Model Based Test Generation for Microprocessor Architecture Validation

Sreekumar V. Kodakara*, Deepak A. Mathaikutty†, Ajit Dingankar‡,
Sandeep Shukla† and , David Lilja*
* The University of Minnesota, Minneapolis, MN 55455
† CESCA, Virginia Tech, Blacksburg, VA 24061
‡ Validation Technology, Intel Corporation,Folsom, CA 95630
{sreek,lilja}@ece.umn.edu, {mathaikutty, shukla}@vt.edu, and ajit.dingankar@intel.com

Abstract

Functional validation of microprocessors is growing in complexity in current and future microprocessors. Traditionally, the different components (or validation collaterals) used in simulation based validation, like simulators and test generators, to validate the system, architecture, microcode, and RTL abstractions of the processor, were manually derived from the specification document. The incomplete informal specification document along with manual translation introduces inconsistency and bugs in the validation collaterals, resulting in increased cost and time to validate the processor. We envision a novel metamodeling based microprocessor modeling and validation environment (MMV) to address this problem. MMV provides a language independent modeling environment to describe the processor at various abstraction levels, a refinement flow to consistently move from one abstraction to the next lower abstraction and code generators to automatically generate the validation collaterals from the models. As a first step towards our vision, in this paper, we describe architectural modeling in MMV and automatic generation of random and coverage directed test suites from the models. We demonstrate the practicality of our approach for validating real world Instruction Set Architectures (ISA) by modeling and generating test cases for eight complex instructions from Intel [1]®Virtualization Technology.

[1]Intel is a trademark or registered trademark of Intel Corporation or its subsidiaries in the United States and other countries.

Model Based Test Generation for Microprocessor Architecture Validation

I. Introduction

Increasing design complexity, shrinking time to market, and high cost of fixing a bug in a released product [1] make functional validation of microprocessors a key ingredient in the product development cycle. The time and cost to validate next generation microprocessors is increasing with design complexity, making validation a very crucial and challenging research problem.

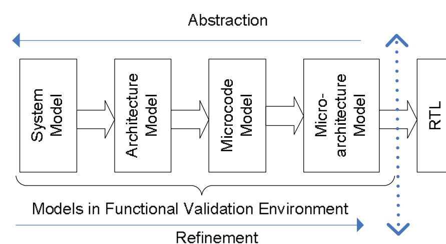


Fig. 1. Different Levels of Abstraction in a Microprocessor

Functional validation is typically done on system, architecture, microcode and RTL levels of abstraction, as shown in figure 1. In the past, the processor models at the architecture and microcode levels were comparatively simpler and easier to design and validate when compared to RTL model. Designing and validating RTL model is hard due to the presence of micro-architectural performance optimizations like out-of-order processing, super-scalar and speculative execution and due to the parallel execution model of hardware. Current trends in microprocessor design, like multiple processing cores and/or threads, and support for software applications like virtualization [2] and security [3], increases the design and validation complexity at processor abstraction levels above RTL.

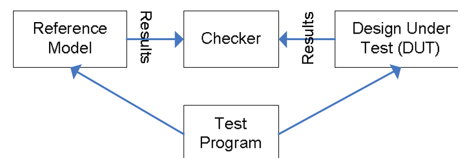
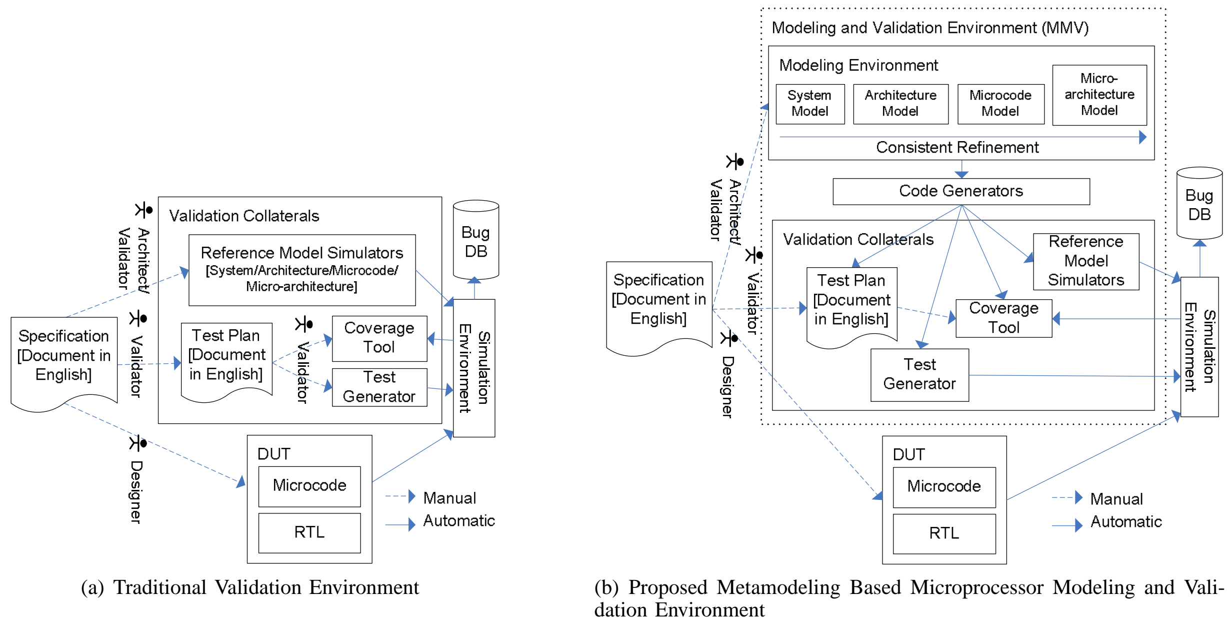


