    M = ∅
15
16    while [S ≠ ∅] {
17        // N is the next set of start domains to start bottom-up traversal analysis from S
18        N = ∅
19        foreach [s ∈ S] {
20            // Check if current domains in s covers domains in cumu_comb; if so, this
21            // implies there are missing attribute combinatorial values between cumu_comb
22            // and the current bottom-up traversal analysed at this domain level.
23            if [cumu_comb ≤ s] {
24                // Include the missing unexercised attribute domains at s if not in M already
25                if [s ∉ M] {
26                    M = { attribute domain values in s } ∪ M
27                }
28
29                // Gather the next upper parent level of domains from s to perform
30                // bottom-up traversal analysis to
31                foreach [p ∈ parent domains of s] {
32                    if [p ∉ N] {
33                        N = {p} ∪ N     // Add to next domains to continue traversal
34                    }
35                }
36            }
37        }
38        S = N   // Set up next set of domains to continue top-down traversal analysis
38    }
39
40    return M
41 }
```

Figure H.21 Bottom up post analysis pseudo code

Despite its conceptual benefits, the coverage post-analysis work has not been fully investigated and implemented in our research. It remains an area of work that is an avenue for future research. However, it is clear that identification of unexercised attribute domain values missing between the cumu_comb and targ_comb would no doubt aid in the evaluation of functional SoC operations tested, and direct further snippets based verification.

H.18 Coverage method enhancements and future work

The research described in this chapter opens up a number of avenues for future work. Currently, the coverage measuring is based on traversing control graphs whereby exercised combinations are mapped to individual snippet nodes as part of sequences of snippets possible from a test program. To facilitate more efficient coverage measuring and reduce the processing required at each node traversal, one could abstract the information captured at the nodal control graph level. Instead of a node for each snippet, a node could represent a set of common snippets (e.g., set of DMA or UART snippets) or short sequences of snippets. At the graph level, instead of individual graph traversal for each test, the control graph can be used to represent multiple tests as well. By abstracting and raising the level of information captured by control graphs and nodes, it would be interesting to investigate how this improves the performance of coverage measuring and handling of larger coverage attributes model; and how coverage metric results are affected.

The present coverage method employs a windowing strategy to break down the update processing operation of coverage data into a divide-n-conquer approach (Section 7.10 Chapter 7) The selection of this window size involves a well-thought-out process given a number of confliction factors must be considered (Section 7.12.2 Chapter 7 and Appendix H.15). Previously, the window size was determined statically before commencement of the measurement process, and this fixed window is used throughout coverage measuring.

Given that the exercised combinations and the coverage measuring in general is dynamic, and can produce different measuring conditions, it could be beneficial for the window size to change and adapt to these changing conditions rather than stay as a fixed size. An adapting window size is then able to continually facilitate optimal coverage data processing during the measuring operation. For example, at the end of every window stage after the most recent stored exercised combinations update/reduction is carried out, the current window size can be re-evaluated and modified to suit the current coverage measuring conditions. The divide-n-conquer approach is no longer segregated evenly, but the windowing and coverage data processing will be carried out whenever deem most appropriate to realise efficient update/reduction. For example, update/reduce only when necessary and with sufficient coverage data accumulated.

Another windowing option could be to employ a sliding windowing technique as well. Rather than using a fixed window whereby uniquely divided coverage combinations are processed each time, the windowing process could process both new combinations exercised within the window, and also overlapping combinations from the previous window of coverage data. That is, the window slides

through the sequence of coverage combinations data, and processes whatever combinations are covered by the window at any appropriately chosen stage. The sliding window could include combinations that were already processed previously but are still covered by the current window. With a variable window size as well, the coverage measuring process can be adapted to execute most efficiently and perform update/reduction operation only whenever needed, and according to the status of the measurement process.

Due to the limitations of available research resources and duration available, no further investigations were carried out to integrate the attribute combinations coverage method more closely with SALVEM, in particular the randomised and genetic evolutionary test generation processes in Chapters 3, 4, and 6. It would be fruitful to investigate the benefits of using the attribute combinations process to directly drive SALVEM test generation, given the types of functional information deduced from this coverage method. In fact, a manual coverage driven example was described in our experiments in Section 7.12.1 Chapter 7, whereby the functional attributes data was used to deduce what missing DMA transfers were not tested and to manipulate the DMA snippets to recreate tests for these missing SoC operating scenarios. With such examples and given the goals of the coverage method, attribute combinations coverage should be able to explicitly drive SALVEM test generation automatically.

One remaining possible benefit of the attribute combinations coverage that remains unexamined is how to use functional coverage data to further develop other snippets. Specifically, snippets to capture uncovered test space and enhance the snippets library. Employing such functional information from the attribute combinations coverage would facilitate a well-directed scheme for developing snippets of other designs. As yet, the strategy employing such coverage to create snippets is still to be developed.

During testing, the combinations of exercised attributes are recorded directly from SoC hardware so that the coverage metric is evaluated immediately. The coverage method uses these attribute combinatorial values and the graph base coverage measuring infrastructure. Specifically, STE trajectory checking to traverse a graph based coverage model is performed allowing coverage contributions to be recorded against snippet based graph nodes. In fact, the range of SoC behaviours tested can not only be deduced from the attribute combinations, but also the percentage of coverage graph model traversed. Furthermore, the untested functional behaviours can be identified by examining at each graph node, the SoC register values that were not exercised. These features promote SALVEM to be adapted as a coverage driven test flow. Currently, the coverage measure quantification is largely focused on attribute combinational evaluation only. However, there is scope for future extension of the coverage method to incorporate graph measuring capabilities. So far, the usage of STE based graphs is solely to facilitate attribute combinations coverage measuring as described in Section 7.9 Chapter 7.