Assertion Checkers – Enablers of Quality Design

Marc Boulé and Zeljko Zilic, *Member, IEEE*

***Abstract*—This paper outlines the MBAC tool for the generation of assertion checkers in hardware. We begin with a high-level presentation of the automated compilation of assertions into checkers, and proceed to overview the multitude of applications of resource-efficient circuit-level checkers in the field of logic design and verification. A summary of experimental results is also given to show the current state of the MBAC tool, compared to the best known checker generator from IBM.**

***Keywords*—Assertion, Checker, Design, Verification, Debug.**

I. Introduction

HIGH-quality design is of paramount importance in the semiconductor industry. With the development cost of integrated circuits in the millions of dollars, delayed time to market, product recalls and loss of market shares are but some of the consequences of providing faulty designs. In mission-critical applications the consequences can be more dire especially when human lives enter the equation. With the growing complexity of designs (billions of transistors in a chip), increasingly a realistic goal is not the elimination of all errors but rather the reduction of critical design faults. Complexity of verification is increasing at a rate higher than the design complexity itself; in the industry this is referred to as the verification gap.

Assertion-based verification has started to emerge as a valuable paradigm in countering the increased difficulties with verification. Assertions are a means of formally specifying correctness properties of a design, and are expressed in high-level verification languages (HVLs). While HVLs differ from traditional hardware design languages, the new standards for VHDL, as well as SystemVerilog, include full assertion languages—PSL and SVA, respectively. Assertions are important as a type of formal documentation, but their real benefits are exploited when they can be processed by EDA (Electronic Design Automation) tools. Running a simulation, for example, wherein the simulator understands the assertions and flags their violations, greatly assists in debugging; however, the power of assertions can not typically be exploited outside the realm of software-based verification tools (simulators, model checkers).

This research widens the scope of applicability of assertions by providing the efficient means to automatically generate assertion-checking circuits from the assertions. Once assertions are converted into circuits, assertion checkers can be used in a variety of applications outside the traditional simulation and formal verification techniques. For example, instrumenting a design with a set of checkers allows the power of assertions to be used in prototype verification and debugging. These areas are notoriously difficult for debugging since the observability of the design's signals is relatively low. This research can also benefit software-based verification tools by allowing them to seamlessly support assertions, even when not designed to. In another application, checker generation allows for a form of high-level design whereby assertion specifications are converted into design modules automatically. Increasing the abstraction level of design practices has been a recurrent theme in EDA, and the proposed tool finds applications there as well.

Efficient assertion checker synthesis is critical for the acceptance of assertion-based techniques in a number of applications, as the assertions could take enormous resources if not handled well. This paper outlines the wide array of applications of checker synthesis, which is done for the first time (to the authors' knowledge) in a single paper. In all applications, the common theme is that of integrating checkers in an efficient manner in order to enable and extend the many benefits of assertions.

M. Boulé and Z. Zilic are with the Department of Electrical and Computer Engineering, McGill University, 3480 University Street, Montreal, Quebec H3A 2A7, Canada, Tel.: 514–398–7110, Fax: 514–398–4470, e-mails (first is contact author): marc.boule@elf.mcgill.ca, zeljko.zilic@mcgill.ca

II. Assertions and Assertion Checkers

Assertion languages allow the specification of complex temporal expressions (that do not lend themselves directly to hardware implementations) in a compact and elegant form. Assertions are high-level statements built on temporal logic that formally express the correctness properties of a specification.

In Assertion-Based Verification (ABV), the observation of an assertion failure helps to identify design errors, which are then used as a starting point for the debugging process. Hardware assertions are usually written in PSL (Property Specification Language, IEEE Std. 1850–2005) or SVA (SystemVerilog Assertions, part of the SystemVerilog language, IEEE Std. 1800–2005).

At the core, assertion languages are based on temporal logic languages (most notably LTL) and regular expressions, augmented by a number of "sugaring" operators. Sequential regular expressions are used to specify chains of events of Boolean expressions. Property operators such as `always`, and the suffix implication `|->` are used to model how sequences and Booleans should behave.

For example, consider the sequences in the PSL bus protocol assertion (`;` is temporal concatenation):

$$\texttt{assert always } (\{!req\ ;\ req\} \mid\!\!\rightarrow \{!gnt[*0{:}15]\ ;\ gnt\});$$

Whenever a bus request is issued, a bus grant must be given within 16 clock cycles. More specifically, whenever the sequence in the antecedent of the implication occurs, the sequence in the consequent must be observed or else the assertion has failed. The implication is under the scope of the `always` keyword, thus it is continually checked. PSL and SVA are explained in more detail in books [1] and [2].