Fig. 2. Simulation Environment

Simulation is widely used to validate large systems like microprocessors. In simulation based validation, shown in figure 2, a test is executed against the golden reference model as well as against the design under test (DUT). A bug is recorded if the checker detects a difference in the results between the two. A simulation based validation environment for microprocessors consists of test generators, simulators for reference models, coverage analysis tools, collection of test cases, test plans etc, which we collectively refer to as *validation collaterals* or *targets* in this paper.

In the traditional approach to validation, shown in figure 3(a), all the collaterals are derived manually from the specification document written in the English language. The manual translation would result in an optimized, high performance and application-tuned collateral, providing the validator with finer control on the details. Nevertheless, the ambiguous informal specification and the subsequent patches in the manual translation introduces many bugs and renders the different targets to be inconsistent with each other. The increasing complexity in the next generation processors at all abstraction levels will further complicate the problems associated with traditional validation environment, resulting in an increased time and cost for the validation cycle.



(a) Traditional Validation Environment

(b) Proposed Metamodeling Based Microprocessor Modeling and Validation Environment

Fig. 3. This figure shows the traditional and the proposed simulation based microprocessor modeling and validation environments. Note that in the traditional environment all validation collaterals are derived manually from the specification document. In the proposed environment *only the models* are derived manually from the specification. All the tools that are used in the validation environment are automatically generated from the models using code generators. In both cases the validation collaterals and DUT is given as input to the simulation environment, as shown in Figure 2

The next-generation validation environments for future microprocessors require a modeling framework to uniformly describe the processor at all abstraction levels and maintain consistency across abstraction levels through a well-defined refinement methodology. They should also be highly re-targetable, by providing code generators that automate the generation of validation collaterals uniformly from the processor model. This consistency during the translation process ensures the removal of mis-matches between the targets. Since all the collaterals are derived automatically from the same processor model, the cost and time associated with modifying the collaterals due to changes in the specification is tremendously reduced. Furthermore, the performance and scalability of these collaterals should be comparable or better than those generated manually.

Our grand vision is the **Microprocessor Modeling and Validation environment** (MMV) shown in figure 3(b). MMV provides a modeling language to describe the processor at various levels of abstraction as well as provide a refinement flow to consistently move from one level to the next lower level. It enables analysis and transformation of the processor model into various validation collaterals through re-targetable code generation derived from the language-independent representation. The key ingredients of MMV are its metamodeling capability and language-independent representation. The metamodeling capability lets MMV to create a customizable multi-abstraction modeling framework using generic modeling concepts. The inter-operable representation allows implementing back-end passes that performs various interpretations and translations of the processor model. We are building MMV on top of a generic modeling environment (GME) [4] that allows for metamodeling and model capture into a data-structure to which they provide well-defined querying mechanism.

As a first step toward our vision, in this paper we demonstrate architectural modeling in MMV by specifying eight complex instructions from Intel®Virtualization Technology [2] and generate tests from the models. At the architectural abstraction in MMV, the models describe the state of the processor as well as the instruction behavior as defined in the Instruction Set Architecture (ISA). The code generator in MMV analyzes the models and generates constraint satisfaction problems (CSPs). The test generator uses a constraint engine based on iLog Solver to solve the CSPs and to generate random and coverage directed test suites. We show that, using MMV, the models can be uniformly converted into CSPs and tests for the models can be generated efficiently. In section VI we demonstrate that our test generation method works efficiently on some of the most complex instructions in modern ISAs, e.g., `vmEnter` in Intel®Virtualization Technology, with generation speeds of about 2000 tests/sec.

The remainder of this paper is organized as follows. Section II briefly introduces the terms used in the paper. Section III describes the modeling framework and Section IV describes how CSP is formulated from the models

and how iLog Solver is used in the test generator to solve CSP's of instruction description. Section V presents the methodology used to evaluate our framework and Section VI presents the results. Section VII presents the related work done in simulation based validation and Section VIII concludes the paper.

II. Background

A. Generic Modeling Environment (GME)

The Generic Modeling Environment (GME) [4] is a configurable toolkit that facilitates the easy creation of domain-specific modeling and program synthesis environment. GME has a set of generic concepts that can be customized to create a new domain specific modeling environment. The customization is accomplished through metamodels which specify the modeling language (syntax & semantics) of the domain. It contains all the syntactic, semantic and presentation information regarding the domain and defines the family of models that can be created using the resultant modeling framework. These models can then be used to generate the applications or to synthesize input to different COTS analysis tools.

