Coverage Measurement for Software Application Level Verification using Symbolic Trajectory Evaluation Techniques

Adriel Cheng[†*] Atanas Parashkevov[*] Cheng-Chew Lim[†]

[†]*School of Electrical and Electronic Engineering*
The University of Adelaide
Adelaide, SA, Australia 5005
{acheng,cclim}@eleceng.adelaide.edu.au

[*]*Motorola EDA/IC*
2 Second Avenue, Mawson Lakes
Adelaide, SA, Australia 5095
{Adriel.Cheng,Atanas.Parashkevov}@motorola.com

Abstract

Design verification of a systems-on-a-chip is a bottleneck for hardware design projects. A new solution is a design verification methodology that applies coverage driven verification at the embedded software application level. This methodology currently lacks an appropriate coverage measurement technique. This paper proposes a new coverage model for the software application level. Using this coverage model, a novel technique to represent and measure coverage is described. This technique uses ideas such as control graph structures and checking algorithms to estimate the completeness of software application verification.

1. Introduction

Verifying the correctness of systems-on-a-chip (SoC) designs has become increasingly difficult and is a significant bottleneck in modern design flows. This difficulty is due to faster and highly complex SoCs. The increased complexity and functionality means design bugs are now even harder to find and detect. Therefore, given that exhaustive verification of a SoC design is not practical, design verification requires a structured and efficient technique.

Design verification involves functionally verifying the behaviour of a SoC to ensure it operates according to the design specifications. Currently, design verification is mainly undertaken at the assembly language or pin level, using assembler instructions test programs or input stimulus vectors as test cases [1,9].

A new design verification methodology is to apply coverage driven verification (CDV) at the embedded software application level. This methodology is at a higher level of abstraction from current methods, and is an ongoing development at Motorola, Australia.

The underlying aim of CDV is to gain maximum coverage (hence maximise the probability of detecting errors) using minimal test resources. CDV involves measuring the coverage of a test suite. The coverage measured is analysed and provides *useful* information that guides further test generation [2,4]. Coverage information helps generate new tests that target previously unverified or critical portions of the design. Coverage also indicates the quality and completeness of verification - this helps design managers decide when to stop verifying and approve tape-out.

One important aspect in this new software application level verification methodology (SALVEM) [5] is coverage. Coverage is essential for driving CDV, but no suitable or effective method has been proposed.

The main contributions of the paper are twofold: (1) a method to apply functional attribute based coverage that is suitable for SALVEM, and (2) a novel technique to represent and measure functional attribute coverage using ideas from symbolic trajectory evaluation (STE). An extended version of this paper is available in [3].

The remainder of the paper is organized as follows. Section 2 introduces SALVEM and STE. Section 3 proposes the attribute coverage model and defines the coverage metric. Section 4 examines the use of STE control graphs to represent the coverage model. Section 5 introduces the checking algorithm that checks and measures attribute coverage. Section 6 presents a simple example of the proposed coverage representation. Section 7 concludes the paper.

2. Background

SALVEM uses software application test cases to verify functionality exercised by typical applications of the SoC when it is used in customers' end products. The test cases mimic the different types of uses of the SoC under its intended operating environment. Once these real-life functionalities are verified, more focused and directed verification may be performed to target any remaining corner cases.

Proceedings of the Second IEEE International Workshop on Electronic Design, Test and Applications (DELTA'04)
0-7695-2081-2/04 $ 20.00 © 2004 IEEE

There are two essential elements in SALVEM: (1) the test generation process that creates the software application tests, and (2) coverage measurement. Currently, one of the test generation methods is to use a snippet based technique developed at Motorola.

Snippets are small, modular pieces of software *fragments* that perform a specific task using the peripherals and processor units on a SoC. Snippets contain software code statements such as peripheral driver application programming interface (API) function calls, operating system (OS) routines, and other simple code statements. Test generation algorithms automatically compose these snippets into application tests. Various combinations of snippets can be used to create different software application tests. The composition of snippets, API libraries and OS are compiled, linked and loaded as a test image in the SoC for simulation.

STE [7] is a formal verification technique using symbolic simulation to verify assertions that express properties of circuit behaviours.

