Automated Debugging of SystemVerilog Assertions

Brian Keng[1], Sean Safarpour[2], Andreas Veneris[1,3]

***Abstract*—In the last decade, functional verification has become a major bottleneck in the design flow. To relieve this growing burden, assertion-based verification has gained popularity as a means to increase the quality and efficiency of verification. Although robust, the adoption of assertion-based verification poses new challenges to debugging due to presence of errors in the assertions. These unique challenges necessitate a departure from past automated circuit debugging techniques which are shown to be ineffective. In this work, we present a methodology, mutation model and additional techniques to debug errors in SystemVerilog assertions. The methodology uses the failing assertion, counter-example and mutation model to produce alternative properties that are verified against the design. These properties serve as a basis for possible corrections. They also provide insight into the design behavior and the failing assertion. Experimental results show that this process is effective in finding high quality alternative assertions for all empirical instances.**

I. Introduction

Functional verification and debugging are the largest bottlenecks in the design cycle taking up to 46% of the total development time [1]. To cope with this bottleneck, new methods such as assertion-based verification (ABV) [2], [3] have been adopted by the industry to ease this growing burden. ABV in particular has shown to improve observability, reduce debug time as well as improve overall verification efficiency [2]. However even with the adoption of ABV, debugging remains an ongoing challenge taking up to 60% of the total verification time [1].

Modern ABV environments are typically composed of three main components: design, testbench and assertions. Due to the human factor inherent in the design process, it is equally likely for errors to be introduced into any one of these components. Commercial solutions [4]–[6] aim to help the debugging process by allowing manual navigation and visualization of these components. Most existing research in automated debugging [7]–[11] have focused primarily on locating errors within the design. The absence of automated debugging tools targeting testbenches and assertions remains a critical roadblock in further reducing the verification bottleneck.

The adoption of assertions introduces new challenges to the debugging process. Modern temporal assertion languages such as SystemVerilog assertions and Property Specification Language [12], [13] are foreign to most design engineers who are more familiar with RTL semantics. Temporal assertions concisely define behaviors across multiple cycles and execution threads, which creates a significant paradigm shift from RTL. For example debugging the failing SystemVerilog assertion `req |=> gnt ##[1:4] ack`, requires the engineer to analyze four threads over five clock cycles to understand the failure. Moreover, a single temporal operator such as a non-consecutive repetition may map to a multiple line RTL implementation, adding to the debugging complexity. For these reasons, debugging complex temporal assertions remains one of the biggest challenges in their wide spread adoption.

Automated circuit debugging techniques have traditionally relied on localizing an error in a circuit. In a similar manner, it is possible to synthesize assertions [14]–[16] and allow one to apply similar circuit localization techniques to assertions. However, this proves ineffective in debugging assertions due to their compact nature, also shown later in the paper. For example, applying path-tracing [7] to the assertion `valid ##1 start |=> go`, will return the entire assertion as potentially erroneous. Moreover, this type of localization does not provide help in directing the engineer towards correcting it. This suggests an urgent need for a departure from traditional circuit debugging techniques so that we can debug assertions effectively.

In this work, we propose a novel automated debugging methodology for SystemVerilog assertions (SVA) that takes a different approach. It aids debugging by generating a set of properties closely related to the original failing assertion that have been validated against the RTL. These properties serve as a basis for possible corrections to the failing assertion, providing an intuitive method for debugging and correction. They also provide insight into design behavior by being able to contrast their differences with the failing assertion.

In summary, our major contributions are as follows:

- We introduce a language independent methodology for debugging errors in assertions that produce a set of closely related verified properties to aid the debugging process. These properties are generated by an input set of modifications that mutate existing assertions.
- We propose a particular set of modifications for the SystemVerilog assertion language to mutate the failing assertion in order to generate closely related properties.
- We introduce two techniques dealing with vacuity and multiple errors to enhance the practical viability of this approach.

An extensive set of experimental results are presented on real designs with SystemVerilog assertions written from their specifications. They show that the proposed methodology and modification model are able to return high quality verified properties for all instances. In addition, the multiple error and vacuity techniques are able to filter out inessential properties by an average of 23% and 34% respectively.

The remaining sections of the paper are organized as follows. Section II provides background material. Our contributions are presented in Section III, Section IV and Section V. Section VI presents the experimental results and Section VII concludes this work.

