MVP: A MUTATION-BASED VALIDATION PARADIGM

Jorge Campos and Hussain Al-Asaad
Department of Electrical and Computer Engineering
University of California, Davis, CA
E-mail: {jcampos, halasaad} @ece.ucdavis.edu

Abstract*—A mutation-based validation paradigm that can handle complete high-level microprocessor implementations is presented. First, a control-based coverage measure is presented that is aimed at exposing design errors that incorrectly set control signal values. A method of automatically generating a complete set of modeled errors from this coverage metric is presented such that the instantiated modeled errors harness the rules of cause-and-effect that define mutation-based error models. Finally, we introduce a new automatic test pattern generation technique for high-level hardware descriptions that solves multiple concurrent constraints and is empowered by concurrent programming.*

1 INTRODUCTION

The task of creating a validation paradigm is an open-ended problem. An enormous amount of research has already been performed to study the sub-topics that are incorporated into a validation system including coverage metrics, design error modeling, finite state machine (FSM) analysis, equivalence checking at various layers of abstraction, and various automatic test pattern generation (ATPG) techniques. Validation systems for high-level microprocessor implementations also tend to borrow techniques from compiler technologies and software testing; these techniques include topics such as data-dependency graphs, assertion-based software testing, and mutation-based software testing. All these above mentioned techniques and others act as pieces that can be used to create ground-breaking validation methodologies, but significant effort has not been made to conceptualize a novel method of integrating these pieces to produce a validation paradigm that can impact a wide range of validation-based problems.

It is the goal of our research to create such a versatile validation paradigm by adopting mutation-based testing as a means to translate design errors from an abstract point-of-view into a physical one. This translation can be done by first identifying how such abstract design errors affect the system under validation, then producing design error models that follow the rules of cause-and-effect similar to that of physical fault models. This approach is advantageous because a collection of modeled errors can be efficiently simulated on a hardware description concurrently by incorporating high-level fault-list propagation into the simulation system. Another significant advantage, if we visualize an undetected modeled error's location to be its activation criteria (activation site), is that each test sequence can be aimed at the corner of the design space with the greatest density of undetected modeled errors.

The basis for this mutation-based approach comes from an adaptation of the Coupling Effect [4] in software testing, which allows us to argue that a test sequence capable of exposing simple design errors is also implicitly capable of exposing hard-to-detect design errors [2]. To completely develop this mutation-based validation paradigm (MVP), it is necessary to describe the steps of the validation process in the following order: (*i*) coverage measure selection, (*ii*) design error modeling, (*iii*) instantiating a set of modeled errors from each design error model, and (*iv*) concurrent modeled error simulation/automatic test pattern generation.

We next introduce various simulation-based validation environments that do not represent the complete set of existing environments, but introduce some good examples.

SymFony [12]. Many common validation environments that employ symbolic automatic test pattern generation methods rely on a gate-level implementation that has been previously synthesized. The drawback of this approach is that it forces the symbolic methods to be applied onto the complete implementation at the logic-gate level. This is a drawback because symbolic methods are only capable of generating a solution to circuits that contain few registers.

SymFony attempts to circumvent this limitation by extracting circuit macros from the gate-level implementation, which act as reduced problems for the symbolic solver. It employs two main algorithms: *Forbidden* is used to identify all reachable states, and *Justify* is used to generate the implications required by the FAN solver. To improve the run-time performance, a pre-processing phase is used to identify (from the synthesized gate-level implementation) *F macros* that will be used by *Forbidden* and *J macros* that will be used by *Justify.* At the register-transfer level, SymFony consider control macros that are composed of each FSM state register, and data-path macros that are composed of data registers/data-manipulating combinational logic.

SymFony's ATPG process can be applied to only small and medium circuits because the *Forbidden* pre-processor computes the FSM for the complete circuit, and the input/output constraint BDDs (derived from the logic BDDs) are intersected with the complete FSM BDDs (FSM next-state and FSM output BDDs) during each symbolic solver process. The *Forbidden* algorithm prunes out the invalid state configurations as a preprocessor step, but an effort is not made to prune the state space prior to performing the expensive operations that intersect BDDs.

Genesys-Pro [13]**.** Unlike many other verification tools, the Genesys-Pro verification tool is capable of performing verification on complete processor systems by implementing model-based test pattern generation. The model-based approach provides the building blocks found in processor implementations to simplify the effort of creating a processor model. A processor model is composed of a declarative description, and testing knowledge for this model. A processor model is verified through the use of a test template language, such that a test template describes architecture-level characteristics that should be tested. This test template is converted into a verification program via the model-based test pattern generator (implemented by a pseudorandom test pattern generator) that uses the model's included testing knowledge. The strength of this technique is that it allows a validation engineer to create test templates that are not burdened by implementation details. This, of course, requires significant human foresight when creating the testing knowledge included with the processor model itself. Applying human effort into generating this testing knowledge is only advantageous when there is extra manpower or when the model belongs to a family of processors with long lifetime expectancy. Furthermore, it cannot be guaranteed that the modeling engineer has not left out important corner cases that are difficult to stimulate.

μGP [14]. Some validation environments employ an instruction library that contains a collection of mini-programs (known as program macros) that are capable of exercising interesting corners of

the processor. These program macros can be combined in various sequences to achieve test programs that are more effective than purely random test generation methods. This approach requires that the simulation method connect the processor implementation onto a simulated memory unit that contains the test program, as opposed to forcing a fixed test sequence into the processor's primary inputs.

The methods in μGP employ an instruction library, and achieve effective test programs via a genetic algorithm. The program macros used in μGP are fine-grained such that it does not render a sequence of program macros incapable of stimulating interesting interactions among the instructions in a pipeline. The goal of μGP is therefore to use a genetic algorithm to generate a test program (composed of a sequence of program macros) that achieves high design coverage. μGP implement the genetic algorithm as follows: μGP first generate μ individual test programs by combining program macros in a random order—this set of test programs act as the first generation. Then for each generation, λ new individual test programs are generated by pairing the existing test programs and generating their offspring through mutation and crossover operations. The next generation is then defined by rating the set of μ + λ test programs with a fitness function that is based on the coverage metric, and eliminating all but the highest-rating μ test programs.