Assertions are in the form of [*'antecedents' leads-to 'consequents'*]. The assertion implies for every sequence of circuit states that satisfy the antecedents, the consequent should be satisfied too. For more complex assertions, Jain [8] uses control graphs, and applies checking algorithms whilst traversing the graph to verify the assertion.

Ideas from STE are adapted for use in two areas, (1) coverage model representation, and (2) coverage combinations realization check and measurement.

3. Coverage model and metric

Current coverage methods [10,11] are not suitable for SALVEM because they were developed for low level conventional simulation technique. Instead, a coverage measure that is more related to the functionality and application domain of the SoC is desired.

Our coverage model is based on the functional coverage methods from IBM Haifa Labs [4,6]. Our coverage model relies on the identification of coverage attributes and coverage elements. Attributes are variables in the design that influence how the SoC operates. Coverage elements decompose the entire coverage space into manageable subsets. Each coverage element is associated with a set of attributes, and each attribute has a defined domain set of allowable values. A particular combination of attribute values exercises an instance of the coverage element (and SoC) in a particular manner. To attain high coverability, all legal combinations of attribute values must be exercised for each coverage element.

In SALVEM, coverage elements are peripherals (and processor units) of the SoC. For each peripheral, coverage attributes are extracted from configurable registers. For example, a communications SoC uses the Ethernet or Direct Memory Access (DMA) controller as its coverage elements. The attributes are either entire configurable registers or sub-divided fields of a register that control and react to peripheral operations (eg. DMA transmit destination address register, or data size sub-field of a control register). During testing, configurable register values indicate how a peripheral was initialised, and the results or status of various functions carried out. The range and combinations of register (attribute) values is measured to estimate what functionalities were invoked and how widespread the application domain was tested.

The coverage metric of a peripheral, is defined as the number of exercised attribute combinations over the total number of realisable combinations. Formally, the quantitative coverage metric is defined as

$$\frac{|CMB_{ex}|}{|CMB| - |CMB_{ill}|} \times 100\%$$

where $|CMB_{ex}|$, $|CMB_{ill}|$, and $|CMB|$ are the number of exercised combinations, illegal combinations, and total combinations, respectively.

Each attribute a_i is associated with a domain set V_i of possible concrete values the attribute may hold, with

$V_i \in \{0,\ldots,v_{max}\}$ where $v_{max} = (2^{(\text{number bits of } a_i)} - 1)$.

Therefore, the number of concrete values a_i can hold is $|V_i|$, and the total number of combinations is

$$|CMB| = \prod_{i=1}^{n} |V_i| \text{ where } n \text{ is the number of attributes.}$$

Initially, a coverage metric figure is calculated for each peripheral and its set of attributes. The aim is to maximise the coverage of each peripheral first, before considering the cross-product combinations of multiple peripherals, and eventually the entire SoC. Given the potential size of some coverage models, this enables the coverage to be strategically enhanced in a structured, bottom up, divide and conquer approach.

A coverage combination specifies the set of values currently assigned to coverage attributes at a particular time step. A coverage combination is defined as a tuple of n elements, $\langle a_1, \ldots, a_{n-1}, a_n \rangle$.

Attributes in a coverage combination are represented using the ternary $\{0,1,X\}$ set, where X is the unknown value. The domain of attribute values is

$D_i = V_i \cup \{X^b\}$ where X^b assign X to each bit of an attribute and b is the number of bits in the attribute (i.e. the attribute value is irrelevant and may hold any allowable concrete value regardless). The set of all coverage combinations is defined as,

$CMB = \{\langle a_1, \ldots, a_{n-1}, a_n \rangle : a_i \in D_i \text{ and } i = 1...n\}$.

4. Coverage model representation

To represent our coverage model, a new technique using modified STE control graphs is proposed. Control graphs are used to describe all possible attribute combinations realizable during testing. The allowable combinations defined by the coverage model are encapsulated in a compact and abstract form using these graphs. During verification, control graph coverage information assists in coverage measurement.

Control graphs are extracted for each snippet from a snippets database. These snippet control graphs compose the overall application test case control graph. Each test case control graph node encapsulates a lower level snippet control graph, resulting in a hierarchical graph structure (Figure 3).

Throughout test execution, certain configurable registers are read or written at different times, corresponding to operations performed by snippets. For example, during DMA data transfers, some status registers may be repeatedly read to check for successful transactions. The control graph only need to specify the attributes accessed when describing the realisable sequence of combinations.

