Static Netlist Verification for IBM High-Frequency Processors using a Tree-Grammar

Christoph Jäschke, Ulla Herter, Claudia Wolkober, Carsten Schmitt, and Christian Zoellin
IBM Deutschland Research & Development GmbH, Schönaicherstr. 220, 71032 Böblingen, Germany
Email: jaeschke,ulla.herter,ck,carsten.schmitt,czoellin@de.ibm.com

***Abstract*—This paper introduces a new static verification technique using tree-grammars. The core contribution is the combination of a structural netlist traversal with parser generation techniques for tree-grammars. Today's commercial static analysis tools offer a rich set of parameterized connectivity checks, but their predefined nature prevents effective checks on the highly customized structures found in high-end processor designs. The method presented here allows to formulate the required connectivity using a tree-grammar, thus combining high checking flexibility with convenient specification. Unlike other grammar based structural verification approaches, this method does not require the complete netlist to be matched against the production rules, which allows short runtimes even on large multi-core chip netlists. Results are presented for the most recent 22nm high-end processor designs.**

I. Introduction

Verification of today's server processors requires the complete range of available verification techniques. They can be subdivided into dynamic verification, i.e. simulation, and static verification techniques like analysis of the netlist structure or formal verification. We developed a new static analysis approach based on netlist traversal and tree-parsing techniques. It has proven to yield valuable coverage during the verification of meanwhile five processors. This paper is presenting the method for the first time.

Formal verification[11] checks functional properties on a canonical representation of a gate-level netlist. Although the canonical form is advantageous in terms of general applicability, many applications still suffer from serious capacity limitations when considering processor chip netlists.

Structural netlist properties can be verified directly on a netlist without capacity limiting conversions. Unfortunately not all functional properties can be formulated easily by structural properties, but there is a wide range perfectly suited for a structural specification. Typically these are chip spanning structures, far exceeding the capacity of any formal verification method, or structures hard to specify functionally because of their pervasive nature[20]. Examples are the clocking and test infrastructure, fencing of regions, as well as debug and sensor networks.

Part of the test infrastructure can be inserted automatically for a mature design technology with standard clocking schemes, examples are the scan-wiring[8] or array BIST (built-in self-test) logic[6]. These approaches do not work well on leading edge technologies used in high-end processor design. This domain is often leveraging advanced clocking schemes whenever necessary, exploiting a wide variety of array styles including a sophisticated twisting of functional and test paths. Consequently, high-end processor design uses automatic correct-by-construction approaches only in rare situations; a verification method for the mostly hand-crafted structures is definitely required.

II. Related Work

There is a great variety of static verification methods dealing with netlist structures. On the transistor- or gate-level, electrical rule checkers (ERC) verify technology dependent properties like CMOS-compliance or driving strength limits. There are methods raising the abstraction level in order to simplify later checks on the resulting structure by transformations from the transistor-level[21] or gate-level[10] up to the module-level.

In the structural verification expert systems[15], [19] programmed checking rules are triggered by matches of sub-structures in the netlist. The system requires all matches starting on the transistor-level to be done before first checking rules can be triggered.

The GRASP[9] system leverages a graph grammar[18] approach, which is the translation of sub-graphs of an original graph into a new graph. Generally these replacements are recursive until the top-level grammar production is referenced, denoting the validity of the graph instance according to the grammar.

Both methods are targeting the structural validation on the transistor-level and require the lowest netlist level to be fully matched for checking. The full matching requirement leads to a large host-machine requirement in terms of memory and runtime for a full processor chip netlist. Even if this could be satisfied, checks on higher levels are considering the function of certain blocks, which would require dedicated structure matches or grammar productions for every function block, as otherwise the functional information is lost in upper levels.

Although the interpreter approach in [17] does not require to process the full netlist completely, the netlist is stored flat in memory; a modern full processor netlist would be impractical to handle efficiently this way. Checks are written using set-functions. Higher-level checks are of course possible, but need to be composed by hand. Raising the abstraction level by netlist sub-structure matching and

978-3-9815370-8-6/17/$31.00

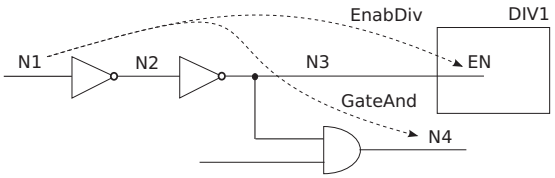

Fig. 1. Forward-traversal and resulting tokens.

