A Novel Mutation-Based Validation Paradigm for High-Level Hardware Descriptions

Jorge Campos, *Member, IEEE*, and Hussain Al-Asaad, *Senior Member, IEEE*

***Abstract*—We present a Mutation-based Validation Paradigm (MVP) technology that can handle complete high-level microprocessor implementations and is based on explicit design error modeling, design error simulation, and model-directed test vector generation. We first present a control-based coverage measure that is aimed at exposing design errors that incorrectly set control signal values. We then describe MVP's high-level concurrent design error simulator that can handle various modeled design errors. We then present fundamental techniques and data structures for analyzing high-level circuit implementations and present various optimizations to speed-up the processing of data structures and consequently speed up MVP's overall test generation process. We next introduce a new automatic test vector generation technique for high-level hardware descriptions that generates a test sequence by efficiently solving constraints on multiple finite state machines. To speed up the test generation, MVP is empowered by learning abilities via profiling various aspects of the test generation process. Our experimental results show that MVP's learning abilities and automated test vector generation effectiveness make MVP significantly better than random or pseudorandom validation techniques.**

***Index Terms*—Simulation-based design verification, concurrent design error simulation, high-level deterministic test generation, design error modeling.**

I. Introduction

Digital circuit design methodologies have reached a highly optimized state, but the circuit validation methods used in industry are still subjective to the validation engineer on the job. Many circuit validation methods for high-level hardware descriptions are available and are in competition with one-another and most of these methods do not provide a stand-alone solution. As a result, circuit validation is still an art mastered by an engineer through experience and observations, as opposed to a systematic technique that can be easily disseminated. These validation engineers develop an intuition on *how* to perform circuit validation, and most importantly, on *how much* circuit validation is necessary. It is therefore possible to reduce the time and money required to create circuits by reducing the human effort required in circuit validation through a systematic and easily-reproducible system comprised of software tools and deterministic practices, both in software and human effort.

"Black-box" circuit validation is a strategy that does not require the validation system to have prior inside knowledge of the circuit under validation. It currently relies on random or pseudo-random test patterns to validate the "black box" because such a validation system has no way of efficiently deciphering *how* to generate the most effective instruction sequence for circuit validation.

Current industry standard practices rely on random and pseudo-random instruction sequence generation techniques that bet on the statistical nature of their practices to eventually explore a large-enough portion of the circuit. The simplicity in these practices allows for a high-frequency of simulations, but the ability of the simulator to traverse unexplored architectural states quickly diminishes over time. Deterministic practices, on the other hand, guarantee continued forward progress for effectiveness as they allow the circuit validation engineer to attack the problem head-on.

Unfortunately, current validation systems are neither efficient nor effective enough to perform *deterministic* "black box" validation on complete large circuit implementations. Therefore, when handling complete large circuits such as microprocessors and ASICs, companies rely on "white box" circuit validation to attain an increase in circuit coverage and validation effectiveness. To achieve this, circuit design teams provide inside information to their validation strategy by building "self-testing" knowledge into their implementations and validation system.

In this paper, we describe the design and implementation of MVP, a mutation-based validation paradigm. MVP is a circuit validation tool for high-level hardware descriptions, and its purpose is to provide expert deterministic validation methods to the average design engineer. MVP provides a complete and automated strategy for analyzing high-level hardware descriptions that only leaves the circuit design engineer to decide what portions of the circuit to validate, and not how to validate it. These circuit analysis abilities allow MVP to perform automated white-box circuit validation on high-level RTL descriptions while providing the simplicity of black-box validation to its users.

MVP does not require a priori information on the circuit under validation for it to be effective, but instead gathers this

Manuscript received September 23, 2006; revised July 16, 2007. This work was supported by the National Science Foundation under Grant No. 0092867. Preliminary parts of this paper appeared in [1]-[5].

Jorge Campos is with the Park Vaughan & Fleming patent law firm, Davis, CA (e-mail: jcampos@ucdavis.edu).

Hussain Al-Asaad is with the Department of Electrical and Computer Engineering, University of California, Davis, CA (e-mail: halasaad@ece.ucdavis.edu).

information real-time. This allows semiconductor companies to analyze large circuit implementations in their entirety, and allows them to analyze these projects even before they are ready for circuit synthesis. The generality and completeness in MVP's design allows it to be used for all validation strategies: from formal to simulation-based, static assertion-based to dynamic assertion-based, deterministic vector generation to pseudorandom vector generation, and from a static code-coverage metric to any functional coverage metric.

A. Related Work

We next describe a few good examples of simulation-based circuit validation systems aimed at ensuring the correctness of complete large circuit implementations.

SymFony [6]. Many common validation environments that employ symbolic automatic test pattern generation methods rely on a gate-level implementation that has been previously synthesized. Consequently, these 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 finite state machine (FSM) state register, and data-path macros that are composed of data registers/data-manipulating combinational logic.

SymFony's automatic test vector generation (ATVG) 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 Binary Decision Diagrams (BDDs) are intersected with the complete FSM BDDs (next-state and output BDDs) during each symbolic solver process.

Genesys-Pro [7]. Unlike many other verification tools, the Genesys-Pro verification tool is capable of performing verification on complete processor systems by implementing a model-based test pattern generation approach. The 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 effort into generating this testing knowledge that 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 [8]. 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.

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.

B. Overview of MVP's Validation Process

A diagram depicting MVP's validation process is provided in Fig. 1. In the *description phase*, a microprocessor implementation and a collection of abstract design error models are created. Given that these implementations are inherently C++ code, a standard C++ compiler is used to create the runtime Simulation/ATVG system during the *construction phase*. The *design error/fault injection* application programming interface (API) is required to implement the runtime simulation environment, and the *ATVG engine* API is required to implement the runtime ATVG environment. The target block of a double-lined arrow denotes an object produced by the source block of the double-lined arrows.

The *simulation/ATVG phase* executes the compiled program where the microprocessor implementation and a collection of user run-time configurations and commands are used as inputs. The user specifies what outputs to produce and what modeled errors to consider during the simulation and ATVG efforts.

The Simulation/ATVG program creates the output files during the *reporting phase*, including the instruction sequence that detects the optimal set of modeled errors, the collection of

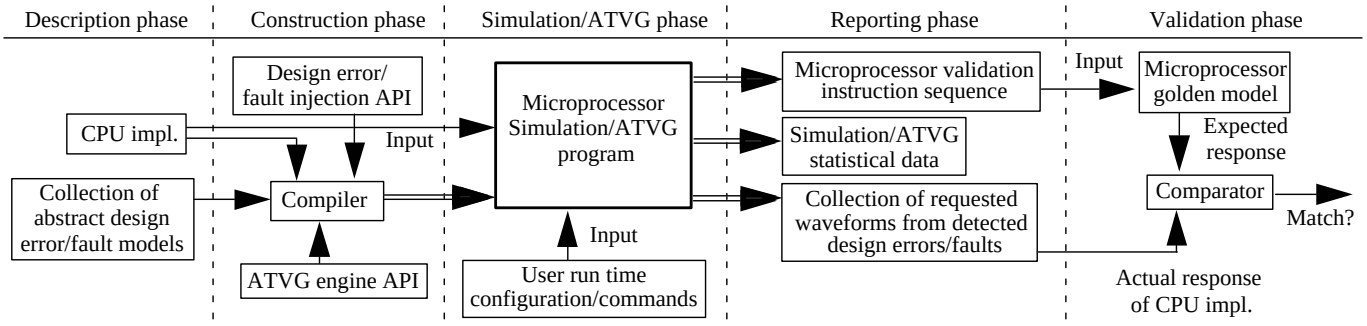


Fig. 1. MVP's validation process overview.

simulation/ATVG statistical data, and the actual response of the microprocessor implementation when the instruction sequence is used as an input, as well as a collection of responses that correspond to a user-specified set of modeled errors. The instruction sequence is fed into the golden model (specification) during the *validation phase* to generate the expected response, which is compared with the actual response of the implementation. Any detected discrepancy needs to be analyzed to determine if it is a result of an actual design error, or if it corresponds to an invalid input sequence.

