

MMV: Metamodeling Based Microprocessor Validation Environment

Ajit Dingankar*, Deepak A. Mathaikutty[†], Sreekumar V. Kodakara[‡],
Sandeep Shukla[†] and , David Lilja[‡]

*Validation Technology, Intel Corporation, Folsom, CA 95630

[†] CEsCA, Virginia Tech, Blacksburg, VA 24061

[‡] The University of Minnesota, Minneapolis, MN 55455

ajit.dingankar@intel.com, {mathaikutty, shukla}@vt.edu, and {sreek, lilja}@ece.umn.edu

Abstract—With increasing levels of integration of multiple processing cores and new features to support software functionality, recent generations of microprocessors face difficult validation challenges. The systematic validation approach starts with defining the correct behaviors of the hardware and software components and their interactions. This requires a new modeling paradigm that supports multiple levels of abstraction. Mutual consistency of models at adjacent levels is crucial for manual refinement of models from the full chip level to production RTL, which is likely to remain the dominant design methodology of complex microprocessors in the near future. In this work, we present MMV, a validation environment based on metamodeling, that can be used to create models at various abstraction levels and to generate most of the important validation collaterals, viz., simulators, checkers, coverage and test generation tools. We illustrate the functionalities in MMV by modeling a 32 bit RISC processor at the system, instruction set architecture and microarchitecture levels. We show by examples how consistency across levels is enforced during modeling and also how to generate constraints for automatic test generation.

I. INTRODUCTION

Increasing design complexity, shrinking time to market, and high cost of fixing a bug in a released product make validation of microprocessors a key ingredient in the product development cycle. The complexity and hence the cost of validating a microprocessor increases from one generation to the next, making it a very important and challenging problem for current and future designs.

Designers model the microprocessor at different levels of abstraction. The complexity of models increases as we go down the abstraction hierarchy. Validation complexity at the microarchitecture level of abstraction dwarfs the complexity at higher levels of abstraction because of the additional features introduced in the microarchitecture to support performance optimizations like out-of-order, superscalar and speculative execution. Hence most of the validation effort is spent in validating the functionality introduced in the microarchitecture. This trend may change in the future generations of multi-core processors which are designed with multiple cores in a single die and with hardware support for software applications like virtualization [1]. This shift in design paradigm increases the validation complexity at *all* levels of abstraction of the processor. For example, in Intel processors, eight complex instructions and two modes of operation are added to the Instruction Set Architecture (ISA) to support virtualization. Multiple cores and the associated communication protocols increase the complexity at the system level abstraction.

In order to deal with the increasing complexity in processor validation, there is a need for a modeling and validation environment in which all levels of abstraction of the processor can be modeled in a unified manner. The modeling environment should support stepwise manual refinement (e.g., by architects, designers and validators) from one level to another, enforcing consistency of refinements to a predefined degree. The environment should also support creation of tools for model analysis at any level of abstraction for the purpose of generating validation collaterals like simulators, coverage analysis tools and test content.

In this paper we propose a visual Microprocessor Modeling and Validation Environment (MMV). MMV is built on top of

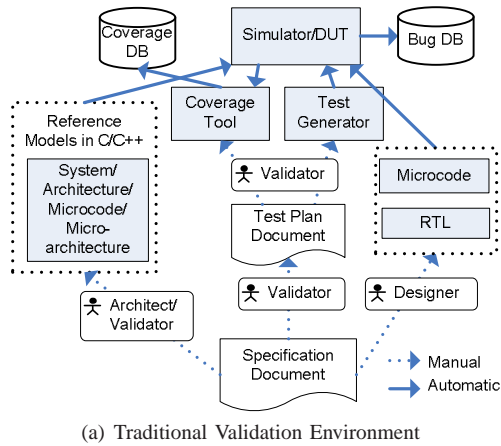
a metamodeling framework which enables retargetable code generation and facilitates creating models at different levels of abstraction, without tying them to any specific programming language. This language-independent representation enables the analysis and transformation of the models uniformly into various validation collaterals. It also allows a manual refinement methodology and enforces consistency within and across abstraction levels. As an example, we model a 32 bit RISC processor at the system level, as well as architecture and microarchitecture levels to illustrate MMV's modeling capability and discuss how to extract constraints for test generation at each of these levels of abstraction.

II. BACKGROUND

The design teams pass through many levels of abstraction, from system to microarchitecture, when designing a microprocessor. They develop functional specifications for each level of abstraction which then becomes the standard document against which the implementation of the processor is validated. The goal of functional validation is to ensure that the microcode, RTL and the fabricated product in silicon implement the behavior defined in the system, architecture, microcode and microarchitecture specifications [2].