Coverage attribute information is coupled to each snippet control graph node in terms of antecedent or consequent node formulas, called attribute specification formulas (ASF). ASFs describe the minimal sequence of coverage attribute values that must be realised, and identify the range of combinations coverable at each node.

The use of control graphs was motivated by similarities between STE circuit states and coverage attribute combinations. An attribute combination realized by a peripheral is like a circuit state exercised by the circuit. As tests are executed, the coverage attributes advances from one combination to the next.

In 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 is also set up (Figure 3). This enables modified STE algorithms to check for realisation of desired and expected coverage combinations. Antecedents specify write and read-write registers, whilst the consequent denote read and read-write registers. Configurable registers containing write-only, read-only or read-write sub-fields are sub-divided into separate attributes in the coverage combinations.

4.1 Formal definition

The control graph g_s of a snippet s is formally defined as a tuple of the form,

$g_s = [N_s, E_s, st_s, ed_s, \alpha_{ant}, \alpha_{con}]$, where

- N_s is the set of nodes in the control graph;
- E_s is the set of unidirectional and unlabelled edges; ($E_s = \{(n_{src}, n_{dest}) : n_{src} \in N_s$ and $n_{dest} \in N_s\}$);
- st_s is the start node and ed_s is the end node; ($st_s \in N_s$, $ed_s \in N_s$, st_s has no incoming edges and ed_s has no outgoing edges.);
- α_{ant}, α_{con} are formulation functions that associates attribute data with a node, defined $\alpha_{antlcon}: N_s \rightarrow ASF$.

The start and end nodes specify where graph traversal for coverage measurement begins and terminates. Except st_s and ed_s, all other nodes may have any number of incoming and outgoing edges, allowing divergent and cyclic graph behaviours to describe various snippet operations. The formulation functions analyses the snippets to label each node with its required ASFs.

A snippet control graph node (Figure 1) is defined as a tuple

$N_s = [id, ts, IN, OUT, ant, con, covset]$, where

- *id* is an unique identifier for the snippet node;
- *ts* is the time step associated with the node; (It specifies when *ant*, *con*, and *covset* applies.);
- *IN*, *OUT* is the set of incoming and outgoing edges of the node, respectively; ($IN \subseteq E_s$ and $OUT \subseteq E_s$);
- *ant*, *con* are the ASFs of the antecedent and consequent node formulas, respectively; (*ant* and *con* specify the desired write, expected read, and read-write registers values.);
- *covset* is a coverage set defined later in this section.

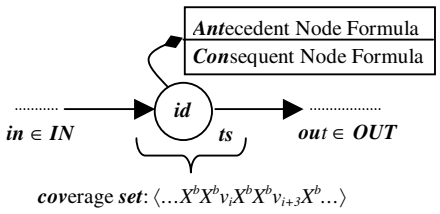

Figure 1. Snippet graph node

ASFs describe attribute information required at each snippet graph node, and are defined recursively as,

- simple predicate: $ASF = (a_i \text{ is } v_i)$
  (a_i is required to hold the value v_i.)
- conjunction: $ASF = f_1 \wedge f_2$
  (f_1 and f_2 are *ASFs* that hold at the current node.)

The ASF constructs ensure only the desired and expected attributes that need to be checked are specified (i.e. the minimal set of attributes that must be realised to ensure consistency with snippet behaviour).

ASFs can be solved by many coverage combinations. A function δ, which determine the set of

satisfying coverage combinations, is defined as follows.

δ:*ASF*$\rightarrow$*COVSET*

- simple predicate: $\delta(a_i \text{ is } v_i) = \langle \ldots X^b X^b, v_i, X^b X^b \ldots \rangle$
- conjunction: $\delta(f_1 \wedge f_2) = \delta(f_1) \text{ lub } \delta(f_2)$

(where lub is the least upper bound operator)

δ constrains only the attributes specified in the simple predicates, the remaining attributes are denoted X^b. A coverage combination containing X's is *abstract* and deemed a coverage set. A coverage set encapsulates all other possible *exact* coverage combinations if these X's were replaced with any combinations of their concrete values in V_i. AFSs are translated to equivalent coverage sets so checking algorithms can check for and measure coverage combinations. A coverage set is formally defined as,