The generic concepts describe a system as a graphical, multi-aspect, attributed entity-relationship (MAER) diagram, which internally are UML diagrams. Such a MAER diagram is indifferent to the dynamic semantics of the system, which is determined later during the model interpretation process. The generic concepts supported are hierarchy, multiple aspects, sets, references and explicit constraints, which are described using constructs such as <<Atom>>, <<Model>>, <<Connection>>, etc [4]. *Constraints* in GME are articulated based on the predicate expression language called Object Constraint Language (OCL). OCL constraints are used to express relationship restrictions, rules for containment hierarchy and the values of the static semantics of the domain.

B. Constraint Programming

Constraint Programming (CP) is a study of computational systems based on constraints [5][3]. We model test generation as a constraint satisfaction problem (CSP). The formal definitions of a constraint and CSP are given below.

1) Constraint and Constraint Satisfaction Problem: A constraint is defined as a logical relation among several variables, each taking a value from a given domain. A constraint is formally defined as follows:

- Let $X = \{x_1, x_2, ...x_n\}$ be a set of variables
- Each x_i is associated with a finite set of possible values D_i. D_i is also called the *domain* of x_i.
- A constraint C on $x_1, x_2, ...x_n$ is a subset of $D_i \times ... \times D_n$.
- A CSP is a set of constraints restricting the values that the variables in X can simultaneously take.

A *solution* to a *CSP* is an assignment of one value to every variable x_i in X from its domain D_i, such that all constraints are satisfied at once.

C. Test Case and Test Suite

A test case is defined as a set of inputs, execution conditions and expected outputs that are developed for a particular objective, such as to exercise a particular path in the implementation or to verify its compliance with a specific requirement. A test suite is a collection of test cases.

D. Structural Coverage Criteria

Structural coverage criteria are used to determine if the code structure is adequately covered by a test suite. Research in software testing has shown that structural coverage criteria is effective in finding large classes of bugs [6]. Since sequential execution of instructions at architecture abstraction is similar to sequential execution of software, we decided to use structural coverage criteria for generating coverage directed tests. In this section we define the two different structural coverage criteria that is currently modeled in our framework.

1) Statement Coverage: Statement coverage is used to verify if every statement in the code is reachable. To achieve 100% statement coverage, a test suite should invoke every executable statement in the code. Statement coverage does not take control flow into consideration and hence is considered as a weak coverage metric.

2) Branch/Decision Coverage: Branch coverage is used to test the control constructs in the code. To achieve 100% Branch coverage, 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 and is extensively used in software testing. Statement coverage is automatically satisfied when branch coverage is satisfied.

[3]Here we use constraints in two contexts: that of test generation problem using constraint programming and the other of model consistency using OCL.

III. MODELING FRAMEWORK

The microprocessor domain-specific syntax and static semantics is captured as a *metamodel* that provides the multi-abstraction modeling language. A modeler instantiates this metamodel to perform microprocessor modeling. There are two type of users in **MMV**, firstly the metamodel creator that we call the *metamodeler* and secondly the *modeler* that uses the metamodel for the modeling activity. Both these user work with GME, but the metamodeler define the modeling language and the respective interpreters/analysis engines, whereas the modeler uses the modeling framework (metamodel + code generator) to create the microprocessor model and then through buttons on the console of the framework generates the corresponding target code.

The metamodel is captured through generic concepts described using UML diagrams. This provides customizability, which is necessary criteria for next generation microprocessor modeling frameworks. The metamodel also consist of *metamodeling rules* described as constraints specified in OCL. These rules ensure consistency as well as enforce modeling guidelines that prompts the user of model-time violations. The UML-capture enables visualization that is provided through an editor, where the modeling is performed and the OCL constraints are activated to flag any modeling violations. The framework can also be used in a non-visual description mode, where the metamodel's equivalent textual language is used for modeling. The textual language and the UML-based metamodel have a one-to-one correspondence, which is exploited to visualize the textual model description and vice-versa, hence taking advantage of a front-end and resolving the scalability issues associated with a visual framework. In this paper, our contribution is the metamodel that depicts an architectural abstraction of a microprocessor, where the modeler is allowed to provide register-level and instruction-level description of the processor. Therefore, our modeling framework provides a visual and formal specification process for the ISA as opposed to it being an architectural specification document, where each instruction is explained in English and pseudo code.

3) Architecture Modeling: At this abstraction, the processor consist of the **Core**, **Main_Memory** and **ISA**, which are the main constituents of the metamodel. The **Main_Memory** entity is used to model the memory by capturing information on the address size, and number of bytes/location, whereas the **Core** is used to describe the CPU states of the processor. The core at this level is captured through a register-level description, which boils down to specifying the different registers and their part-whole relationship (such as AX is a part of EAX in Intel®'s architecture).

The definition of the instruction set is main part of architectural abstraction of the processor. The ISA specification is provided through the **Instruction** entity in the metamodel, which is shown in Figure 4. Each instruction has a behavior which is described using attributes and the various entities shown in the Figure, which together constitute the modeling language. *Group* is an attribute of each instruction that enumerates the possible instruction categories such as data transfer, arithmetic, etc and a modeled instruction should belong to one of these.