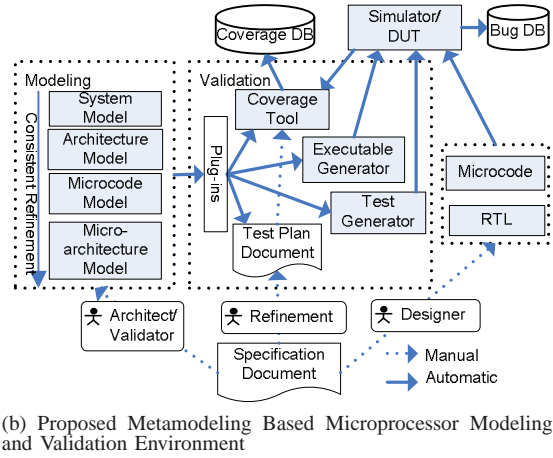
Simulation based validation is widely used to validate microprocessors in the industry. A traditional simulation based validation framework is shown in figure 1(a). In this framework, the models of the microprocessor at different levels of abstraction is manually derived from the specification document. These models act as reference models against which the RTL/microcode/silicon of the processor is validated. A test plan document, which describes the scenarios that needs to be tested during validation is manually derived from the specification. The test plan acts as the reference document for manually guiding the test generators and coverage analysis tools. The generated tests are run on the RTL/microcode/silicon and the validated processor models. The results produced by the test is compared for a mismatch.

There are several drawbacks in the traditional validation environment. First, most of the models and tools are generated from the specification document or the test plan. The language in the documents could be ambiguous and could be mis-interpreted by validators developing different models and tools. Secondly, the rapid changes in the specification is common during the design and development process. These changes should be manually propagated into the different models and tools making it an erroneous process. Thirdly, the models in a traditional validation environment are developed in a modeling framework that depends on the target usage of the model. For example, models used to generate simulators are typically written in programming languages like C/C++ and those for test generation are written in specification languages like 'Specman e'. Modeling in a specific programming language makes the modeling process to be highly dependent on the artifacts specific to the underline programming framework. For example, modeling frameworks based on C/C++ are operational in nature. They do not offer formal semantics for refinement of the models and there is no proof of correctness for the defined refinement maps.



(a) Traditional Validation Environment

*



(b) Proposed Metamodeling Based Microprocessor Modeling and Validation Environment

Fig. 1. This figure shows the traditional and the proposed simulation based microprocessor modeling and validation environments. Note that in the traditional environment all the models and tools 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 plug-in's. Further more, the modeling environment ensures that the models in different abstractions are consistent with one another.

III. MICROPROCESSOR MODELING AND VALIDATION ENVIRONMENT (MMV)

The **MMV** environment facilitates microprocessor modeling across various abstractions and uniformly generates code that can be targeted towards various validation platforms as shown in Figure 1(b). The requirements for such an environment are: (i) language-independent modeling syntax and semantics, (ii) a uniform way to capture the various modeling abstractions, (iii) a way to express rules that enforce well-defined model construction in each abstraction level and consistency across abstractions, (iv) a visual editor to facilitate the modeling based on the specified syntax & semantics and to enforce the modeling rules through checkers during design-time and (vi) finally the capability to plug-in code generators/interpreters to allow for model analysis and translation into executables, validation collaterals, test plans, coverage tools, etc. The key enabling capability for **MMV** is *metamodeling*, which is provided through the generic modeling environment (GME). In this paper, we demonstrate the multi-abstraction modeling framework and code generation for one specific validation target.

There are two types of users of MMV namely the modeler and the metamodeler. The metamodeler customizes MMV based on the requirements of the processor modeling and validation domain and also creates code generators for different validation targets. The modeler creates microprocessor models in MMV and uses the code generators to generate validation targets. When dealing with complex microprocessors, scalability of a visual modeling tool is a concern. Therefore MMV allows two modeling modes: (i) Visually based on the microprocessor metamodel (MMM) and (ii) Textually based on the microprocessor description language (MDL). Having a textual input addresses the scalability issue with visual modeling tools and allows the modeler to take advantage of MMV's validation tools and checkers.

A. Metamodeling Framework

A *metamodeling framework* facilitates the description of an application domain by capturing the domain-specific syntax and static semantics into an abstract notation called the *metamodel*. This metamodel is then used to generate a *modeling framework* that the designer can use to construct domain-specific models. Static semantics refer to the well-formedness of syntax in the modeled language, and are specified as invariant conditions that must hold for any model created using the modeling framework. The modeling framework of **MMV** is built on top of GME's metamodeling framework.

B. Generic Modeling Environment (GME)

The Generic Modeling Environment (GME) [3] 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.

The generic concepts describe a system as a graphical, multi-aspect, attributed entity-relationship (MAER) diagram, which internally are UML diagrams. The generic concepts supported are hierarchy, multiple aspects, sets, references and explicit constraints, which are described using the constructs such as `<<Atom>>`, `<<Model>>`, `<<Connection>>`, etc detailed in [3]. *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.