II. Preliminaries

This section gives a brief overview of the SystemVerilog assertions (SVA) language as well as concepts used extensively throughout this paper. For a more detailed treatment please refer to [12], [17]. SVA is a concise language for describing temporal system behavior for use in verifying RTL designs.

[1]University of Toronto, ECE Department, Toronto, ON M5S 3G4 ({briank, veneris}@eecg.toronto.edu)

[2]Vennsa Technologies, Inc., Toronto, ON M5V 3B1 (sean@vennsa.com)

[3]University of Toronto, CS Department, Toronto, ON M5S 3G4

TABLE I
COMMON SVA OPERATORS

seq_expr	::=	bool_expr
	|	seq_expr time_range seq_expr
	|	bool_expr bool_abbrv
	|	seq_expr seq_op seq_expr
	|	`first_match` (seq_expr)
	|	bool_expr `throughout` seq_expr
	|	...
time_range	::=	| ##k | ##[k_1:k_2]
bool_abbrv	::=	| [*k] | [*k_1:k_2] | [=k] | [=k_1:k_2]
		| [$->$ k] | [$->$ k_1:k_2]
seq_op	::=	`and` | `intersect` | `or` | `within`
prop_expr	::=	seq_expr
	|	seq_expr |$->$ property_expr
	|	seq_expr |=> property_expr
	|	property_expr `or` property_expr
	|	property_expr `and` property_expr
	|	...

Each system behavior can be specified by writing a SVA property which in turn can be used as an assertion.

Properties are derived from several different types of expressions that build upon each other. A *sequence expression* specifies the behavior of *Boolean expressions* over one or more clock cycles using operators such as delays and repetitions. These can be combined to define concise sequence expressions that define multiple linear sequences known as *threads*. For example, `start ##[1:3] stop` specifies three separate threads of `start` followed by `stop` with varying lengths of delay in between. If one of the threads evaluates to true then the sequence is said to *match*. Threads allow great ease in specifying common design behaviors but also lead to increase difficulty in debugging. A *Property expression* specifies more complex combinations of sequence expressions such as implications. A grammar of common SVA operators is listed in Table I.

The implication property operator (|$->$, |=>) is similar to an if-then structure. The left-hand side is called the *antecedent* and the right-hand side is called the *consequent*. With respect to the implication operator, a property is said to be *vacuously* [18] true if every evaluation of the antecedent is false. Example 1 gives an overview of how several common SVA operators are used.

Example 1 *Consider the specification: "If* `start` *is active,* `data` *repeats 2 times before* `stop` *and one cycle later* `stop` *should be low." We can interpret this as the waveform given in Figure 1 with respect to* `clk`*. This can be concisely written as the SVA property:*

```
start |=> data[->2] ##1 stop ##1 !stop
```

III. ASSERTION DEBUGGING METHODOLOGY

This section presents a methodology that automatically debugs errors in failing assertions. It is assumed that errors only exist in the assertions and the design is implemented correctly. This follows the convention in circuit debugging [7]–[11] literature where the verification environment is assumed to be correct. If this is not the case, then the methodology still can provide value for debugging by giving insight into the design behavior. Note that this methodology makes no assumptions about the assertion language or the types of errors as these are functions of the input model.

The methodology aids debugging by returning a set of verified properties with respect to the design that closely relate to the failing assertion. We denote this set as P. This set of closely related assertions aids in the debugging process in several ways. First, P serves as a suggestion for possible corrections to the failing assertion. As such, it provides an intuitive method to aid in the debugging and correction process. Second, since the properties in P have been verified, they provide an in depth understanding of related design behaviors. This provides critical information in understanding the reason for the failed assertion. Finally, P allows the engineer to contrast the failing assertion with closely related ones, a fact that allows the user to build intuition regarding the possible sources and causes of errors.

The overall methodology is shown in Figure 2 and consists of three main steps. After a failing assertion is detected by verification, the first step of applying *mutations* is performed. This step takes in the failing property along with the mutation model and generates a set of closely related properties, denoted as P', to be verified. Each property in P' is generated by taking the original failing assertion and applying one or more predefined modifications, or *mutations*, defined by the mutation model. This model defines the ability of the methodology to handle different assertion languages as well as different types of errors. We define a practical model for SVA in the next section but different models based on user experience are also possible.