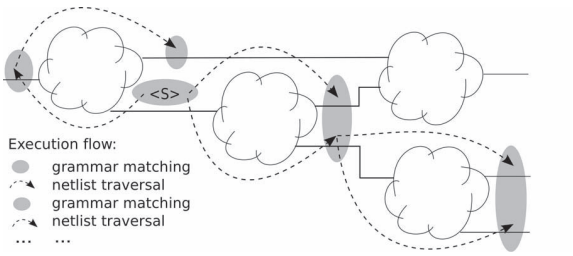

Fig. 2. High-level view on netlist traversal and grammar matching.

replacement is possible, but needs to be done on the full netlist like in the previously mentioned approaches.

Commercial static analysis tools[5], [3], [2] not only offer static lint-like checks on the hardware description language (HDL) level, but also check on the netlist connectivity in varying degrees. Clock domain crossing (CDC) checks are quite common. Ensuring the connectivity of BIST controls with their memories[1] belong to the more advanced checks. An exposure of explicit (structural) net connection checks of a specific type (AND, OR, or XOR) to the user have been found by us only in [2]. Although parameterization capabilities may be excellent, the above commercial tool checks are predefined. This makes their application easy to use, but on the other hand prevents creating more advanced netlist checks.

III. Our New Method

The method presented in this paper allows a flexible formulation of structural checks. It is leveraging the specification conciseness of a grammar, but does not need the full netlist to be processed. This leads to a simplification of the grammar compared to the aforementioned approaches. In the experimental results section, we prove the performance benefits by short turnaround times on full-chip netlists of our latest IBM high-end processors.

The input to our method is a hierarchical and technology independent gate-level netlist with storage elements either directly instantiated (HDL specified) or synthesized from process statements. As processor design demands highly optimized operators, handcrafted gate-level netlists are commonly used for RT-level operators. Our method still supports complex gates synthesized from RT-level operators and therefore an application on the RT-level.

Classical text parsing tokenizes a character stream first and verifies this stream against the grammar afterwards. For example, this happens when the IBM design compiler reads in the HDL source and builds our input netlist.

Our new verification method, instead of running on the HDL source text, is applied on the connectivity structure of this output netlist. It is using netlist traversal operations and tree-grammar matching in an alternating sequence. The netlist traversal is generating tokens according to the netlist structure and given grammar. The tree-parser checks the conformance of the generated tokens with the grammar, which is based on the syntax of ANTLR[16] .

A classical text tokenizer might generate a token INT for the characters "256". Without elaborating on the details now, our netlist traversal might generate a token EnabDiv for an exemplary traversal path (upper dashed arrow in figure 1) from net N1 over N2 over N3 to net EN (inside DIV1). The classical token text "256" corresponds to the net path N1, N2, N3, to DIV1.EN of our approach.

Figure 2 sketches the alternating sequence of netlist traversal and grammar matching from a high-level view. Net names are not shown and logic is abstracted as clouds. An exemplary forward and backward traversal is drawn from the start net of the top-level rule depicted by <S>. The traversals are represented by dashed trees. Each tree-root is a traversal start, and the tree-leaves (arrow heads) are the traversal end-points generating the tokens. The grammar matching is represented by gray shaded areas. They have a glue-function to the trees on the syntactic level and form higher level structures.

This approach is able to verify only a part of the complete netlist structure. The coverage is explicitly defined by the start nets required for every top-level rule[1], and implicitly by the grammar and returned tokens of the netlist traversal phases. Whenever the netlist neighborhood should be checked further, new netlist traversals can be called recursively from the parser (at points specified by the grammar) in forward or backward direction. Not continuing these traversals would keep the checks local to the netlist processed so far.

This has several advantages: rules need to be specified only for the interesting chip structures; the reduced rule set can be much more specific, and processing only a netlist subset keeps runtime and memory consumption low.

Like symbol-tables are used in programming language compiler frontends[7], our method uses C++ objects to represent high-level information of interesting netlist structures. These are accessed in the C++ action code of the grammar and serve as anchor points in checking general graph structures as well as enabling checks using explicitly coded meta information not present in the netlist structure. Due to space limitations in this paper, we have to omit the details on this part.