C. Microprocessor Metamodel (MMM)

The domain-specific syntax and some required static semantics for modeling microprocessors in different levels of abstraction is captured as a metamodel in MMV. The metamodel is built in GME using UML constructs and OCL constraints. It depicts four distinct modeling abstraction of a microprocessor and provides a unified language to enable the specification process. The usage of UML to construct the microprocessor modeling syntax and semantic provides language-independence and as well as interoperability through an XML-based intermediate representation/exchange medium. It alleviates the language-specific artifacts such as pointer manipulation, strong typing in C/C++ as well as overcomes the modeling restriction of languages such as VHDL/Verilog with object-orientated features embedded within them.

The abstractions are: (i) System-level view, (ii) Architectural view, (iii) Microcode view and (iv) Microarchitecture view. The modeling language provides a refinement methodology across abstraction levels where the syntactic-level transformation are well-defined and enforced during refinement. Each of these abstractions correspond to a view of the metamodel and is described using the notion of aspects in GME. In the following subsection, we outline the metamodel description for some of these abstractions.

1) *System-level view (SV)*: This abstraction provides a protocol-level description of various microprocessor features, which can be thought of as a software model for the processor component. The modeling framework at this abstraction

provides a high-level algorithmic specification capability. All the entities used to describe the high-level model need not refined to the lower abstraction, since they may not have any significance at another level and are artifacts of the SV. However most of the entities when refined to the next-level, which is the architectural view, are mapped to one or more sequence of instructions.

Figure 2 shows a snapshot of the system-level aspect of the MMM. The topmost entity of the metamodel is a **Platform**, which acts as a container for the different constituents of a processor. The constituents visible at the SV are **Core**, **Main Memory** and entities that describe the communication structure. The **Core** is used to describe the CPU of the processor at the system-level and the **CoreType** attribute is used to inform the framework that an existing core is being modeled.

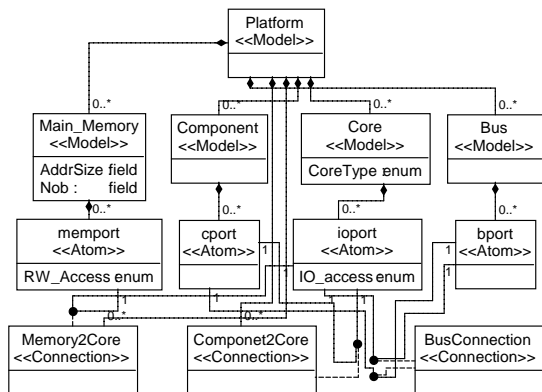


Fig. 2. System-level View of MMM

Another important constituent of the **Platform** is the **Main Memory** entity, which is used to model the memory by capturing information on the address size, and number of bytes/location. In SV, memory is virtual with no physical existence, therefore the attributes are only configurable at lower abstractions. This is enforced as a constraint to restrict the users from mixing the abstractions inconsistently. In SV, we have the notion of function calls, which is modeled through ports, in order to capture the communication between the memory and the core. The core has **ioports** that could be input or output specified through **IO_access** and the memory has **memports** that could be specified as read or write ports using **RW_access**. Point-to-point communications are modeled using these ports. We also allow for bus-based communication, which is enabled through the **BUS** and **busport** entities. The entities **Memory2Core** and **BusConnection** are of type **<<Connection>>** in GME, which captures the communication between the memory and core as well as their communication through a bus by instantiating ports. The metamodel also has an entity which is a place holder for some block that models external interaction with the processor and memory. The block models a functionality at the protocol-level such as arbitration and scheduling and is described using **Component**, **cport** and **Component2Core**. The protocol specification is captured using a variant of hierarchical finite state machines (HFSM).

2) **Architecture view (AV)**: Figure 3 shows a snapshot of the architectural aspect of the microprocessor metamodel.

In AV, the memory abstraction is similar to SV and does not undergo a refinement but uses **mem_arch_port** to communication with the core. The core undergoes a refinement to include register definitions that is described using the **RegisterFile** entity with a **Size** attribute, which is shown in Figure 4. This attribute is used in the later abstractions to map the registers to the physical address. The **<<connection>>**s in this abstractions are not shown due to lack of space.