The rest of this paper is organized as follows. Section II discusses a control-based coverage metric that is used by MVP. Section III introduces MVP's concurrent mutation-based circuit simulation techniques that provide real-time coverage analysis. Section IV introduces MVP's low-level mechanisms for performing high-level circuit analysis. Section V expands on the mechanisms of Section IV to apply them into MVP's ATVG algorithm. Section VI discusses the real-time profiling techniques that allow MVP to learn and adapt onto any circuit description. Section VII presents experimental results from applying MVP onto an open-source Motorola 6800 microprocessor implementation, and Section VIII concludes this paper.

II. A Control-Based Coverage Metric

Several coverage metrics have been proposed in the literature. An exhaustive coverage metric would attempt to apply every possible input vector onto every architectural state of the processor. This obviously leads to the state explosion problem. Two popular alternatives that reduce this coverage space include state coverage (attempting to reach every reachable state in the design) and transition coverage (attempting to traverse every possible transition among the states of the design) [9][10]. Another popular metric is the FSM path-coverage metric [9] that 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. Finally, due to the extensive use of modern hardware description languages (HDLs), a series of code-based coverage metrics appeared including line coverage, transition coverage, and path coverage [9][11].

It is important to employ a coverage metric that reduces the search space from an exhaustive coverage metric without notably degrading the quality of the tests that result from it. This step is critical to the development of a validation paradigm because an inefficient coverage metric will require too many test vector 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; this being because 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 processor implementations are modular in nature, we assume that modules are validated against their description before the microprocessor is validated as a whole. Consequently, 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 [12] shows that over two-thirds of design errors are related to the control logic.

At this point, it is obvious that the coverage metric becomes one that ensures every possible value for each control signal is exercised for every possible processor state. This may seem like an even harder coverage metric to employ than the state coverage metric, but it can actually reduce the state space by ignoring redundant and irrelevant processor states.

Error Modeling (Mutants). It is obvious that an error results in the generation of an erroneous value under a specific state of the system. So, circuit design errors and physical faults are governed by the rules of cause-and-effect. We can harness this

cause-and-effect characteristic to define the mutant construct. Similar to some fault injection campaigns [13][14], we are defining an error model by three basic characteristics: (*i*) the activation criteria, (*ii*) the consequence of activation, and (*iii*) error injection. Furthermore, let us define mutants as the many instantiations of an error model that span a given design space. Numerous mutants are to be simulated concurrently; therefore each mutant must have a unique identification number.

A mutant's activation criteria specify a set of signals and the conditions that they must satisfy before the mutant is activated. Once the activation criteria of a mutant are met, its code segment is executed as the consequence of activation to generate a set of mutations for a corresponding set of injection points. The mutant values are injected into the specified signals within the circuit during error injection, which follows immediately thereafter.

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 [15] 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 simultaneously appear at multiple instantiations of a component if necessary, and it correctly models aliasing in the case where these mutant values mask each other's propagation across the circuit.

The Mutation Control Error Model. Given the objective to stimulate every control signal under every processor state, we employ a modified version of the mutation control error (MCE) model [16] as the preliminary error model for our validation environment.

Our modified model accommodates for micro-architecture-based (FSM-based) processor implementations and is defined as the quadruplet (*s*, *c*, *vc*, *ve*) [1] such that the explicit processor state *s* and a correct value *vc* of the control signal *c* act as the activation criteria, signal *c* is mutated from *vc* to an erroneous value *ve* as a consequence, and injected back into signal *c*. This modification is possible because the arrival of an instruction *i* into a processor cycle *c* of a structural microprocessor implementation can be interpreted as a processor state.

Implementing the Control-Based Coverage Metric. It is possible to automate the generation of mutants that span the control-based coverage measure. In fact, performing automated generation of these mutants is expected to be most influential for superscalar microprocessor implementations because of their inherent complexity. If one were to analyze the data dependency of a control signal onto the set of registers and primary inputs, one would see that each control signal is dependent on only a subset of the control registers. 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 state-space *s*. Then for every correct value *vc* of control signal *c* under every possible value of *s*, generate a set of mutants that modifies *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, but it is more effective because it only generates useful mutants 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 (pState) is introduced and defined in Section IV and can be generated as discussed there. Each pState for a given signal defines a possible data dependency that satisfies the signal constraint, and the set of control requirements that allow for that data dependency to be satisfied. All pStates generated from the control signal must have its solution space searched for relevant control registers.

III. Concurrent Mutant Simulation

Significant work has been performed on mutation-based analysis and verification, especially in the software verification arena. A lot of the foundation of mutation-based hardware verification is an adaptation from the software area. However, when it comes to hardware, building the fault dictionary and injection of the faults requires special attention. The fault injection needs to be dynamic in nature for certain types of faults. In this section, we describe a concurrent mutant simulator that is characterized by a dynamic injection of modeled design errors (mutants) where only the design errors that are activated are injected back into the circuit.

Error modeling for circuit validation is used to create an artificial collection of simple modeled design errors (mutants) that span throughout the corner cases of an implementation. As a consequence to the coupling effect between simple and complex design errors [17], a test sequence that is capable of detecting these known simple modeled errors is implicitly capable of detecting actual complex design errors as well. Therefore, one application for a concurrent mutant simulator is to grade a test sequence's ability to traverse the design space by concurrently and efficiently applying it to the complete set of mutants and reporting its coverage. A more important application of mutation-based circuit simulation is that it can be used as a part of the mutation-based testing engine as we have done in our MVP.

A. Limitations of Modern Error Injection Campaigns

Analysis of controllability and observability measures through concurrent simulation methods has been previously investigated via a tag simulation calculus [18]. Under this simulation method, a single tag is propagated throughout the simulation to designate a possible change in a signal value due to an error. This method, however, results in an estimation of observability given that mutations on a signal are represented by a tag that only represents a positive or negative polarity. Furthermore, this method requires the modification of the hardware description when condition statements are involved in order to compute the effects of the fault model when it

causes the wrong path to be taken.

The methods in [19] are an improvement as behavioral fault simulations are implemented with fault lists, such that Petri Nets are used to propagate the fault lists in their event-triggered simulation environment. They fell short of creating a concurrent fault simulation environment because they use an initial fault free simulation phase to guide the subsequent simulation with fault-list propagation phase. This prevents the simulator from being used alongside the automatic test pattern generator in a fine-grained fashion, therefore preventing the possibility of a closed-loop validation strategy.

Other related papers discuss methods of generating mutations of a hardware description as a means to find a test pattern that can distinguish a program from its faulty versions [13][14]. These techniques, however, generate a collection of mutant implementations by modifying the original implementation. This results in a collection of separate implementations that require independent simulations.

B. The Concurrent Approach

The initial step in developing the high-level concurrent error simulator consists of determining how a signal should maintain its fault list, and how the basic signal operations should be performed on the complete fault lists. To accomplish this, a signal is first defined as an object that consists of a fault-free value and a list of mutant values, where each mutant m in the signal S is obtained as a result of the corresponding parent error model. Let us denote the parent construct of a mutant value m by $\pi(m)$. It is common that aliasing occurs between the fault-free value and one or more mutant values, in which case it is advantageous to collapse the error lists as a means to minimize the memory demand and the number of operations required by each list.

The simulator takes as input a collection of mutants E, which are used to generate and insert a mutant into a specific fault site when appropriate. Let a_i be the set of fault values in signal A, such that $a_{i=0}$ denotes the fault-free value and $a_{i\neq0}$ denotes the mutant value associated with the mutant construct $\pi(a_i)$ that has an ID value i. Given that the fault list-enabled signals implement fault collapsing, an arbitrary mutant value a_i will only exist in signal A when all of the following conditions are met:

- The mutant construct $\pi(a_i)\in E$ has been activated.
- The corresponding error has been injected or propagated into signal A, thus producing the mutant value a_i.
- The corresponding mutant value a_i is not aliased by the fault-free value a_0 ($a_i \neq a_0$).