The second step of the methodology quickly rules out invalid properties in P' through simulation with the failing counter-example. A counter-example in this context is a simulation trace that shows one way for the assertion to fail. The intuition here is that since the counter-example causes the original assertion to fail, it will also provide a quick filter for related properties in P'. It accomplishes this by evaluating each property in P' for each cycle in the counter-example through simulation. If any of the properties in P' fail for any of the evaluations, they are removed from P'. The resulting set of properties is denoted by P''.

The final step of the methodology uses an existing verification flow to filter out the remaining invalid properties in P''. This is the most time-consuming step in this process, which is the reason for generating P'. The existing verification flow can either be a high coverage simulation testbench or a formal property checker. In the case of the testbench, the properties in P will have a high confidence of being true. While with the formal flow, P is a set of proven properties for the design. In most verification environments, both these flows are automated resulting in no wasted manual engineering time. The final set of filtered properties P are verified by the environment and can be presented to the user for analysis.

IV. SYSTEMVERILOG ASSERTION MUTATION MODEL

This section describes a practical mutation model for SystemVerilog assertions to be used with the methodology described in Section III. These mutations are designed to model

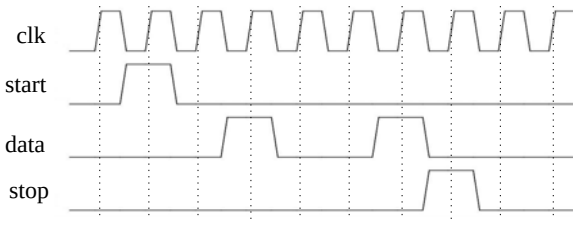

Fig. 1. Example 1: Timing Specification

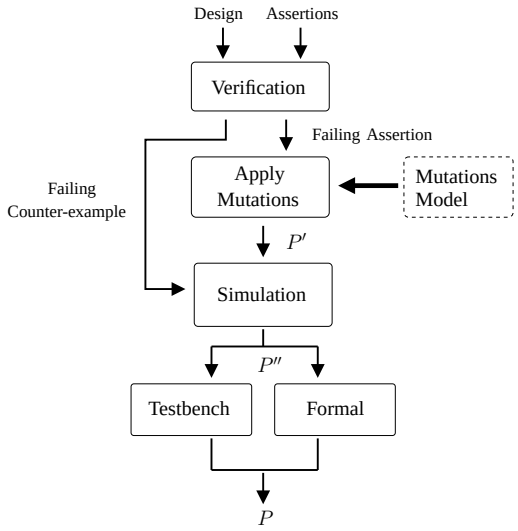

Fig. 2. Assertion Debugging Methodology

common industrial errors [2], [19] as well as misinterpretations of SVA [17]. This model is created based on our discussions with industrial partners, own experiences writing assertions, as well as common errors in cited literature. Note that other mutation models can be developed based on user experience.

Each mutation modifies the assertion either by adding operators, changing operators or changing parameters to operators. Each new property is generated by applying a fixed number of these mutations to the failing assertion. The number of mutations is defined to be the *cardinality* of the candidate and depends on the number of additions or changes to the assertion. In some cases, multiple or complex errors may require higher cardinality to model. The rest of the section will describe the various types of mutations in this model.

The first group of mutations involves modifying Boolean expressions. SVA provides operators for signal transitions across a pair of clock cycles and previous values. These operators can frequently be misused. For example, a common error occurs when interpreting the word "active". It is ambiguous whether the intent is a rising edge (`$rose(sig)`), or a level sensitive trigger (`sig`). Similarly with the `$past(sig,`k`)` operator, the number of cycles (k) to evaluate in the past is a common source of errors. The mutation can model this error by adding or subtracting an integer i. The following table gives the mapping from Boolean operators to the set of possible mutations where `<s>` is a given signal.

Name	Operator/Expression	Mutation
Boolean Expressions	`<s>`, `$past`, `$rose`, `$fell`, `$stable`, `$changed`	Replace with {`<s>`, `!<s>`, `$rose(<s>)`, `$past(<s>,` $k \pm i$`)`, `$fell(<s>)`, `$stable(<s>)`, `$changed(<s>)`}

The next group of mutations involve commonly used arithmetic, logical and bitwise operators. They replace such operators with their related counter-parts. For example, if `&&` is mutated into `||`, it may relax a condition to generate a passing property. This can provide insight into possible places where the error may have occurred. Another example might be replacing `>` with `>=` which might also correct a commonly made error. The mutations for these types are operators are in the table below.