978-1-4244-2921-9/08/$25.00 ©2008 IEEE

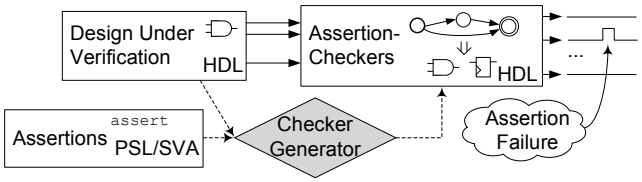

Fig. 1. Checker Generator for Hardware Verification.

To allow assertions to be processed in hardware, a *checker generator* is used to produce assertion checkers [3], [4], [5], which are typically expressed in a Hardware Description Language (HDL). An assertion *checker* is a circuit that captures the behavior of a given assertion, and can be included in the design for in-circuit assertion monitoring.

Figure 1 shows a high-level view of the assertion-based verification methodology, and the roles played by the assertions, the checkers and the checker generator. At the left of the figure are the given inputs to the tool, namely the Design Under Verification (DUV) and the assertions. In this example the circuit is described in a HDL and the assertions are specified in PSL or SVA. The checker generator produces an assertion circuit (a checker) for each input assertion.

A. Checker Generation Techniques

There are two other stand-alone tools in the literature for generating hardware checkers (PSL only). IBM's FoCs Property Checkers Generator [3], [6] (v2.04) is the oldest such tool. Concurrently to us, the HORUS checker generator is being developed at the TIMA laboratory [5], and is based on a library of primitive digital components for PSL operators. A modular approach was employed in a previous version of our tool, and also in the HORUS tool, whereby sub-modules for each property operator are built and interconnected according to the expression being implemented.

As visible in Figure 1, in the current version of our checker generator (called MBAC) assertions are transformed into an intermediate representation in automaton form (the graph in the top-right box), for subsequent conversion into RTL (Register Transfer Level). Automata for assertions extend the classical automata theory algorithms [7] with many new algorithms to support the vast array of assertion operators [8]. A critical aspect of our automata algorithms for generating hardware checkers is the novel use of *partial nondeterminism* and the unique minimization algorithm. Nondeterministic automata are well suited for hardware and do not need to be fully determinized, and we used that to obtain automata and checkers that are more concise than any others in literature [4].

In addition to original automata algorithms, rewrite rules play a key role in MBAC to help handle the large variety of "sugaring" and other more involved temporal operators found in PSL [4], and SVA to a lesser degree. These rules were proven correct using automated theorem proving techniques. Using rewrite rules and specialized automata algorithms, our checker generator supports the full synthesizable subset of PSL, and SVA, and produces behaviorally correct checkers in experimental benchmarks with simulators and model checkers.

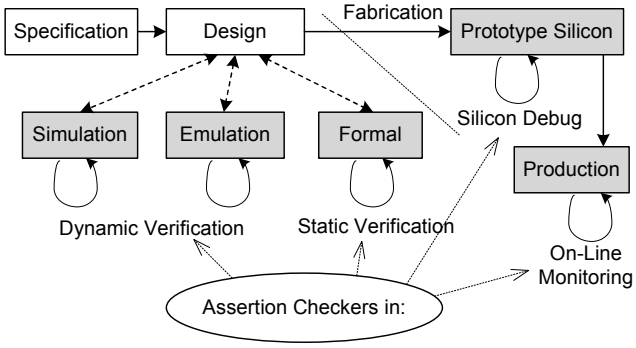

Fig. 2. Assertion Checkers in Hardware Verification, Silicon Debugging and On-Line Monitoring.

III. Applications of Checkers

Assertion-based design and verification [9] are based on the specification of correctness requirements in the form of assertion statements. Assertions can (and should) be added before the verification step, and should be part of the digital design process as well. Figure 2 shows a summary of the main engineering tasks leading to a finished integrated circuit. Ideally, assertions should also be used in the specification stage to allow the formal documentation of requirements in a clear and unambiguous way.

The uses of checkers in verification and silicon debug are presented in the next two subsections. In the third subsection, on-line monitoring is cast in the more general theme of using a checker generator and assertions for automating certain types of logic design, in a high-level manner.

A. Checkers in Verification

Quality-driven verification consists in ensuring that a given design respects is specification. Verification mainly takes place before the fabrication step, and can also be referred to as pre-fabrication verification. Two main classes of verification are dynamic verification (ex.: simulation and hardware emulation) and static verification (ex.: model checking).