Based on the above definition of a signal's fault list, we developed MVP's concurrent mutant simulator that allows us to propagate fault lists across operations in high-level hardware descriptions. The simulator devises a novel method to implement conditional execution on signals containing a fault list. The problem of executing a statement based on a fault list-enabled condition is that the condition will be met by some of the mutants and not by others. As a result, the fault list of the signals in the condition statement must be split into two partitions: the set of mutants that meet the condition, and the set of mutants that do not. The complete details of the simulator and its implementation are described in [1].

C. Integrating Mutant Value Generation into the Simulation Environment

The core concurrent mutant simulator does not produce mutant values; its purpose is simply to propagate them. The mutant values are generated by separate engine(s) that we denote as mutant value generator(s). As a result, we can have a simulation environment that is adaptable to any design-based/fault-based error model by creating the appropriate mutant value generator(s) that are in charge of inserting the appropriate mutant values into the appropriate signal(s) under the appropriate condition(s).

A mutant generator is a unit within a simulation environment in charge of activating any of its mutants when the proper activation criteria are met, at which point it generates the corresponding mutant value and injects it into the circuit. Therefore for each mutant generator, the collection of signals in the circuit that act as activation criteria to any of its mutants needs to propagate any change in value to this mutant generator. Also, whenever an activation criterion propagates into a mutant generator, the mutant generator needs to search through its set of mutants and identify every mutant that needs to be activated. When developing a mutant value generator, the effects of propagation complexity (the number of extra mutant signal propagations per simulation iteration such that a mutant value generator is the target) and activation complexity (the number of mutants that need to be considered for activation upon the propagation of a signal into a mutant value generator) need to be taken into consideration.

There are three implementation alternatives for the distribution of mutant constructs among mutant value generators: (*i*) a centralized mutant generator where only one unit in the simulation environment is in charge of generating mutant values (low propagation complexity and high activation complexity), (*ii*) distributed mutant generators where one mutant value generator is assigned to each mutant construct (high propagation complexity and low activation complexity), and (*iii*) hybrid (clustered) mutant generators where mutant constructs are "clustered" into groups that have common activation criteria, therefore maintaining the propagation complexity and activation complexity per mutant value generator at feasible levels.

In a validation system where multiple error models are being used, the hybrid mutant generation technique gives us the flexibility of keeping mutants of disjoint activation criteria in separate clusters and allows us to optimize the search algorithm of each mutant value generator by introducing a "clustering and partitioning" technique. The technique reduces the search space per mutant generator by selecting the signal that acts as the most common activation criteria in that cluster and designating this signal as the partitioning point. A detailed description of our "clustering and partitioning" technique can be found in [2].

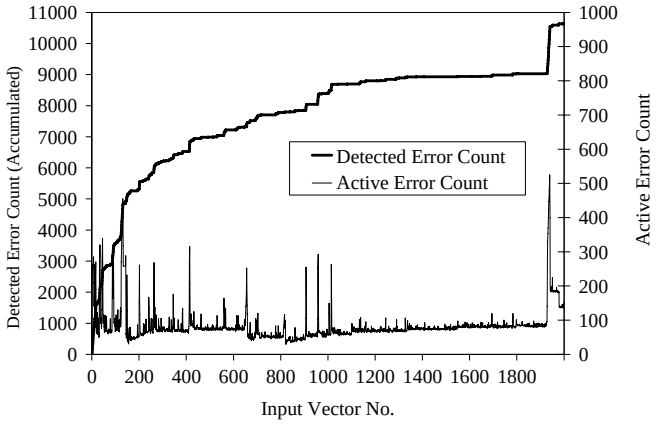


Fig. 2. Detected/active error comparison [1].

D. Concurrent Simulation Results

We have used the modified MCE model in conjunction with an automatic mutant generator for the micro-architectural implementation of the Motorola 6800 microprocessor by John E. Kent [20]. The exhaustive set of MCEs for this implementation consists of 300,092 mutants, and they have been simulated using a random test sequence. The simulation results demonstrate that there is a significant correlation between the number of mutants that are detected and the number of mutants that are active per simulation iteration (Fig. 2); therefore establishing that it is fruitful to focus our ATVG efforts on maintaining the number of active mutants at its highest possible value.

Fig. 2 also brings to our attention the difficulty in generating tests to detect mutants given that a random test sequence activated a mere peak of 525 mutants out of a collection of 300,092 (less than 0.2%) mutants. The low activation rate for mutants and the strong correlation between the active and the detected mutant counts serve to encourage the implementation of a concurrent mutant simulator for our MVP validation technique because an exhaustive set of mutants can be simulated with an acceptable performance cost and significant observability rewards.

IV. High-Level Circuit Analysis

Once a set of mutants have been generated, they are efficiently simulated on the hardware description under validation to utilize the test sequence and track its effect. Now, to promote an effective validation paradigm, we need to devise a systematic test pattern generation strategy that can satisfy the simultaneous constraints specified by any mutant. To do this, we first need to convert the set of simultaneous constraints into a solution space. This solution space must list all target architectural states that can satisfy all simultaneous constraints, and any of these target architectural states can be used as the starting point when generating a test sequence.

It is important to take the time to identify the complete set of few target architectural states because doing so prunes out the many irrelevant architectural states. Given that only one of these target architectural states is necessary, let us denote each target architectural state in the solution space as a *prospect state (pState)*. Also, let us define each pState by: (*i*) the simultaneous constraints to be satisfied, and (*ii*) the data and control dependencies among the constraints' identifiers in the circuit under validation. Each of the above components of a pState is implemented using a type of constraint dependency graph (CDG) that we describe next.

A. Constraint Dependency Graphs (CDGs)

The CDG is a structure that can represent possible solutions by using range information akin to fuzzy logic. Fig. 3(a) depicts an example CDG produced by the "when others" block of a case statement for signal *A*, such that the guards for the case statement's two explicit cases are: (*i*) "when 1" and (*ii*) "when 3". In solving CDGs, we would prefer to avoid operators that impose solutions with multiple disjoint ranges in 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 splitting the solution space into two explicit value ranges. Let us define such operators as "disjoining" operators. Instead of solving a CDG by transferring multiple value ranges across CDG operators as a result of disjoining operators, we can restructure a CDG into an equivalent graph that does not contain these disjoining operators.

In our restructured CDG representation, there is only one disjoining operator that we allow to remain in our CDG structure as it binds all disjoint range of values into a set. We use the Boolean *OR* operator to reference the CDG structures that define a specific explicit value range, and a set of these CDG structures is linked by a chain of Boolean OR operators. As a result, we get a CDG structure in the form of a disjunction of conjunctions, such that each conjunction defines a specific explicit range in values for its identifiers. 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 CDG follow a hierarchy in the following order: OR operators, AND operators, relational operators, computational operators, and literals/identifiers. The Boolean OR and Boolean AND operators are propagated towards the first and second layers in the CDG, respectively, via DeMorgan's Theorem.

MVP's pStates are generated and solved throughout the ATVG process to justify a set of constraints, making the algorithms that restructure and solve the CDGs to be MVP's limiting factor. Therefore optimizing the worst-case scenario for the algorithm that restructures and solves a CDG will have a significant impact on MVP's overall performance. Case statements (commonly used in hardware descriptions) can be significantly large, especially when they are used to describe the functionality of an FSM. These large case statements will be the limiting factor for MVP's performance because we need to analyze the assignment statements and control requirements for every block in the case statement.