The **registerFile** allows for three type of registers namely (i) general purpose register (**GPR**), (ii) segment register (**SR**)

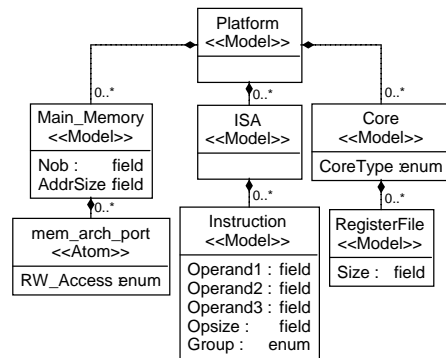


Fig. 3. Architectural View of MMM

and (iii) status & control register (**SCR**). Furthermore, to capture the part-whole relationship between registers we have the notion of references to registers.

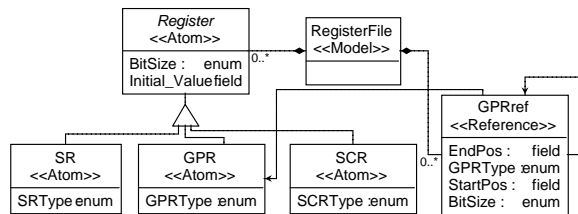


Fig. 4. RegisterFile Entity

A constituent of the **Platform** not seen in SV is the **ISA** entity, which is only relevant at the architectural abstraction and provides the language to describe the instruction set.

Another refinement in this AV is the definition of the instruction set architecture by using the **ISA** entity. The **ISA** provides an instruction-level abstraction of a processor and is modeled using the **Instruction** entity. The attributes **Operand1**, **Operand2**, **Operand3** and **Opsize** are used to characterize the instruction by describing the operands and their size. The **group** attribute has a set of categories that a modeled instruction belongs to, which allows the framework to enforce constraints and flag an illegal instruction. For example, one of the category is shift and rotate and a constraint enforced on instructions in 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.

The **Instruction** entity of the MMM provides the modeler with a language to describe the instruction behavior, which has constructs to write functions, statements and define identifiers.

3) **Microarchitecture View (MV)**: The virtual memory is made concrete by defining the memory map using the **Placement** entity providing the physical address. The core on the other hand undergoes a refinement, where the registers defined in AV is bound using a map function. In the MV, we provide a **RegisterMap** entity to map the defined registers to addresses corresponding to the **RegisterFile**. During the refinement of the Core, invalid register map functions need to be avoided, since the registers are defined in a different abstraction, this inconsistency is common. Therefore, the refinements are enforced as constraints and the modeler is required to conform to these to arrive at a consistent description of the microprocessor across the abstractions. Furthermore, a new constituent is inserted as a refinement, which is the **Pipeline** entity. A Pipeline acts a collection of **Stages** and instruction registers (**IRegisters**).

Pipeline communicates with the memory to read & write data and address, however this communication should not be

visible at SV. The **Main Memory** should not grant read/write access to the pipeline through the **mempport**, since it is meant exclusively for system-level communication with the processor core. As a result, the **Main Memory** entity has a new set of ports namely **mem_march_ports**, which renders this communication possible only at MV. **Pipeline** also communicates with its internal stages acting as an interface to read/write the memory and provide it to/from the different stages. The pipeline registers are modeled using the **IRegister** entity. The functionality of the stages of a pipeline vary widely, but can be described using some basic building blocks such as MUX, ALU and adders. The MV is the most complex level in MMV and varies distinctly across processors based on whether the execution is speculative, out-of-order, etc. Therefore, we do not provide a framework with the such elaborate modeling capability. However, one of the foremost advantage of a metamodeling-driven modeling framework such as MMV is the ease of extension. As a result, MMV's microarchitecture view can be quickly extended to in-cooperate memory arrays and state machines to model re-order buffers and register tables.

D. Code Generation

During modeling, the metamodel entities are instantiated, characterized and connected to describe the microprocessor model. These entity instances and connections are collected and populated into a model database. GME provides appropriate API calls to interface with the database, which facilitates easy access to model elements and a way to perform analysis. In MMV, model interpretation is carried out and the result is a translation of the model into some target that allows microprocessor validation. Some of the common targets are functional simulators, test generator, coverage tools etc. During translation the model is analyzed in its current abstraction as well as across abstractions and the elements are extracted into homogeneous sets. These sets are treated uniformly during target-specific code generation. Therefore, our translation has two stages: (i) Extraction process and (ii) Target-specific code generation. In Section IV, we discuss how the models are converted into Constraint Satisfaction Problems for test generation.

Note that the extraction process blindly follows the containment hierarchy and aspect partitioning in MMM and populates the respective structure. These structures are not shared between abstractions and this lets the code generator to be oblivious to the abstraction level. The extraction is detailed in [4]. The TICG is a templated function that is an integral part of the code generation and is very target-specific. It performs translation based on the instance passed to it as an argument. We do not describe its implementation specific details in this paper due to lack of space.