The quality of the test program generated by the validation system can only be as good as the set of program macros that define it. It is therefore dependent on the validation engineer to develop a diverse enough set of program macros such that a combination of program macros exists for every corner of the design.

In our previous work [1], we have developed a method to simulate a collection of modeled errors concurrently on a high-level hardware description. This method is implemented by giving signals and variables a fault list, such that mutant values from modeled errors are injected into corresponding fault lists upon activation, and removed from the fault list once they have propagated to an observation point.

We have introduced our "clustering and partitioning" technique as a data structure that minimizes the simulation overhead caused by error injection [2]. Clustering and partitioning balances the trade-off between signal propagation overhead and modeled error activation overhead by clustering modeled errors that have common signals in their activation criteria, and partitioning these clusters based on the value for the signals in the activation criteria. Each cluster acts as another target for fault list propagation, and is partitioned using hashing-and-chaining. As a result, this data structure allows the simulation environment to only consider the partitions that correspond to the current processor state during modeled error activation.

We have also proposed a method to identify the most effective ATPG goals based on the statistical information on the distribution of modeled errors that is obtained from the clustering and partitioning data structure [2]. Observations on our simulation results have made it clear that there is a strong correlation between the number of active modeled errors per simulation iteration and the number of detected modeled errors per simulation iteration. This observation allows us to argue that it is fruitful to focus our ATPG efforts on activating the largest number of modeled errors per ATPG iteration (deterministic activation and probabilistic propagation) for as long as there are sufficient modeled errors that are easy to detect. Once we reach to a state where there are not enough modeled errors that are easy to be detected, our ATPG effort switches to deterministic activation and propagation.

2 Coverage Measure Selection

An exhaustive coverage metric would attempt to apply every possible input vector onto every state of the design which often leads to the state explosion problem. Other FSM-based coverage metrics include state coverage, transition coverage, and path-coverage [15][16]. The path coverage metric encapsulates transition coverage, and can potentially be more complex than the exhaustive coverage metric. It attempts to exercise every possible state sequence that is within a given length.

With the use of hardware description languages (HDLs) came a series of code-based coverage metrics (synonymous to FSM coverage metrics) for circuit designs including line coverage, transition coverage, and path coverage [5][15].

A validation paradigm should employ a good coverage metric because an inefficient coverage metric will require too many test pattern generation iterations, and an over-simplified coverage metric will sacrifice the effectiveness of the resulting test sequence. A control-based coverage measure is being employed for our mutation-based validation paradigm, and the reasoning behind it is as follows. A microprocessor's explicit processor state can be defined by combining all the control registers. A simple 16-bit processor with a 5-stage pipeline would therefore consist of a state register that is at least 64 bits wide (each of the last four pipeline stages contains a control register that holds the currently residing instruction). The state register would be even larger for superscalar microprocessor implementations because they employ a distributed control methodology through many disjoint and cooperating functional and control units. Any attempt to even perform a complete state coverage would face the wrath of the state explosion problem, so a more effective method must be employed. Given that modern superscalar microprocessor implementations are modular by nature, it is reasonable to request for each of these functional and control units to be validated against their description before the microprocessor is validated as a whole. With this request, the microprocessor-wide validation problem is reduced to one of validating the control signals that merge these units together. A study on bug occurrences in pipelined and superscalar microprocessor implementations [9] shows that over two-thirds of design errors are related to the control logic.

At this point, it is obvious that the control-based coverage metric becomes one which ensures that every possible value for each of the control signals is exercised for every possible processor state. This may seem like an even harder coverage metric to employ than the state coverage metric. However, we can actually reduce the state space by ignoring those processor states which do not assert control signal values that cannot be asserted by other processor states. The general implementation technique for this coverage metric is discussed in the next section.

3 Design Error Modeling

A design error results in the generation of an erroneous value under a specific state of the system; therefore we can harness this cause-and-effect characteristic to define the design error construct. Similar to some fault injection techniques [7][8], the description of a design error model is based on three basic characteristics: (*i*) the activation criteria, (*ii*) the consequence of activation, and (*iii*) error injection. Multiple design errors are to be simulated concurrently; therefore each design error must have a unique identification number.

A design error's activation criteria specify a set of signals and the conditions that they must satisfy before the design error is activated. Once the activation criteria is met, the code segment particular to a design error is executed and generates a set of mutations for a corresponding set of injection points. A mutant value is injected into the specified signal immediately after it is generated.

Given that a design error on an implementation of a modular component will appear on every instantiation of that component, all design error models have to obey the hierarchical error model [6] where every instantiation of a modular component will have the same set of design errors with corresponding identification numbers. This is important because it allows a design error to simulta-

neously appear at multiple instantiations of a component if necessary, and it correctly models aliasing in the case where these mutant values mask each others' propagation across the circuit.

It is possible to automate the generation of modeled design errors such that they span the control-based coverage measure of Section 2. In fact, performing automated generation of these modeled errors is expected to be most influential for superscalar processor implementations because of their inherent complexity. If one were to analyze the data dependency of a control signal onto the set of registers (data and control) and primary inputs, one would see that each control signal is dependent on only a subset of those sources. It is therefore possible to prune the state space without consequences as follows: For each control signal *c*, first identify the set of control registers that affect the value of that control signal and denote this set as *s*. Then for every correct value *vc* of control signal *c* under every possible value of *s*, generate a set of modeled errors that mutates *c* from *vc* to all erroneous values *ve* (such that $vc \neq ve$) and inject their corresponding *ve* back into *c*.

This error modeling technique follows closely from the modified MCE model [2], but it is more effective because it only generates useful modeled errors for every control signal by first identifying the relevant control registers. The set of relevant control registers for a particular control signal can be found easily by analyzing that control signal's set of prospect states. A prospect state is introduced and defined in Section 6 and can be generated as discussed there.

