Efficient Equivalence Checking with Partitions and Hierarchical Cut-points

Demosthenes Anastasakis
Synopsys, Inc.
2025 NW Cornelius Pass Rd.
Hillsboro, OR 97124
demosthenes.anastasakis@synopsys.com

Lisa McIlwain
Synopsys, Inc.
2025 NW Cornelius Pass Rd.
Hillsboro, OR 97124
lisa.mcilwain@synopsys.com

Slawomir Pilarski
University of Washington, Tacoma
Tacoma, WA 98402-3100
sp5@u.washington.edu

ABSTRACT

Previous results show that both flat and hierarchical methodologies present obstacles to effectively completing combinational equivalence checking. A new approach that combines the benefits while effectively dealing with the pitfalls of both styles of equivalence checking is presented.

Categories and Subject Descriptors

B.5.2 [**Design Aids**]: Verification

General Terms: Design, Verification

Keywords: Logic Design, Verification, Equivalence Checking

1. INTRODUCTION

Formal equivalence-checking tools check an *implementation* version of an electronic design for functional equivalence against a *reference* version of the same design, which simulation and/or formal methods have proven correct.

Commercially available tools perform *combinational* equivalence checking[3][14]. That is, they verify the equivalence of combinational cones of logic between matched primary inputs, internal registers, and primary outputs of the two designs. Each primary input and internal register is viewed as an *input point* to downstream cones of logic, and each internal register and primary output is viewed as a *compare point*, which terminates a cone of logic and must be verified. Input points and compare points in the reference are matched, typically by name, function, or topology, with corresponding input points and compare points in the implementation, and matched compare point pairs are compared for functional equivalence. If each reference primary output has a matching implementation primary output and combinationally equivalent driving cones define each matched pair of compare points, then it follows that the two designs produce identical compare point values for all possible combinations of binary values at input points and are functionally equivalent.

Although combinational equivalence checkers can definitively prove that two designs are equivalent, they generally cannot definitively prove non-equivalence. They may report *false negatives* for the following reasons:

(1) A complete match between reference and implementation input and compare points may be difficult to find, or nonexistent[2][5][7], even though the designs' sequential behavior, as observed at primary outputs, is identical. Such matching issues are beyond the scope of this paper.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
DAC 2004, June 7–11, 2004, San Diego, California, USA
Copyright 2004 ACM 1-58113-828-8/04/0006…$5.00.

(2) Because of their combinational view of sequential behavior, they may report a difference in combinational function that does not actually result in a difference in sequential behavior as observed at primary outputs. However, experience indicates that the typical use of these tools does not require verification of design transformations that modify combinational function, so this limitation is not currently of great practical significance.

(3) Because they view input points as unconstrained variables, these tools may report differences in function that cannot actually occur unless functional constraints imposed by logic outside the design being verified are violated[8][9]. Unfortunately, this limitation often *is* of great practical significance, particularly when verifying individual hierarchical blocks of a design in isolation. Users may define external constraints to avoid false negatives, but such set-up is often difficult and time-consuming.

The computational difficulty of checking the equivalence of two designs depends largely on their degree of internal structural similarity, or the number of equivalent points internal to their cones. If internal structure has been radically modified and the cones are very large, the equivalence-checking problem may become intractable. This is more likely to occur when verifying a post-synthesis netlist against a pre-synthesis RTL description (*RTL-to-gate* verification) than when verifying later transformations to a gate-level netlist (*gate-to-gate* verification).

To mitigate verification complexity, equivalence checkers may insert cut-points at potentially equivalent nets[8][9]. If these nets are proven equivalent, they can be treated as input points for verification of downstream logic, simplifying the cones to be verified. However, not only can potentially useful cut-points be missed, but also some solvers may be unable to use the ones that were found. Therefore it is often helpful to explicitly bound cones to be verified by performing the verification *hierarchically*.