As a part of code generation, we translate the microprocessor models into an intermediate XML representation (XML-IR), which has a well-defined schema. The XML-IR is detailed in [4]. The XML-IR functions as a middleware between the visual modeling framework and the textual interface provided through MDL. MDL has a one-to-one correspondence with XML-IR and a simple Perl parser provides the conversion. Similarly, the XML-IR is converted using a non-trivial XML parser into XME that MMV can visualize.

IV. CONSTRAINTS FOR TEST GENERATION

MMV facilitates automatic test generation as one of its validation targets. A test generator takes the processor behavior as input and automatically formulates valid test cases. 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. For this target, the TICG converts the description into a program flow graph on which it performs static single assignment (SSA) analysis and writes out print-streams that translate the program flow graph into constraint satisfaction problems (CSP). The CSP is given as input to a test generator that solves these constraints and generates test cases.

Definition 4.1: Constraint Satisfaction Problem (CSP) [5] Let $X = \{x_1, x_2, \dots, x_n\}$ be a set of variables. Each x_i is associated with a finite set of possible values from D_i the domain of x_i . A constraint C on x_1, x_2, \dots, x_n is a subset of $D_1 \times \dots \times D_n$. A CSP is formally defined a set of constraints restricting the values that the variables in X can simultaneously.

A solution to a CSP is the assignment of a value to every variable x_i in X from its domain D_i , such that all constraints are satisfied. Consider the behavior of a hypothetical instruction *foo* and its corresponding constraints shown in Figure 5 as an illustration for CSP generation in MMV. Here a and b are the inputs and c is the output. Let the variables a_0, b_0, c_0 represent the values of a, b and c before the execution of *foo*. The output c can get a new value during the execution of the instruction that is captured by introducing a new variable c_1 . The constraint states that if the condition $(a_0 > 1 \ \&\& \ b_0 < 2)$ is true then c_1 is set to 2 else c_1 is set to c_0 .

Behavior: foo() { if($a > 1 \ \&\& \ b < 2$) $c = 2$ }	Constraints: if ($a_0 > 1 \ \&\& \ b_0 < 2$) $c_1 = 2$ if (!($a_0 > 1 \ \&\& \ b_0 < 2$)) $c_1 = c_0$
Two possible test cases:	
1. $a_0 = 2, b_0 = 1, c_0 = 0, c_1 = 2$	
2. $a_0 = 1, b_0 = 3, c_0 = 0, c_1 = 0$	

Fig. 5. Behavior & Constraints corresponding to instruction *foo* and test case generation

The test generator will solve the CSP and generate values for a_0, b_0, c_0 and c_1 such that the two constraints are satisfied. Two example test cases generated by the test generator is given in the Figure 5. In the first example, $a_0 = 2$ and $b_0 = 1$. These values satisfy the first constraint and hence c_1 is set to 2. c_0 can take any value from its domain. In this example it takes the value 0. In the second example, the values assigned for a_0 and b_0 satisfies the second constraint. This implies that c is not modified by the instruction *foo*. Hence c_1 , the variable that represents c after the execution of *foo*, should take the value of c_0 , the value of c before the execution of *foo*. In this example, c_0 takes the value 0, and hence c_1 also assigned 0.

V. CASE STUDY: MODELING VESPA IN MMV

We illustrate the modeling of Vespa [6], a 32-bit RISC processor using MMV at the system, architecture and microarchitecture abstractions.

A. System-level Modeling

Vespa System Model: We model the interaction of the real-time software scheduler, processor and timer of Vespa at the system-level, which is shown in Figure 6. The state machine description of each component is shown in figure 7.

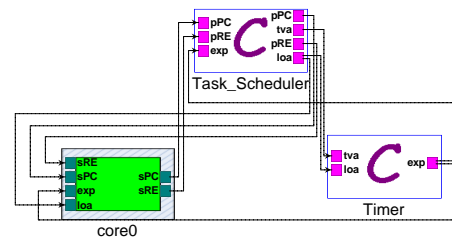


Fig. 6. System-level model of Vespa

TIMER: It is modeled as a down counter with three states namely INIT, DECREMENT and EXPIRED. The timer has an internal variable *tcnt* that holds the counter value. It is modeled as a **Variable**, which is initialized by an input in the INIT state. The timer has three inputs: (i) timer's initial value *tval*, (ii) load flag and (iii) clock. It also has an output called *expiry* that is used to say when the timer has reached zero. These are modeled as **ioports**. The timer's internal *tcnt* is initialized with *tval* when the *loadtimer* input is activated. Then the timer moves into DECREMENT, where in the subsequent clock cycles the *tcnt* becomes *tcnt* - 1. When *tcnt* becomes zero, the timer transitions to EXPIRED and the output *expiry* is set.

