An Automated Framework for Correction and Debug of PSL Assertions

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

***Abstract*—Functional verification is becoming a major bottleneck in modern VLSI design flows. To manage this growing problem, assertion-based verification has been adopted as one of the key technologies to increase the quality and efficiency of verification. However, this technology also poses new challenges in the form of debugging and correcting errors in the assertions. In this work, we present a framework for correcting and debugging Property Specification Language assertions. The methodology uses the failing assertion, counter-example and mutation model to produce alternative properties that are verified against the design. Each one of these properties serve as a basis for possible corrections. They also provide insight into the design behavior and the failing assertion that can be used for debugging. Preliminary experimental results show that this process is effective in finding alternative assertions for all empirical instances.**

I. Introduction

Functional verification and debugging remains one of the largest challenges in modern VLSI design flows taking up to 46% [1] of the total design time. In recent years, new methods have been developed to cope with the verification challenge. To increase observability in order to reduce overall debug time, assertion-based verification (ABV) [2], [3] has been adopted to improve the verification and debug efficiency. Despite this fact, debugging still remains a significant bottleneck taking up to 60% of the total verification time [1].

Modern ABV verification environments are typically made up of three components: design, testbench and assertions. Bugs can be introduced into any one of these components as an engineer is as likely to make a mistake in writing the design as they are in the testbench or assertions. Most debugging solutions have either focused on providing greater efficiency for manual debug [4]–[6] or have targeted automated solutions for the design [7]–[10]. The inability of current solutions to provide automation in debugging testbenches and assertions remains a roadblock to reducing the overall debug efficiency.

Temporal assertions present a unique challenges to the debugging process over traditional circuit debugging. Modern temporal assertion languages such as Property Specification Language (PSL) and SystemVerilog assertions [11], [12] introduce new constructs and semantics compared to traditional RTL languages. The complex temporal behaviors over multiple cycles and execution threads makes it difficult for engineers to write high-quality bug-free assertions. This fact leads to one of the biggest challenges in their wide spread adoption.

Localizing the error has traditionally been the main goal of automated circuit debugging techniques. This proves effective when the circuit has many components and most of the time is spent locating erroneous components. In a similar way, it is possible to synthesize assertions [13], [14] and apply similar localization techniques. However, this approach is less effective for debugging assertions. For example, applying path-tracing [7] to {`valid; start`} `-> next(go)` will return the entire assertion as potentially erroneous. In addition, localization does not directly help the engineer towards correcting the assertion. This suggests a need to develop new techniques beyond localization in order to effectively debug assertions.

In this work, we propose a novel framework for correcting and debugging PSL assertions that takes a different approach from previous circuit debugging techniques. It aids correction and debugging by generating a set of closely related properties to the failing assertion that have been validated against the RTL. These properties provide both suggestions for possible corrections as well as provide insight into the design behavior. This is accomplished by introducing a language independent methodology for correction and debugging of errors in assertions that produce a set of closely related verified properties. These properties are generated by an input set of modifications that mutate existing assertions. Additionally, we propose a particular set of modifications for PSL to mutate the failing assertion in order to generate closely related properties. Two techniques dealing with vacuity and multiple errors are also presented to enhance the practical viability of this approach.

Preliminary experimental results on real designs with PSL assertions written from their specifications are presented. They show that the proposed methodology and modification model are able to return verified properties for all instances. In addition, the extensions to deal with multiple error and vacuity techniques are shown to provide value in filtering out inessential properties.

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 Property Specification Language (PSL) as well as concepts used extensively throughout this paper. For a more detailed treatment please refer to [2], [12].

PSL is a concise language for describing temporal system behavior for use in verifying RTL designs. Each system behavior can be specified by writing a 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 *sequential extended regular expression* (SERE) describes a behavior of one or more cycles over *Boolean expressions* such as delays or repetitions. A *property* specifies temporal relationships among various Boolean expressions, SEREs and other properties. A simplified grammar of common PSL operators is listed in Table I.

[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 PSL OPERATORS

SERE	::=	boolean
	|	SERE ; SERE
	|	SERE : SERE
	|	Comp_SERE
Comp_SERE	::=	Repeated_SERE
	|	Comp_SERE | Comp_SERE
	|	Comp_SERE & Comp_SERE
	|	Comp_SERE && Comp_SERE
	|	Comp_SERE within Comp_SERE
	|	...
Repeated_SERE	::=	Boolean Repeat_op
Repeat_op	::=	[*k] | [*k_1:k_2] | [=k] | [=k_1:k_2]
	|	[$->$ k] | [$->$ k_1:k_2]
Property	::=	Boolean
	|	{ SERE }
	|	always Property
	|	never Property
	|	next Property
	|	next_e Property
	|	next_a Property
	|	Property -> Property
	|	...

Informally, a property is *vacuously* true if it only passes for some trivial or unintended reason. For example, when the left hand side of the logical implication operator ($->$) is always false, the right hand side never gets evaluated thus the property is vacuous.