In a *hierarchical* methodology each hierarchical block is verified in isolation, with lower-level blocks excised. It is assumed that exact functional equivalence at hierarchical boundaries is preserved, and that the equivalence of each block does not depend on constraints imposed by higher- or lower-level blocks—in contrast to a *flat* methodology, in which the entire design is verified at once, disregarding hierarchical boundaries.

A hierarchical approach can enable an intractable or lengthy verification to complete in reasonable time. A second advantage is that it is likely to require less memory than flat verification. However, flat verification offers one major advantage: if it completes, it is more likely to produce the correct result. False negatives arise in hierarchical verification when either (a) functional equivalence at hierarchical boundaries has not been preserved, or (b) a block of the design is functionally equivalent only when the containing or contained blocks that drive its input points properly constrain them. These conditions occur frequently enough to severely limit the applicability of hierarchical verification.

Table 1 shows run-time, memory, and verification results for several designs, verified flat and hierarchically. These designs are "real world" verifications, both RTL-to-gate (identified as "R2Gn") and gate-to-gate (identified as "G2Gn"). Verifications exceeding a 4Gb process size or 40

hours of CPU time were deemed to have "maxed out." All tests were run on a 750 MHz Sun Microsystems Sparc 4800 with SunOS Release5.8.

The table shows that hierarchical verification generally uses much less time and memory than flat verification. For these examples, automatically generated scripts performed the hierarchical verification, including set-up information defining simple constraints derived from each block's containing and contained blocks, such as input constants, known input equivalences, and pin matching information. Nevertheless, the table shows that hierarchical verification is highly subject to false negatives.

2. SYSTEM ARCHITECTURE

A new approach to combinational equivalence yields both the accuracy and ease of setup of a fully flat methodology and the reduced complexity and memory requirements of a fully hierarchical methodology. It achieves low memory consumption by partitioning the designs sequentially instead of by hierarchical blocks. It achieves performance comparable to that of hierarchical verification, without its false-negative risk, by effective management of cut-points on nets crossing hierarchical boundaries.

The overall system architecture consists of several major components. First, to control memory requirements, the *partition manager* groups compare points into partitions, and the verification model of the first partition is built. Second, to reduce verification complexity, the *cut-point manager* inserts cut-points selectively at hierarchical boundaries within the current partition. Third, the *solver controller* verifies the given partition. Deployment of verification solvers is beyond the scope of this paper[6][12]. If failures are found, hierarchical cut-points are progressively removed and downstream compare points and cut-points are re-verified, repeating until either no failure exists, i.e. verification has succeeded, or no cut-point exists, i.e. verification has failed. Verification continues until all partitions have been processed.

In one extreme, the case when the designs are hierarchically equivalent, verification will succeed with no need to remove any hierarchical cut-points, therefore reaching the correct result as fast as a traditional hierarchical verification approach. In the other extreme, the case when the designs are equivalent only when verified completely flat or are simply not equivalent, this approach will progressively remove all hierarchical cut-points and re-verify in an effort to eliminate false negatives. This is the worst-case scenario for this approach.

In practice, experience shows that most designs have a high degree of similarity at hierarchical boundaries and this approach reduces verification complexity by optimally exploiting such similarity. Hierarchical cut-points are present for the portions of the design that can be verified hierarchically, while they are removed for the portions of the design that need to be appropriately constrained by surrounding logic in order to achieve verification success.

2.1 Partitioning Management

The purpose of partitioning is to limit memory consumption. A *partition* is defined as a group of related compare points and their driving cones. Building the verification model for, and verifying, only one partition at a time limits memory consumption.

To avoid the performance penalty of reprocessing already-processed sub-cones and to maximize the benefits of previously identified internal equivalences, a straightforward criterion for partitioning is to group compare points that share large portions of logic. However, experience has shown that this simple criterion is insufficient, because partitioning decisions also affect the complexity of the verification. Verification can become hard or easy depending on compare point grouping. This is because verification solvers often save information learned during processing shallow cones and re-use it while processing deeper cones of the same partition. Also, the performance and effectiveness of some BDD-based solvers can be hampered by dynamic variable re-ordering when same-partition sub-cones that depend on the same set of input points impose conflicting requirements on the ordering of these input points.