$covset = \langle a_1, \ldots, a_{n-1}, a_n \rangle$

where $a_i \in D_i$ and $\exists a_i$ such that $a_i = X^b$

The coverage set at each graph node, n is obtained by evaluating, $covset = \delta(\alpha_{ant}(n)) \text{ lub } \delta(\alpha_{con}(n))$

The coverage set associated with each node restricts and narrows down the realizable attribute combinations. Each coverage set defines a searchable space of *exact* combinations that directs where the verification effort should focus on. Hence, the test generator can target combinations that have a greater likelihood of coverability (i.e. realisable and not illegal). Any wasted effort or resources from randomly targeting combinations that turn out to be illegal is prevented.

The aim is to exercise as many coverage set combinations by making full use of the existing snippets database before creating further snippets and test cases. Many different test variants generated, will adhere to the ASFs but exercise different combinations from the coverage set space each time (i.e. different combinations of values for X's). This increases coverage, exercises previously unverified scenarios, and enables more robust testing.

5. Coverage check and measure algorithm

The coverage algorithm performs two main functions, (1) it determines whether the control graph coverage sets are realisable, and (2) it measures the exercised combinations to calculate the quantitative coverage metric.

In the first stage, the algorithm uses modified STE checking techniques to check coverage. If a coverage set is deemed realisable according to the exercised combination, coverage measurement of the combination proceeds. If a coverage set is unrealisable, this implies uncovered attribute combinations. Efficient and accurate comparison is achieved by examining only the required concrete attributes (non-X's) in the coverage set.

The checking algorithm traverses the graph in a depth first manner. The attributes specified by antecedent and consequent node formulas are checked using compatibility (~) and reduced partial ordered ($\subseteq$) operators, respectively. If neither the antecedent nor consequent is satisfied, the combinations are added to the unrealisable set for further investigation.

The coverage measurement stage increments the coverage metric if the exercised coverage combination was not previously covered. Once a combination is exercised, it is stored in a database for comparison with future combinations to prevent duplicate coverage measurement. Efficient storage (using coverage sets) and comparison ($\subseteq$) is achieved as a single exercised combination is checked against many combinations in the database simultaneously.

The coverage checking approach uncovers unrealised combinations within the coverage space covered by the test cases. These combinations are analysed so existing or new test cases can be modified to realise these combinations.

6. Simple example

Consider a simple SoC containing a main processor and two peripherals – a DMA controller and a memory module. An application of this SoC is data transfers between any of the three SoC modules. For example, data transfers between memory and an external IO device via the DMA, or data block copies from the processor on-chip cache to memory (and vice versa). These common data transfer applications exercise a range of functionalities such as DMA device set up, DMA transfers in burst mode, or memory reads/writes.

The coverage model (Table 1) covers these application functionalities by selectively choosing suitable configurable registers and registers sub-fields to be coverage attributes. These attributes are separated into read, write, or read-write categories so they may be specified in antecedent or consequent nodes.

DMA set up snippet:

1. Enable DMA. *Set(ct_dma_enable is 1)*
2. Set up the source and target address, and request a channel. *Set(sr is 300)*, *Set(tg is B00)*, *Set(ct_ch_req is 1)*
3. Set the number of bytes to transfer, and ensure the channel is activated. *Set(dc_count is F)*, *Check(st_ch_act is 1)*