Details of the netlist traversal are provided next. The operation of the parser as well as the interaction of both are explained afterwards.

[1]Clocking infrastructure checks would e.g. select PLL and clock controller outputs as start nets.

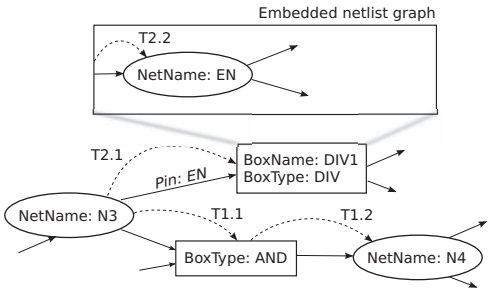

Fig. 3. Available attributes during the netlist traversal.

A. Netlist Traversal

The netlist traversal basically abstracts the netlist connection structure by generating tokens for the parsing process. Each generated token is of a specific type, referring to the reason why the traversal has stopped. As mentioned before, the path of nets to the traversal stop is attached to every token. The location information of such a token is net-based, as opposed to the line/column-based location of classical text tokens.

The netlist can be modeled as a set of bipartite graphs with two types of vertices and directed as well as undirected edges, the latter for bidirectional signal connections. The first type of vertices are boxes: either (primitive) logic function gates, state keeping elements like latches, or hierarchical boxes. A hierarchical box is either referencing another embedded netlist graph, or the enclosing box providing the IOs of the current level. The second type of vertices are nets. Graph edges always connect a net vertex with a box vertex. Edges can be labeled with pin attributes of the connected box, for example the name or function of the box pin.

As mentioned before, the netlist traversal (in forward or backward direction) starts on a specific net provided by the parser. The traversal steps from a net vertex over all out-going (in direction of traversal) edges to all connected box vertices in a depth-first manner. From a primitive box vertex, the traversal continues over all out-going edges to all connected net vertices. For a hierarchical box vertex, the traversal suspends at the current netlist graph and continues from the enclosing box vertex of the referenced embedded netlist graph. A stack is keeping track of the visited hierarchical box vertices. The traversal repeats stepping from one net to the next until a stop is detected or a net has no more out-going edges.

Along with the start net and direction, the parser is passing the token type specifications to the netlist traversal, defining where to stop. Opposed to classic text tokenizing, where the token match is done only against the input text characters, the token generation of our netlist traversal can reference multiple attributes of box and net vertices. These attributes are attached to the netlist by the HDL compilation process. Examples are instance names, types of nets and boxes as well as their HDL attributes.

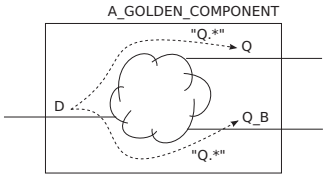

Fig. 4. A jump traversal using "Q.*".

Figure 3 shows the graph model of a part of the netlist in figure 1. A netlist traversal step in figure 1 from N3 to N4 corresponds to the pre-step T1.1 and post-step T1.2 in figure 3. The attributes BoxType AND (T1.1) and NetName N4 (T1.2) can be used by the traversal to check for any stop-conditions. Similarly, a traversal step from N3 to DIV1.EN can evaluate the attributes BoxName DIV1 and BoxType DIV (T2.1) as well as NetName EN (T2.2).

Let us complete the traversal example using figure 1, starting from net N1 in forward direction. We assume the parser has passed the token type specifications EnabDiv and GateAnd to the netlist traversal:

```
EnabDiv: BoxType == "DIV" && NetName == "EN";
GateAnd: BoxType == "AND";
```

These token definitions are called stop-rules, because they are defining where to stop the traversal. A stop-rule is a conjunction of at least one stop-condition: a regular expression (using PCRE [4] syntax) on an attribute. Every stop-rule is checked against the referenced attributes after each traversal step. The stop-rule EnabDiv will match at net EN inside box DIV1 (upper dashed arrow). Stop-rule GateAnd will match at net N4 (lower dashed arrow). Using <token-type>@<location> as the textual representation, the traversal will pass EnabDiv@DIV1.EN and GateAnd@N4 back to the parser.