1) Dynamic Verification: In dynamic verification, the design is exercised with a given stimulus and its response is analyzed for the presence of errors. When simulation times become excessive, designs are often emulated in custom hardware in order to run orders of magnitude faster than in simulators. When simulators or emulators do not support PSL and SVA, generating assertion checkers and adding them to the source design is an effective way of allowing the continued use of assertions. The checker is connected to the design under verification, as shown in the top part of Figure 3. The output signal of the checker can be observed and any violation can be identified in the trace.

2) Static Verification: When formal proofs of correctness are required, static verification is performed. Although formal tools show greater challenges in scaling to large designs, model checkers can provide a guarantee that a property holds, or it can provide counterexamples when it does not. When formal verification tools such as model checkers do not support PSL and SVA, generating assertion checkers and

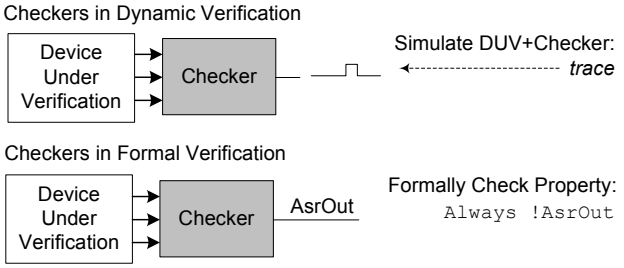

Fig. 3. Using Checkers in Formal (Static) and Dynamic Verification.

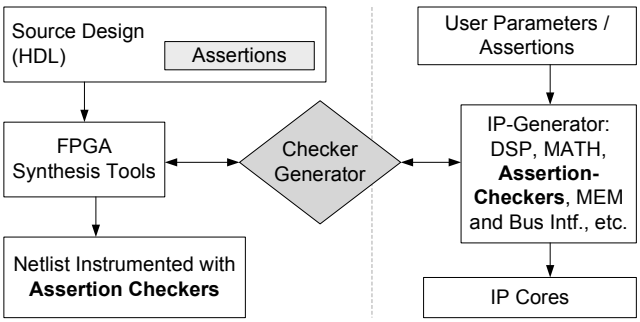

Fig. 4. Use of the Checker Generator in Synthesis Tools (left) and in Core-Generating Tools (right).

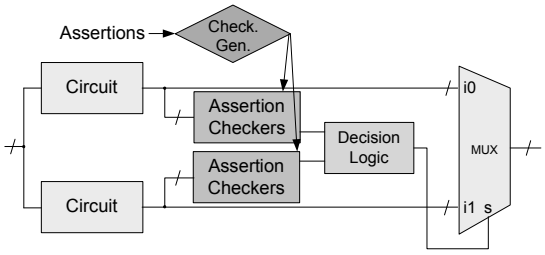

Fig. 5. Example in High-Level Design: Run-time Status Monitoring Using Assertion Checkers for Redundancy Control.

adding them to the source design also allows the continued use of assertions.

In the formal verification case (bottom part of Figure 3), a simple property postulates that the checker output(s) are always false. This property is implicitly checked over all possible execution paths. Using checkers in formal verification is straightforward for safety-type properties (invariants), but would require some adaptation for liveness-type properties, which apply to infinite executions.

B. Checkers in Silicon Debug

Verifying a fabricated design for faults is known as post-fabrication verification or silicon debug. Although it is much more cost effective to find bugs before fabrication, some errors sensitive to timing constraints can only be checked in the actual fabricated silicon. In silicon debugging with assertions, checkers can be purposely left in the fabricated IC for debugging purposes, where they can greatly help in finding the cause of a failure.

When the prototype silicon is found to operate as desired, these checkers can be removed in the production spin, or they can be purposely left in the chip for in field testing. Using assertions with a checker generator can be an effective way of automating the design of a response analyzer for a form of built-in self test. As ICs become more complex and harder to realize as first-time-correct, assertions become useful in areas such as emulation and post-silicon debug, where full simulation traces are not easily available for analysis.

C. Checkers in EDA Tools

The checker generator also finds an array of applications in existing EDA (Electronic Design Automation) tools such as synthesis tools and core generators (IP generators). We also present our view of high-level assertion-based design.

1) Synthesis Tools: As assertions become more and more integrated in the design process, the need to synthesize them will only gain in importance. The checker generator can also be integrated to conventional synthesis tools that need to process assertions. The left side of Figure 4 shows how MBAC can interact with an FPGA synthesis tool in order to allow users to incorporate assertions in their implementation.