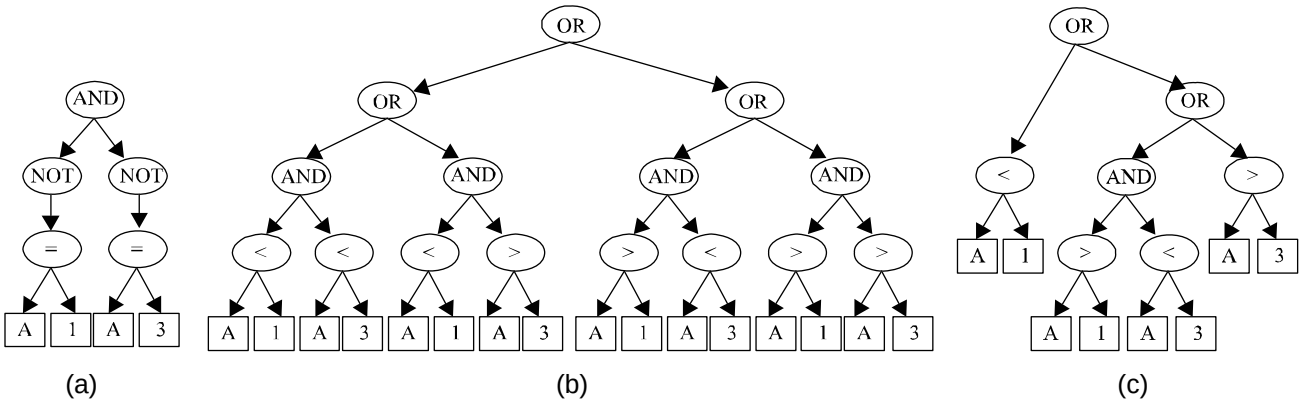


Fig. 3. (a) A sample CDG and its corresponding restructured CDG with (b) no optimization and (c) complete optimization.

If a case statement contains a "when others" block, its control requirements (guard) will be the conjunction of the negated guards of all explicit cases. This block's control requirements will therefore be a conjunction of disjoining operators, where a disjoining operator is an operator that imposes a disjoint range of values onto any identifier operand. Our goal is to convert this graph into a disjunction of conjunctions such that each conjunction defines a contiguous range of values for all discrete identifiers within it, therefore we must restructure the CDG into an equivalent graph that is free of disjoining operators.

Restructuring the graph into our desired form forces us to recursively replace each sub-tree rooted at a disjoining operator with an equivalent tree that is free of disjoining operators, but is bigger in size. An inequality operator is replaced by a disjunction of relational operators, which unfortunately is a complete tree with twice the number of leaf nodes than the original. The size complexity is exacerbated by the modified graph's conjunction of disjunctions structure. Performing a brute-force restructuring process to convert this graph into a disjunction of conjunctions through the use of DeMorgan's Theorem produces a graph that is exponential in size in terms of the number of disjoining operators. This size complexity quickly becomes a burden because restructuring requires an exponential runtime complexity, and soon thereafter becomes a limitation because it may easily consume all available memory. Fig. 3(b) shows the restructured CDG (with no optimization) using the method described above.

By analyzing the restructured graph of Fig. 3(b), we can notice the presence of sub-trees that can be removed early in the restructuring process because they evaluate to *false*. We consider these sub-trees as *unconditionally false* and we can identify them by attempting to force a Boolean *true* value onto any Boolean operator or relational operator. A sub-tree will only be able to satisfy the *true* value if the range of values imposed onto all identifiers at that sub-tree intersects with the range of values imposed on corresponding identifiers at all other sub-trees of the same conjunction.

In addition to the above, we can notice the presence of sub-trees that can be removed early in the restructuring process because they evaluate to *true*. These sub-trees occur when a comparison on an identifier does not reduce the range of values imposed on that identifier; therefore we consider these sub-trees as *unconditionally true*. Applying the above optimizations to the CDG of Fig. 3(b) produces the CDG shown in Fig. 3(c).

The optimization presented above effectively reduces the size complexity and runtime complexity of the restructure process. More details about this optimization method are presented in [4].

B. Generating a Prospect Code Path

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 ATVG goals. Before attempting to identify the sets of implications that satisfy the ATVG goals, we can reduce the search space by first identifying, for each ATVG goal, the basic blocks of HDL code that can assign the required value onto the required signal. For each of these identified basic blocks, we need to extract the guards (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 ATVG goal's constraint, such that the assignment statement can be reached when the identified control requirements are satisfied. A prospect code path for an ATVG goal will therefore contain: (*i*) the constraint to be satisfied, and (*ii*) the assignment statement and control requirements that allow this constraint to be satisfied. It is important to mention that generating a prospect code path for an ATVG 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 ATVG goal exists.

To implement the environment that extracts the prospect code paths for a given module, a statement tree is created such that it 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 requirements that allow 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. An HDL 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 [21][22], which allows us to extract the prospect code path for such a statement just as we would for an assignment statement in a process.

There may be numerous assignment statements capable of satisfying any given constraint, so a new prospect code path needs to be generated for each of these alternatives. A process is spawned to convert a constraint into a set of prospect code paths as follows: If the statement references a non-shared variable, then the scope of this variable is expanded by inserting the previous assignment statement to that variable within the current code path. Shared variables are not supported because of their nondeterministic behavior when multiple processes modify the same shared variable at the same simulation iteration [22].

The program segment of Fig. 4(a) is from the Motorola 6800 microprocessor implementation from John E. Kent; 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 CDG for the signal *pc* gives us Fig. 4(b). 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 CDGs for an ATVG 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 CDGs are shown in Fig. 4(c).

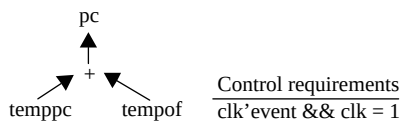
Any of the prospect code paths deduced from these eight CDGs may be used to satisfy the ATVG goal on signal *pc*, and obviously some choices are better than others. By inspection we see that CDGs (5) and (6) may be used in sequence to effectively satisfy a constraint on signal *pc*, primarily because they have access to primary input signals.

C. Generating Prospect States

At this point, we have for each constraint a collection of prospect code paths that are capable of performing the desired signal assignment. For each of these code paths, we have a

```
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;
```

(a)



(b)

(c)

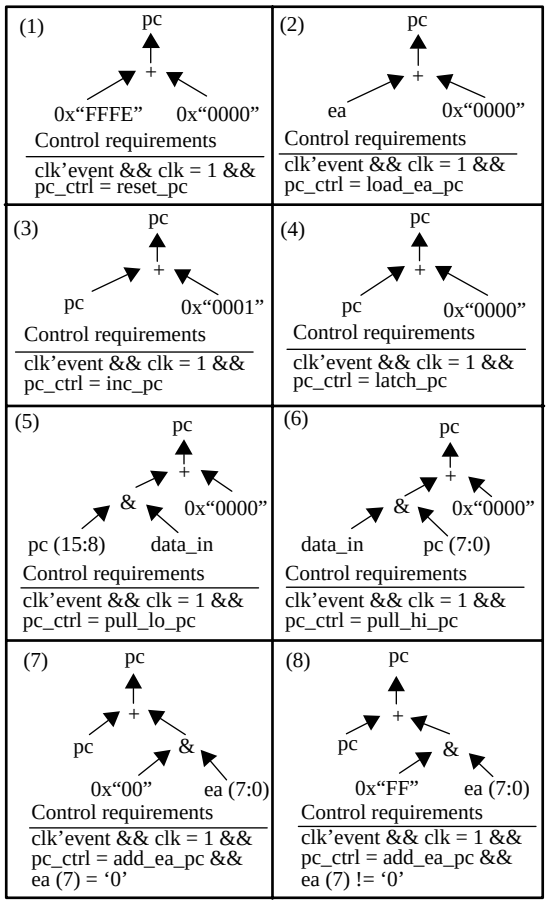


Fig. 4. (a) An example process implementation which uses signals and variables and its corresponding (b) incomplete CDG for the signal *pc* and (c) possible CDGs for an ATVG goal on the signal *pc*.

collection of control constraints that need to be satisfied in order for the corresponding assignment statement to be reached. It is possible to narrow down the search space into a collection of the target architectural states that can satisfy a set of constraints across module boundaries. For the remainder of this paper, we will refer to each of these prospect architectural states as pStates.