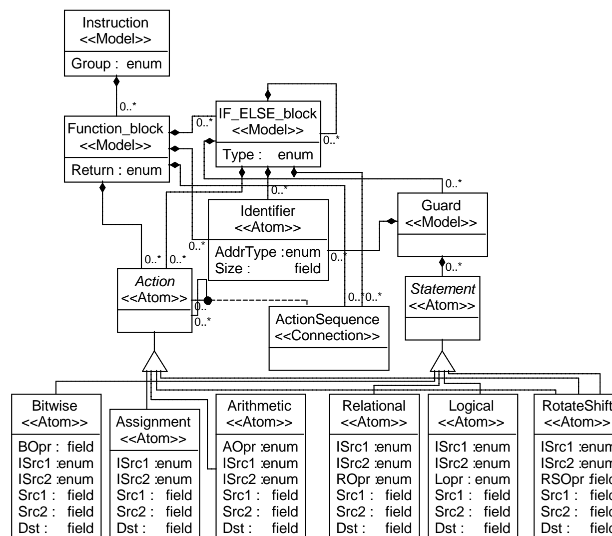


Fig. 4. Instruction-level Description

The language has constructs to write functions, statements and define identifiers. A **Function_block** acts as a collection of the different statements. An **IF_ELSE_block** captures conditional expressions of the behavior. It can

be seen as a guard-action pair, where a **Guard** is a logical/relational/bitwise/rotate & shift statement and an **Action** is an arithmetic/assignment/bitwise/rotate & shift statement. Three of the attributes of each statement capture the two source and destination operands, whereas the other two attributes that start with an *I* specify whether these operands need to be inverted during the expression evaluation. The **Opr* attribute enumerates the list of possible operators for each of these statements. The operand input for a statement correspond to an **Identifier** entity, which needs to be instantiated and characterized before their usage within a statement (enforced by a constraint). The *AddrType* attribute is used to capture the addressing mode of these identifiers as they serve as operands to the instruction. The metamodel also contains **ActionSequence**, which allows for sequential specification of statements. Similar sequencing construct exist for if-else blocks and actions, which is not shown due to lack of space. The internal representation of the visual model of the ADD instruction and its corresponding textual representation is shown in Figure 5.

```
Function_block: add
Identifier: src                              add(src, dst)
Identifier: dst                              {
Arithmetic Statement: S0 := src + dst          dst = src + dst;
Assignment Statement: S1 := dst = S0         }
ActionSequence: A0 := S0 → S1
```

Fig. 5. The internal representation of the visual model of ADD instruction and its corresponding textual description is shown. For the sake of readability, we use the textual representation in all the examples in this paper.

Another important part of the modeling framework is the constraint manager, which enforces modeling restrictions (OCL constraints) and checks for violation at model-time. For example, one of the instruction group is shift & rotate and a constraint enforced on the instructions that belong to this group is that the destination's addressing mode must be immediate and cannot be a register or memory. Similarly, a set of constraints have been written for each of the instruction group and based on the value chosen for the *Group* attribute, the constraints are activated and enforced during modeling. Furthermore, the **Core** is used to describe the CPU of the processor and has an attribute *CoreType* that is used to inform the framework that an existing core is being modeled. This allows the framework to enforce constraints based on the characteristics of the chosen core. An example would be that the Pentium Pro® core's register description is allowed 32, 16 and 8-bit registers, but some of the recent core allow for 64-bit registers. If the user has set the *CoreType* to *Pentium Pro*, then the framework activates a trigger that will flag a violation whenever a 64-bit register is created. Many constraints are written in MMV to enforce such static semantics and enable a correct-by construction modeling approach.

IV. CSP Formulation and Test Generation

The instruction model in MMV is converted into CSPs by the code generator. The test generator solves the CSP to generate test cases and coverts them into test program binaries. We chose constraint solver over SAT solver as the reasoning engine in our test generator due to the presence of variables with large domains and arithmetic constraints in instruction descriptions. It is easier and efficient to translate variables with large domains and arithmetic constraints into CSP rather than boolean propositional formulas required by SAT solvers [7]. The constraint solver used in our framework is iLog Solver [8]. iLog solver is a commercial solver designed for performance and scalability. It is widely used to solve complex combinatorial problems in diverse areas like production planning, resource allocation, personnel scheduling etc.

A. CSP Formulation For Generating Random Test Cases

Many functionally equivalent CSPs can be derived from an instruction model. In this paper, we follow the rules described below to convert the instruction model into one such CSP.

1) Every state element (ie., registers or memory) that is used in the instruction is represented as a constraint variable with a finite domain [min,max]. The constraint variables are always positive. An additional boolean constraint variable is associated with integer variable to act as sign bit to represent both positive and negative values where required.
2) Every instruction has a set of input and output constraint variables. The input constraint variables represent state elements that are *used* in the instruction description and output constraint variables represent state elements that are *modified* by the instruction. Constraints for a sequence of instructions are generated by using the output constraint variables of one instruction as input constraint variable of the next instruction in sequence.

3) Every constraint variable in CSP is the target of only one assignment. If a state element is assigned a new value in the instruction, a new constraint variable is generated for that state element and future references to that state element will refer to the newly generated constraint variable. This is very similar to static single assignment form (SSA) [9] used in compiler analysis.
4) The instruction behavior could have if-then-else statements. The statements within the *then* block or the *else* block should be exercised only when the guard is true or false, respectively. We use a predicate variable for each statement to enforce this rule. If any state element is assigned a new value (and hence a new variable) within the *then* or *else* block, then constraints should be added such that statements following the if-then-else block will use the correct variable depending on the path taken by the instruction.