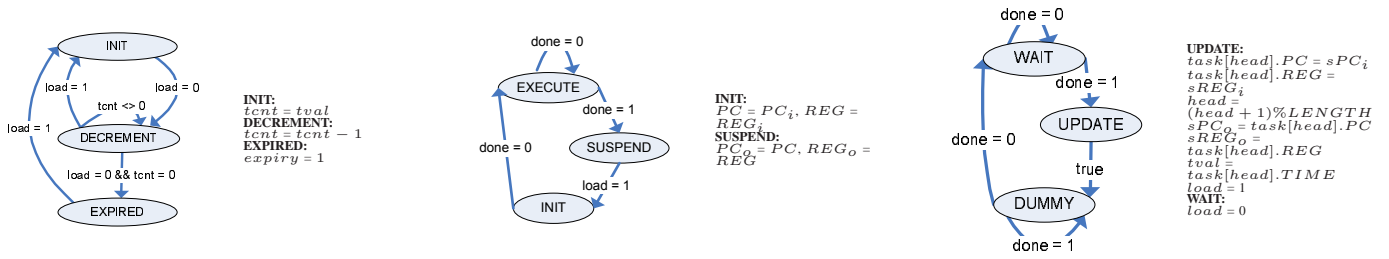


Fig. 7. Behavior of the timer, processor and scheduler

In this state if the *load* is activated, the timer will de-assert *expiry* and jump to INIT.

PROCESSOR: It is modeled as an FSM with three states INIT, EXECUTE and SUSPEND. When the processor receives the *done* input, it goes to the SUSPEND state and sends the status of the executing task to its output ports pPC_o and $pREG_o$. On receiving the *load* input, the processor will transition to the INIT state and initialize PC and REG with the values in the input ports PC_i and REG_i and then transitions to the EXECUTE state to begin executing the new task.

SCHEDULER: The scheduler controls which task will execute in the processor core at any given time. We model a simple static scheduling algorithm in the scheduler [7]. All the tasks are assumed to be periodic and the time taken to execute any task is assumed to be constant and deterministic. The scheduler maintains the details of all the tasks in a task queue *task* modeled as a **Structure**. When the scheduler receives the timer *done* input (*expiry*), it wakes up and saves the state of the suspended task in its task queue. It increments the head pointer to point to the next task in the queue and sends the PC and register state of the next task to its output ports sPC_o and $sREG_o$. It sets *tval* with the period of the task and asserts the *load* signal.

The state machine representation at the system level hides the actual implementation of the different components. In the subsequent levels, these state machines will be manually refined to include more details. In the architecture level, the model of the processor core will be refined to include the details in the ISA and the scheduler will be implemented using the assembly language of Vespa. In the microarchitecture level, the timer will be refined as a hardware block that interacts with the pipeline of the processor using interrupts. In Section V-B and V-C we show how the processor is refined in the architecture and microarchitecture abstractions. Due to lack of space we do not show how the scheduler and the timer is refined in the subsequent abstraction levels.

Constraints for System Model: The CSP for the scheduler in the system model is shown in figure 8. The code generator in MMV generates an *if* constraint for every transition in the FSM. The *if* constraint will guard the action performed in the state which the system will transition to. The CSP represents the behavior of the HFSM for one time step. Variables subscripted by p and n stores the state before and after the time step, respectively. CSP's for multiple time steps can be generated by generating one CSP for every time step and chaining the variables. The variables representing the *next* state of one CSP will be used as the *previous* state of the next CSP.

Scheduler:
 $if(expiry_p == 1)$
 $task[head_p].PC = sPC_{i_p} \ \&\& \ task[head_p].REG = sReg_{i_p}$
 $head_n = (head_p + 1) \% LENGTH$
 $sPC_{o_n} = task[head_n].PC$
 $sREG_{o_n} = task[head_n].REG$
 $tval_n = task[head_n].TIME$
 $load_n = 1$

Fig. 8. Constraints generated for the system model of the processor

B. Architecture Model

Vespa Architecture Model: The architecture of Vespa consists of 32 General Purpose Registers (GPRs), four flag bits namely, the carry bit (C), the overflow bit (V), the sign bit (S) and the zero bit (Z) and a set of 30 RISC Instructions [6]. All instructions defined in the architecture of Vespa were modeled in MMV. Due to lack of space, we only show the behavior of *add* instruction, its model in MMV and the CSP generated by the code generator.