Name	Operator/Expression	Mutation
Logical	`!`, `&&`, `||`	Replace with {`&&`, `||`}; Remove negation
Bitwise	`~`, `&`, `|`, $\wedge$	Replace with {`~`, `&`, `|`, $\wedge$}
Arithmetic	`+`, `-`	Replace with {`+`, `-`}
Equality	`<`, `<=`, `==`, `>=`, `>`, `!=`	Replace with {`<`, `<=`, `==`, `>=`, `>`, `!=`}

SVA sequential binary operators make up the next group of mutations. These operators have subtle differences that are easy to misinterpret. For example, `intersect` is similar to `and` except with the added restriction that the lengths of the matched sequences must be the same. Another example is replacing `intersect` with `within` which might produce a passing property that can give insight into the underlying error. The following table shows the possible mutations.

Name	Operator/Expression	Mutation
Sequential Binary Operators	`and`, `intersect`, `or`, `within`	Replace with {`and`, `intersect`, `or`, `within`}

The next group of mutations involve the sequential concatenation operator. This family of operators specifies a delay and it is frequently used in properties. Two types of delays are possible, a single delay or a range of delays. The mutations involve changing the delays by integers i or j, or changing a single delay into a ranged delay. When mutating using i or j, the cardinality will be increased by the absolute value of the integer. For example, changing `##1` to `##3` will increase the cardinality by 2. The following table describes the mutations.

Name	Operator/Expression	Mutation
Sequential Concatenation	`##`k_1, `##[`k_1`:`k_2`]`	Change single delay to range delay; Change constants to $k_1 \pm i$, $k_2 \pm j$

Sequential repetition operators are the next group of mutations. These operators allow repetition of signals in consecutive or non-consecutive forms with subtle differences. For example, the non-consecutive goto repetition `sig[->`k_1`]` is frequently confused with the `sig[=`k_1`]` operator. The difference between the two is whether the sequence ends with strictly k_1 matches, or if multiple cycles are allowed before the next occurrence. The mutations may either change the repetition operator or the number of repetitions performed using i and j. The following table shows the mutations.

Name	Operator/Expression	Mutation
Repetition	`[*`k_1`]`, `[*`k_1`:`k_2`]` `[=`k_1`]`, `[=`k_1`:`k_2`]` `[->`k_1`]`, `[->`k_1`:`k_2`]`	Replace op with {`[*]`, `[=]`, `[->]`}; Change constant to $k_1 \pm i$, $k_2 \pm j$

The last group of mutations involve the implication operators. This family of operators are often used because most properties are evaluations based on a condition. The first type of mutation is a change between the overlapping ($|-\!>$) and non-overlapping ($|=\!>$) implication. This accounts for when there is extra or missing delay between the antecedent and consequent. The next mutation extends this idea by allowing a multiple cycle delay after the antecedent with the addition of the `##`i operator. The third type of mutation in this group involves adding the `first_match` operator to the antecedent of the implication. This addresses a subtlety of SVA where the consequent of each matching antecedent thread must be satisfied. The `first_match` operator handles this subtlety by allowing only the first matching sequence to be used. The following table outlines the possible mutations.

Name	Operator/Expression	Mutation
Implication	`|=>, |->`	Replace with {`|=>, |->`}; Append `|=> 1'b1 ##`i; Add `first_match` on antecedent

The following example shows how mutations can be applied in the context of an entire property.

Example 2 *Consider the assertion described in Example 1. We can generate properties in P' by mutating the failing assertion with each of the applicable rules. Here we show several examples of mutated properties with different cardinalities.*

```
// Cardinality 1
P1:start |=> data[->2] ##2 stop ##1 !stop
P2:start |-> data[->2] ##1 stop ##1 !stop
// Cardinality 2
P3:$rose(start)|=> data[=2] ##1 stop ##1 !stop
P4:$rose(start)|=> data[->2] ##2 stop ##1 !stop
// Cardinality 3
P5:start |=> !data[->2] ##1 stop ##3 !stop
```

V. Practical Considerations and Extensions

The methodology outlined in Section III along with the model in Section IV generates a set of closely related properties, P. However practically speaking, they are only useful if the number of properties returned by the methodology is small enough to be analyzed by an engineer. Two techniques that greatly reduce the number of properties are discussed here.