In addition to forward and backward traversals, a jump traversal can be specified by the grammar. Instead of stop-rules, the parser is passing possible jump targets using regular expressions to the netlist traversal. Figure 4 shows a jump traversal starting from net D using "Q.*" as jump target specification. The jump traversal is a mechanism to abstract known components, as for example edge-triggered latches. Obviously this abstraction requires to have the component verified in a sufficient way by other means. A jump is intentionally restricted to occur within the same hierarchy, which actually limits how checks can be skipped.

B. Tree-Parsing

The parser calls the netlist traversals and check the returned tokens against the grammar. The grammar effectively specifies the structural relationship of all interesting net paths (generated by the traversals) to each other.

1) Alternating parser and traversal sequence: The traversal is started from within the parser by three new traversal operators ->, -<, and -^, starting a forward, backward, and jump traversal respectively.

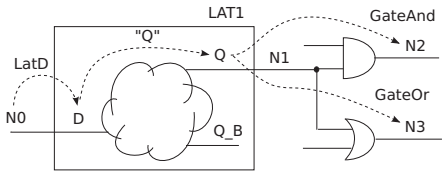

Fig. 5. Alternating sequence of netlist traversal and parsing.

Listing 1. A simple hardware grammar and stop-rules.

```
start: <S> -> (latD);
 latD: LatD -^ ("Q" -> (gates));
gates: GateAnd GateOr;

  LatD: BoxType == "Lat" && NetName == "D";
GateAnd: BoxType == "AND";
 GateOr: BoxType == "OR";
```

Listing 1 shows a grammar and stop-rules able to accept the netlist in figure 5. The traversal sequence is shown as dashed arrows. Although the grammar is not very meaningful, it is simple enough to demonstrate the interaction.

Once the parser code has been generated and compiled, the user code has to load the start location (Start@N0 in our example) into the token stream first. The token stream serves as the means to pass the results from the netlist traversal back to the parser. The call of the top-level rule "start" causes the parser now to match the grammar against the token stream content, i.e. the current grammar symbol <S> is matched against Start@N0. This causes the token stream pointer to advance after the Start@N0 token.

The next grammar symbol in our example is the forward traversal operator ->. A traversal operator causes all stop-rules at the current level to be collected. The current level is bounded by a pair of parenthesis directly following the traversal operator, not containing any embedded traversal operators and their associated stop-rules.

The collected stop-rules are passed to the traversal code, which is executed after consuming the traversal operator. Once the traversal is done, the returned tokens are inserted into the token stream just after the current point of parsing, where the parser resumes parsing them. The tokens would also be enclosed by ForwTrav@N0 and EndOfTrav@N0, basically causing a linearization of the token tree into the token stream.

In listing 1, the collected stop-rules consist of only LatD, collected through sub-rule latD. The traversal returns LatD as expected. The parser resumes by matching LatD[2] and proceeds to the jump traversal on the same line. The process repeats with collecting the stop-rule "Q", starting a jump in the netlist, returning the token with the net location LAT1.Q. The parser consumes this token and starts another traversal using sub-rule gates, specifying two stop-rules GateAnd and GateOr.

[2]Matching of tree-linearization tokens like ForwTrav@N0 and EndOfTrav@N0 will not be mentioned.

The simple example shows how parallel and serial structures in the netlist are expressed in the grammar. Parallel structures are just sequences of tokens in the grammar (see GateAnd and GateOr in the third line). Serial structures are expressed by nested traversal operators.

2) Advanced Grammar Constructs: Constructs like recursion, alternatives or quantifiers allow a grammar to accept more variations of a netlist. The grammar

```
gateTree: <S> -> (contAnd);
contAnd: GateAnd -> (contAnd | GateOr);
```

would basically accept an arbitrary chain of AND-gates until an OR-gate is recognized. The grammar

```
gateTree: <S> -> (contAnd* GateOr*);
contAnd: GateAnd -> (contAnd* GateOr*);
```

would accept an arbitrary AND-gate tree where every net can optionally connect to OR-gate inputs.

3) Using Labels and Logic Expressions: Labels are user-defined character strings and can be passed to any traversal. By default, any logic input hit during this traversal will be tagged with the passed labels. A traversal of a fence signal started with label "fence" would therefore place this label at every logic driven. Another grammar can now verify that we have a fence present at a signal before driving some other net. The following grammar will do the trick:

```
fenceMe: AndFence -> (FencedInput) ;
AndFence: NetMarkLogic == "AND[(]fence,>[)]";
```

It requires any net traversed with rule fenceMe to be fenced (ensured by stop-rule AndFence) before reaching FencedInput (not shown here). The NetMarkLogic stop-condition investigates the logic driving the net to be stopped at. The string matches any binary AND where the other (non-traversed) input has a "fence"-label, the actually traversed input is represented by the ">".