We would like to minimize the number of times any cone is processed; i.e., each cone should appear in as few partitions as possible, ideally just one. However, it is more important to minimize the number of times a complex cone is processed than the number of times a large cone is processed. Unfortunately, there is no known method of pre-computing the complexity of verifying a given cone. However, experience indicates that cone depth offers a good tradeoff between complexity and size when predicting the cost of processing a cone.

The partition manager first assigns a predicted cost, based on cone depth, to each shared cone, and then sorts the shared cones based on their cost. The current partition is initialized with the compare points in the fanout of the highest-cost unprocessed shared cone. Populating the partition continues by exploring the fanout of the next unprocessed shared cone already in the current partition. The partition cost threshold is based on the size of the designs being verified. When the partition cost threshold is exceeded, populating stops. It may also stop before the cost threshold is

Table 1. Hierarchical vs. Flat Verification.

			Flat			Hierarchical		
Test ID	Compare Points	Correct Result	Result	Memory (Gbytes)	CPU Time (seconds)	Result	Memory (Gbytes)	CPU Time (seconds)
R2G1	157819	fail	fail	1.27	33791	fail	0.73	1262
R2G2	73870	fail	fail	2.13	20641	fail	0.64	2137
G2G1	68854	fail	fail	2.16	4153	fail	0.73	10804
R2G3	185344	fail	n/a	**maxed out**	n/a	fail	1.32	1771
R2G4	120306	fail	n/a	**maxed out**	n/a	fail	1.96	11011
R2G5	214961	fail	n/a	**maxed out**	n/a	fail	2.1	9532
R2G6	42988	succeed	succeed	0.59	10215	succeed	0.18	581
R2G7	36813	succeed	succeed	0.84	780	succeed	0.76	777
G2G2	51033	succeed	succeed	1.63	2342	**false neg**	0.65	1514
R2G8	39691	succeed	succeed	2.28	2675	**false neg**	1.24	4604
R2G9	78331	succeed	succeed	2.97	86039	**false neg**	0.73	1039
R2G10	72012	succeed	succeed	3.25	110186	**false neg**	0.79	4080
R2G11	28573	succeed	n/a	n/a	**maxed out**	**false neg**	0.92	1486
R2G12	21707	succeed	n/a	n/a	**maxed out**	**false neg**	1.66	61220

exceeded, when a group of compare points is found to be disjoint from every other group of compare points (they share little or no logic).

2.2 Hierarchical Cut-point Management

The approach described here reduces the size and complexity of cones to be verified by conditionally inserting and removing explicit cut-points on nets that cross hierarchical boundaries within each partition.

2.2.1 Cut-point Insertion

A net cut-point consists of a new compare point and input point pair. The compare point verifies the function of the cut net, and the input point replaces the function of the cut net in the cones of downstream compare points. If verification of the new compare point and all compare points downstream from the new input point succeeds, then the complete cones are equivalent. Initially, we create cut-points on nets that cross hierarchical pin boundaries, when the hierarchical pins are *matched* and *not constant.*

Hierarchical pins may be matched by any method used to match compare points. Cut-points representing hierarchical pins that are not matched are omitted because they cannot enable verification to succeed.

Cut-points representing hierarchical pins that are known to be constant are also omitted because they would not simplify the cones of downstream compare points and would block constraint information (i.e. the constant) that may be required for verification success.

Omitting cut-points representing unmatched or constant hierarchical pins prevents many of the false negatives associated with traditional hierarchical verification. In traditional hierarchical verification, every pin of every separately verified hierarchical block is an input point or compare point, regardless of whether it is matched or constant, unless the user supplies specific external constraint information. Even relatively simple design transformations, such as clock-tree synthesis or test insertion, introduce unmatched hierarchical pins that cause traditional hierarchical verification to produce false negatives. In this system, such pins do not become input points or compare points; instead, their driving logic all the way back to true input points is included in the verification of downstream compare points, as it would be in a traditional flat verification. But by simultaneously including cut-points representing matched, non-constant hierarchical pins, the desired reduction in cone size and complexity is obtained.