The first technique deals with *vacuous assertions*. Assertions that are vacuous typically are considered erroneous since their intended behavior is not exercised. Similarly, all verified properties that are found to be vacuous for all evaluations are removed from P, reducing its size significantly as seen in the experimental results.

The second technique deals with *multiple cardinalities*. As the cardinality increases, the size of the mutated properties, P', increases exponentially. This may become unmanageable at higher cardinalities. To deal with this, P' can be reduced by eliminating properties with mutations that have been verified at lower cardinalities. For example if the property `P1` from Example 2 is found to be a verified property, it would remove `P4` from consideration since it contains the same mutation from `P1`. The intuition here is that the removed properties do not add value because they are more difficult to contrast with the original assertion. This proves to be very effective in reducing the size of P' for higher cardinalities by removing these inessential properties.

VI. Experiments

This section presents the experimental results for the proposed work. All experiments are run on a single core of a dual-core AMD Athlon 64 4800+ CPU with 4 GB of RAM. A commercial simulator or property checker is used for all simulation and verification steps. All instances are generated from Verilog RTL designs from `OpenCores` [20] and our industrial partners. SVA is written for all designs based on their specifications.

To generate unbiased results, we do not artificially insert errors directly into the assertions and then try to fix them. Instead, we add errors into the RTL to create a mismatch between the RTL implementation and SVA assertions. We then assume that the RTL is correct and the SVA is erroneous, to create a failing assertion. It should be noted that in some cases there may be no possible corrections to the SVA since the RTL error may drastically change the design behavior.

TABLE II
Instance, Localization and Supplemental Data

Instance Info				Localize		Supplemental		
Inst	gates (k)	dffs	SVA ops	sus	sus (%)	tb sim cyc	tb cyc (k)	form cyc
hp1	20.3	2164	12	9	75	20479	27	47
hp2	20.3	2164	16	13	81	20466	27	43
mips1	73.4	2670	19	17	89	25	2	N/A
mips2	73.4	2670	9	7	78	3393	12	N/A
risc1	15.8	1371	14	N/A		N/A		10
risc2	15.8	1371	16	N/A		N/A		14
spi1	1.6	132	12	6	50	35	7	27
spi2	1.6	132	12	6	50	34	7	26
tlc1	2.5	42	7	6	86	29	0.5	15
tlc2	2.5	42	14	7	50	495	0.5	11
usb1	39.2	2349	4	4	100	11	0.4	11
usb2	39.2	2349	16	11	69	219	0.4	9
vga1	89.4	17110	8	4	50	43	8377	7
vga2	89.4	17110	8	6	75	45	8424	7
wb1	5.2	96	6	3	50	13	5	N/A
wb2	5.2	96	7	6	86	15	5	N/A

The RTL errors that are injected are based on the experience of our industrial partners. These are common designer mistakes such as a wrong state transition, incorrect operator or incorrect module instantiation. It should be emphasized that RTL errors typically correspond to multiple gate-level errors.

To create instances for the experiments, for each RTL error, one assertion is selected as the mutation target among the failing assertions for a design. Each instance is named by appending a number after the design name. Table II shows the information for each of these instances. The table is divided into three parts. The first section shows instance information, while the other two parts show localization results and supplementary information. The first four columns show the instance name, the number of gates and state elements in the circuit followed by the number of operators plus variables in the assertion. The remaining columns are described in later subsections.

The following subsections presents two sets of experiments. The first subsection demonstrates the ineffectiveness of circuit based localization techniques in debugging assertions motivating the proposed methodology. While the second presents the experimental results from the proposed assertion debugging methodology.

A. Localization

To motivate and illustrate the impact of the proposed methodology, results of a circuit localization technique applied to debugging assertions are presented in this subsection. The instances from Table II that had simulation testbenches are used in these experiments. A path tracing [7] strategy is implemented to locate which operators or variables could be responsible for an error in the assertion. This was done by replacing variables or nodes with any constant values and evaluating if the property passed. The counter-example used in these experiments is generated from its simulation testbench.

The results of these experiments are presented in columns five and six of Table II. Column five describes the number of operators or variables that are potentially erroneous which we denote as *suspects*. Column six shows the suspects divided by the total number of operators and variables as a percentage.