Mutants will commonly have multiple constraints as their activation criteria that must be satisfied simultaneously. The set of constraints can originate from distinct module instantiations within the circuit implementation, but each of the prospect code paths has a scope that does not reach past its module instantiation. Therefore, each pState serves as a specific focal point for the constraint solver such that every constraint's possible solution space is directly specified by a unified CDG. Since each module instantiation contains a statement tree and contains references to all its embedded modules, then the collection of pStates can be generated as follows: Each module instantiation is responsible for creating a prospect code path for every constraint that resides inside itself. It is also responsible for generating the complete set of pStates from the set of prospect code paths that reside inside itself; these pStates have a domain that does not surpass its module's scope.

Each module instantiation uses the pStates it receives from its children to generate the pStates at its level of scope. When a module instantiation receives a pState from any of its children, it will first replace the child module's primary inputs with the corresponding local signals as specified by the port map. Then it will create a cross-product of the pStates from its child with its own (if any exist) into an expanded set of pStates. It does this by merging the data dependencies and control requirements from all its local pStates with those of its entire child's pStates to generate all acceptable merges. Once it merges its local pStates with those of all its children, each of these pStates encompasses all constraints that lie at or below this point in the module hierarchy. The pState framework allows us to apply the CDG solver and the circuit's HDL knowledge onto a given constraint to handle one time frame in the ATVG problem.

V. High-Level ATVG

It was the efficiency and effectiveness of our simulation strategy that inspired the development of MVP's efficient and effective constraint solver. Section IV introduced the deterministic circuit analysis methods that make up this constraint solver, and it is the purpose of this section to exploit MVP's simulation and circuit analysis abilities to generate a test sequence that exposes an optimal set of mutants after every ATVG iteration.

A. Identifying the Most Effective ATVG Goals

We can take advantage of the outcome of the "clustering-and-partitioning" technique introduced in Section III and [2] to produce an ATVG algorithm (Fig. 5) that gives priority to the activation of the partition with the highest density of

Precondition: Lp = list of all partitions from every cluster

```
ATVG-ITERATION(Lp)
1.  Sort Lp into descending order of member size
2.  P ← first partition in Lp
3.  SUCCESS ← false
4.  while P exists and SUCCESS = false
5.    if activation criteria for P is not met
6.      then TP ← generate activation pattern(s) for any
                  inactive error in P while dropping
                  errors from unsuccessful ATVG attempts
7.           if activation is successful
8.             then SUCCESS ← true
9.             else P ← next partition in Lp
10.     else TP ← generate propagation pattern(s) for any
                  active design error in P while dropping
                  errors from unsuccessful ATVG attempts
11.          if propagation is successful
12.            then SUCCESS ← true
13.            else P ← next partition in Lp
14. if SUCCESS
15.   then return TP
16.   else fail
```

Fig. 5. Algorithm for each ATVG iteration.

undetected mutants (deterministic-activation) and only performs deterministic-propagation (to any pre-designated observation point) in the case where probabilistic-propagation is insufficiently effective. Line 1 of the algorithm sorts the list of partitions into the order of descending member size to ensure that any unsuccessful attempt to generate a test for a partition *P* is followed by an attempt on the next best partition during the subsequent iteration of the while loop. Line 6 attempts to generate a test sequence that activates an inactive mutant in *P*, and any failed attempt results in the removal of that mutant from *P* (fault dropping). These dropped mutants are marked as *unexcitable*. Line 10 handles the case where the activation criteria for the partition *P* are already met, which is expected to happen whenever probabilistic propagation on the set of active mutants from *P* is insufficiently effective. Therefore line 10 is used to generate a test sequence that propagates an active mutant in *P* to a primary output, and any failed attempt results in the removal of that mutant from *P*. These dropped design errors are marked as *undetectable*.

At the start of the ATVG effort, it is expected that deterministic activation on the dominant partition will be effective in causing the detection of enough mutants from this partition so as to demote it from its dominant status. The probabilistic-propagation technique will continue to be effective for as long as there are enough mutants with simple propagation requirements. Whenever the ATVG algorithm encounters a partition that has an insufficient number of mutants with simple propagation requirements, the deterministic activation iteration can be followed by deterministic propagation iteration on the same dominant partition.

B. FSM Analysis for Test Sequence Generation

We can argue that a control signal is only dependent on a

subset of a design's internal registers. This encourages us to decompose an RTL implementation into a set of interacting FSMs, such that one FSM is generated per internal register. Given a set of simultaneous constraints, we can identify – using a corresponding pState – the set of registers (and their corresponding target values) that help satisfy these constraints. Thus the final step in ATVG requires us to trace the target values for these registers to their current simulation values by analyzing their corresponding FSMs.

Analyzing the solution space for multiple simultaneous FSMs at a time interval is similar to the problem of analyzing the solution space for multiple simultaneous constraints, such that contradictions cannot exist in the control values of the solution space. For multiple simultaneous FSMs, the control values that define the state transition of one FSM cannot contradict the control values that define the state transition for any other FSM in the same time frame. Even though this inter-dependency between FSMs does complicate the ATVG problem, we can exploit it to identify contradictions early in the search process, similar to the generation of a pState.

Finite state machines are often described as a directed graph, such that each state transition can be represented by a function $y = \delta(s, x)$ [6]. In this function, y represents the next state, s represents the current state, and x represents the input to the FSM. By using pStates, we are converting a set of simultaneous constraints to a target state y', and we are identifying the internal register values s and primary input values x that allow y' to be reached. This allows us to generate a test sequence by stepping backwards in time starting at y'. All FSM graph edges have equal weight, thus we are limited to employing either a depth-first-search (DFS) or a breadth-first-search algorithm when generating a test sequence. A breadth-first-search algorithm guarantee finding the shortest test sequence, but it will require a significant amount of extra memory to store all pending paths being searched. Therefore, our FSM search algorithm is best implemented using DFS, such that we specify a maximum path length l to eliminate lengthy solutions and limit the search space.

Concerning multiple simultaneous FSMs, we can adapt the DFS algorithm as shown in Fig. 6. In this algorithm, *TS* holds the test sequence that will be returned, *Y* holds the constraints to be satisfied (target state), *S* returns the control requirements that satisfy the constraints to *Y* (previous state to *Y*), and l is the size limit to the instruction sequence. This multi-FSM DFS algorithm gives us the advantage of only searching the relevant portion of a microprocessor's FSM, as it allows us to generate a test sequence using a subset of the microprocessor's registers.

To illustrate the operation of the multi-FSM DFS algorithm across multiple time frames, an example scenario is shown in Fig. 7. In this example, we begin from the right with two signal constraints $\{Sig_\alpha = y_\alpha, Sig_\beta = y_\beta\}$ whose data dependency is mapped to three registers $\{R_1 = y_1, R_2 = y_2, R_3 = y_3\}$ by a specific pState. From here on, the multi-FSM DFS algorithm identifies for each register an incoming transition that is compatible with all other registers' incoming

```
multiFSM_DFS(testSequence TS, constraintSet S,
             constraintSet Y, int l)
1.  If (l = 0) return FAIL
2.  If (Y ≠ Ø)
3.    let v ∈ Y
4.    U ← getAllIncomingTransitions(v)
5.    For each t ∈ U
6.      If (S = Ø)  T ← t
7.      Else        T ← S ∩ t
8.        If (T ≠ Ø && multiFSM_DFS(TS, T, Y– v, l)
              = SUCCESS)
9.          TS ← TS + getPrimaryInputs(S)
10.         return SUCCESS
11. Else
12.   TS ← TS + getPrimaryInputs(S)
13.   return SUCCESS
14. return FAIL
```

Fig. 6. Multi-FSM DFS algorithm for each ATVG iteration.

transitions, such that all transition information ($\delta(s, x)$ in Fig. 7) is combined to define the control space (*S* in Fig. 6) for that specific time frame. This control space consists of a set of register values and primary input values, thus it defines the state space for the previous time frame. At some intermediate time frames, this control space will introduce a dependency on a new register (i.e. introduction of R_4 @ t=2, R_5 @ t=1 in Fig. 7) whose FSM will also need to be analyzed. Similarly at some other time frames, this control space will no longer denote a dependency on a specific register (i.e. absence of R_3 @ t=0 in Fig. 7); this can happen at time frames when a data register is assigned the required data value. Once the reset state is reached, the recursive multi-FSM DFS algorithm reports the primary input values in chronological order as it returns; reporting a test sequence $\{X_0, X_1, X_2, X_3\}$ in the case of Fig. 7.

