Design Fault Directed Test Generation for Microprocessor Validation

Deepak A. Mathaikutty, Sandeep K. Shukla
FERMAT Lab, Virginia Tech
Blacksburg, VA 24061
{mathaikutty,shukla}@vt.edu

Sreekumar V. Kodakara, David Lilja
The University of Minnesota
Minneapolis, MN 55455
{sreek,lilja}@ece.umn.edu

Ajit Dingankar
Validation Tools
Intel Corporation
Folsom, CA 95630
ajit.dingankar@intel.com

Abstract

Functional validation of modern microprocessors is an important and complex problem. One of the problems in functional validation is the generation of test cases that has higher potential to find faults in the design. We propose a model based test generation framework that generates tests for design fault classes inspired from software validation. There are two main contributions in this paper. Firstly, we propose a microprocessor modeling and test generation framework that generates test suites to satisfy Modified Condition Decision Coverage (MCDC), a structural coverage metric that detects most of the classified design faults as well as the remaining faults not covered by MCDC. Secondly, we show that there exists good correlation between types of design faults proposed by software validation and the errors/bugs reported in case studies on microprocessor validation. We demonstrate the framework by modeling and generating tests for the microarchitecture of VESPA, a 32-bit microprocessor. In the results section, we show that the tests generated using our framework's coverage directed approach detects the fault classes with 100% coverage, when compared to model-random test generation.

1. Introduction

Functional validation has become a key ingredient in the development cycle of current and future microprocessors. Simulation is widely used to validate large systems like microprocessors. In simulation based validation, a test is executed in a golden reference model as well as in the design under test (DUT), which in this case is the RTL implementation of the microprocessor. The success of simulation based validation depends on the quality of tests. A potential fault in the RTL is exposed only if the test resulted in a state deviation of the RTL from that predicted by the golden model. Coverage metrics inspired from software validation (statement, branch and path coverage), state machine representation (state, transition and path coverage) and functionality of the microprocessor is used to measure the quality of tests. Effectiveness of coverage directed test generation depends on the *types of faults* that can occur and the strength of the *coverage metric* to detect these faults.

We analyze the bug descriptions and modeling errors reported during the study of microprocessor validation in [2, 13] and relate them to the fault types observed during software validation [7]. The outcome is a high correlation between the fault types and the modeling errors seen in the control logic of microprocessors. Hence, we propose a model based test generation framework that detects these fault classes.

The proposed framework allows modeling the microprocessor at the architectural and mircoarchitectural abstraction. The modeling language is developed as a *metamodel*, which provides the syntax and semantic necessary to describe the architectural and microarchitectural models. The framework translates these models into constraint satisfaction problems [4] that are resolved through a highly scalable constraint solver. The architectural model is converted into architectural constraints that are solved to generate random test suites that validates the *Microcode*. The microarchitecture model is converted into microarchitectural constraints that are solved to generate random test suites that validates the RTL implementation. However, to improve the quality of the test suite generated, we enable our framework with coverage-directed test generation capability. The coverage metric is used to generate additional constraints that are given to the solver along with the constraints obtained from the model. The validation flow is shown in Figure 1.

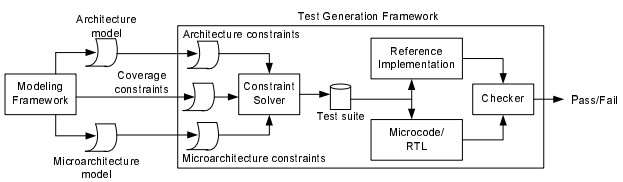


Figure 1. Validation Flow

We support traditional coverages such as statement and branch as well as modified condition decision coverage and design fault coverage. Modified Condition Decision Coverage (MCDC) [6] is a coverage metric that is widely used for the validation of critical software systems like avionics. The test suite generated as result of MCDC-directed test generation is found to detect most of the design fault classes proposed in [7]. Therefore, we propose MCDC as a metric for microprocessor validation. The design fault classes not covered as result of MCDC is detected by generating additional constraints that tune the solver to generate solutions that test them. We illustrate our test generation framework by creating test suites for VESPA [8], a 32-bit pipelined microprocessor. We provide results to highlight the importance of MCDC-directed test generation for design fault coverage.

978-3-9810801-2-4/DATE07 © 2007 EDAA

2. Related Work

Test generation for simulation based validation can be broadly classified into random and coverage directed test generation. Random test generators [3] are extensively used to validate microprocessors. Several techniques [5, 9, 12] were proposed in the literature for coverage directed test generation, which differ in the coverage metric being targeted and the techniques used to generate test cases.