From the table, we see that path tracing returns suspects covering a large part of the assertion for all instances. This

ranges from 50% for wb1 to 100% for usb1. On average, 71% of the assertion operators and variables are returned confirming the ineffectiveness of circuit debugging techniques for use with assertions. This motivates the need for the mutation based debugging methodology presented in this work.

B. Assertion Debugging Methodology

The experimental results from implementing the proposed methodology, mutation model and enhancements are presented in this section. For each instance, two sets of experiments are run. The first uses the accompanying simulation testbench to generate the initial counter-example and the final verification step. The counter-example is generated by stopping the simulation at the point of first failure, while the verification runs through the entire testbench. We refer to these experiments as *testbench*. The second set instead uses a formal property checker for these tasks and we refer to them as *formal*. Note that not all instances have both a testbench and formal environment, thus some entries will be not available and are denoted by N/A. In addition for the same instance, different environments may produce different results due to assumption constraints in formal which are not respected in testbench. Each instance is run through the methodology varying the cardinality from one to three. Table III shows the results of these experiments.

The first two columns show the instance name and cardinality. The next eight columns show the testbench experiments. Columns three and four show the size of the initial candidates, P', without and with the multiple cardinality enhancement from Section V, respectively. Columns five to seven show the number of passing assertions from P' after simulation with the failing counter-example (P''), the number of vacuous properties found in P'', and the final number of verified properties (P). Column eight shows the percent reduction of the cardinality and vacuity enhancements from the unoptimized P'. Column nine and ten show the counter-example simulation time followed by the total testbench verification time. The last eight columns show same respective results for the formal experiments. The last three columns of Table II present supplemental data regarding the number of clock cycles in the testbench counter-example, testbench final verification step and formal counter-example respectively. Run-times to create the mutated properties (P') take less than two seconds and are not shown in the table due to space considerations. Time-outs for each step of the methodology are set at 3600 seconds and are indicated by TO.

The applicability of the proposed technique is apparent when analyzing the final number of verified properties in P. Despite the large number of initial properties in P', the methodology successfully filters properties to a manageable size. This is important for debugging because if P is too large then it becomes impractical to use. Moreover we see that in the case of formal, most instances are able to return proven properties that precisely specify the behavior of the design.

The SVA mutation model also proves to be helpful in generating high quality properties that pass verification. For example, for hp1 one of the cardinality three properties is: `$rose(rd) && !tim_cas ##(1) !rd[*4] |-> $rose(wr_safe)`. The mutation here changed the repetition operator from `[*7]` to `[*4]`. This directly corresponds to the RTL error which changed the timing between the read and write phases. Showing a different case for spi2, a cardinality one solutions is: `wfre |-> $stable(rfwe)[-> 1] within $rose(state==1) ##0 (state==0)[-> 1]`. The mutation added `$stable` to rfwe. This gives insight into the design behavior where the error in the RTL is that rfwe does not toggle in the correct state.

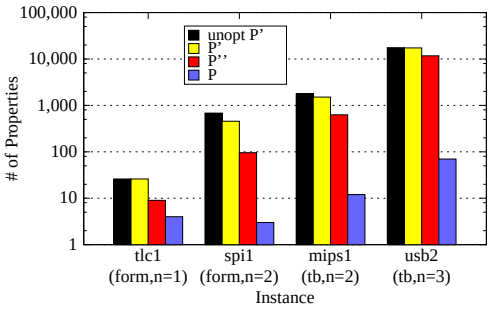

Fig. 3. Verified Properties for Several Instances

Figure 3 shows the size of each set of properties on a log-scale for several sample instances. From the figure, we see that the multiple cardinality technique from Section V helps to reduce the size of P' from the original unopt P'. This results in an average reduction across all instances of 23%. Next, we see that simulation with the counter-example efficiently reduces the size of P' to P'' by an average of 59% across all instances. This is critical to ensure that the run-time of the final verification step is minimized.

For columns nine and sixteen in Table III, we see that the number vacuous solutions also contributes to the reduced size of P from P''. As a percentage of P'', these verified vacuous properties represent an average of 34% of the set. In addition, the enhancements from Section V, shown in columns eight and sixteen, reduce the size of the unoptimized properties in P' by 27% across all instances.

Finally from Table II, we see that the run-time for the counter-example simulation and final verification step increases with the number of properties. For cardinality one, this does not significantly impact performance. For higher cardinalities, this becomes more costly causing time-out conditions in certain formal cases. This degraded run-time is the trade-off for being able to model more complicated types of errors. However, this can be avoided with more precise input mutation models so that a costly increase in cardinality is not needed.