C. Test Generation Using Prospect States

If we carefully analyze the multiFSM_DFS algorithm of Fig. 6, we can see that lines 3-10 simply map a constraint set *Y* to any constraint set *S*, such that satisfying *S* results in *Y* as the next state; notice this is the inherent purpose of a pState. Knowing this, we can easily modify the multiFSM_DFS algorithm to use pStates when generating an instruction sequence. Doing this gives us the algorithm in Fig. 8, which is easier to understand, and its implementation works well with the definition and implementation of a pState. In line 4, we convert a set of constraints (target state) into a pState; this pState has a defined set of data dependencies, and needs to have its control requirements identified and solved. Line 4 solves this pState as discussed in Section IV, and stores all possible solutions into the set *P*. We only need to use one solution in *P*, therefore the FOR loop starting at line 5 continues to iterate until a solution is found or all entries in *P* have been explored. To explore the previous time frame, the control requirements from the current pState *t* are passed as the constraints to the next recursive call to the multiFSM_DFS algorithm. Finally, the recursive multi-FSM DFS algorithm reports the test sequence in chronological order; this happens

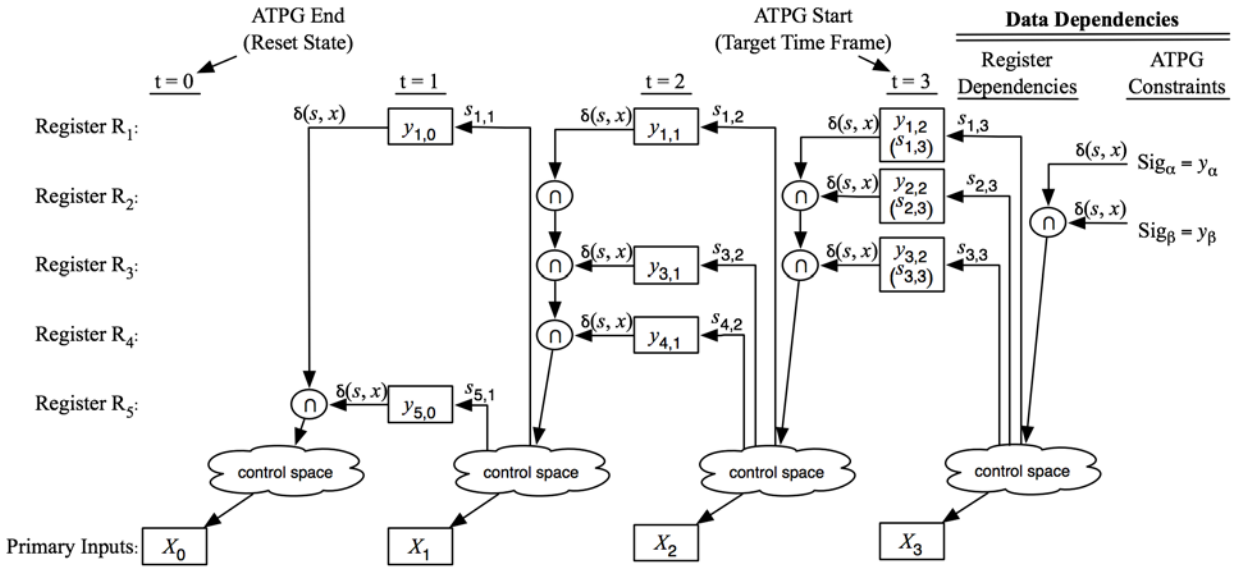


Fig. 7. Multiple-timeframe example for multi-FSM DFS algorithm.

in lines 8 and 9 of Fig. 8.

VI. Real-Time Circuit Profiling

The methods presented in Section V provide for an effective test pattern generator that is capable of exposing complex circuit design errors. Unfortunately, these methods used alone are burdened by the analysis of irrelevant HDL code segments, and by the traversal of already-explored architectural states. We can significantly improve MVP's run-time performance by implanting mechanisms that enable it to *learn* important details of the circuit under validation as a means to avoid irrelevant circuit scenarios. These mechanisms [5] can exist as a pre-processor that gathers circuit information prior to the circuit validation process, and can exist as run-time entities that allow MVP to learn from its experience.

A. Pre-processor Circuit Profiling

The pre-processor to MVP's circuit validation process should be a light-weight task that provides MVP with valuable insight capable of directing its test pattern generation process towards a solution. The pre-processor should not attempt to solve actual constraints, but rather solve early the sub-problems that provide MVP with the most valuable information. Instead of analyzing the implications that the circuit has onto each statement in the HDL as is done in the real-time circuit analysis process, the pre-processor should analyze the implications each statement has onto the overall circuit.

```
multiFSM_DFS(testSequence TS, pStateSet Y, int l)
1.  If (l = 0) return FAIL
2.  If (Y = reset state)
3.    return SUCCESS
4.  P ← solve(Y)        //Generates set of solutions
5.  For each t ∈ P
6.    S ← get_previous_timeFrame(t)
7.    If (multiFSM_DFS(TS, S, l-1) = SUCCESS)
8.      TS ← TS + getPrimaryInputs(S)
9.      return SUCCESS
10. return FAIL
```

Fig. 8. Using prospect states during ATVG.

Pre-processor circuit profiling concentrates on the following categories:

- *Assignment statement profiling*: Solving a constraint involves exploring all relevant assignment statements that can satisfy its unresolved data implications. Doing this requires a significant amount of work that is often repeated for a great deal of assignment statements that cannot help satisfy the constraint. Much of this dead-end work can be prevented by indexing each assignment statement with the identifier value implications that it has onto the hardware description.
- *Implicit memory profiling*: MVP explores all signals in the hardware description in search for implicit memory elements. It does this by negating the explicit guards to all assignment statements onto the signal being analyzed, and inserting them into a single conjunction (unified by Boolean AND operators). This process exploits MVP's efficient CDG solver, and a CDG that does not evaluate to false signifies an implicit memory element.
- *Basic-block guard profiling*: In most cases where a data contradiction is encountered when solving a constraint, the contradiction arises from the union of the guards in the multiple prospect code paths. Experiencing an identifier value contradiction within the guard of a basic block is significantly more costly than experiencing a contradiction within the statement itself because the aggregated guards leading up to a basic block is larger in most cases. Therefore, having pre-computed knowledge as to the constraints imposed by the guard of a statement can help reduce the number of pStates that are generated and computed.

B. Runtime Circuit Profiling

MVP's run-time circuit validation process should be a complete task focused on exploring uncharted territory within the processor. Complete FSM coverage commonly requires a significant amount of redundant state exploration. Therefore as MVP gets further into its validation process, it will be forced to retrace more of the previously-explored state space. Also, there are many architectural states that have a high occurrence frequency as they are a precursor to a wide range of other architectural states, thus retaining some of their pre-solved information can optimize MVP's performance in the long run.

Finite State Machine Profiling. When a specific target state can be reached by multiple states, we can use a weight scheme such that the state *s* with the lowest weight provides MVP with two advantages:

- When the pStates have never been explored (thus they are un-indexed), it will allow MVP to choose the state *s* with the least number of constraints that will need to be satisfied at the subsequent ATVG iteration.
- When any of the pStates has been previously explored, its weight will be lower than all unexplored pStates, and will provide MVP with guidance towards the reset state.