In the following subsections, we describe how we apply these rules to convert the instruction descriptions of increasing complexity into CSPs.

1) ADD Instruction: Figure 6 shows the behavior of an `add` instruction. Let A and B represent two registers of the CPU. The description states that A and B are added and the result is stored in A. Let a_0 and b_0 be the constraint variables corresponding to the state elements A and B, respectively. If the A and B are 32 bit registers, then the domain of a_0 and b_0 is $(0, 2^{31} - 1)$. Since A gets a new value during the execution of the instruction, we define a new variable a_1 and assign the calculated value $(a_0 + b_0)$ to a_1. The equation $a_1 = a_0 + b_0$ represents the constraints of the `add` instruction. The variables (a_0, b_0) and (a_1, b_0) represent the input and output constraint variable set of `add`. (a_1 and b_0) will be the input constraint variable set for the next instruction in sequence.

Table 6(c) shows one possible test case generated by the test generator. The constraint variables a_0, b_0 and a_1 are assigned values 5,7 and 12 respectively. The test generator converts this test case into a program binary which contains an `add` instruction and the registers A and B initialized to the values stored in a_0 and b_0 respectively.

ADL Description of ADD Instruction

add()
{
A=A+B;
}

Converting the Description to Constraints
Initial State Variables: a_0,b_0 Constraints: $a_1 = a_0 + b_0$; Final State Variables:a_1,b_0

A possible test case
$a_0 = 5$, $b_0 = 7$, $a_1 = 12$

Fig. 6. ADL Description of simple `add` instruction, its corresponding constraints and a sample test case

2) Instruction with a conditional statement: iLog Solver allows conditional constraints of the form if-then in CSPs. We use this feature to generate constraints for instruction descriptions with conditional statements in it. Figure 7 shows an ADL description of a hypothetical instruction `foo` which has a conditional statement and an assignment statement in its description.

The constraints for this instruction is given in figure 7(b). The constraint variables for the state elements A and B are a_0 and b_0 respectively. We also declare a boolean constraint variable t_0 that act as predicate variable for the decision (ie., condition in the *if* statement). The first two constraints relate the decision to t_0. If t_0 is set to 1, a_0 and b_0 will be assigned values such that the decision ($a_0 == 0$ `AND` $b_0 == 1$) is true and vice-versa. Since B gets a new value within the then block, a new variable b_1 is generated for B. The next constraint relate the statement $B = 2$ to t_0. If t_0 is set to 1, then the statement $B = 2$ should be executed, which follows the semantics of if-then statement. Hence if $t_0 == 1$, b_1 is set to 2, else b_1 can take any value within its domain. The state element B after the execution of `foo` is represented either by b_0 or b_1. We generate a new variable b_2 to represent the value of B after the execution of `foo` and add the final two constraints which assigns b_0 or b_1 to b_2 depending on the value of t_0. This completes the constraints for `foo`.

It should be noted that the CSP can be formulated without using predicate variable t_0. t_0 is added to make the constraints more readable in complex instruction flows and for easy manipulation of constraints when generating test cases for branch coverage. The memory and performance overheads due to predicate variables were found to be negligible in our experiments.

3) Instructions with Loops: Some complex instructions in modern day processors have loops in their behavior. If the number of iterations of the loop is deterministic, the loop is first unrolled and constraints are generated for unrolled version of the instruction behavior. An example of such an instruction is $MOVS$ in IA32 architecture, which moves data from one set of memory locations to another. For loops with non-deterministic loop counts, we use the trial and error method followed in [10] to generate CSP.

ADL Description of `foo` with if statement

`foo()`
{
if($A == 0$ AND $B == 1$)
{
$B = 2$;
}
}

Converting the Description into Constraints

Initial State Variables: a_0,b_0
Predicate Variables: t_0
Constraints for ADL Description:
if($t_0 == 1$) ($a_0 == 0$ AND $b_0 == 1$)
if($t_0 == 0$) !($a_0 == 0$ AND $b_0 == 1$)
 	
if($t_0 == 1$) ($b_1 = 2$)
 	
if($t_0 == 0$) ($b_2 = b_0$)
if($t_0 == 1$) ($b_2 = b_1$)
Final State Variables: a_0,b_2

A possible test suite to get 100% branch coverage	
Additional Constraint	**Test Case**
$t_0 = 1$	$a_0 = 0$, $b_0 = 1$, $t_0 = 1$ and $b_1 = 2$ (taken)
$t_0 = 0$	$a_0 = 2$, $b_0 = 3$, $t_0 = 0$ and $b_1 = 0$ (not taken)

Fig. 7. ADL Description of an instruction with a decision

B. CSP Formulation For Coverage Directed Test Generation

Every decision in the code have to be *true* and *false* atleast once to satisfy branch coverage. A set of test cases that satisfies this requirement is said to form a test suite that satisfies branch coverage criteria. Each test in the test suite is generated to exercise one decision in the behavior. This is ensured in MMV by generating constraints on the predicate variables of each decision in the code in addition to those generated from the instruction behavior.