VII. Conclusion

In this work, we present a methodology, mutation model and additional techniques to automatically debug errors in SystemVerilog assertions. The methodology works by using the assertion, counter-example and mutation model to generate a set of alternative properties that have been verified against the design. These properties serve as a basis for possible corrections as well as provide insight into the design behavior and failing assertion. Experimental results show the methodology is effective in generating high quality alternative properties for all empirical instances.

References

[1] H. Foster, "Applied assertion-based verification: An industry perspective," *Foundations and Trends in Electronic Design Automation*, vol. 3, no. 1, pp. 1–95, 2009.

[2] H. Foster, A. Krolnik, and D. Lacey, *Assertion-Based Design*. Kluwer Academic Publishers, 2003.

TABLE III
ASSERTION DEBUGGING METHODOLOGY RESULTS

Instance Info		Testbench								Formal							
inst	N	unopt P'	P'	P''	tb vac	P	% red	sim time (s)	tb time (s)	unopt P'	P'	P''	form vac	P	% red	sim time (s)	form time (s)
hp1	1	34	34	11	8	2	24%	1	2	34	34	20	5	0	15%	1	425
	2	529	467	220	177	0	45%	2	6	529	529	403	121	9	23%	1	TO
	3	5001	4129	2415	1824	38	54%	7	73	5001	4710	3801	0	0	6%	6	TO
hp2	1	52	52	17	11	3	21%	1	2	52	52	25	3	4	6%	1	454
	2	1257	1116	441	334	2	38%	3	14	1257	1116	615	69	3	17%	2	TO
	3	18748	15596	6521	4771	11	42%	54	197	18748	15682	8443	0	0	16%	55	TO
mips1	1	62	62	20	15	5	24%	1	2	N/A							
	2	1797	1514	627	611	12	50%	3	29	N/A							
	3	32338	24235	11811	11660	14	61%	84	594	N/A							
mips2	1	35	35	7	7	0	20%	1	2	N/A							
	2	539	539	181	173	3	32%	4	6	N/A							
	3	4791	4714	1952	1846	52	40%	26	91	N/A							
risc1	1	N/A								62	62	25	5	2	8%	1	49
	2	N/A								1797	1684	944	277	12	22%	3	853
	3	N/A								32338	28643	18290	0	0	11%	112	TO
risc2	1	N/A								53	53	36	11	1	21%	1	98
	2	N/A								1502	1450	1184	398	5	30%	2	TO
	3	N/A								28501	27109	24089	0	0	5%	78	TO
spi1	1	38	38	10	1	9	3%	1	1	38	38	13	1	7	3%	1	15
	2	682	400	33	24	5	45%	1	1	682	455	96	0	3	33%	1	42
	3	7638	3339	372	267	52	60%	5	8	7638	4091	0	0	4	46%	2	808
spi2	1	39	39	4	0	4	0%	1	3	39	39	6	0	2	0%	1	15
	2	693	527	2	0	0	24%	1	1	693	623	59	0	0	10%	1	62
	3	7445	4951	38	0	0	33%	7	3	7445	6339	497	0	0	15%	9	1648
tlc1	1	26	26	9	3	4	12%	1	1	26	26	9	2	4	8%	1	9
	2	285	203	67	40	10	43%	1	1	285	203	69	27	8	38%	1	16
	3	1706	896	250	201	0	59%	1	1	1706	920	267	137	0	54%	1	70
tlc2	1	42	42	13	1	2	2%	1	1	42	42	12	1	0	2%	1	9
	2	803	727	315	81	6	20%	1	2	803	803	345	48	16	6%	1	88
	3	9250	7737	3383	1279	103	30%	14	15	9250	8728	3687	0	0	6%	13	TO
usb1	1	13	13	3	0	3	0%	1	1	13	13	9	0	3	0%	1	17
	2	65	38	11	0	4	42%	1	1	65	38	24	0	3	42%	1	24
	3	163	62	28	0	6	62%	1	1	163	63	45	0	5	61%	1	42
usb2	1	51	51	20	16	0	31%	1	1	51	51	25	8	0	16%	1	23
	2	1206	1206	705	592	6	49%	2	3	1206	1206	840	318	13	26%	2	598
	3	17553	17301	11736	10239	70	60%	32	54	17553	16891	13134	0	0	4%	29	TO
vga1	1	19	19	6	0	4	0%	1	244	19	19	6	0	4	0%	1	14
	2	151	90	24	0	0	40%	1	526	151	90	23	0	0	40%	1	15
	3	655	278	96	0	9	58%	1	1691	655	278	92	0	1	58%	1	133
vga2	1	22	22	7	2	5	9%	1	248	22	22	13	1	0	5%	1	14
	2	223	132	38	13	4	47%	1	572	223	223	171	11	1	5%	1	37
	3	1358	567	170	40	2	61%	1	2172	1358	1341	1022	57	1	5%	2	620
wb1	1	25	25	7	5	1	20%	1	1	N/A							
	2	260	240	98	85	3	40%	1	1	N/A							
	3	1446	1238	616	560	9	53%	2	2	N/A							
wb2	1	22	22	6	4	2	18%	1	1	N/A							
	2	205	167	50	47	0	41%	1	1	N/A							
	3	1050	754	235	214	0	49%	1	2	N/A							