Ur and Yadin [12] presented a method for generation of assembler test programs that probe the microarchitecture of a PowerPC processor. Benjamin *et al.* [5] describe a verification methodology that uses coverage of formal models to specify tests for a superscaler processor. In [9], Mishra *et al.* propose graph-based test generation for validation of pipelined processors that achieves functional coverage. Andrew Piziali [11] has presented a comprehensive study on functional verification coverage measurement and analysis. To the best of our knowledge, there are no previous approaches that propose a model based test generation for *design fault* coverage on microcode and RTL by applying MCDC and CSP formulation.

3. Background

We briefly outline the different fault classes and introduce the coverage metrics satisfied by our test generation.

Table 1. Fault Classes

Fault Class	*Description*
Expression Negation Fault (ENF)	A subexpression of S is negated
Term Omission Fault (TOF)	A term t_i is omitted
Term Negation Fault (TNF)	A term t_i is erroneously negated
Literal Omission Fault (LOF)	A literal x_j^i is omitted from a term t_i
Literal Insertion Fault (LIF)	An extra literal $x_{k_i+1}^i$ is inserted into a term t_i
Literal Negation Fault (LNF)	A literal x_j^i is erroneously negated in a term t_i
Literal Reference Fault (LRF)	A literal y is referenced ins tead of x_j^i in a term t_i
Disjunctive Operator Reference Fault (ORF[+])	A boolean operator (+) in S is replaced by (.)
Conjunctive Operator Reference Fault (ORF[.])	A boolean operator (.) in S is replaced by (+)

Fault Classes: Lau and Yu [7] proposed a comprehensive set of nine fault classes related to boolean expressions in disjunctive normal form (DNF). Let us consider a boolean expression in DNF form with m terms $S = t_1 + .. + t_m$. Let t_i denote the i^{th} term in S such that t_i is a conjunction of k_i literals. Let x_j^i denotes the j^{th} literal in t_i, where $1 \leq j \leq k_i$. S_F^E is defined as the faulty implementation of S, where F is the name of the fault class and E identifies the exact fault in S. Definition 2.1 formally describes a fault type and Table 1 lists the nine different fault classes.

Definition 3.1. Term Omission Fault (TOF)
If a term t_i is omitted from S then $S_{TOF}^{t_i} = t_1 + ... + t_{i-1} + t_{i+1} + ... + t_m$ where $1 \leq i \leq m$

Coverage Metrics: We define the three structural coverage criteria, namely statement, branch and MCDC, for which tests are generated in our framework.

To achieve *statement coverage*, a test suite should invoke every executable statement in the code. *Branch coverage* mandates that the test suite should execute both the *true* and *false* outcomes of all *decisions* (eg., *if* statements) in the code. This metric is stronger than statement coverage, but is still weak due to the presence of *conditions* with in a *decision*. A *condition* is defined as a boolean expression with **no** boolean operators. A *decision* is defined as a boolean expression with **zero or more** boolean operators. For example, a decision (A $>$ 10 $\|$ B) has two conditions (A $>$ 10) and B.

MCDC [6] was developed to test the effect of *conditions* within a *decision*. MCDC requires that each *condition* with in a *decision* should *independently* affect the outcome of the *decision*. This coverage ensures that the effect of each *condition* is tested relative to other *conditions* within the *decision*. This implies that no *condition* is left untested due to logical masking. For a decision with m conditions, MCDC requires $m + 1$ test cases [6]. It is difficult to attain when compared to statement and branch. For example, consider a decision d with three conditions (A $<$ 0), B and (C $<>$ 10), the test cases generated are shown below:

Example:
d = ((A$<$0) $\wedge$ B $\vee$ (C $<>$ 10))
Test cases generated for MCDC:
case (A$<$0): T_1 = [0, 1, 0] (d=0) and T_2 = [1, 1, 0] (d=1)
case B: T_3 = [1, 0, 0] (d=0) and T_4 = [1, 1, 0] (d=1)
case (C $<>$ 10): T_5 = [0, 1, 0] (d=0) and T_6 = [0, 1, 1] (d=1)
Test suite: { T_1, T_2, T_3, T_6 }

4. Motivation

Velev [13] and Campenhout [2] conducted empirical studies on error/bug categories observed in academic microprocessor projects. The bug categories identified were very similar to the faults observed during validation of software systems [6]. The faults were analyzed and an extended fault model comprising of nine types of faults on boolean expressions was proposed by Lau *et al.* shown in Table 1. Being able to detect these basic fault classes would result in identifying more complex faults, which follows from the *fault coupling effect hypothesis* [7]. We correlate the bug categories in [2, 13] to the extended fault model in [7] and motivate the need for a test generation framework that provides coverage for the fault classes.