Table 7(c) shows one possible test suite generated by the test generator to get 100% branch coverage on the behavior of `foo`. To generate the first test case, an additional constraint $t_0 = 1$ is added to the CSP. This forces a_0 and b_0 to 0 and 1 respectively, which causes the decision to be taken. Similarly in the second test case, an additional constraint $t_0 = 0$ is added to the CSP. This causes a_0 and b_0 to take any value other than 0 and 1 respectively, which causes the decision to be not taken.

For instructions with multiple decisions in its behavior, the generated constraints should also ensure that the execution flow reaches the current decision. In MMV, the additional constraints for each test case is generated by analyzing the control flow graph (CFG) of the instruction behavior.

C. Constraint Solving using iLog Solver

ILog Solver is not designed specifically to handle operations derived from instruction behavior, i.e., some operations that arise naturally and frequently in the description of instruction execution are not natively supported in iLog Solver. For example, iLog solver does not support bitwise logical operation in the constraints. Bitwise logical operations are used in many complex instructions for masking and extracting specific bits from control registers as well as in logic instructions. iLog solver also restricts the size of the integer constraint variables. In their current 32-bit implementation, the maximum size of the integer constraint variable is only 31 bits. Hence it cannot be directly used to represent 64-bit and 128-bit registers that are found in current day processors.

We overcame this limitation by adding a library on top of iLog Solver, which converts the bitwise operations and variable size integer variable into constraints that are natively supported in iLog Solver. In our library, we have two representations for each state element. One is an array of boolean variables and the other is an array of integer variables. Constraints are added between the two representations, to keep them consistent with each other. Each boolean variable in the boolean array represents one bit of the state element. Bitwise operations between two state elements in the instruction description are represented as constraints between its corresponding boolean variables. The integer array is used to support state elements that is larger than 31 bits (for example, 64-bit registers). Each integer variable in the integer array represents a part of the state element. All arithmetic constraints on state element is converted into constraints for each variable in the integer array.

The test generator has a memory and performance overhead due to two different representations for each variable in the instruction description, which could affect its scalability. To reduce the memory requirements, we generated the two representations only for those state elements that had both logical and arithmetic operations performed on it. In our experiments we found that the average memory consumed by the test generator is less than 8 MB for a CSP with 6,500 constraint variables.

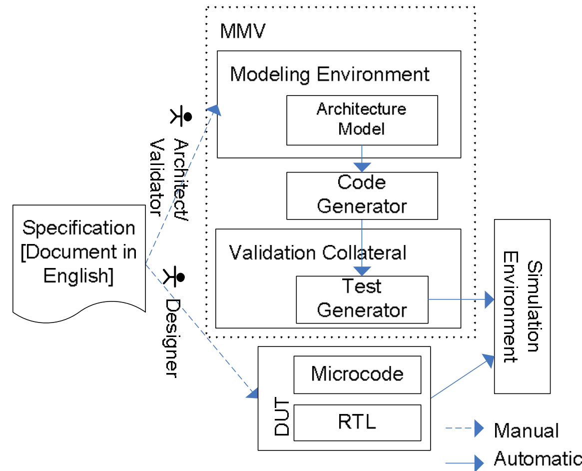


Fig. 8. Model Based Test Generation Framework

```
VMXOFF ()
{
 //VIRTUAL_MODE_CHECK
 if(CR4.VMXE == 0)
  {
  UD == 1; //RAISE_UD_EXCEPTION
  return;
  }

 //COMPATIBILITY_MODE_CHECK
 if(IA32_EFER.LMA == 1 && CS.L == 0)
  {
  UD == 1; //RAISE_UD_EXCEPTION
  return;
  }
}
```

Fig. 9. Behavior of VMXOFF Instruction

V. Evaluation Methodology

In this section, we describe the test generation framework, instructions for which the tests were generated and the metric used to evaluate the framework.

A. *Test Generation Framework*

Figure 8 shows the test generation framework evaluated in this paper. The behavior of the instruction (ie., model) is modeled in the visual modeling environment. The convert this model into CSPs using the steps outlined in section IV. The CSP is given as input to the test generator and random and coverage directed test cases are generated. The test cases are converted into test program binaries, which are then executed in the reference models and DUT in the simulation environment.

B. *Modeled Instructions*

Eight instructions from VT were modeled in this study. Table I briefly describes the instructions in evaluated in this paper. VT instructions, also known a VMX operations, provide architectural support to run multiple operating systems concurrently on a processor. The processor operates in *root* and *guest* modes when virtualization is enabled. A thin software layer called Virtual Machine Monitor (VMM) runs in the root mode and multiple operating systems

can run in the guest mode. An OS running in guest mode has restricted privileges to perform operations that affect the global state of the processor. When an OS in guest mode attempts to perform a restricted operation, the processor stops the execution of the guest OS and transparently switches control to the VMM. The VMM performs the operation and switches the control back to the guest operating system. The instructions and hardware support is provided in VT to switch from guest mode to root mode and back efficiently and transparently. The state of the guest operating system is stored in a special data structure called VMCS. The VMCS is maintained in the main memory of the system. Given the nature of the VMX operations and the complexity of IA32 [11], many consistency checks should be performed during the execution of VMX instructions. Figure 9 shows the first two consistency checks performed in the `vmOff` instruction. `vmOff` instruction is used to exit virtualization mode and it should be executed only when the processor is in virtualization mode. In the first consistency check, `if(VMXE == 0)`, the instruction checks to see if the processor is in virtualization mode. If not, the processor raises an undefined-opcode exception (UD). In this paper, we modeled all the non-System Management Mode [11] consistency checks and generated tests to exercise the checks.