4 HIGH-LEVEL ATPG

Once a set of modeled errors has been generated as specified in the previous section, they are simulated on the hardware description under validation to rate the effectiveness of the test sequence. In our previous work, we have developed an efficient method of simulating modeled errors on a high-level hardware description [1], and we have proposed a method of identifying the ATPG goals that result in the most effective test sequence per ATPG iteration. This combination of simulation efficiency and ATPG effectiveness allows us to employ a deterministic ATPG algorithm as opposed to the pseudo-random algorithms that are widely used. The efficiency of our ATPG algorithm is demonstrated in Section 10.

Many researchers have attempted innovative methods for automatically identifying the register(s) on which a microprocessor's FSM is based [5]. The motive is that the test pattern generation efforts are greatly simplified when a processor's complete FSM has been predetermined. Once an FSM is identified, a test sequence that maps the current processor state to a target state can be generated by identifying the corresponding transition sequence. The work in [5] identifies an FSM by searching for the control signals that change most frequently and denoting these signals as control states. It also acknowledge that other control signals also play an influence on the FSM, and so denote these signals as state associative control signals. This approach, however, was developed for processors with micro-architectural implementations which inherently rely on an explicit FSM. For superscalar implementations, it is advantageous to conceptualize a microprocessor's control functionality to be implemented by numerous interacting FSMs.

The current proposed method for generating a set of predefined FSMs involves having the user provide basic architectural information. First of all, the user would identify the dominant control registers that should be considered for ATPG by attaching an attribute to the corresponding signals in the hardware description; FSMs will only be generated for these control registers. User-configured control registers are an example of control registers that do not require pre-defined FSMs because they behave like data registers. The user would also specify the range for each of the listed control registers so it may be accessed at runtime; this step is believed to be a reasonable request given that microprocessor implementations are usually accompanied by useful documentation. The range for these control registers can be specified by partially-defined vectors [5]. An *n*-bit register that is missing explicit range information will have to be assigned an initial exhaustive 2^n-valued range. As the FSM for the register is identified, the range is pruned out. Any control register that holds the op-code serves as an example of a control register that requires a pre-specified range—the range is the complete set of possible instructions.

The process of identifying the FSM for a particular control register involves first identifying its relationship to primary inputs and internal registers, such that this information is stored through a set of data dependency graphs (DDGs). These DDGs are generated via the methods discussed in Section 5. For the purpose of our validation environment (MVP), the FSM input is identified using the DDG and can be any primary input or control register. There usually are multiple assignment statements for each control signal, such that each assignment statement is activated by a given set of control requirements; therefore each DDG is naturally accompanied by a corresponding set of control requirements.

An FSM graph for an *n*-bit register has a total number of transitions equal to 2^{2n}. Fortunately, the allowable set of values for a control register in a microprocessor implementation is usually a subset of the complete range of values that it can hold. We can therefore use a predefined range for a control register to reduce the size of its FSM graph.

The FSM can be identified for a given control register by applying simple ATPG methods for every acceptable combination of FSM state and output; this technique can employ any logic-gate-level ATPG algorithm such that the FSM state is predefined and the goal is to find any acceptable FSM input value. Given that the process of identifying a set of FSMs that exist in the microprocessor implementation uses the same techniques as the run-time ATPG process, it should become clear that the goal for this preprocessing step is to invest effort to obtain the solutions to the reoccurring ATPG problems for long-term benefit. Designating too many control registers with attributes will overburden the preprocessor, but designating too few control registers will overburden the runtime ATPG process with redundant problems and backtracking. Performing case studies with the finalized validation environment should result in interesting data that hopefully gives insight as to what combination of control registers can achieve a balance for optimal performance.

At this point, it is clear that our MVP relies on giving modeled errors the capability to specify multiple concurrent constraints. Each constraint will therefore specify one or more paths in the HDL code that serve as candidates for test pattern generation (not all possible combinations of paths for a modeled error can be satisfied due to conflicting control requirements in the set of constraints). In fact, it is expected that when automatic generation of modeled errors is used, many of the modeled errors will contain an activation criteria that cannot be asserted. Therefore it is necessary to identify the combinations of code paths that do not lead to a solvable problem as a means to prune the search space and to identify the modeled errors that cannot be activated.

It is expected that the microprocessor implementations under consideration will be hierarchical in nature, therefore forcing an error model specification that resides in an embedded module to be implemented at every instantiation of that module. To complicate the situation even further, an error model specification might include activation criteria that span multiple module instantiations, therefore requiring a global technique of gathering the control requirements for each activation criterion (constraint). It is possible to collapse hierarchical implementations of hardware descriptions into a collection of interconnected processes [17], and in fact many simulation environments rely on this to implement a distributed simulation system [10]. Analyzing a flattened implementation for

test pattern generation is appealing because it can be solved by analyzing a single layer of scope. However, analyzing the implementation in its hierarchical format is beneficial because a multi-threaded solution can be used in a depth-first fashion to efficiently identify the combinations of code paths for a collection of constraints that can be possibly satisfied.

The researchers in [11] have attempted to satisfy a set of constraints for large high-level circuits by dividing it into smaller, independent, cascaded problems. Each time the hybrid solver is not able to solve one of the individual components; their software tool will divide that component into two cascaded components and attempt to solve the end-most component first. This transforms a large circuit problem into one of cascaded circuit components. The limitation with this cascaded-circuit decomposition scheme is that it requires a component's FSM to be capable of satisfying the following component's input constraints. To facilitate this, the component's FSM should have the ability to preserve its state.

MVP decomposes a circuit into the subset of control registers on which the set of constraints are dependent. Unlike the work in [11], the ATPG methods of MVP do not require a control register's FSM to have the ability to preserve its state. The reasoning behind this is that satisfying a set of constraints on a sequential circuit will most likely require a sequence of test vectors. As a result, a set of constraints will be satisfied by starting the ATPG process at any target state that satisfies these constraints. Then these constraints are justified to bring the target state one step closer to the current state. During each intermediate step, the set of constraints will require specific control requirements in order to reach the appropriate assignment statements in the hardware description; these control requirements are used as the constraints for the next ATPG iteration (the previous time frame).