In [2], Campenhout *et al.* analyzed modeling errors commonly seen in microprocessors and generated error distributions. We segregated the errors related to signals and gates in the control logic of the processor into three groups: (i) wrong, (ii) missing and (iii) extra. The study reported 33% of the errors occur due to wrong signal source and 4.9% due to wrong gate type. The missing gate errors accounted to 7.4% and missing input(s) to 11.5%. The errors from extra inversion contributed 1.6%. We relate these errors to the fault classes based on omission, insertion and incorrect reference (shown in Table 1) and summarize the correlation in Table 2. Note that the exact description of the modeling error were not provided and certain assumptions were made to arrive at the correlation.

Table 2. Correlation Study

Fault Class in [7]	*Campenhout* et al. *[2]*	*Velev [13]*
ENF, LNF	1.6% (extra)	-
TOF	7.4% (missing)	56.8%(159/280)
LOF	11.5% (missing)	11.4% (32/280)
LIF	-	-
LRF	32.8% (wrong)	26.4%(74/280)
ORF	4.9% (wrong)	0.4% (1/280)

Velev [13] studied three microprocessors to perform a bug col-

lection in pipelined and superscaler designs and arrived at the following bug categories: (i) incorrect control logic, (ii) incorrect connections, (iii) incorrect design for formal verification and (iv) incorrect symbolic simulation. His work identified 280 bugs and presented a neat description of each bug, which was subjected to our analysis. We found that 95% of the bugs identified were subsumpted by the fault classification in [7] and the remaining 5% were artifacts of the formal verification framework in [13]. Our test generation is directed to detect all the design faults except LIF, since none of the errors observed was classified as a LIF.

5. Modeling & Test Generation

The modeling framework facilitates microprocessor modeling through an abstract representation called the *metamodel* [1]. The metamodel defines the syntax and semantic for the language in which the modeler describes the architecture and microarchitecture of the processor.

Architecture Modeling. The processor is specified through a register/memory-level description (registers, their whole-part relationship and memory schematic) and an instruction set capture (instruction-behavior). The metamodel for architectural modeling provides the user with constructs to describe function blocks, if-else blocks, statement sequences and loops as well as the register map and memory layout. On the left of Figure 2, we show the jump-if-not-zero instruction (JNZ) that positions the instruction pointer (IP) to the location specified in source (src) when the zero flag (ZF) is not set. The IP and ZF are references to registers modeled as a part of the architecture. The register map specifies all the registers as well as their whole-part relationship as shown on the right in Figure 2, whereas the memory schematic describes the memory hierarchy. The *src* is an identifier that translates to a register/memory reference or an immediate value during decoding.

Function block: JNZ
Identifier: src
Register reference: IP
Register reference: ZF
If block:
Guard:
Relational Statement: ZF = 0
Action:
Arithmetic Statement: IP = IP + src

Pentium® IV
Register: RAX [63:0]
Reg reference: EAX [31:0]→ RAX
Reg reference: AX [15:0] → EAX
Reg reference: AH [15:8] → AX
Reg reference: AL [7:0] → AX

Figure 2. Architectural Modeling

Microarchitecture Modeling. The language allows the modeler to instantiate and configure different pipe stages, provide ports to interface with the memory and register file, insert instruction registers and describe the control logic needed to perform data forwarding and stalling. It provides a set of basic configurable components such as MUX_n, ALU, INC_n, etc that can be combined to create complex stages that interact with each other as well as with the architectural entities through access ports. In Figure 3, we show the updation of the program counter in the instruction fetch stage (IF) of the VESPA processor [8] modeled using our constructs. The selection criterias of the MUXes are not shown, since they map to statements similar to the ones in architectural modeling. PC and PC2 are program counters, where PC belongs to IF and PC2 is an input to instruction decode (ID) stage. PC2 is updated with the PC or previous PC2 value depending on whether the pipeline is stalled or not. PC is updated with PC+4 (location of the next instruction), the previous PC/PC2 (pipeline stall) or a new address at port Z (branch instruction). The value at Z is obtained from the EX stage.