IV. Experimental Results

One of the primary targets of our checking environment is the clocking infrastructure shown in figure 6. We verify every non-dashed connection in the drawing, but of course the picture is a considerable simplification. As an example, there are over a dozen clocking areas asynchronous to each other, the required muxing and control is verified as well.

The clocking reaches from the PLL over the local clock buffers[13] (Lcb) to latches (Lat) and arrays. The dynamic control is starting from the clock controllers (Cc) over staging latches (Plat), local clock controllers (Lcc), optional dividers (Cdiv), and smaller control blocks (LcbOr) to Lcbs. The mode control is verified from dedicated control blocks (LcbCntl) to the LcbOrs and Lcbs. Table I shows how many elements of a certain type are present in the POWER8[3][12] and z13[3][14] processor chips.

We consider it impossible to describe every check we do using our new method. Let us instead explain the basic

[3]Trademarks of IBM in USA and/or other countries.

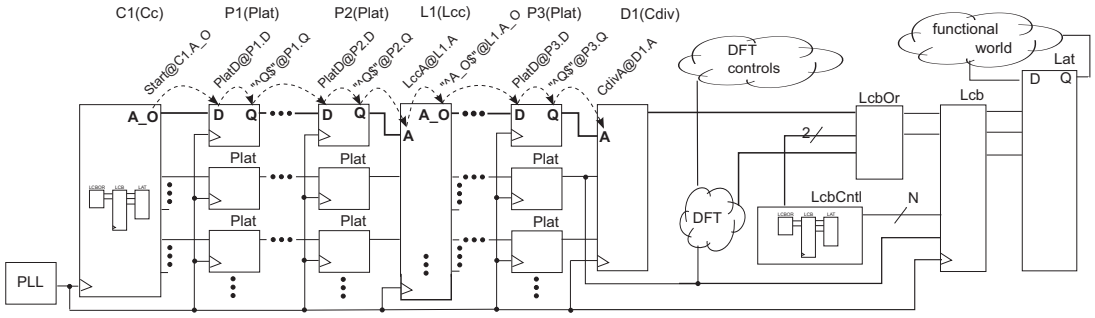

Fig. 6. Clocking infrastructure overview with simplified align path in bold.

TABLE I
ELEMENT COUNTS OF CLOCKING INFRASTRUCTURE.

Clocking Element	POWER8[3]	z13[3]
Cc	19	18
Plat[4]	144 K	160 K
Lcc	142	69
Cdiv	7517	3995
LcbOr	109 K	107 K
Lcb	345 K	260 K
Lat[4]	2.73 M	2.55 M
LcbCntl	44.8 K	39.4 K

principles of our new approach by two examples: the align signal structure and the first level error reporting.

The (simplified) align signal connectivity can be verified by the following grammar and stop-rule definitions:

```
startA[int d]: <S> -> (contA[d]);
contA[int d]: lccA[d,ti]* platA[d]*
  (n = CdivA { hitCdiv(n,d,"A"); })*;
lccA[int d]: n = LccA { hitLcc(n,&d,"A"); }
  -^ ("A_O" -> (contA[d]));
platA[int d]: n = PlatD { hitPlat(n,&d,"D"); }
  -^ ("Q" -> (contA[d]));

CdivA: BoxType == "Cdiv" && NetName == "A";
 LccA: BoxType == "Lcc"  && NetName == "A";
PlatD: BoxType == "Plat" && NetName == "D";
```

The verification environment knows about all clock controller instances. For this example, the top-level rule "startA" is called with just C1.A_O, causing the dashed arrow path in figure 6 to be verified. The tokens exchanged between netlist traversal and parser are depicted along this path. The verification task of the grammar is to complain about invalid connectivity, the task of the C++ objects (updated via hitCdiv, hitLcc and hitPlat) is to complain about missing connectivity as well as to apply explicitly coded meta-information for additional checks.