5 EXTRACTING PROSPECT CODE PATHS

As previously mentioned, an activation criteria denotes a collection of signal instantiations, and a corresponding set of values that these signals are required to satisfy. These activation criteria are used as the initial set of ATPG goals. Before attempting to identify the sets of implications that satisfy the ATPG goals, we can reduce the search space by first identifying, for each ATPG goal, the basic blocks that can assign the required value onto the required signal. For each of these identified basic blocks, we need to extract the guards (sensitivity list from process statements, and conditions from condition statements) that allow this block to be reached and combine the identified guard constraints to form the set of control requirements. Let us therefore define a prospect code path as one of the many assignment statements that may be able to satisfy an ATPG goal's constraint, such that the assignment statement can be reached when the identified control requirements are satisfied. A prospect code path for an ATPG goal will therefore contain: (*i*) the data dependency between the constraint's signal to the set of registers and primary inputs, and (*ii*) the control requirements that allow its corresponding assignment statement to be reached. A prospect code path can be conceptualized as one of many possible high-level cones of logic (local to a module) for a given signal or variable. It is important to mention that generating a prospect code path for an ATPG goal is performed independently of all other prospect code path generations for other goals, and it only need consider the scope of the module in which the ATPG goal exists.

Some control registers can retain their values across clock cycles when they are provided with a feedback loop or their assignment statements have extra control guards. In order to give the test pattern generation algorithm more flexibility in choosing when to defer the justification of a control register's value, the code paths that preserve a control register's value will not be generated explicitly. Instead, all prospect code paths that correspond to an assignment statement which modifies the register's value will be

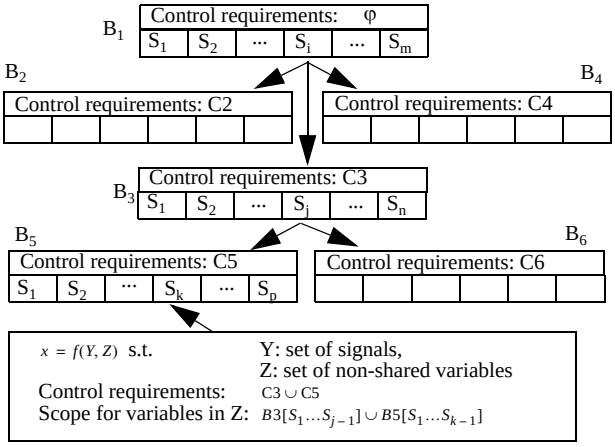

Figure 1 Diagram depicting statement tree of nested condition statements.

appended by a *deferrable* flag. When this flag is set, the prospect code path will be provided with the control requirements that allow the control register's justification to be deferred. If the control register has a feedback loop, the control requirements that allow this feedback loop to defer its justification will be the control requirements that correspond to the code path that activates this feedback loop. If the control register's assignment statements have extra guards that can prevent any of its statements from being reached, the control requirements that defer its justification can be found by creating a conjunction of negated guards for all its assignment statements. If this control requirement evaluates to false unconditionally, then it is clear that the control register's justification cannot be deferred.

To implement the environment that extracts the prospect code paths for a given module, a statement tree (Figure 1) can be created that preserves the structural integrity of all statements in the module and is able to provide an absolute path and control requirements to a given statement. The tree is implemented by a collection of *StatementList* nodes that contains a series of statements, and the control requirement for any of these statements to be reached. The root level only contains the concurrent items in the module and thus does not impose any control requirements. Any of these concurrent items can be a statement outside of process declarations, or they can be a process declaration. All other levels contain sequential items. A process is created into a sequential node by inserting all statements in the order in which they appear, such that a child *StatementList* node is created for any nested condition statements and a link to it is inserted in its place. Conditional assignment statements that exist outside of a process can themselves be converted into a process for their implementation [10][17], which allows us to extract the prospect code path for such a statement as we would for an assignment statement in a process.

A prospect code path is created in the form of a DDG as it is extracted from the module specification, and it includes the control requirements for that code path. Limiting the test pattern generation algorithm to RTL microprocessor implementations allows us to reject designs that employ loops in their processes. For each constraint, all assignment statements to the signal in that constraint will result in a set of prospect code paths.

A process will be spawned to convert an assignment statement into a prospect code path as follows: If the statement contains a non-shared variable, then this variable will be replaced by the previous assignment statement to that variable within the current code path. Shared variables will not be supported because of their nondeterministic behavior when multiple processes modify the same shared variable at the same simulation iteration [17]. If the state-

```
pc_mux: process(clk, pc_ctrl, pc ,out_alu, data_in, ea)
variable tempof: std_logic_vector(15 downto 0);
variable temppc: std_logic_vector(15 downto 0);
begin
      case pc_ctrl is
      when add_ea_pc =>
            if ea(7)='0' then tempof:="00000000" & ea(7 downto 0);
            else tempof:="11111111" & ea(7 downto 0);
            end if;
      when inc_pc =>
            tempof:="0000000000000001";
      when others=>
            tempof:="0000000000000000";
      end case;
      case pc_ctrl is
      when reset_pc =>
            temppc:="1111111111111110";
      when load_ea_pc =>
            temppc:=ea;
      when pull_lo_pc =>
            temppc(7 downto 0):=data_in;
            temppc(15 downto 8):=pc(15 downto 8);
      when pull_hi_pc =>
            temppc(7 downto 0):=pc(7 downto 0);
            temppc(15 downto 8):=data_in;
      when others =>
            temppc:=pc;
      end case;
      if clk'event and clk = '1' then
            pc <= temppc + tempof;
      end if;
end process;
```

Figure 2 Example process implementation which uses signals and variables [opencores.org].