//PC Multiplexing
$\mathbf{MUX_4}$: PC_MUX
$\mathbf{PC\_MUX_{ip}}$: Z, PC, PC+4, PC2
$\mathbf{PC\_MUX_{op}}$: PC
//Update PC2 after incrementing PC
$\mathbf{INC_4}$: INC_PC
$\mathbf{INC\_PC_{ip}}$: PC
$\mathbf{INC\_PC_{op}}$: PC2
//PC2 Multiplexing
$\mathbf{MUX_2}$: PC2_MUX
$\mathbf{PC2\_MUX_{ip}}$: PC, PC2
$\mathbf{PC2\_MUX_{op}}$: PC2

Figure 3. PC behavior of the IF in VESPA [8]

Metamodel-based modeling is very similar to having a library of configurable components that the modeler instantiates and characterizes to perform a certain functionality. Such a language need not be highly expressive, but it should be extendable. The customizability obtained from the metamodel makes the modeling framework easily extendable with specialized components needed for speculative and out-of-order execution such as re-order buffers and memory arrays.

The test generation framework as shown in Figure 4 converts the architectural and microarchitectural model into Constraint Satisfaction Problems (CSP)s [4] and passes them through a constraint solver and a test case generator to create test suites that validates the microcode and RTL. In order to correctly formulate the CSP, we convert the architectural and microarchitectural model into flow graphs and generate constraints in the form single assignments.

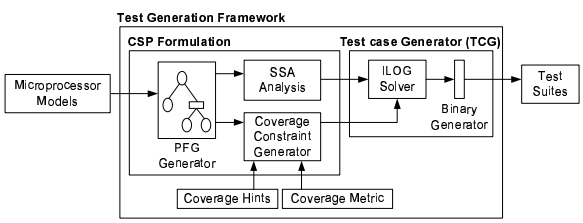
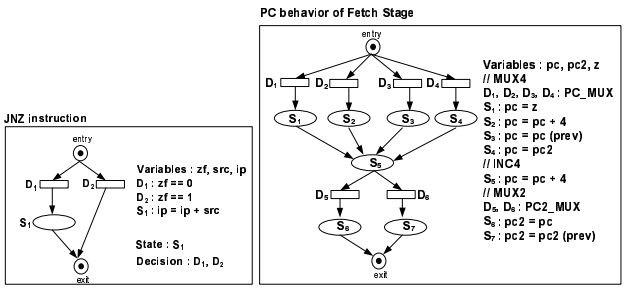


Figure 4. Test Generation Framework

CSP Formulation. A model created using our modeling framework is converted into a Program Flow Graph (PFG), where all the architectural/microarchitectural constructs are mapped to a bunch of states, decisions and transitions. For an architectural model, the statements translate to states and the if-else constructs map to decisions which capture the guard that causes a transition. Function blocks map to hierarchical states, whereas loops lead to cyclic graphs. The identifiers, registers, memory locations and register references are used to instantiate variables that undergoes a change in value whenever a statement updates them. In the microarchitecture model, the basic components translate to a bunch of if-else sequences that when evaluated to *true* execute a sequence of statements. For example, the MUX is basically an *If block* sequence with *Assignment Statement*s that execute based on which selection

criteria is valid in that run. The PFG for the JNZ instruction and the PC behavior of the fetch stage of VESPA is shown in Figure 5.



(a) Instruction model (b) Pipeline stage model

Figure 5. Program Flow Graph

For a graph with a unique path (no branches), the behavior is converted into CSP in a straightforward manner by generating constraint variables and assignment constraints. However for a model with multiple paths, the graph generated undergoes Static Single-Assignment (SSA) analysis [10] to construct the correct set of constraints for the CSP formulation for one complete run (entry to exit). The analysis focuses on states with multiple incoming transitions, which maps to a statement that follows a set of conditional branches. At these join states, parent resolution and variable renaming are performed very similar to how *dominance frontier* [10] is computed in SSA during compiler analysis. The outcome of the analysis is variables and decisions being inserted to help in the CSP formulation. The resulting variables from the SSA analysis for the PC behavior in the IF stage is shown on the left in Figure 6, where $(D_i \implies S_i)$ means if D_i is true then S_i is executed.

Var $pc_0, pc_1, pc_2, pc2_0, pc2_1, z_0$
$D_1 \implies S_1$: $pc_1 = z_0$
$D_2 \implies S_2$: $pc_1 = pc_0 + 4$
$D_3 \implies S_3$: $pc_1 = pc_0$
$D_4 \implies S_4$: $pc_1 = pc2_0$
S_5: $pc_2 = pc_1 + 4$
$D_5 \implies S_6$: $pc2_1 = pc_2$
$D_6 \implies S_7$: $pc2_1 = pc2_0$