The second example is checking the first level of a simplified error reporting structure: ErrRpt blocks provide error signals over an optional OR-tree and optional staging latches to fault isolation registers (Firs). The below grammar and stop-rules can be used to verify the structure from the Firs backwards to the ErrRpt outputs:

```
startE: <S> -< (contE);
 contE: orE+ | latE | n = ErrO { hitE(n,"O"); };
  orE: GateOr -< (contE);
 latE: LatQ -^ ("D" -< (contE));

GateOr: BoxType == "OR";
  LatQ: BoxType == "Lat" && NetName == "Q";
  ErrO: BoxType == "ErrRpt" && NetName == "O";
```

The POWER8[3] error reporting structure has about 21400 ErrRpt outputs and 3750 Fir error inputs. The top-level rule "startE" is called for all error input nets of all Firs. A backward traversal is started for every net with the expectation formulated by the sub-rule "contE": either an OR (sub-rule "orE"), a latch output (sub-rule "latE"), or an ErrRpt output net O.

When the traversal steps back over an OR gate, it evaluates the stop-rule for both inputs and therefore generates two tokens, thus the quantifier "+". A latch is continued at its D input via a jump traversal. The hitE call in the action code is recording the actual connectivity of ErrRpts to the Fir inputs. Thus, ErrRpt outputs not feeding any Firs are easily detected. The connections found are written out for documentation and other verification purposes.

Verification Runtimes: To our best knowledge, there is no grammar based structural verification method capable of running on a modern processor full chip netlist. Direct comparison is therefore not possible. We can still present numbers to illustrate the advantage of a concise grammar specification as well as checking runtimes for our latest processor chips.

Runtime numbers for the clocking checks are collected on a POWER7[3] IBM 9179-MHB host at 3.86 GHz (single execution thread) in table II. The memory footprint for the POWER8 is 31.5 GBytes, for the z13 it is 27.6 GBytes.

Problem Takedown: The processor HDL development is organized along releases with a fixed pace of a couple of days. Figure 7 shows the problems detected by the structural verification over the releases of the z13[3] chip. With more logic implemented in a release, a higher, more restrictive structural checking phase (denoted in the header line) is turned on over time. Within a phase color, a dark bar color denotes unclassified problems which

[4]Bit-width of the Plat- and Lat-objects are not presented here.

TABLE II
CHECKING RUNTIMES.

Task	POWER8 [min]	z13 [min]
Base inits	26.7	34.0
Mark application	77.19	75.04
Object inits	11.9	11.9
Traversals	130.8	175.8
Object checks	3.96	5.59
Overall	**250.5**	**302.3**

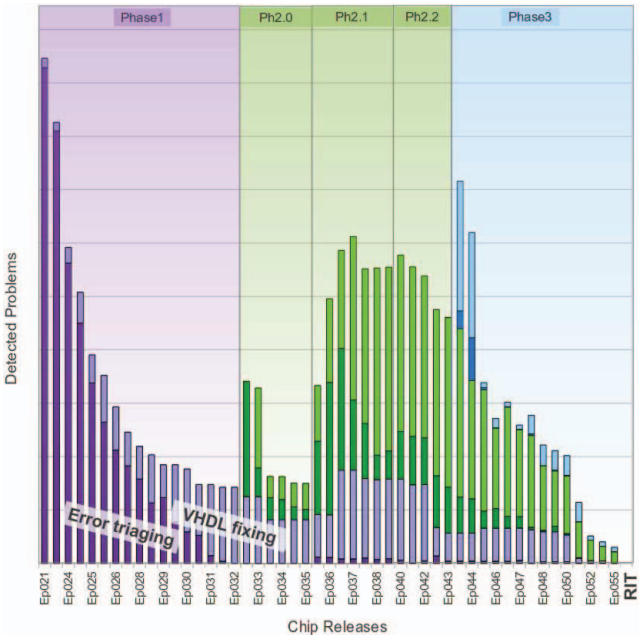

Fig. 7. z13 problem take-down curve.

designers are still investigating. A light bar color denotes the problems where designers put a "classification" label on the box or net. The graph shows that the problem count increases with nearly every new phase enabled. The unclassified (dark color) problems usually go down more quickly than the actual fixing (light color). This approach ensures that a problem is not handled by multiple people, and allows to get a good overview about the quality of the problems.

V. Conclusions

A new static netlist verification method has been presented. The tree-grammar based approach is leveraging a netlist traversal doing a first abstraction of the netlist by generating tokens. The grammar allows the structural property specification on a high level of abstraction, without requiring to specify the full chip.