2.2.2 Cut-point Removal

Even with a selective approach to hierarchical pin cut-point insertion, a preliminary false negative may be seen, for example due to boundary optimization. In this case we selectively remove cut-points that may contribute to false negatives, and re-attempt verification of only the affected cones. The object is to remove cut-points that contribute to false-negatives, but avoid removing any cut-points that do not contribute to false negatives.

Two conflicting goals in cut-point removal complicate the selection of candidates. The first goal is to avoid making the verification very difficult. This goal argues for very conservative cut-point removal, removing as few cut-points as possible for each re-attempt. The second goal is to complete the verification in as few attempts as possible. This goal argues for very aggressive cut-point removal, or in the logical extreme, never inserting any—this is illustrated most obviously in true failing verifications, for which ultimately all cut-points in all failing cones must be removed. An effective cut-point removal algorithm must strike a balance between overly conservative and overly aggressive cut-point removal.

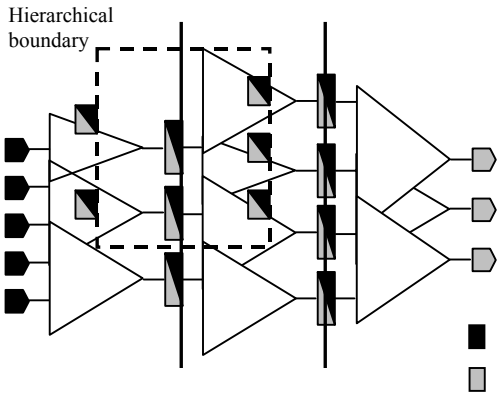

Figure 1. Hierarchical Cut-points.

The method described here makes decisions based in part on the difficulty of previous verification attempts, taking a more conservative approach if previous verification was very difficult, and a more aggressive approach otherwise. Characterization of the previous verification as "difficult" versus "easy" is a heuristic derived from the relative expense of verification solvers it employed.

The "conservative" approach removes cut-points "bottom-up"; that is, it removes cut-points representing lower-level block boundaries before those representing higher-level block boundaries. Within a hierarchical level, it removes cut-points representing boundaries that were "easy" to verify before removing those that were "difficult." In addition, it removes cut-points representing boundaries that have themselves failed verification before those representing boundaries that have succeeded.

The "aggressive" approach also removes cut-points representing boundaries that have themselves failed verification before those representing boundaries that have succeeded, but without regard to hierarchical level or previous verification difficulty.

Although this method is fairly naïve, it has yielded good results. More-sophisticated analysis of previous verification attempts and the relationships between failure points might be expected to yield even better results. Selectively removing cut-points for individual pins, instead of all pins of a given sub-block, is another obvious avenue to explore.

3. EXPERIMENTAL RESULTS

An equivalence checker implementing the described architecture verified the test cases presented in Table 1. This implementation includes underlying equivalence-checking solvers identical to those used in obtaining the previous results. We refer to this implementation as "new."

Figure 2 compares memory and CPU time used by the three methodologies, "flat", "hierarchical", and "new". In 5 of 14 verifications, flat verification did not complete within 40 hours and 4 Gbytes. In 6 of 8 expected succeeding verifications, hierarchical verification yielded a false negative. In contrast, the new approach yielded the correct result and completed within 40 hours and 4 Gbytes in all cases.