Var $pc_0, pc_1, pc_2, pc_3, pc2_0, pc2_1, z_0$
$D_1 \implies S_1$: $pc_1 = z_0$
$D_2 \implies S_2$: $pc_1 = pc_0 + 4$
$D_3 \implies pc_0$
$D_4 \implies S_4$: $pc_1 = pc2_0$
Additional Decisions
$D_1 \vee D_2 \vee D_4 \implies S_8$: $pc_2 = pc_1$
$D_3 \implies S_9$: $pc_2 = pc_0$
S_5: $pc_3 = pc_2 + 4$
$D_5 \implies S_6$: $pc2_1 = pc_3$
$D_6 \implies S_7$: $pc2_1 = pc2_0$

Figure 6. SSA analysis outcome of the PFG in Figure 5.b

Consider the decision D_3 and state S_3, which does not change the current value of pc but forwards the previous value ($pc_1 = pc_0$). Omission of S_3 results in a contention at S_5 with multiple paths, where most of them update the pc register except for one (S_3). Therefore, a parent resolution is performed that inserts additional decisions and variables as shown on the right in Figure 6. The SSA is non-trivial when considering nested-ifs and loop unrolling. The CSP formulation guidelines are outlined below:

Algorithm 1: CSP Formulation

// Given a program flow graph, with variables, states, decisions and transitions
Step 1 For a **variable**, an input constraint variable with a finite domain [min, max] is generated.
Step 2 For a **decision**, the following state should be exercised only when it evaluates to *true* or *false*, respectively and the decision is converted into a conditional constraint of the form *if-then*.
Step 3 For a **state**, a set of input constraint variables are used to update an output constraint variable using an assignment constraint (=).
Step 4 If a state assigns **a new value** to a variable, a new constraint variable is generated for that variable (renaming act in SSA) and future references to the variable will refer to the newly generated constraint variable.
Step 5 If a state assigns **multiple new values** for a program variable, a new constraint variable is generated and for each decision that results in a value, the corresponding conditional constraint is generated such that the new variable can be assigned that value (parent resolution in SSA).

The language used to print out the CSP consist of assignment and conditional constraints. The CSPs developed for the above examples are shown in Figure 7. On the left, the architectural constraints for the CSP generated from the JNZ instruction are shown and on the right the microarchitectural constraints for the CSP generated from the PC behavior in the IF stage are shown.

if($zf_0 = 0$) then $ip_1 = src_0 + ip_0$
else $ip_1 = ip_0$

if($D_1 = 1$) then $pc_1 = z_0$
else if($D_2 = 1$) then $pc_1 = pc_0 + 4$
else if($D_4 = 1$) then S_4: $pc_1 = pc2_0$
Step 5 Illustration for pc_2
if($(D_1 \vee D_2 \vee D_4) = 1$) then $pc_2 = pc_1$
else if ($D_3 = 1$) then $pc_2 = pc_0$
$pc_3 = pc_2 + 4$
if($D_5 = 1$) then $pc2_1 = pc_3$
else if($D_6 = 1$) then $pc2_1 = pc2_0$

Figure 7. CSP formulation for Figure 5.a & 6 (right side)

Test case Generator (TCG). The CSP is given as input to the constraint solver in the TCG. The constraint solver used in our framework is ILOG, which is a commercial solver designed for performance and scalability. The solution generated by the solver consists of a possible value for **every constraint variable** in the problem such that the CSP is satisfied. This solution is called a *test case*. Examples of some possible solutions generated for Figure 7 are shown in Figure 8. Test case T_1 causes the instruction to jump to the location src, whereas T_2 does not. Test case T_3 causes the IF stage to fetch a branch instruction from #1100 and T_4 causes both IF and ID stages of the pipeline to be stalled.

Test cases for JNZ instruction
$T_1 = \{zf_0 = 0, src_0 = 1001, ip_0 = 5, ip_1 = 1006\}$
$T_2 = \{zf_0 = 1, src_0 = 1001, ip_0 = 5, ip_1 = 5\}$
Test cases for PC behavior in IF
$T_3 = \{pc_0 = 1000, pc2_0 = 996, z_0 = 1100, pc_1 = 1100\ (D_1 = 1), pc_2 = 1100, pc_3 = 1104, pc2_1 = 1104\}$
$T_4 = \{pc_0 = 1000, pc2_0 = 996, z_0 = 1100, pc_1 = 1000\ (D_3 = 1), pc_2 = 1000, pc_3 = 1004, pc2_1 = 996\}$

Figure 8. Possible test cases generated for Figure 7

The binary generator creates a program binary by combining the model with the registers and memory locations initialized to the values stored in the input constraint variables obtained from the test case. A program binary for the PC's behavior initializes Z, PC and PC2 to the values of z_0, pc_0, and $pc2_0$ respectively. The output constraints pc_3 and $pc2_1$ are the results obtained by executing the test case against the reference model.