The aforementioned global FSM profiling effort is meant to interpret the low-level FSM profiling information and identify the shortest FSM path that can reach the circuit's reset state. A low-level FSM profiling effort is often focused on depositing information onto each statement in the hardware description during runtime to record its scope (range in values) and the success it provides (proximity to FSM reset state). Conversely, the global FSM profiling effort is focused on unifying the information gathered from all statement sources that represent a given solution as a means to avoid costly or irrelevant scenarios.

Explored State-Space Tracking. Preventing the ATVG algorithm from revisiting a pState that is visited earlier in the same test sequence will avoid analyzing FSM loops. Furthermore, preventing the ATVG algorithm from revisiting a pState that was visited by a previous test sequence branch that failed to generate a result will prevent analyzing unsuccessful paths more than once.

VII. Experimental Results

Results on MVP's effectiveness have been generated by following the key steps in a validation paradigm: coverage metric definition, error modeling, circuit simulation, and ATVG. The strategy for each of these steps has been covered by this paper in that order, and the results are provided in this section. All tests have been performed on a Dual 2.5GHz G5 workstation under OS X Tiger using gcc 4.0. MVP has been implemented as a library using GNU's autotools (autoconf, automake, libtool) in 20K physical lines of C++ code.

The collection of MCEs for the Motorola 6800 implementation [20] are generated to bind all possible combinations between the explicit *state* signal to all other control signals. The set of control signals also includes the *next_state* signal, which allows us to stimulate the data paths as well as the FSM transitions. Having a combination of values between a control signal and the explicit state as ATVG constraints allows us to stimulate every combination of the control signal's set of resulting data paths at every explicit microarchitectural state. Furthermore, having a combination of values between the *state* signal and the *next_state* signal allows us to stimulate every transition in the microarchitectural FSM.

The brute force approach of implementing the control-based coverage metric has resulted in a collection of 300,092 mutants. The activation criteria of each mutant is designated by (*i*) an activation criterion (described by a signal/value pair) for the explicit "state" signal, (*ii*) an activation criterion for one of the explicit control signals, and (*iii*) an error injection for the control signal that mutates it to a value other than it's activation criterion. From this exhaustive collection, MVP was able to easily identify that 287,565 mutants in the collection are irrelevant because the corresponding constraints are not supported by the M6800 implementation. These irrelevant mutants weren't a burden to MVP's runtime performance because they were each identified and removed in under 0.01 seconds.

MVP's true effectiveness is due to its ability to continuously traverse the unexplored portions of a circuit's architectural state-space. By applying a given set of mutants onto MVP's concurrent mutant simulator, we can directly compare the effectiveness in MVP's approach to the random methods commonly used to expose circuit design errors. Fig. 9 presents MVP's effectiveness at stimulating mutant peaks, resulting in a continuous mutant detection rate. After input vector 700, MVP has already stimulated every mutant by activating (or removing) it and is now making a second pass to attempt in exposing mutants in the remaining partitions.

Fig. 10 shows an effectiveness comparison between MVP's ATVG results and random ATVG results. From this figure, we can see that random ATVG is initially more effective than MVP's deterministic ATVG. The reason for this initial lead for random ATVG is because every input sequence generated

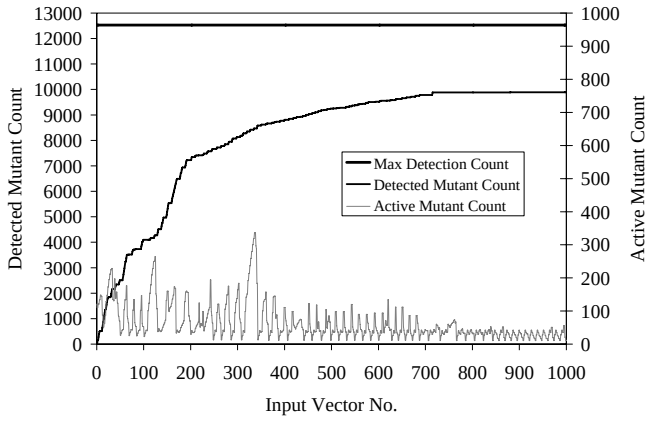


Fig. 9. MVP's ATVG effectiveness.

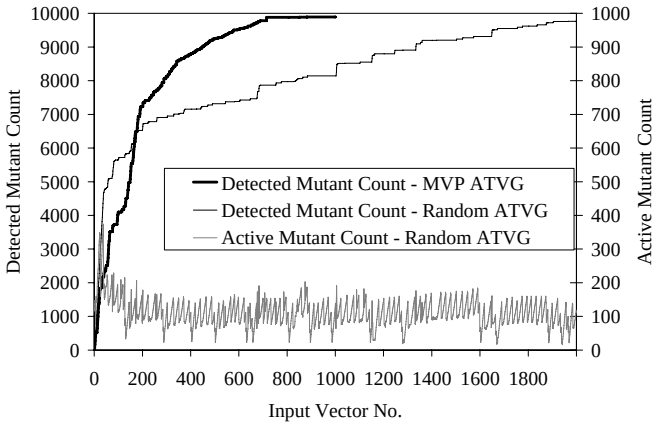


Fig. 10. MVP ATVG vs. random ATVG.

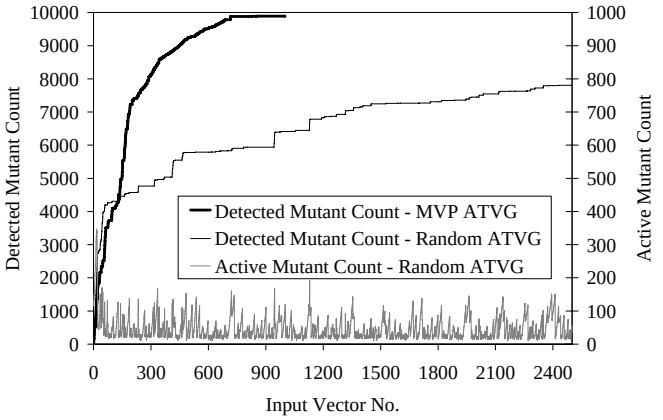


Fig. 11. MVP ATVG vs. random ATVG w/reset toggles.

by MVP begins at the circuit's reset state. As a result, the initial input vectors in a sequence from MVP traverse already-explored architectural states. Each of these input sequences has an average length of 15 vectors, resulting in a constant toggling of the reset signal. It is for this reason that the active mutant count of Fig. 9 (MVP's ATVG input stimuli) is so low in comparison to the active mutant count of Fig. 10 (random ATVG input stimuli).

MVP's use of the reset state as the common input sequence starting point provides us with two advantages: (*i*) the ATVG unit is able to perform real-time profiling such that each HDL line of code can hold its weight with respect to its known shortest distance to the reset state, and (*ii*) any given input sequence that exposes an actual circuit design error is self-sustained and can be utilized independently, as it begins with the circuit's reset state and ends when the circuit design error is exposed.

Now, if we look past input vector number 200 of the simulations in Fig. 10, we can see that MVP's deterministic ATVG is significantly more consistent and effective than a random ATVG approach. The results from the random ATVG simulation soon transforms into a linear trend that relies on sudden bursts of productivity, which only happens when the random input vectors happen to stimulate an unexplored portion of the state-space. These sudden bursts of productivity cannot be predicted, and are commonly a source of false-positives in circuit verification because of the inflection points that it introduces. MVP's deterministic ATVG, however, has *consistent* bursts of productivity due to MVP's closed-loop verification strategy between circuit simulation and automated ATVG.

The simulation results of Fig. 11 are provided to demonstrate MVP's effectiveness to navigate through a circuit implementation despite its dependency on the reset state. It compares MVP's detection rate (from Fig. 9) with the detection rate of a random ATVG effort where the reset signal is toggled at roughly the same rate as MVP's simulation run (every ~15 vectors). What Fig. 11 shows is that the random ATVG effort is initially decent as expected, but quickly has trouble in traversing a unique path in the circuit's state space following every toggle of the reset signal.