The pseudo code of the behavior of *add* instruction is shown in figure 9. The source operands and destination operands of *add* are *src1*, *src2* and *dest* respectively. The first statement in the pseudo code performs the *add* operation and it is followed by statements that calculate the value for the flag bits based on the result of the addition.

```

1. dest = src1 + src2
2. carryflag = (((src1 & src2) |
                 (src1 & !dest) |
                 (src2 & !dest)) &
                2^BitSize-1) >> BitSize-1)
3. zeroflag = (dest == 0)
4. signflag = dest >> BitSize-1
5. overflowflag = (((src1 & !src2 & !dest) |
                   (!src1 & src2 & dest)) &
                   2^BitSize-1) >> BitSize-1)

```

Fig. 9. Pseudo code for the behavior of *add* instruction

Constraints for ADD Instruction: The CSP generated by MMV for *add* instruction is shown in figure 10. The generated CSP is a direct translation of the behavior *add* instruction. Variables with subscript p and n holds the value of the GPR's and the flag bits before and after the execution of *add*. Complex instructions have conditional statements in their behavior. The code generator will generate an *if* constraint for every conditional statement in instructions behavior. CSP's for multiple instructions can be generated by assigning the output variables of one instruction as the input variable to the next instruction in sequence.

```

dest_n = src1_n + src2_n
c_n = (((src1_n & src2_n) |
        (src1_n & !dest_n) |
        (src2_n & !dest_n)) & 2^31) >> 31)
z_n = (dest_n == 0)
s_n = dest_n >> 31
v_n = (((src1_n & !src2_n & !dest_n) |
        (!src1_n & src2_n & dest_n)) & 2^31) >> 31)

```

Fig. 10. Constraints generated for ADD in MMV

C. Microarchitecture Model

Vespa Microarchitecture Model: Vespa is modeled as a scalar, single-issue, 5-stage pipelined processor in the microarchitecture level of abstraction [6]. The microarchitecture consists of five pipeline stages, pipeline registers, data forwarding logic and ports to access memory. The ports are visible only at this level of abstraction. The register file in the microarchitecture abstraction refers to the register set defined in the architecture abstraction. All registers defined in the architecture abstraction should be mapped to a physical address in the register file at the microarchitecture abstraction. This

OCL constraint was added in MMV to maintain consistency across the abstraction levels. Due to lack of space we only show the model of IF stage and its CSP generated by the code generator. The behavior of the other stages of Vespa can be found in [6].

Figure 11 shows the internals of the IF pipe stage. IF stage communicates with the ID stage by updating IR2 and PC2 registers respectively. The address of the current instruction is stored in PC. During normal operation, the IF stage fetches the current instruction from the memory via the port *Addr* and *Data* using the address stored in PC. It then increments the PC by 4 to point to the next instruction in sequence. The multiplexers IRMUX, PC2MUX and PCMUX control the value that updates the IR2, PC2 and PC registers respectively. The control signals for the multiplexers are generated by a hazard detection and control logic in the processor. IR2 can either be updated with the instruction fetched from the memory, a NOP or the recirculated value of the current IR2 in case of a pipeline stall. Similarly PC2 can either be updated with PC + 4 or the current value of PC2 depending on whether the pipeline is stalled or not. PC can be updated with PC + 4, the current value of PC or the branch target address in port *z4*. The value in *z4* is calculated in the EX stage of the processor.

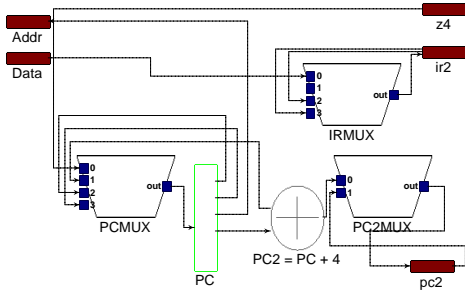


Fig. 11. The model of the Instruction Fetch State of Vespa Processor

Constraints for IF Stage: The CSP is generated for one time frame and CSP's for multiple time frames are generated similar to the previous two abstractions. The CSP of the IF stage is shown in figure 12. The code generator formulates an *if* constraint for every multiplexer in the model. It represents the control input to the multiplexers using an integer variable. *ir2mux*, *pc2mux* and *pcmux* are integer variables that controls the operation of the IR2MUX, PC2MUX and PCMUX, respectively.

$$\begin{array}{ll}
 \text{if}(ir2mux_p == 0) & \text{if}(pcmux_p == 0) pc_n = z4_n \\
 ir2_n = Mem[pc_n] & \text{if}(pcmux_p == 1) \\
 \text{if}(ir2mux_p == 1) ir2_n = NOP & pc_n = pc_n + 4 \\
 \text{if}(ir2mux_p > 1) ir2_n = ir2_n & \text{if}(pcmux_p > 1) pc_n = pc_n \\
 \text{if}(pc2mux_p == 0) pc2_n = pc_n + 4 & \text{if}(pc2mux_p > 0) pc2_n = pc2_n
 \end{array}$$