III. ASSERTION CORRECTION AND DEBUGGING METHODOLOGY

This section presents the methodology to automatically generate a set of verified properties to aid in correction and debugging for errors in failing assertions.

In this methodology, it is assumed that errors only exist in the assertions and the design is implemented correctly. 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 returns a set of verified properties (P) with respect to the design that closely relate to the failing assertion. 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 correction and debugging 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 1 and consists of three main steps. The first step applies *mutations* which are performed after a failing assertion is detected by verification. 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 PSL in the next section but different models based on user experience are also possible.

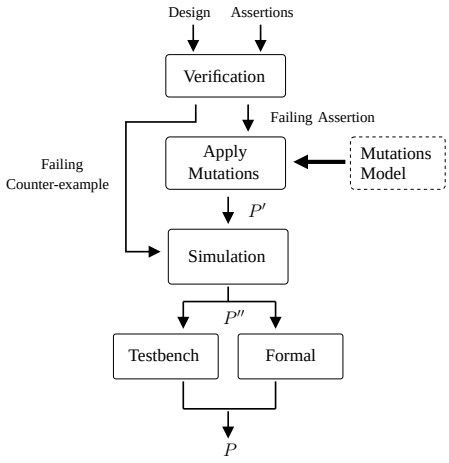

Fig. 1. Assertion Debugging Methodology

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. PSL ASSERTION MUTATION MODEL

This section describes a practical mutation model for the PSL assertions to be used with the methodology described in Section III. These mutations are designed to model common industrial errors [2], [15] as well as misinterpretations of PSL. 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 intuition behind this strategy is that the engineer typically has written the assertion so that it is syntactically similar to the correct one. This means that it is not necessary to explore every possible combination of expressions, only those that are similar to the failing assertion. As previously

TABLE II
PSL MUTATION MODEL

Name	Operator/Expression	Mutation
Boolean Expressions	<s>, prev, rose, fell, stable,	Replace with {<s>, !<s>,rose(<s>), prev(<s>, $k \pm i$), fell(<s>), stable(<s>),
	!,&&,||	Replace with {&&,||}; Remove negation
	~,&,|,∧	Replace with {~,&,|,∧}
	+,-	Replace with {+,-}
	<,<=,==,>=,>,!=	Replace with
		<,<=,==,>=,>,!=
Sequential Binary Operators	|, &, &&, within	Replace with {|, &, &&,within}
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 $k_1 \pm i$, $k_2 \pm j$
Simple SERE	;, :	Change operator to {;, :}; Add a delay after operator { [*1] ; }
Property Operators	next [k] next_e [k_1:k_2] next_a [k_1:k_2]	Change constant to $k_1 \pm i$; Change constant to $k_1 \pm i$, $k_2 \pm j$; Change operator to {next_e, next_a}
	->	Append -> next [i];

mentioned, even if the set of mutations is not exhaustive for every possible error, they still can provide value when debugging.

Table II lists the different types of mutations in this model. The three columns list the name, type of source operator and the mutation that is applied to the source operator. The various types of mutations are broken up into several types which will described below.

The first group of mutations involves modifying Boolean expressions. PSL provides built-in operators for signal transitions across a pair of clock cycles and previous values. These operators can frequently be misused. For example the `prev(sig,`k`)` operator, the number of cycles (k) to evaluate in the `prev` 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.

PSL sequential binary operators make up the next group of mutations. These operators have subtle differences that are easy to misinterpret. For example, `&&` is similar to `&` except with the added restriction that the lengths of the matched sequences must be the same. The following table shows the possible mutations.

Sequential repetition operators are the next group of mutations. These operators allow repetition of signals in consecutive or non-consecutive forms with subtle differences. Mutations either involve changing the repetition operator or changing the number of repetitions by integers i or j. 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.

The next group of mutations involve the concatenation (`;`) and fusion (`:`) operators used to define a single thread of behavior for SERE. These mutations either change the operator used, or append a delay to model any errors in timing. The following table shows the mutations.

The last group of mutations involve the property layer operators. The first type of mutations involve the `next` family of operators. These mutations aim to ensure the correct operator is used (`next_a` vs. `next_e`) or modify the number or range on the statement. The second type of mutation involve the implication operator. In this case, the mutation allows a multiple cycle delay after the antecedent with the addition of `next [`i`]` operator. The following table outlines the possible mutations.

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. 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 experimental results for the proposed PSL correction and debug framework. The experiments are run on a single core of a dual-core AMD Athlon 64 4800+ CPU with 4 GB of RAM. A commercial simulator and property checker were used for the simulation and verification steps. The instances were generated from Verilog RTL designs from `OpenCores` [16] and our industrial partners. The PSL assertions are written from the specifications of the design.

To remove bias from the results, we do not artificially insert errors into the assertions. Instead, we insert an error into the RTL so that there is a mismatch between the RTL and PSL assertions. We then assume the RTL is correct and PSL is erroneous and try to correct it. For each RTL error, an instance is created by selecting a failing assertion to be the mutation target of our methodology. Each instance is named by the design name followed by a number.