} set up DMA

4. Make sure the DMA is ready to perform transfers. *Check(st_dma_ready is 1)*

} check DMA is ready

Figure 2. Example snippet

Table 1. Example coverage model

Element	Configurable Registers		Attributes (Regs or Reg Sub-fields)	Domain Values	Coverage Combination
DMA	Source (sr)	**RW**	$sr_{0\text{-}31}$	0-(2^{32}-1) - any valid source address	⟨sr, tg, ct_dma_enable, ct_ch_req, ct_bw, ct_burst, ct_channel, dc_packet_size, dc_count, st_transfer_status, st_error_type, st_ch_act, st_dma_ready⟩
	Target (tg)	**RW**	$tg_{0\text{-}31}$	0-(2^{32}-1) - any valid target address	
	Control (ct)	**R**	$ct\_dma\_enable_0$	0- disable, 1-enable	
			$ct\_ch\_req_1$	1- request channel	
			$ct\_bw_{2\text{-}5}$ (bandwidth)	0- 64byte bw, 1-128b bw, etc	
			ct_burst_6	0- normal, 1-burst mode	
			$ct\_channel_{7\text{-}10}$	0- ch. 0, 1- ch. 1, etc	
			etc		
	Data Control (dc)	**W**	$dc\_packet\_size_{0\text{-}3}$	0- 1byte, 1- 2byte, 2- 4byte, etc	
		RW	$dc\_count_{4\text{-}19}$	0-(2^{16}-1) - count of data transfer	
			etc		
	Status (st)	**R**	$st\_transfer\_status_{0\text{-}1}$	0- success, 1-error, etc	
			$st\_error\_type_{2\text{-}3}$	0- no error, 1- erroneous data, 2- incomplete transaction, etc	
			$st\_ch\_act_{4\text{-}5}$	0- channel deactivated, 1- activated	
		RW	$st\_dma\_ready_6$	1- ready for transmit, etc	
			etc		
Processor	Status (st)	**RW**	$st\_sup\_usr_0$	0- user mode, 1- supervisor	⟨st_sup_usr, st_ext_int, st_reset,…, ca_en, ca_write,…etc⟩
			$st\_ext\_int_1$	0- enable, 1- disable (ext interrupts)	
			st_reset_2	1- soft reset	
			etc		
	Cache (ca)	**R**	ca_en_0	0- cache disable, 1- enable	
			$ca\_write_{1\text{-}2}$	0- write-back, 1- write-thru, etc	
			etc		
Memory	Source (sr)	**RW**	$sr_{0\text{-}31}$	0-(2^{32}-1)- any valid source address	⟨st, tg, ct_size, ct_read_write,…etc⟩
	Target (tg)	**RW**	$tg_{0\text{-}31}$	0-(2^{32}-1)- any valid target address	
	Control (ct)	**W**	$ct\_size_{0\text{-}3}$	0- 1byte, 1- 2b, 3- 4b, 4- 8b, etc	
			$ct\_read\_write_4$	0- read, 1- write	
			etc		

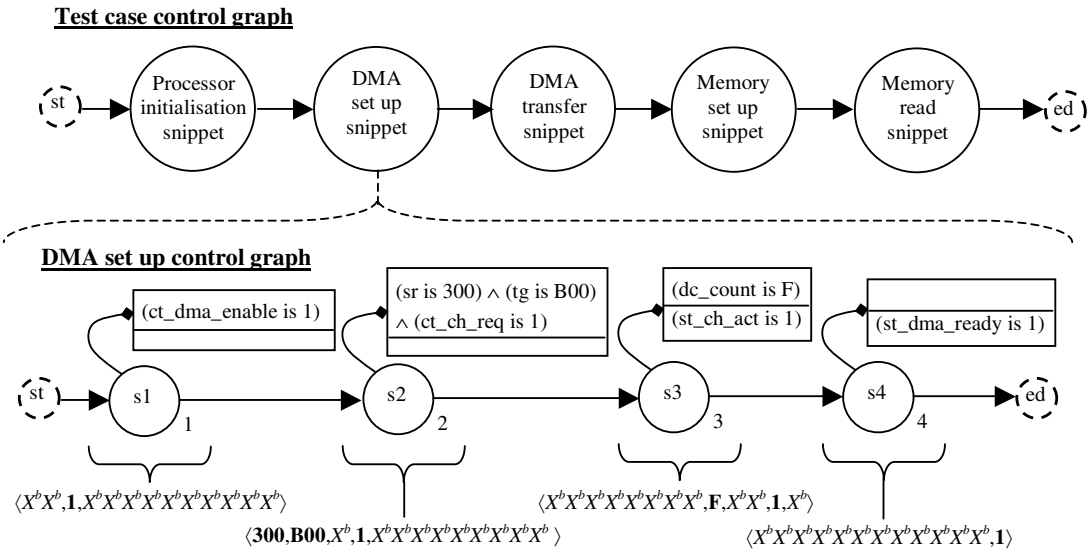

Figure 3. Test case and snippet control graphs

Snippets perform small, simple tasks. Some examples are,

- Processor initialisation snippet – carry out the SoC software initialisation sequence after reset;
- DMA set up snippet – enable and set up the DMA to perform data transactions;
- DMA transfer snippet – execute different modes of transactions between source and target devices.