Fig. 12. Constraints Generated for IF stage

VI. RELATED WORK

Different simulation based modeling and validation environment were proposed in the literature. They fall under the category of Architecture Description Languages (ADLs) and GME based environments. ADL's [8]–[14] are proposed to model the architecture and pipeline structure of the processor. ADL's are narrow spectrum languages. They are designed for a specific target application such as generating an ISA simulator, cycle accurate microarchitecture simulator and instruction descriptions for the back-end of the compiler. Since ADL's are developed for a specific application, it would be difficult to re-target the model developed in an ADL to another application. For example, depending on the ADL, there might not be a neat and consistent way to convert a model written for generating a simulator into constraints that could be used for test generation. Also, these languages do not distinguish between different abstraction levels. This makes it difficult

to verify if two models defined in different abstraction levels are consistent with one another. Bakshi et al [15] proposed MILAN, a framework built using GME, for designing microprocessors at the RTL level of abstraction. MILAN is a modeling tool, while MMV is used for microprocessor validation. Furthermore, MILAN does not facilitate modeling the microprocessor in different abstraction levels. Bin et al [16] proposed a random test generation framework for system, architecture and microarchitecture levels of abstraction. The paper does not clearly mention the modeling environment used to create the different models.

VII. CONCLUSION

In this paper, we proposed a microprocessor validation environment based on a metamodeling framework. The environment supports the creation of models at various abstraction levels, and keeping models at neighboring levels of abstraction consistent. This is an important feature for validation, along with the requirement that this consistent set of models be utilized for generating important validation collaterals.

We demonstrated the concept by modeling a 32 bit RISC processor, Vespa, at the system, instruction set architecture and microarchitecture levels. We showed how consistency across levels is enforced during modeling and gave an example of generating constraints from the model for automatic test generation.

In the future, we intend to extend the environment for creating other validation collaterals, viz., software simulation, hardware synthesis for emulation, and coverage analysis.

REFERENCES

- [1] 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.
- [2] B. Bentley, "Validating the Intel Pentium 4 Microprocessor," in *DAC '01: Proceedings of the 38th conference on Design automation*.
- [3] A. Ledeczi, M. Maroti, A. Bakay, and G. Karsai, "The Generic Modeling Environment," Vanderbilt University. Institute for Software Integrated Systems Nashville, 2001.
- [4] A. Dingankar, D. Mathaikutty, S. Kodakara, S. Shukla, and D. Lilja, "MMV: Metamodeling Based Microprocessor Validation Environment," *FERMAT Technical Report*, no. 2006-13, 2006.
- [5] K. Marriott and P. J. Stuckey, "Programming with Constraints: An Introduction." The MIT Press, 1998.
- [6] D. J. Lilja and S. S. Sapatnekar, "Designing Digital Computer Systems with Verilog." Cambridge University Press, 2005.
- [7] C. Fridge, "Real-Time Scheduling Theory," in *Software Verification Research Center, The University of Queensland, Tech. Report*, no. 02-19, 2002.
- [8] A. Halambi, P. Grun, V. Ganesh, A. Khare, N. Dutt, and A. Nicolau, "Expression: A language for architecture exploration through compiler/simulator retargetability," *DATE*, vol. 00, p. 485, 1999.
- [9] M. Freericks, "The nML machine description formalism," in *Fachbereich Informatik, Tech. Univ. Berlin, Berlin, Germany, Tech. Rep.*, 1991, pp. 299–302.
- [10] A. Ayari, D. Basin, and A. Podolski, "LISA: A specification language based on WS2S," in *11th International Conference of the European Association for Computer Science Logic (CSL '97)*, M. Neilsen and W. Thomas, Eds., no. 1414. Springer-Verlag, 1997, pp. 18–34. [Online]. Available: citeseer.ist.psu.edu/article/ayari98lisa.html
- [11] G. Hadjiyiannis, S. Hanono, and S. Devadas, "ISDL: An instruction set description language for retargetability," in *Design Automation Conference*, 1997, pp. 299–302. [Online]. Available: citeseer.ist.psu.edu/hadjiyiannis97isdl.html
- [12] R. Leupers and P. Marwedel, "Retargetable code generation based on structural processor descriptions," 1998. [Online]. Available: citeseer.ist.psu.edu/leupers98retargetable.html
- [13] J. Gyllenhaal, "A machine description language for compilation," 1994. [Online]. Available: citeseer.ist.psu.edu/gyllenhaal94machine.html
- [14] F. S.-H. Chang and A. J. Hu, "Fast specification of cycle-accurate processor models," *iccd*, vol. 00, p. 0488, 2001.
- [15] A. Bakshi, V. Mathur, S. Mohanty, V. K. Prasanna, C. S. Raghavendra, M. Singh, A. Agrawal, J. Davis, B. Eames, A. Ledeczi, S. Neema, and G. Nordstrom, "MILAN: A model based integrated simulation framework for design of embedded systems," *ACM SIGPLAN Notices*, vol. 36, no. 8, pp. 82–87, 2001.
- [16] 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.