Usage Mode. There are two modes of usage for the test generation framework: (i) Model-random and (ii) Coverage-directed. A test suite in the model-random mode is generated by converting the model into a CSP and solving it multiple times through the solver. Attaining a coverage goal in this mode requires a large number of test cases. However, validators are interested in design-fault based testing, the test cases for which are difficult to obtain using the model-random approach. As a result, the framework also provides a coverage-directed approach that generates one possible test suite,

which attains 100% coverage of a specific goal. In this mode, coverage constraint are given as input to the solver that directs the test generation towards the goal as shown in Figure 4.

6. Coverage Constraints

The coverage constraint generator (CCG) takes two set of inputs besides the program flow graph. Firstly, the coverage annotations that the modeler provides to expedite the reachability of the coverage goal. Secondly, the coverage type based on which the additional constraints are generated. The coverage hints are embedded into the flow graph for certain coverage types and for the others specialized constructs that serves as collectors are provided. The annotations are done by setting the attribute **MARK** associated with a state and decision. The coverage types supported are: (i) Statement Coverage, (ii) Branch Coverage, (iii) MCDC and (iv) Design Fault Coverage. The test suite created for statement and branch-directed test generation is a subset of the test suite created for MCDC. Therefore, we only explain how the coverage constraints for MCDC are generated.

Algorithm 2: MCDC Constraint Generation

// Given a program flow graph G, D = $\mathbf{collect_d}$(G) and $T_S = \phi$
$\mathbf{collect_d}$(G) - gets all decisions in G with the MARK attribute set.
$\mathbf{collect_c}$(d) - gets all conditions in decision d.
$\mathbf{arrange_d}$(D) - sorts the decision list D in ascending order based on depth.
find_path(G,d) - gets a path from the root node *entry* in G to the node d.
gen_constraint(P) - generates the constraints for the path P.
cr_constraint(t) - creates a constraint that enforces t.
check_boolean_opr(d) - checks if d is a *condition* with no boolean operators or a *decision* with one or more operators.
Step 1 $\forall\ d \in \mathbf{arrange_d(D)}$
  Step 1.1 P = **find_path**(G, d)
  Step 1.2 if **check_boolean_opr**(d) == *false* then
    T_T = **gen_constraint**(P) $\cup$ **cr_constraint**(d = 1)
    T_F = **gen_constraint**(P) $\cup$ **cr_constraint**(d = 0)
    $T_S = T_T \cup T_F$
  Step 1.3 else
    Step 1.3.1 $\forall\ c \in \mathbf{collect_c}$(d)
      T_T = **gen_constraint(P)** $\cup$ **MCDC_cov**(c, d, *true*)
      T_F = **gen_constraint(P)** $\cup$ **MCDC_cov**(c, d, *false*)
      $T_S = T_T \cup T_F$ and $T_T = T_F = \phi$
  Step 1.4 rm_duplicate(T_S)

MCDC: The objective of MCDC is that every condition within a decision should independently affect the outcome of the decision. The CCG works off the flow graph and generates the constraints necessary to attain the objective using Algorithm 2. The validator is allowed to tag the problematic decision points and the CCG collects them using $\mathbf{collect_d}$. It then orders these collected nodes using $\mathbf{order_d}$ based on their depth in the graph before generating the constraints. For every collected decision, the CCG identifies a path that reaches it using **find_path** and generates the constraints for the path as well as the additional MCDC constraints for that decision. The constraints for the path is generated using the **gen_constraint** function. For a decision thats just a condition, branch and MCDC generates identical test suites. However for a decision with one or more boolean operators found using **check_boolean_opr**, the additional constraints are generated to tune the evaluation of the decision to the condition's effect. For each condition, the decision should evaluate to *true* and *false* based on whether it is assigned a '0' or a '1'. This results in m test cases for the *0-value* scenario and another m for the *1-value* scenario. After removal of the duplicate test cases using **rm_duplicate** MCDC results in $m + 1$ test cases. To perform MCDC on a decision d with m conditions, w.r.t the j^{th} condition, CCG generates constraints for both the *0* and *1-value* scenarios using Function 1.