Snippets are defined in terms of software code [5], and may be analysed to describe their sequence of configurable register accesses (Figure 2). These register accesses enable control graphs to be extracted.

The test case control graph (Figure 3) tests a DMA transfer and memory read after the SoC is initialised from reset. The hierarchical control graph is made up of a number of snippets. The DMA set up graph (extracted from Figure 2) shows the sequence of attribute accesses when the DMA is initialised from reset into a functional transfer ready state.

Exercised coverage combinations must realize both antecedent and consequent attribute values in the coverage sets. The DMA set up graph describes a *set up – check response* pattern. To initialise the DMA, the first three nodes set up antecedent attributes that are written. The last two nodes specify the expected consequent attribute values to indicate the DMA is initialised.

The snippet graph nodes specify only required attribute values that must be satisfied. The DMA set up graph specifies attributes essential for initialisation only (eg. enable DMA, initialise data source address, etc). The remaining attributes are unrestricted. Additional test variants may execute data transactions using different combinations of channels, bandwidths, burst mode, or data transaction sizes. These attributes are X^b in the coverage sets. Hence, the generated test variants cover more combinations and verify functionalities under different scenarios.

7. Conclusion

The paper proposed a new coverage method for the software application level, based on a functional attribute coverage model. The coverage model relies on the realisation of many coverage combinations. Coverage combinations are combinations of configurable register (attribute) values that indicate different tasks performed by the SoC. Using this coverage model, STE control graphs are adapted to represent the attribute combinations in an efficient manner. STE based checking algorithm is also used to check for, and measure a range of realizable combinations. The coverage representation and measurement approach can direct the test generator to make full use of existing snippets and focus on a smaller coverage space. Further research will be conducted to extend the current coverage method to facilitate a feedback coverage driven test generator.

References

[1] R. Ho, C. Yang, M. Horowitz, and D. Dill, "Architecture Validation for Processors," *International Symposium of Computer Architecture*, 1995.

[2] G. Nativ, S. Mittermaier, S. Ur, and A. Ziv, "Cost Evaluation of Coverage Directed Generation for the IBM Mainframe," *IBM Research Verification Technology Publications*, 2001.

[3] A. Cheng, A. Parashkevov, and C.C. Lim, "Coverage Measurement for Software Application Level Verification using Symbolic Trajectory Evaluation Techniques," *Extended Version, Internal Paper, http://www.eleceng.adelaide.edu.au/Personal/acheng/ste/ste.html*, November 2003.

[4] S. Fine and A. Ziv, "Coverage Directed Test Generation for Functional Verification using Bayesian Networks," In *Proceedings of the 40th Design Automation Conference*, June 2003.

[5] D. Sharman, D. Nutchey, A. Parashkevov, and M. McGeever, "VeriSoC Requirements Book," *Motorola EDA/IC Document*, 2003.

[6] O. Lachish, E. Marcus, S. Ur, and A. Ziv, "Hole Analysis for Functional Coverage Data," In *Proceedings of the 39th Design Automation Conference*, June 2002.

[7] C. Seger and R. Bryant, "Formal Verification by Symbolic Evaluation of Partially-Ordered Trajectories," *Formal Methods in System Design*, Vol. 6, No. 2, pages 147-190, March 1995.

[8] A. Jain, *Formal Hardware Verification by Symbolic Trajectory Evaluation*, Ph.D. thesis, School of Electrical and Computer Engineering, Carnegie Mellon University, July 1997.

[9] S. Ur and Y. Yadin, "Micro Architecture Coverage Directed Generation of Test Programs," In *Proceedings of the 36th Design Automation Conference*, June 1999.

[10] F. Fallah, S. Devadas, and K. Keutzer, "OCCOM – Efficient Computation of Observability-Based Code Coverage Metrics for Functional Verification," In *IEEE Transaction on Computer-Aided Design of Integrated Circuits and Systems*, Vol. 20, No. 8, August 2001.

[11] Y. Hoskote, D. Moundanos, and J. Abraham, "Automatic Extraction of the Control Flow Machine and Application to Evaluating Coverage of Verification Vectors," *International Conference on Computer Design: VLSI in Computers and Processors*, 1995.