TABLE I
VT INSTRUCTION DESCRIPTION

S.No	Name	Description
1	VMXON	Enter VMX Operation
2	VMXOFF	Leave VMX Operation
3	VMREAD	Reads an entry from the VMCS control Structure
4	VMWRITE	Writes an entry from to the VMCS control Structure
5	VMPTRLD	Loads the current VMCS Pointer from Memory
6	VMPTRST	Store the VMCS Pointer to Memory
7	VMLAUNCH	Launches a Virtual Machine in guest mode
8	VMCLEAR	Copy VMCS data cached in the processor to VMCS region in memory

C. Metrics

The metric used to evaluate our test generation framework is the time to generate tests for each instruction. All measurements were done on a 64-bit 1.6 Ghz IBM Power4 machine with 15 GB memory running AIX 5.2. The measurements were repeated several times and the average time is reported. The results includes the time taken by the solver to solve the CSPs and does not include the time taken by the code generator to generate the CSPs or the time taken to print the solutions to files. The former is done only once and takes very negligible amount of time, while the latter is a property of the file system of AIX operating system.

VI. PERFORMANCE AND COMPLEXITY

Table II shows the number of constraint variables in the CSPs for the instructions evaluated in this paper. It includes the constraint variables representing the state elements accessed by the instructions and the predicate variables created by the code converters to represent decisions in the instruction descriptions. We use the number of variables as an indicator for the complexity of the instruction. The instructions can be divided into three classes based on its complexity. `vmEnter` is the most complex instruction with 6560 variables, followed by `vmOn` with 765 variables and the rest of the instructions each with 358 variables. We evaluated the performance of the test generator by generating random and coverage directed tests. The time taken to generate $10,000$ random tests is shown in table III. The test generator takes 5.13s to generate $10,000$ random tests for `vmEnter`, the most complex instruction in VT, which corresponds to *only* 0.51ms to generate one test. The average time taken to generate one random test for instructions with 765 and 358 variables is only 0.022ms and 0.013ms respectively. Table IV shows the time taken to generate a test suite that satisfies branch coverage. The number of test cases in the test suite, the time taken to generate the test suite and the average time taken to generate one test in the test suite is shown. The average time to generate a test in the test suite ranges from 2.01ms to 4.26ms, which is higher than the time reported for random test case generation. This is because of the initial overhead in iLog solver to create an internal representation of the CSP. Once the internal representation is created, the solver can generate multiple solutions in

TABLE II
SIZE OF CSP

Instruction	Vars
vmEnter	6560
vmClear	358
vmOff	358
vmOn	765
vmPtrLoad	358
vmPtrStore	358
vmRead	358
vmWrte	358

TABLE III
TIME TAKEN TO GENERATE 10,000 RANDOM TESTS

Instruction	Time (s)	Time/test (ms)
vmEnter	5.13	0.513
vmClear	0.13	0.013
vmOff	0.13	0.013
vmOn	0.22	0.022
vmPtrLoad	0.13	0.013
vmPtrStore	0.13	0.013
vmRead	0.13	0.013
vmWrite	0.13	0.013

significantly less time than initial overhead for the CSPs of the instruction descriptions evaluated in this paper. In random test case generation, the initial overhead was amortized over 10,000 solutions while for coverage directed test generation it is amortized over smaller number of tests. This causes the time to generate coverage directed tests to be higher than the time to generate random tests. Nevertheless, the time taken to generate a test is still very low.

`vmEnter` is probably the most complex instruction in any existing ISA, and the results clearly indicate that the proposed test generator works very well for the instruction. Hence we can conclude that the CSP formulation and test case generation is practical for validating complex real-world instruction descriptions.

VII. RELATED WORK

Test generation for simulation based validation can be broadly classified into Manual and Automatic Test generation. In manual test generation, the test writer understands the specifications and manually generates the test cases for a specific scenario. High quality test cases that exercise different corner cases can be obtained from

TABLE IV
TIME TAKEN TO GENERATE TEST SUITE THAT SATISFIES BRANCH COVERAGE

Instruction	No. of Test Cases	Time	
		Time (ms)	Time/test (ms)
vmEnter	153	413.36	2.70
vmClear	5	10.06	2.01
vmOff	5	10.06	2.01
vmOn	8	34.10	4.26
vmPtrLoad	5	10.06	2.01
vmPtrStore	5	10.06	2.01
vmRead	5	10.06	2.01
vmWrite	5	10.06	2.01

manual test generation. Due to the complexity of current and future microprocessors, manually generating test cases is very time consuming, tedious and error prone.