Without the use of its runtime profiling techniques, MVP was not effective at generating a meaningful input sequence because the runtime for each justification ATVG iteration exceeded 25,000 time frames. Therefore the purpose of MVP's pState weighing scheme is to help it forecast the easiest path to the circuit's reset state by selecting the pState with the least number of simultaneous constraints. This optimization alone has allowed MVP to achieve a feasible runtime by allowing it to satisfy all ATVG problems at fewer than 1000 time frames during FSM analysis. Fig. 12 is a testament to the significant contribution provided by MVP's weight estimation scheme, as it shows how manageable FSM traversal can be.

If we compare the number of time frames analyzed per ATVG iteration between the two graphs of Fig. 12, we can see that MVP's effectiveness in reaching the reset state is highly optimized by incorporating FSM weight indexing. By using weight indexing, the typical FSM search space was reduced from over 300 time frames down to approximately 100 time frames per justification ATVG iteration. There are still occasional justification problems that are difficult to solve as shown by the large bars around vectors 600 and 1000, but they do not dominate the problem space and their solutions

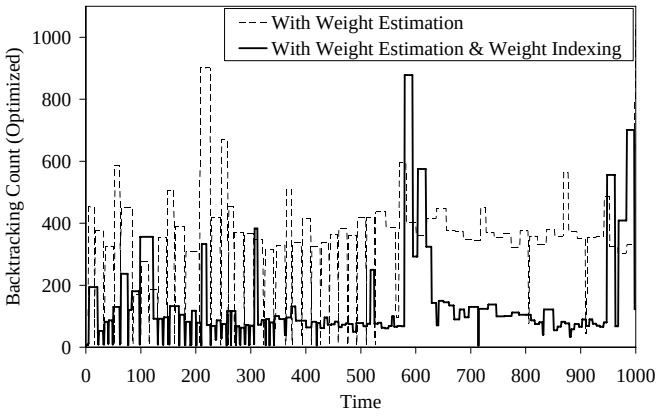


Fig. 12. ATVG search space with runtime FSM profiling.

contribute to MVP's FSM learning process.

VIII. Conclusions

We have presented a Mutation-based Validation Paradigm (MVP) technology that automates the design-to-verification process and contains fundamental techniques for analyzing high-level circuit implementations. MVP is unique in the way it exploits these techniques to validate circuit implementations. MVP's methods help deliver certainty into a circuit verification project in two ways: (*i*) it provides real-time observability into the validation effort through a concurrent mutant simulator that quantifies the circuit coverage (certainty level) at every simulation time-frame, and (*ii*) it employs deterministic circuit analysis techniques that, together with the observability provided by its concurrent mutant simulator, allow it's ATVG effort to consistently explore new corners in a circuit's architectural landscape. These contributions enable MVP to mitigate the risk of verification false-positives due to unexposed bugs, which is commonly encountered when random or pseudorandom ATVG fail to travel towards unexplored portions of the circuit under validation. Furthermore, the smooth slope of MVP's mutant detection rate allows a verification engineer to predict when enough circuit verification has been performed, given that further input vectors cannot promise much observability.

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. International High-Level Design Validation and Test Workshop*, 2004, pp. 81-86.

[3] J. Campos and H. Al-Asaad, "MVP: A mutation-based validation paradigm", *Proc. International High-Level Design Validation and Test Workshop*, 2005, pp. 27-34.

[4] J. Campos and H. Al-Asaad, "Search-space optimizations for high-level ATPG", *Proc. Microprocessor Test and Verification Workshop*, 2005, pp. 84-89.

[5] J. Campos and H. Al-Asaad, "Circuit profiling mechanisms for high-level ATPG", *Proc. Microprocessor Test & Verification Workshop*, 2006, pp. 9-14.

[6] F. Corno, U. Glaser, P. Prinetto, M.S. Reorda, H.T. Vierhaus, and M. Violante, "SymFony: A hybrid topological-symbolic ATPG exploiting RT-level information," *IEEE Transactions on Computer-Aided Design*, Vol. 18, pp.191-202, February 1999.

[7] A. Adir, E. Almog, L. Fournier, E. Marcus, M. Rimon, M. Vinov, and A. Ziv, "Genesys-Pro: Innovations in test program generation for functional processor verification," *IEEE Design and Test of Computers*, Vol. 21, pp.84-93, March-April 2004.

[8] F. Corno, E. Sanchez, M.S. Reorda, and G. Squillero, "Automatic test program generation: A case study," *IEEE Design and Test of Computers*, Vol. 21, pp.102-109, March-April 2004.

[9] S. Tasiran and K. Keutzer, "Coverage metrics for functional validation of hardware designs," *IEEE Design and Test of Computers*, Vol. 18, pp.36-45, July-August 2001.

[10] D. Moundanos, J.A. Abraham, and Y.V. Hoskote, "Abstraction techniques for validation coverage analysis and test generation," *IEEE Transactions on Computers*, Vol. 47, pp.2-14, January 1998.

[11] 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, pp. 67-81, February-April 2000.

[12] 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.

[13] E. Jenn, J. Arlat, M. Rimen, J. Ohlsson, and J. Karlsson, "Fault injection into VHDL models: The MEFISTO tool," *Digest of Papers: International Symposium on Fault-Tolerant Computing*, 1994, pp. 66-75.

[14] L. Berrojo, I. Gonzalez, F. Corno, M.S. Reorda, G. Squillero, L. Entrena, and C. Lopez, "New techniques for speeding-up fault-injection campaigns," *Proc. Design Automation and Test in Europe*, 2002, pp. 847-852.

[15] L.-C. Wang, M.S. Abadir, and J. 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.

[16] H. Al-Asaad, *Lifetime Validation of Digital Systems via Fault Modeling and Test Generation*, Ph.D. Dissertation, University of Michigan, Ann Arbor, September 1998.

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

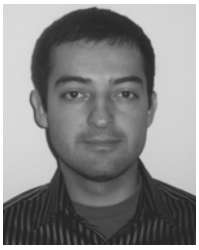
[18] F. Fallah, S. Devadas, and K. Keutzer, "OCCOM: Efficient computation of observability-based code coverage metrics for functional verification", *Proc. Design Automation Conference*, 1998, pp. 152-157.

[19] F. Dominique, B. Paul, and S. Jean-Francois, "Behavioral fault simulation: Implementation and experiments results", *Proc. IEEE International Workshop on Electronic Design, Test and Applications*, 2002, pp. 81-85.

[20] http://www.opencores.org/projects.cgi/web/system68/overview.

[21] 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.

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



Jorge Campos (S'00–M'07) received the B.S. degree in computer engineering in 2002, the M.S. and Ph.D. degrees in electrical and computer engineering in 2005 and 2007, respectively, all from the University of California, Davis, CA.

He is currently a Patent Engineer with the Park Vaughan & Fleming patent law firm in Davis, CA. He specializes in preparing and prosecuting patent applications on computer hardware and software technologies. His current research interests include design verification, fault tolerant computing, digital integrated circuits, and high-performance computing.

Hussain Al-Asaad (S'92–M'99–SM'05) received the B.E. degree (with distinction) in computer and communications engineering from the American University of Beirut, Lebanon, in 1990, the M.S. degree in computer engineering from Northeastern University, Boston, in 1993, and the Ph.D. degree in computer science and engineering from the University of Michigan, Ann Arbor, in 1998.

He is currently an Assistant Professor with the Department of Electrical and Computer Engineering, University of California, Davis, CA. His current research interests include design verification, testing, and fault tolerant computing. He has published over 35 papers in archival journals and refereed conference/workshop proceedings.

Prof. Al-Asaad is a member of Sigma Xi. He is a recipient of the National Science Foundation CAREER Award. He has served on the program committees of several conferences/workshops including the IEEE International High Level Design Validation and Test Workshop, and the IEEE International Workshop on Microprocessor Test and Verification.