Table III shows the experimental results. The first four columns show the instance name, number of synthesized gates and state elements and the number of variables or operators in the assertion. The next ten columns show the results from the experiments. Column five shows the cardinality, while columns six and seven show the number of initial candidates with and without the cardinality optimization from Section V respectively. Columns eight and nine show the number of cycles in the counter-example, the number of passing properties after simulation with the counter-example, the number of proven vacuous properties, and the final set of verified

TABLE III
PSL CORRECTION AND DEBUGGING METHODOLOGY RESULTS

Instance Info				Results									
inst	gates	DFFs	nodes	N	unopt P'	P'	form cyc	P''	vac	P	% red	sim time (s)	form time (s)
pipeline1	3.2	228	10	1	29	29	29	29	2	6	7%	0.44	20.89
	3.2	228	10	2	366	228	29	228	29	3	46%	0.54	117.39
	3.2	228	10	3	2643	1238	29	1238	170	6	60%	1.12	1392.15
pipeline2	3.2	228	9	1	25	25	32	20	0	1	0%	0.46	20.22
	3.2	228	9	2	267	243	32	174	0	3	9%	0.54	103.09
	3.2	228	9	3	1591	1297	32	907	0	10	18%	1.2	1348.78
pipeline3	3.2	228	11	1	32	32	32	27	0	2	0%	0.48	23.2
	3.2	228	11	2	454	398	32	302	0	2	12%	0.66	385.22
	3.2	228	11	3	3762	3030	32	2230	0	0	19%	2.82	TO
risc1	15.8	1371	16	1	48	48	17	21	3	2	6%	0.6	22.4
	15.8	1371	16	2	1071	983	17	507	101	2	18%	1.16	668.92
	15.8	1371	16	3	14766	12854	17	7114	0	0	13%	30.89	0
risc2	15.8	1371	19	1	55	55	19	55	13	1	24%	0.54	106.49
	15.8	1371	19	2	1421	1421	19	1421	0	0	0%	1.64	TO
	15.8	1371	19	3	22946	22946	19	22946	0	0	0%	124.04	1460.81
tlc1	2.5	42	9	1	26	26	17	26	7	6	27%	0.44	9.73
	2.5	42	9	2	288	168	17	165	62	0	63%	0.53	22.02
	2.5	42	9	3	1779	783	17	764	227	0	69%	0.9	140.62
tlc2	2.5	42	15	1	42	42	16	3	1	0	2%	0.48	6.11
	2.5	42	15	2	803	803	16	145	42	20	5%	2.04	19.93
	2.5	42	15	3	9252	8605	16	1682	0	0	7%	28.54	0
tlc3	2.5	42	14	1	39	39	14	6	1	2	3%	0.46	6.24
	2.5	42	14	2	686	618	14	121	33	30	15%	1.25	15.98
	2.5	42	14	3	7194	5369	14	997	0	0	25%	12.01	0
usb1	39.2	2349	6	1	14	14	16	14	0	4	0%	0.7	13.06
	39.2	2349	6	2	80	42	16	42	0	3	48%	0.72	21.86
	39.2	2349	6	3	251	88	16	88	0	4	65%	0.73	28.67
usb2	39.2	2349	16	1	45	45	14	46	5	0	11%	0.71	16.99
	39.2	2349	16	2	928	928	14	927	166	10	18%	1.35	742.69
	39.2	2349	16	3	11621	11269	14	11194	0	0	3%	25.74	3373.3

properties respectively. The final three columns show the percent reduction of the cardinality and vacuity enhancements, counter-example simulation time and property checker proving time. The generation of the candidates are all under one second and are not included in the table. In addition, time-outs for each step of the methodology is set at 3600 seconds and is indicated by a TO.

From Table III, verified properties (P) are found for each of the instances when cardinality increased from one to three. An important point is that the size of P is relatively small ensuring that it can be practically analyzed by a human. Moreover, we also notice that each one of these properties is proven with respect to the design. This can be used to help with debugging by understanding the design, even if they do not directly correct the error.

An example of a correction for `risc2` is: {`(instr0 == igoto) && valid; !valid; 1'b1[*2]; valid`} `|-> prev(pc,2) <= prev(tmp,3)`. The RTL bug that was inserted in this case was one of the previous pipeline stages of `pc` was incorrectly concatenated. This shows up indirectly in the verified property but clearly points to a problem with the consequent signals.

The effectiveness of the vacuity and cardinality enhancements can also be seen when analyzing the columns labeled unopt P', P', and vac. The reduction in some cases is quite significant such as in `pipeline1` where it reduces 60% of the candidates when $N = 3$. The average reduction from both enhancements is 20% showing the benefit of the technique.

VII. CONCLUSION

In this work, we present a framework for debugging and correcting PSL assertions. It uses the failing assertion, corresponding counter-example along with a mutation model to produce closely related assertions that are verified against the design. These closely related properties serve as a basis for both correction and debugging of the erroneous assertions. Experimental results show that the framework can find alternative properties for all 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.

[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. 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.

[9] 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.

[10] 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.

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

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

[13] 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.

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

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

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