Function 1: MCDC_cov(c_s, d, val)

// Given a condition c_s in decision d and, *val*-scenario and the additional constraint set $C_C = \phi$
Step 1 d' = rename(copy(d))
Step 2 $C_C \cup$ **cr_constraint**($d = val \wedge d' = !val$)
Step 3 C_d = $\mathbf{collect_c}(d)$ and $C_{d'}$ = $\mathbf{collect_c}(d')$
Step 4 $\forall\ c_i \in C_d$
  Step 4.1 if $c_i = c_s$ then $C_C \cup$ **cr_constraint**($c_i <> C_{d'}$[i])
  Step 4.2 else $C_C \cup$ **cr_constraint**($c_i = C_{d'}$[i])
Step 5 return C_C

Function 2 **MCDC_cov**(c_s, d, val) copies d into d' and renames the conditions in d'. Then, it generates constraints that assert d to val and d' to $!val$ by the **cr_constraint** function. It also enforces that every condition in d and d' have identical values except for the condition c_s in d and its corresponding copy in d' using the same function.

Design Fault Coverage: The design fault classes are provided in Table 1. To attain this coverage goal, test cases that cover each of these fault class are generated by the CCG. *Our probabilistic analysis of the fault detection capability of MCDC revealed that seven of the nine fault classes are covered by the test suite generated as result of MCDC-directed test generation.*

Consider the literal omission fault class (LOF), MCDC detects this fault because to satisfy this coverage, every literal in the decision has to affect the output of the decision. Therefore a pair of test cases that causes the missing literal to affect the decision's output in the reference model exist, which is missing in the implementation. For atleast one of these test cases, the output of the decision in the implementation will differ from the fault-free decision. Therefore, MCDC-directed test generation finds the bug with a probability 1. The only fault class not covered by MCDC with a probability 1 is literal reference fault, for which CCG produces additional constraints that guide the TCG to detect it.

Example:
$d = A \wedge B \vee C$ (fault-free decision)
$d' = A \wedge B$ (LOF'y decision)
Test cases generated:
case A : [0, 1, 0] (d=0) and [1, 1, 0] (d=1)
case B : [1, 0, 0] (d=0) and [1, 1, 0] (d=1)
case C : [0, 1, 0] (d=0) and [0, 1, 1] (d=1)
Test case [0,1,1] executed on d', results in $d <> d'$

Literal Reference Fault Coverage: The annotations to attain this coverage is captured using the **LRF_list** construct, which allows to create a list of variables. The validator associates a LRF_list with every problematic variable v that tells CCG that v can be replaced by an instance of any variable in the given LRF_list, which results in a fault. If a LRF_list is not provided then, the CCG assumes that every variable can be replaced incorrectly by every other variable, which results in a large set of test cases. Therefore, the validator is recommended to specify a possible LRF_list with a variable and mark that variable, such that CCG can take advantage it. The constraints generation for LRF coverage is shown in Algorithm 3.

Algorithm 3: LRF Constraint Generation

// Given a fault-free decision d with m literals in DNF, a LRF_list associated with literal $x_i \in d$, and $T_S = \phi$
Step 1 $\forall\ y_i \in$ LRF_list
  Step 1.1 d_e = copy(d, 0, i-1) $\vee\ y_i \vee$ copy(d, i+1, m)
  Step 1.2 T_F = **cr_constraint**(d = 0 $\wedge$ d_e = 1)
  Step 1.3 $T_F \cup$ **cr_constraint**($y_i <> x_i$)
  Step 1.4 T_T = **cr_constraint**(d = 1 $\wedge$ d_e = 0)
  Step 1.5 $T_T \cup$ **cr_constraint**($y_i <> x_i$)
  Step 1.6 $T_S = T_T \cup T_F$ and $T_T = T_F = \phi$

After specifying the coverage type and inserting the coverages

hints, the CCG generates the coverage constraints. These are given as additional constraints to the TCG to generate the test suite that achieves the coverage goal.

7. Results

We applied our test generation methodology on VESPA [8], a 32-bit pipelined RISC microprocessor to evaluate our framework. The microarchitecture of VESPA was modeled using our metamodel-based language and converted into a program flow graph from which a CSP was formulated. Additional constraints for each coverage metric were generated using the coverage constraint generator. These coverage constraints along with the CSP are resolved through a solver to generate the coverage-specific test suite. Model-random test suites were generated by asserting the statements and decisions in the model with uniform probability.

The model of the microprocessor consist of 330 unique decisions and 400 unique conditions, where the number of conditions in a decision varied between 1 and 10. These decisions translate to boolean expressions in the HDL implementation of VESPA and are prone to modeling errors. Being able to generate tests that detect these errors improve the quality of functional validation.