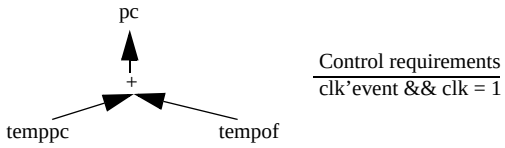

Figure 3 Incomplete DDG for signal *pc*.

ment contains an internal signal, then this signal will be replaced by any assignment statement to it. There may be numerous assignment statements whose control requirements do not conflict with those of a selected prospect code path, so a new prospect code path needs to be generated for each of these alternatives.

The program segment of Figure 2 is from the Motorola 6800 microprocessor implementation from John E. Kent [opencores.org]; it is a process whose purpose is to update the value to the program counter (PC) register. This is the only location in the microprocessor implementation where the PC register is written to, and generating the initial DDG for the signal *pc* gives us Figure 3. Notice that *temppc* and *tempof* are both variables, and the assignment statement to signal *pc* lies at the end of the process. Therefore when generating the DDGs for an ATPG goal on signal *pc*, any assignment to *temppc* and any assignment to *tempof* earlier in the process may be used as long as their control requirements do not conflict. The complete set of prospect code paths for an assignment to signal *pc* is generated and the resulting eight DDGs are shown in Figure 4.

The leaf nodes of all the DDGs in Figure 4 are constants, signals corresponding to registers, or primary input signals. Any of the prospect code paths deduced from these eight DDGs may be used to satisfy the ATPG goal on signal *pc*, and obviously some choices are better than others. By inspection we see that DDGs (e) and (f) may be used in sequence to effectively satisfy a constraint on signal *pc*, primarily because they have access to primary input signals. These eight DDGs are not the complete set of prospect code paths because the control requirements have not been expanded to register signals and primary inputs (*pc_ctrl* is a control signal, not a control-register

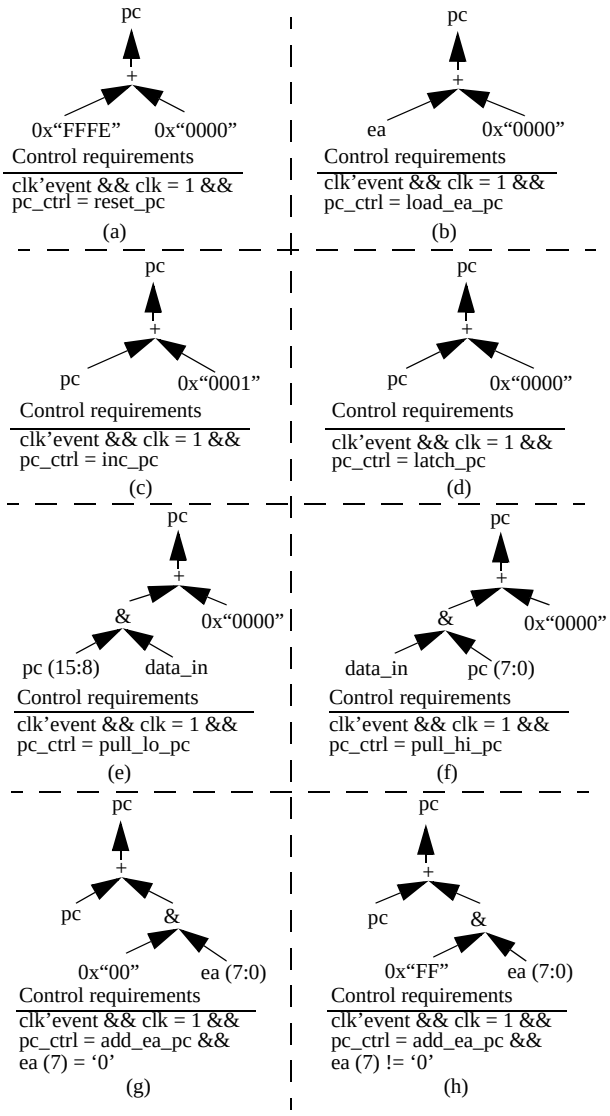

Figure 4 Possible DDGs for an ATPG goal on signal *pc*.

signal). The DDGs occur in the following number of prospect code paths after *pc_ctrl* is expanded: (a):1, (b):1, (c):12, (d):56, (e):3, (f):3, (g):1, (h):1. Therefore a total of 78 prospect code paths would be generated as a result of a constraint on signal *pc*. We next discuss the possible techniques for an effective ATPG.

6 GENERATING PROSPECT STATES

At this point, we have for each ATPG goal a collection of prospect code paths that are capable of performing the desired signal assignment. For each of these code paths, we have a collection of control constraints that need to be satisfied in order for the required assignment statement to be reached. It is possible to narrow down the search space into a collection of target microprocessor states that can possibly satisfy the set of ATPG goals. A prospect state can be conceptualized as a HDL-based state such that each process in an implementation activates a specific code path, and the control requirements of all processes do not contradict one other. It can also be conceptualized as one of many possible high-level cones of logic that traverse module boundaries. These set of prospect states can be generated by cross-referencing the sets of control requirements for all ATPG goals to identify all combinations of prospect code paths

that can satisfy the ATPG goals without resulting in a contradiction between control constraints.

Mutation-based modeled errors will commonly have multiple constraints as their activation criteria that must be satisfied concurrently. The set of constraints can reside in distinct module instantiations within the microprocessor implementation, but each of the prospect code paths has a scope that does not reach past its module instantiation. Each prospect state serves as a specific focal point for the constraint solver such that each ATPG goal's relation to the set of registers and microprocessor primary inputs are directly specified by the collection of DDGs. Therefore the scope of each prospect code path needs to be expanded when generating the complete prospect states. It has previously been stated in Section 5 that each module instantiation contains a statement tree and contains references to all its embedded modules. Therefore the collection of prospect states can be generated as follows: Each module instantiation is responsible for creating a prospect code path for every ATPG goal that resides inside itself. It is also responsible for generating the complete set of prospect states from the set of prospect code paths that reside inside itself; these prospect states have a domain that does not surpass its module's scope.