Adir et.al. [12] described a model based random test generator based on constraint solving. Their modeling environment is specifically designed for modeling the behavior of instructions and for generating test cases. It is not clear if the models can be re-targeted to generate other validation collaterals like simulators or coverage analysis tools or if the modeling environment supports the modeling of processors at different abstraction levels. In MMV, we can model all abstractions of the processor and can generate different validation collaterals from the models. Also, their constraint solver [13] is designed and developed specifically for solving CSPs of instruction descriptions and for generating tests. It is very expensive and time consuming to develop an efficient high performance constraint solver. In our paper, we demonstrate how a commercially available constraint solver can be customized to solve CSPs for instruction behavior.

Mishra et.al. [14] used EXPRESSION to describe the processor and generated tests for a specific coverage metric using model checker. They demonstrated their framework for embedded processors. Since model checker is a formal verification tool, it is not clear how the test generator will scale with the size of the design.

VIII. Conclusion and Future Work

The time and cost associated with functional validation of microprocessors is increasing due to the introduction of multi-core processors with support for software applications like virtualization [2] and security [3]. In a traditional approach to simulation based validation, all validation collaterals were manually derived from the specification document. The incomplete informal specification along with manual translation introduces inconsistency and bugs in the validation collaterals, resulting in increased cost and time to validate the processor. In this paper, we described a novel metamodeling based microprocessor modeling and validation environment (MMV) to address this problem. MMV provides a language independent modeling environment to describe the processor at various abstraction levels, a refinement flow to consistently move from one abstraction to the next lower abstraction level and code generators to automatically generate the validation collaterals from the models. We demonstrated the practicality and speed of our framework by modeling eight complex instructions from Intel®Virtualization Technology and by automatically generating random and coverage directed test suites from the models using constraint programming. We used the number of variables required to model the instructions as a measure of complexity of the instructions and the time taken to generate a test case as a measure of efficiency of our test generator. For `vmEnter`, probably the most complex instruction in IA32 architecture with about 6500 variables in its model, the test generator took *only* about 0.5ms to generate a test case. In the future, we intend to extend MMV to model the processor in the system, microcode and micro-architecture abstractions and use it to study the effectiveness of coverage metrics in detecting functional bugs.

Acknowledgment

This work was supported in part by Intel Corporation, the University of Minnesota Digital Technology Center, and the Minnesota Supercomputing Institute.

References

[1] Price D., "Pentium® FDIV flaw-lessons learned," in *IEEE Micro*, vol. 15, no. 2, April 1995, pp. 86–87.
[2] R. Uhlig, G. Neiger, D. Rodgers, A. L. Santoni, F. C. Martins, A. V. Anderson, S. M. Bennett, A. Kagi, F. H. Leung, and L. Smith, "Intel® Virtualization Technology," in *Computer*, vol. 38, no. 5, May 2005, pp. 48–56.
[3] "LaGrande Technology Preliminary Architecture Specification." Intel®, 2006. [Online]. Available: http://www.intel.com/technology/security/downloads/prelim-lt-spec_d52212.htm
[4] A. Ledeczi, M. Maroti, A. Bakay, and G. Karsai, "The Generic Modeling Environment," Vanderbilt University. Institute for Software Integrated Systems Nashville, 2001.
[5] R. Bartk, "Constraint Programming: In Pursuit of the Holy Grail," in *Proceedings of the Week of Doctoral Students (WDS99).* Prague: MatFyzPress, 1999, pp. 555–564.
[6] A. Dupuy and N. Leveson, "An Empirical Evaluation of the MC/DC Coverage Criterion on the HETE-2 Satellite Software," 2000.
[7] L. Bordeaux, Y. Hamadi, and L. Zhang, "Propositional Satisfiability and Constraint Programming: A Comparative Survey," in *Microsoft Research Technical Report No. MSR-TR-2005-124*. Microsoft, 2005.
[8] J. Puget, "A C++ implementation of CLP," in *Technical report ILOG SOLVER collected papers*. ILOG, 1994.
[9] S. S. Muchnick, *Advanced compiler design and implementation*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1997.
[10] F. Fallah, P. Ashar, and S. Devadas, "Simulation Vector Generation from HDL Descriptions for Observability-Enhanced Statement Coverage," in *Design Automation Conference*, 1999, pp. 666–671. [Online]. Available: citeseer.ist.psu.edu/fallah99simulation.html
[11] "Instruction Set Reference (Volume 2A and 2B)," in *IA-32 Intel® Architecture Software Developers Manual.*
[12] A. Adir, E. Almog, L. Fournier, E. marcus, M. Romon, M. Vinov, and A. Ziv, "Genesys-Pro: Innovations in Test Program Generation for Functional Processor Verification," in *IEEE Design and Test of Computers*. IEEE, 2004.
[13] E. Bin, R. Emek, G. Shurek, and A. Ziv, "Using a constraint satisfaction formulation and solution techniques for random test program generation," in *IBM Systems Journal*, vol. 41, no. 3. IBM, 2002.
[14] P. Mishra and N. Dutt, "Functional Coverage Driven Test Generation for Validation of Pipelined Processors," in *DATE '05: Proceedings of the conference on Design, Automation and Test in Europe*. Washington, DC, USA: IEEE Computer Society, 2005, pp. 678–683.