2) IP Generating Tools: FPGA tools often include core generating tools (or wizards) whereby multiple types of cores such as memory interfaces, DSP blocks and math functions, to name a few, can be automatically generated according to user defined parameters. Assertion checkers can also be offered as a class of cores: users simply enter their assertion (with the signal dimensions) and the tool can generate the corresponding core (checker). This integration is illustrated in the right side in Figure 4. Interfacing a core generator tool to the MBAC checker generator makes this new feature possible, and makes checkers more accessible to hardware designers.

3) High Level Design: In a more general usage scenario, the expressive power of assertions, combined with a checker generator can be used to actually perform explicit logic design, going beyond the bounds of verification and debugging. In this view, any form of monitoring circuit that can be expressed by an assertion, once given to the checker generator, can produce a complex error-free circuit instantly. These circuit-level checkers are in fact more akin to actual design modules rather than verification modules. An example where this technique can be utilized is in designing certain portions of self-test circuits, as mentioned previously.

If checkers are incorporated in the final design, in-circuit diagnostic routines that utilize the checker output can also be implemented. Assertion checkers can be an integral part of any design that attempts to assess its operating conditions on-line in real time. Figure 5 shows an example of how a checker generator can be used to automatically design the monitoring circuits for switching in redundant systems. Designing an array of safety-checking circuits can be more easily performed using assertions and a checker generator.

The use of PSL/SVA and a checker generator can be explored for use in high-level synthesis, similar to the Production Based Specification work [10] and the high-level synthesis of synchronous languages such as Esterel [11]. Using

TABLE I
BENCHMARKING PSL ASSERTIONS. (CHECKERS SENSITIVE TO ACTIVE LOW RESET AND RISING EDGE CLK)

Assertions S1-5: "assert never Sx;" S6-10: "assert always {a} |=> Sx;" P1-5: "assert always Px;"		**Hardware Metrics**						**Equivalence Check**
		MBAC			FoCs v2.04			MBAC ↔ FoCs
		FF	LUT	MHz	FF	LUT	MHz	(FoCs counterexamples)
S1:	{ {{b;c[∗2:4];d}[+]} && {b;{e[–>2:4]};d} }	**20**	**21**	**456**	32	39	383	pass
S2:	{ {a ; b[∗] ; c[∗]} : {d[∗] ; e[∗]} ; f[∗] }	**5**	**7**	**352**	32	56	261	pass
S3:	{ {a ; b[∗] ; c[∗]} : {d[∗] ; e[∗2:4]} ; f[–>] }	**8**	**15**	**338**	26	32	332	pass
S4:	{ {a[∗0:1];b[∗1:2];c[∗]} : {d[∗0:1];e[∗2:4]} }	**6**	**7**	**392**	38	43	340	$\{\sim reset\,;b\wedge e\,;e\}$
S5:	{ {{a[∗];b[∗];c[∗]} && {d[∗5:7]}} : {c[–>]} }	**14**	**16**	**445**	65	127	257	$\{\sim rst\,;a\wedge d\,;d\,;d\,;d\,;c\wedge d\}$
S6:	{ {{c[∗1:2];d}[+]} && {e[–>2]} }	**9**	**22**	**311**	No Output			–
S7:	{ {{b;c[∗1:2];d}[+]} : {{e[–>]};d} }	**22**	**59**	**259**	No Output			–
S8:	{ b ; {{c[∗0:2]} ; {d[∗0:2]}}[∗] ; e }	**3**	**4**	**610**	7	12	331	$\{\sim reset\,;a\,;b\,;d\,;d\,;d\}$
S9:	{ {{c[∗1:2];d}[+]} & {e[–>2]} }	**4**	**4**	**472**	No Output			–
S10:	{ {b ; c[*]} : {d[*] ; e} ; f }	**9**	**17**	**280**	17	44	258	pass
P1:	{a;b;c} |=> never {d[∗0:3] ; e}	**7**	**7**	**419**	Not Supported			–
P2:	(a –> {[∗0:7] ; b}) abort ∼c	**8**	**8**	**667**	Not Supported			–
P3:	e || (a –> ({b;c} until d))	**3**	**5**	**469**	Not Supported			–
P4:	{a} |–> { {b;c[∗]} : {d[*];e;f} }	**8**	**23**	**295**	No Output			–
P5:	a –> next_event_e(b)[1:6](c))	**7**	**8**	445	12	12	**564**	pass

assertions and a checker generator as a means of logic design poses difficulties when it comes to generating complex output signals; however, the design of many types of analysis circuits can benefit directly from these techniques.