Each module instantiation uses the prospect states it receives from its children to generate the prospect states with its level of scope. When a module instantiation receives a prospect state from any of its children, it will first replace the child module's PI signals with the corresponding local signals as specified by the port map. Then it will continue to replace internal signals in the prospect state until all signal references correspond to its primary inputs or any register; a similar process to the generation of prospect code paths of Section 5. After this, it merges the prospect state from its child with its own (if it exists) into an expanded set of prospect states. It does this by cross referencing the control requirements from all its local prospect states with those of all its child's prospect states to generate all acceptable merges. Once it merges its own prospect states with those of all its children, each of these prospect states encompass all ATPG goals that lie at or below this point in the module hierarchy tree.

The prospect state generation technique is clearly a problem that can be optimized through concurrent programming. More specifically, each prospect code path can be generated by an independent process because by definition they are independent of all other prospect code paths. Furthermore, the creation of partial prospect states can be implemented by having each prospect code path generating process transfer control into a *merge()* operation local to its module. This *merge()* operation will merge the prospect code path into its local node such that the partial prospect state is accessed through mutual exclusion. Once the final *merge()* operation of a module has finished executing, it will transfer control to the parent module through the use of a *merge()* operation that itself will access that module's local prospect state through mutual exclusion.

A prospect code path will initially have data and control information that is represented by a DDG with a structure inherited from the hardware description. This DDG structure can vary, and some structures are easier to solve than others. Given that we are representing possible solutions by using range information as discussed in Section 7, we would prefer to avoid operators that impose solutions with multiple disjoint ranges of values. An example of such an operator is the inequality operator. A statement $A/=B$ returns a true Boolean value if $A<B$ or $A>B$, therefore doubling the number of explicit value ranges. Let us define such operators as "disjoining" operators. Instead of solving a DDG by transferring multiple value ranges across DDG operators as a result of disjoining operators, we can restructure a DDG into an equivalent graph that does not contain these disjoining operators. Table 1 contains the set of disjoining operators that we avoid in our DDG representation, and defines the replacement statements that are used for the operators.

Table 1 Replacements for some disjoining operators.

Operation	Equivalent Statement
not A<B	A>=B
not A<=B	A>B
not A>B	A<=B
not A>=B	A<B

Operation	Equivalent Statement
not A=B	A>B or A<B
not A/=B	A=B
A/=B	A>B or A<B
abs A	A or (-A)

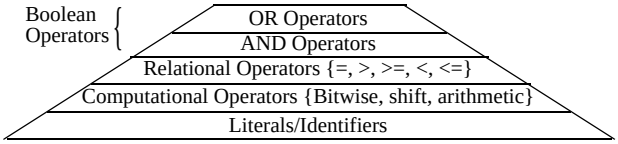

Figure 5 Operator hierarchy for DDG structure.

In our restructured DDG representation, we make an explicit distinction between Boolean operators and bitwise operators because they produce distinct disjoining effects when taking a (two) value range(s) as their operand(s). As an example, a Boolean *NOT* operator applied to an equality operator will double the number of explicit value ranges as previously stated, while a bitwise *NOT* operator applied onto a literal will simply invert every bit in its operand.

Also in our restructured DDG representation, there is only one disjoining operator that we allow to remain in our DDG structure as it binds all disjoint range of values into a set. We use the Boolean *OR* operator to reference the DDG structures that define a specific explicit value range, and a set of these DDG structures is linked by a chain of Boolean OR operators. As a result, we get a DDG structure in the form of a disjunction of conjunctions, such that each conjunction defines a specific explicit range in values. More specifically, all nodes in a conjunction share the range in values for the variables and signals they refer to. The nodes in our restructured DDG follow a hierarchy (Figure 5).

Once disjoining operator nodes are replaced by their equivalent DDG structure, we must propagate all Boolean OR and Boolean AND operators towards the first two layers in the DDG.

7 Representing a Range in Values

Discrete and real data types are the easiest to represent as a value range, given that their range can be explicitly defined by a minimum and a maximum value. Bitvector literals, however, are more difficult to represent. In Section 4 we discussed how the user would specify the explicit range in values for specific control registers in the form of partially-defined bitvectors. This specification forces bitvector operators to support a value range that is specified in the form of a set of partially-defined bitvectors.

A partially-defined bitvector is an array of bits, such that each bit can have a value of zero (0), one (1), or don't-care (x). An x value signifies that the corresponding bit can be used as a 0 or a 1. As a result, we can reduce the number of bitvectors required to represent a given set of values by merging pairs of bitvectors that differ by only one bit into a partially-defined bitvector. Thus a range of values exists as a set of partially-defined bitvectors.

This partially-defined bitvector set can be stored as a tertiary search tree, such that each insertion attempts to reduce the tree by removing bitvectors that are masked by the inserting bitvector, or by merging the inserting bitvector with another bitvector that has at most one corresponding bit with an inverted value. Whenever merging is required, the bitvector in the tree that initiates the merge is removed, and a new insertion operation is performed with the merged bitvector as the operand.

The *std_logic* libraries for VHDL allow performing relational and addition operations on *std_logic_vector* types with integer types. As a result, we require a method to convert an integer value

range into a partially-defined bitvector set. To do this, we can develop a method to generate a partially-defined bitvector set which enforces a minimum (maximum). Consequently, we can generate a partially-defined bitvector set which enforces an integer value range by generating them for the minimum and the maximum, and then intersecting both sets to generate a partially-defined bitvector set that forms an intersection between both sets.

To generate a partially-defined bitvector set for the minimum, we begin by replacing all '0' bits with an 'x' and insert a copy of this bitvector into the new set. From here on, while there is a sequence of '1's (starting with the least significant bit) that is followed by at least one 'x': replace this sequence of '1's by 'x's, replace the proceeding 'x' with a '1', and insert a copy of this bitvector into the set. The final insertion is a bitvector in the form of a series of 'x's that is appended by a series of '1's. Similarly, to generate a partially-defined bitvector set for the maximum, we begin by replacing all '1' bits with an 'x' and insert a copy of this bitvector into the new set. From here on, while there is a sequence of '0's (starting with the least significant bit) that is followed by at least one 'x': replace this sequence of '0's by 'x's, replace the proceeding 'x' with a '0', and insert a copy of this bitvector into the set. The final insertion is a bitvector in the form of a series of 'x's that is appended by a series of '0's. An example where a partially-defined bitvector set for an integer range is generated is shown in Figure 6.