Running these static structural checks on our processor designs is mandatory in the design flow. It catches a majority of problems before they can even reach simulation, saving weeks of complex debug work there.

The flexibility has been proven a number of times. We implemented formerly hand-written checking code for e.g. error reporting or scan-chain analysis into our grammar based environment. Especially when the runway for the check is short, as happening frequently before RIT, we have shown that certain verification tasks are more efficiently done by a static structural analysis using a grammar than by functional simulation or formal verification.

References

[1] Atrenta Inc. An automated approach to RTL memory BIST insertion and verification. http://www.atrenta.com/resources/white-papers.htm5. Accessed November 25, 2014.

[2] Atrenta Inc. Atrenta SpyGlass. http://www.atrenta.com/products/spyglass.htm5. Accessed November 25, 2014.

[3] Mentor Graphics. Questa clock-domain crossing (CDC) verification. http://www.mentor.com/products/fv/questa-cdc/. Accessed November 25, 2014.

[4] PCRE - Perl Compatible Regular Expressions. http://www.pcre.org. Accessed November 25, 2014.

[5] Synopsys Inc. Next-generation CDC static checking. http://www.synopsys.com/tools/verification/static-formal-verification/pages/vc-cdc-ds.aspx. Accessed November 25, 2014.

[6] R. D. Adams, R. Abbott, X. Bai, D. Burek, and E. MacDonald. An integrated memory self test and EDA solution. In *12th IEEE International Workshop on Memory Technology, Design, and Testing (MTDT 2004), 9-10 August 2004, San Jose, CA, USA*, pages 92–95, 2004.

[7] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques and Tools*. Addison Wesley, 1986.

[8] T. Asaka, M. Yoshida, S. Bhattacharya, and S. Dey. H-SCAN+: A practical low-overhead RTL design-for-testability technique for industrial designs. In *Proceedings IEEE International Test Conference 1997, Washington, DC, USA, November 3-5, 1997*, pages 265–274, 1997.

[9] C. Bamji and J. Allen. Grasp: A grammar-based schematic parser. In *DAC*, pages 448–453, 1989.

[10] G. H. Chisholm, S. T. Eckmann, C. M. Lain, and R. Veroff. Understanding integrated circuits. *IEEE Design & Test of Computers*, 16(2):26–37, 1999.

[11] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. The MIT Press, Cambridge, Massachusetts, 1999.

[12] E. J. Fluhr et al. POWER8^{TM}: A 12-core server-class processor in 22nm soi with 7.6tb/s off-chip bandwidth. In *Solid-State Circuits Conference Digest of Technical Papers (ISSCC), 2014 IEEE International*, pages 96–97, 2014.

[13] J. Warnock et al. POWER7^{TM} local clocking and clocked storage elements. In *IEEE International Solid State Circuits Conference*, 2010. Paper 9.3.

[14] J. Warnock et al. 22nm Next Generation IBM System Z Microprocessor. In *Solid-State Circuits Conference Digest of Technical Papers (ISSCC), 2015 IEEE International*, pages 70–71, 2015.

[15] H. D. Man, I. Bolsens, E. V. Meersch, and J. V. Cleynenbreugel. DIALOG: An expert debugging system for MOSVLSI design. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 4(3):303–311, 1985.

[16] T. Parr. *The Definitive ANTLR Reference: Building Domain-Specific Languages*. Pragmatic Programmers. The Pragmatic Bookshelf, Raley, May 2007.

[17] G. Pelz. An interpreter for general netlist design rule checking. In *DAC*, pages 305–310, 1992.

[18] G. Rozenberg, editor. *Handbook of Graph Grammars and Computing by Graph Transformations, Volume 1: Foundations*. World Scientific, 1997.

[19] Spickelmier, R.L. and Newton, A.R. Critic: A knowledge-based program for critiquing circuit designs. In *VLSI in Computers and Processors*, pages 324–327, 1988.

[20] T. Webel, T. Pflueger, R. Ludewig, C. Lichtenau, W. Niklaus, and R. Schaufler. Scalable and modular pervasive logic/-firmware design. *IBM J. Res. Dev.*, 56(1):54–63, January 2012.

[21] L. Yang and C.-J. R. Shi. FROSTY: A fast hierarchy extractor for industrial CMOS circuits. In *ICCAD*, pages 741–747, 2003.