IV. SUMMARY OF EXPERIMENTAL RESULTS

In this section we show recent experimental results comparing the latest version of FoCs to our MBAC tool. In order for checkers to benefit the most the application scenarios mentioned in the previous section, the checkers should utilize the fewest circuit primitives as possible when implemented in hardware. The experimental results contained in this section help motivate this, and we compare against the best known and only other available stand-alone checker generator. Synthetic benchmarks are used to better compare the tools against more temporally complex properties.

The checkers produced by MBAC are compared to FoCs checkers by synthesizing the assertion circuits for a Xilinx XC2V1500-6 FPGA, using ISE 8.1.03i. The number of flip-flops (FF) and four-input lookup tables (LUT) is reported, together with the maximum operating frequency after synthesis. In all the experiments, the checkers were generated instantly by our tool. Model checking by Cadence SMV is further used to compare the behavior of checkers from both tools (MBAC and FoCs). For a given assertion, the checkers generated by both tools are compared using the miter approach (i.e. by asserting that XOR between the two checker outputs is false). For several test cases, the model checker finds counterexamples where the assertion signals from FoCs fail to report the error, as reported in the last column of Table I. From the results, MBAC outperforms FoCs in all the test cases in terms of the circuit size of the checkers, and in all but one case, MBAC checkers have a higher maximum clock frequency.

The MBAC checker generator has consistently been at the forefront of generating resource-efficient checkers since our initial publication on the topic [12], where a particular test case even yielded a 1000× improvement over FoCs, in terms of the Verilog code size of the checker.

V. CONCLUSION

In this paper we have presented a variety of new directions where checker generators can be used, which we expect to materialize further as the assertion paradigm evolves. The MBAC checker generator was shown to be the most effective at compiling assertions into circuit-checkers. New automata methods and rewrite rules play a key role in this tool to help produce efficient run-time checkers for verification and debug, which is important when these checkers compete with the DUV for valuable chip area. This research significantly extends and enhances the many benefits associated to assertions, in the quest for quality designs.

REFERENCES

[1] C. Eisner and D. Fisman, *A Practical Introduction to PSL*. New York, New York: Springer, 2006.

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

[3] Y. Abarbanel, I. Beer, L. Glushovsky, S. Keidar, and Y. Wolfsthal, "FoCs: Automatic Generation of Simulation Checkers from Formal Specifications," in *Proceedings of the 12th International Conference on Computer Aided Verification (CAV'00)*, 2000, pp. 538–542.

[4] M. Boulé and Z. Zilic, "Automata-Based Assertion-Checker Synthesis of PSL Properties," *ACM Transactions on Design Automation of Electronic Systems (ACM-TODAES)*, vol. 13, no. 1, p. Article 4, January 2008.

[5] K. Morin-Allory, E. Gascard, and D. Borrione, "Synthesis of Property Monitors for Online Fault Detection," *Journal of Circuits, Systems and Computers (JCSC)*, vol. 16, no. 6, December 2007.

[6] IBM AlphaWorks, "FoCs Property Checkers Generator, version 2.04," www.alphaworks.ibm.com/tech/FoCs, 2007.

[7] J. Hopcroft, R. Motwani, and J. Ullman, *Introduction to Automata Theory, Languages and Computation*, 2nd ed. Addison-Wesley, 2000.

[8] M. Boulé and Z. Zilic, *Generating Hardware Assertion Checkers – For Hardware Verification, Emulation, Post-Fabrication Debugging and On-Line Monitoring*. Springer, ISBN 978-1-4020-8585-7, 2008.

[9] H. Foster, A. Krolnik, and D. Lacey, *Assertion-Based Design*, 2nd ed. Norwell, Massachusetts: Kluwer Academic Publishers, 2004.

[10] A. Seawright and F. Brewer, "Clairvoyant: A Synthesis System for Production-Based Specification," *IEEE Transactions on VLSI Systems*, vol. 2, no. 2, pp. 172–185, 1994.

[11] S. Edwards, "High-level Synthesis from the Synchronous Language Esterel," in *Proceedings of the International Workshop on Logic and Synthesis (IWLS)*, 2002.

[12] M. Boulé and Z. Zilic, "Incorporating Efficient Assertion Checkers into Hardware Emulation," in *Proceedings of the 23rd IEEE International Conference on Computer Design (ICCD'05)*, 2005, pp. 221–228.