An ordinary bitwise operation is easily performed across partially defined bitvector sets by applying the operation onto all pairs of partially-defined bitvectors from both sets. This process does generate a large number of partially-defined bitvectors, but this set is usually reduced substantially after it is inserted into the tertiary search tree.

Implementing arithmetic operations on partially defined bitvectors is the most difficult part of developing support for partially-defined bitvectors. Let us analyze the truth table for a single-bit adder as shown in Table 2 to discuss the implementation details. Scenarios V, VI, and IX provide alternative outputs that cannot be represented in a single partially-defined bitvector. It is therefore necessary to "split" the addition operation at this point into two independent addition operations, such that each independent addition operation produces its own solution in the form of a partially-defined bitvector. At the end of the addition operation, all the individual partially-defined bitvectors are inserted into the same set, and therefore the set is once again reduced by the insertion process.

Min 11 (00001011): S_{min} = {xxxx1x11, xxxx11xx, xxx1xxxx, xx1xxxxx, x1xxxxxx, 1xxxxxxx}
Max 13 (00001101): S_{max} = {0000xx0x, 0000x0xx, 00000xxx}
Range [11,13]: $S_R = S_{min} \cap S_{max} = \{00001011, 0000110x\}$

Figure 6 Generating a partially-defined bitvector set for an integer range.

Table 2 1-bit addition for a partially-defined bitvector.

	A	B	Cin	Cout	Sum
I:	0	0	0	0	0
II:	0	0	1	0	1
III:	0	0	x	0	x
	0	1	0	II	
IV:	0	1	1	1	0
V:	0	1	x	0	1
				1	0
	0	x	0	III	
	0	x	1	V	
VI:	0	x	x	0	1
				x	0
	1	0	0	II	
	1	0	1	IV	
	1	0	x	V	
	1	1	0	IV	
VII:	1	1	1	1	1
VIII:	1	1	x	1	x
	1	x	0	V	
	1	x	1	VIII	
IX:	1	x	x	1	0
				x	1
	x	0	0	III	
	x	0	1	V	
	x	0	x	VI	
	x	1	0	V	
	x	1	1	VIII	
	x	1	x	IX	
	x	x	0	VI	
	x	x	1	IX	
X:	x	x	x	x	x

Other arithmetic operations, such as negation and subtraction, are implemented using the addition operator.

8 Extracting State Transitions

All prospect states that reach the outer-most module instantiation will consist of DDGs and control requirements that only consist of constants, microprocessor primary inputs, and register signals. The first ATPG iteration will invest its energy in performing *blind* decisions when justifying values on ordinary signals. But for all ATPG iterations afterwards, the ATPG goals will constitute solely of control and data registers. This can be exploited to perform *smart* ATPG decisions by using a pre-computed FSM for each control register.

The pre-computed FSMs for control registers can be used to identify all the transition sequences under a given length that can map the current state for a given register onto the target state for that register. Before a test sequence can be generated that maps all current register states onto a set of target register values, it is necessary to cross reference transition sequences of equal length for distinct control registers. This is necessary to generate sets of transition sequences that are complete (all transition sequences have equal length, and one sequence exists for all FSMs that are relevant to the prospect state) and compatible (no conflicts exist between control requirements in the set).

If a set of complete and compatible transition sequences for all relevant registers (control and data) can be generated a priori, it would be possible to distribute the problem of solving each time frame among a set of server solvers in order to take advantage of the commonality of distributed workstations. With this approach, each server process would be given a collection of register values for two consecutive time iterations so it may identify the implications on the primary inputs that map the first time frame to the second.

9 Solving a Prospect State

The previous sections discuss how to extract all possible prospect states, and how to identify the state transition sequence that can reach a particular prospect state. This section discusses how a prospect state can be solved to identify the PI signal values that can map the target state one step closer to the current state. This could be used to perform either the initial blind test pattern generation process, or a smart test pattern generation process that abides by the identified state transition sequence.

By definition, if a VHDL implementation were to allow a signal to be assigned values from distinct code statements at a given state (control requirements), a bus contention will be observed. As a result, if the collection of code blocks that result from a set of constraints cannot be merged into disjoint DDGs, then the design can be rejected because it allows for bus contentions. There are designs that allow for multiple concurrent assignments to a signal at a given set of control requirements (state) by ensuring that at most one of these assignment statements asserts a non-high-impedance value. To allow this situation, we can generate the DDG by using only the assignment statements that do not assert a high-impedance value.

As previously mentioned, a prospect state is composed of a data requirement and a control requirement, such that both are implemented as a DDG. Also, it has been mentioned that a DDG is restructured into a disjunction of conjunctions, therefore allowing all nodes in a conjunction to share the range in values for the variables and signals they refer to. A data requirement (ATPG goal) is implemented as an equality statement (*signal = value*), therefore multiple data requirements are implemented as a conjunction of equality statements. Furthermore, to identify the set of value ranges that satisfy all equality statements in a data requirement simply requires a true value to be pushed into the root of the data requirement DDG.

10 PRELIMINARY SETUP AND RESULTS

The core of MVP's implementation has been implemented in 14.6K lines of C++ code and is currently being maintained under OS X Tiger using gcc 4.0. It is compiled as a library using GNU's autotools (autoconf, aoutomake, and libtool) such that any structural VHDL implementation can be converted into an ATPG unit by translating it into a series of C++ objects that use MVP's library. The results obtained are executed on a dual 2.5GHz PowerPC workstation.