To evaluate our framework, we inserted modeling errors into the model from the fault classes described in [7]. Seven modeling bugs (LOF, LNF, TOF, TNF, ENF, $ORF[+]$ and $ORF[.]$), were inserted into each of the 330 decisions in the model. For LRF, 50 decisions were randomly chosen and the variables in the decisions were randomly replaced with other variables in the model. For each inserted bug, we compute the test-set that would detect the bug. Next, we compared the test cases in the test suite generated for model-random, statement, branch and MCDC with the set of test cases that detect each inserted bug (the intersection of the two test suite). The quality of different coverage metrics is measured by the number of distinct bugs detected by a test suite generated to satisfy a coverage metric. In order to do a fair comparison, the number of test cases in the test suites being compared were adjusted to be the same. Table 3 shows the percentage of bugs detected by each of the test suites.

Table 3. Fault Coverage of different Metrics

S.No	*Fault*	*Random (%)*	*Stmt (%)*	*Branch (%)*	*MCDC (%)*
1	LOF	92.7	28.9	32.7	100
2	LNF	99.9	79.9	80.2	100
3	TOF	92	32	36	100
4	TNF	100	76.4	77.6	100
5	ENF	100	100	100	100
6	ORF[+]	96	32	36	100
7	ORF[.]	98	66.6	61.7	100
8	LRF	92.5	72.5	95	95

It is seen that MCDC is uniformly better in detecting each of the design faults when compared to statement and branch. MCDC detects 7 of the 8 fault classes 100% of the time. Statement and branch are able to cover some of the fault classes completely and for some they perform badly. Therefore, these metrics are not stable and weaker than MCDC. This illustration supports our proposal to use MCDC as a coverage goal for microprocessor validation. Model-random on the other hand performs better than statement and branch because random tests generated in our framework can assert multiple decisions and statements simultaneously resulting in a higher fault coverage. However, model-random test generation fails to achieve 100% coverage of design faults such as LOF and TOF, which are commonly seen bugs in microprocessor validation as shown in Table 2.

8. Conclusion And Future Work

Simulation-based validation is widely acknowledged as a major bottleneck in current and future microprocessor design due to the increasing complexity. A possible solution is to perform design error based test generation from a high level description of the microprocessor. Our methodology made two important contributions. Firstly, a metamodel-based modeling framework was developed that captures the structure and behavior of the architecture (register file and ISA) and the microarchitecture (pipeline structure, control and data forwarding logic). CSP formulation of these models resulted in test suites that covered design fault commonly seen during microprocessor validation therefore, illustrating our coverage directed test generation approach. Secondly, we show a high correlation between the design faults proposed by software validation and modeling errors/bugs observed during the study microprocessors validation.

Our future work includes extending the test generation framework to perform exception coverage, event coverage and sequencing coverage for instructions. We will also perform empirical evaluation of the bug finding capability of different coverage metrics as well as the stability of different coverage metrics across test suites.

9. References

[1] A. Dingankar et al., *MMV: Metamodeling Based Microprocessor Validation Environment.*, IEEE International High Level Design Validation and Test Workshop, 2006.

[2] David Van Campenhout et al., *Collection and analysis of microprocessor design errors*, IEEE Design and Test **17** (2000), no. 4, 51–60.

[3] E. Bin et al., *Using a constraint satisfaction formulation and solution techniques for random test program generation*, IBM Systems Journal **41** (2002), no. 3.

[4] Lucas Bordeaux et al., *Propositional Satisfiability and Constraint Programming: A Comparative Survey*, Microsoft Research Technical Report No. MSR-TR-2005-124, 2005.

[5] M. Benjamin et al., *A study in coverage driven test generation*, Proceedings of the 36th Design Automation Conference, June 1999.

[6] Kelly Hayhurst, *A Practical Tutorial on Modified Condition/Decision Coverage*, Report NASA/TM, 2001.

[7] Man F. Lau and Yuen T. Yu, *An extended fault class hierarchy for specification-based testing*, ACM Trans. Softw. Eng. Methodol. **14** (2005), no. 3, 247–276.

[8] David J. Lilja and Sachin S. Sapatnekar, *Designing Digital Computer Systems with Verilog*, Cambridge University Press, 2005.

[9] Prabhat Mishra and Nikil Dutt, *Functional Coverage Driven Test Generation for Validation of Pipelined Processors*, Proceedings of the conference on Design, Automation and Test in Europe, 2005.

[10] Steven S. Muchnick, *Advanced compiler design and implementation*, Morgan Kaufmann Publishers Inc., 1997.

[11] Andrew Piziali, *Functional Verification Coverage Measurement and Analysis*, Kluwer Academic Publishers, 2004.

[12] Shmuel Ur and Yaov Yadin, *Micro architecture coverage directed generation of test programs*, Proceedings of the 36th ACM/IEEE conference on Design automation, 1999.

[13] M. Velev, *Collection of High-Level Microprocessor Bugs from Formal Verification of Pipelined and Superscalar Designs*, International Test Conference, 2003.