[3] B. Tabbara, Y.-C. Hsu, G. Bakewell, and S. Sandler, "Assertion-Based Hardware Debugging," in *DVCon*, 2003.

[4] Y.-C. Hsu, B. Tabbara, Y.-A. Chen, and F. Tsai, "Advanced techniques for RTL debugging," in *Design Automation Conf.*, 2003, pp. 362–367.

[5] R. Ranjan, C. Coelho, and S. Skalberg, "Beyond verification: Leveraging formal for debugging," in *Design Automation Conf.*, jul. 2009, pp. 648–651.

[6] M. Siegel, A. Maggiore, and C. Pichler, "Untwist your brain - Efficient debugging and diagnosis of complex assertions," in *Design Automation Conf.*, jul. 2009, pp. 644–647.

[7] S. Huang and K. Cheng, *Formal Equivalence Checking and Design Debugging*. Kluwer Academic Publisher, 1998.

[8] A. Veneris and I. N. Hajj, "Design Error Diagnosis and Correction Via Test Vector Simulation," *IEEE Trans. on CAD*, vol. 18, no. 12, pp. 1803–1816, 1999.

[9] A. Smith, A. Veneris, M. F. Ali, and A. Viglas, "Fault Diagnosis and Logic Debugging Using Boolean Satisfiability," *IEEE Trans. on CAD*, vol. 24, no. 10, pp. 1606–1621, 2005.

[10] K.-h. Chang, I. L. Markov, and V. Bertacco, "Automating post-silicon debugging and repair," in *Int'l Conf. on CAD*, 2007, pp. 91–98.

[11] G. Fey, S. Staber, R. Bloem, and R. Drechsler, "Automatic fault localization for property checking," *IEEE Trans. on CAD*, vol. 27, no. 6, pp. 1138–1149, 2008.

[12] "IEEE Standard for System Verilog-Unified Hardware Design, Specification, and Verification Language," *IEEE STD 1800-2009*, pp. 1–1285, 2009.

[13] "IEEE Standard for Property Specification Language (PSL)," *IEEE Std 1850-2010*, pp. 1–171, apr. 2010.

[14] S. Das, R. Mohanty, P. Dasgupta, and P. Chakrabarti, "Synthesis of System Verilog Assertions," in *Design, Automation and Test in Europe*, vol. 2, mar. 2006, pp. 1–6.

[15] J. Long and A. Seawright, "Synthesizing SVA local variables for formal verification," in *Design Automation Conf.*, 2007, pp. 75–80.

[16] M. Boulé and Z. Zilic, *Generating Hardware Assertion Checkers: For Hardware Verification, Emulation, Post-Fabrication Debugging and On-Line Monitoring*. Springer Publishing Company, Incorporated, 2008.

[17] S. Vijayaraghaven and M. Ramanathan, *A Practical Guide for SystemVerilog Assertions*. Springer, 2005.

[18] M. Samer and H. Veith, "Parameterized Vacuity," in *Formal Methods in CAD*, 2004, vol. 3312, pp. 322–336.

[19] S. Sutherland, "Adding Last-Minute Assertions: Lessons Learned (a little late) about Designing for Verification," in *DVCon*, 2009.

[20] OpenCores.org, "http://www.opencores.org," 2007.