MVP's implementation has been simplified and streamlined by promoting design consistency. The work of the solver process is actually performed when generating a DDG, expanding its scope, and reducing its size by eliminating all conjunctions whose variables and signals experience data contradictions. This technique is used to generate both the control dependencies and data dependencies for a prospect state, and when merging prospect states local to a hierarchical unit into a global set. Therefore once a prospect state reaches the global scope, a solution to the ATPG problem is extracted directly from any global prospect state. Obtaining any solution from any global prospect state implements the *blind* ATPG approach. A NULL set of global prospect states signifies the problem has no solution.

MVP is currently capable of generating and solving prospect states that correspond to multiple simultaneous constraints; these prospect states represent the target architectural states that can satisfy the collection of simultaneous constraints. MVP, however, is not yet capable of generating an FSM for a given microprocessor implementation. FSM analysis should soon follow, as it will exploit the efficiency in generating and solving prospect states.

Six constraints (listed in Table 3) have been selected for ATPG from the Motorola 6800 microprocessor implementation that was introduced in Section 5. Constraints I and II were chosen to represent scenarios involving the FSM. Constraints III and IV were chosen to represent the problem of solving for unique corner-cases, such that their values are not commonly used by the implementation. Similarly, constraints V and VI were each used to represent a signal-value pair that is very commonly encountered. These six constraints create a diverse set of test scenarios, and their results are described next.

Results from the six testbench setups are provided in Table 4. Notice that even though the solution for constraint IV has less prospect states than constraint III, it still requires more computation time. This extra computation time is due to its dependency on a bit-vector, and is a result of the extra complexity of handling bit-vector information. A similar argument can be said about constraints I and II, as the processor's micro-architectural state is highly-dependent on the *op_code* signal.

The solutions in Table 4 correspond to the process of identifying every target architectural state that can satisfy a given set of constraints. This ability to provide such detailed and complete information should be proper justification for the required execution time. Later optimizations will reduce this execution time, and future work will use this solver algorithm to generate an FSM for a given microprocessor implementation.

Table 3 List of individual constraints used.

	Constraint	Characteristic
I	state <= int_pcl_state	FSM-based
II	state <= vect_hi_state	FSM-based
III	nmi_ctrl <= set_nmi	Low-occurrence
IV	alu_ctrl <= alu_cpx	Low-occurrence
V	cc_ctrl <= load_cc	High-occurrence
VI	acca_ctrl <= latch_acca	High-occurrence

Table 4 ATPG performance/results.

	I	II	III	IV	V	VI
Prospect States	11	15	6	3	273	318
Execution Time	2m 4.99s	2m 9.80s	7.91s	35.22s	1m 0.24s	46.68s

ACKNOWLEDGMENT

This material is based upon work supported by the National Science Foundation under Grant No. 0092867.

REFERENCES

[1] J. Campos and H. Al-Asaad, "Concurrent Design Error Simulation for High-Level Microprocessor Implementations," *Proc. AUTOTESTCON*, 2004, pp. 382-388.

[2] J. Campos and H. Al-Asaad, "Mutation-Based Validation of High-Level Microprocessor Implementations," *Proc. HLDVT*, 2004, pp. 81-86.

[3] J. Campos and H. Al-Asaad, "Search-Space Optimizations for High-Level ATPG", To appear in *Microprocessor Test and Verification Workshop*, 2005.

[4] R.A. DeMillo, R.J. Lipton, and F.G. Sayward, "Hints on test data selection: Help for the practicing programmer", *IEEE Computer*, vol. 11, April 1978, pp. 34-41.

[5] J. Shen and J.A. Abraham, "An RTL abstraction technique for processor microarchitecture validation and test generation", *Journal of Electronic Testing: Theory and Applications*, vol. 16, February-April 2000, pp. 67-81.

[6] Li-C. Wang, Magdy S. Abadir, and Jing Zeng, "On Logic and Transistor Level Design Error Detection of Various Validation Approaches for PowerPC Microprocessor Arrays," *Proc. VLSI Test Symposium*, 1998, pp. 260-265.

[7] E. Jenn et al., "Fault Injection into VHDL Models: The MEFISTO Tool," *Digest of Papers: International Symposium on Fault-Tolerant Computing*, 1994, pp. 66-75.

[8] L. Berrojo et al., "New Techniques for Speeding-up Fault-Injection Campaigns," *Proc. Design Automation and Test in Europe*, 2002, pp. 847-852.

[9] M. N. Velev, "Collection of High-Level Microprocessor Bugs from Formal Verification of Pipelined and Superscalar Designs," *Proc. International Test Conference*, 2003, pp. 138-147.

[10] P. A. Wilsey, D. E. Martin, and K. Subramani, "SAVANT/TyVIS/WARPED: Components for the Analysis and Simulation of VHDL," *VHDL Users' Group Spring Conference*, 1998, pp. 195-201.

[11] C.-C. Yen, J.-Y. Jou, and K.-C. Chen, "A Divide-and-Conquer-Based Algorithm for Automatic Simulation Vector Generation," *IEEE Design and Test of Computers*, Vol. 21, March-April 2004, pp. 111-120.

[12] F. Corno et al., "SymFony: A Hybrid Topological-Symbolic ATPG Exploiting RT-Level Information," *IEEE Transactions on Computer-Aided Design*, Vol. 18, February 1999, pp.191-202.

[13] A. Adir et al., "Genesys-Pro: Innovations in Test Program Generation for Functional Processor Verification," *IEEE Design and Test of Computers*, Vol. 21, March-April 2004, pp.84-93.

[14] F. Corno et al., "Automatic Test Program Generation: A Case Study," *IEEE Design and Test of Computers*, Vol. 21, March-April 2004, pp.102-109.

[15] S. Tasiran and K. Keutzer, "Coverage Metrics for Functional Validation of Hardware Designs," *IEEE Design and Test of Computers*, Vol. 18, July-August 2001, pp.36-45.

[16] D. Moundanos, J. A. Abraham, and Y. V. Hoskote, "Abstraction Techniques for Validation Coverage Analysis and Test Generation," *IEEE Transactions on Computers*, Vol. 47, January 1998, pp.2-14.

[17] IEEE Standard VHDL Language Reference Manual. New York, NY, 1993.