It may be noted that for some failing verifications, e.g. G2G1, R2G3, and R2G5, hierarchical verification appears to enjoy a performance advantage over the new approach. This apparent advantage is greatly mitigated by the fact that when hierarchical verification produces a failing result it is generally difficult to ascertain whether the failure is a false negative or not. Further, the failing verification results produced by

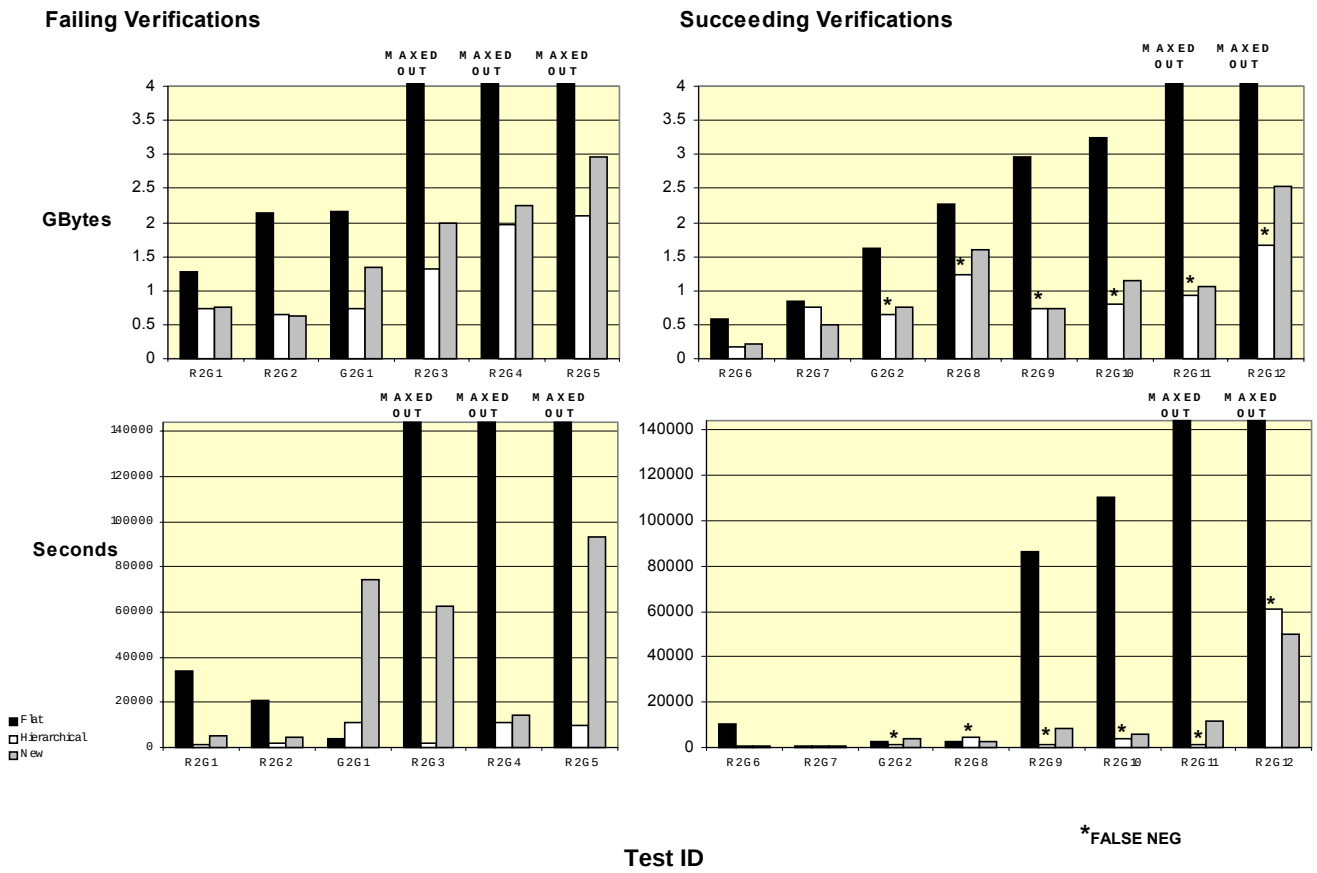

Figure 2. Experimental Results.

hierarchical verification are likely to include false failures in addition to the relevant true failures.

On the other hand, the cost of eliminating false negatives by the new method is modest, as shown by cases G2G2, R2G8, R2G9, R2G10, R2G11, and R2G12.

4. ACKNOWLEDGMENT

Thanks to Daniel Donahue for collecting the data.

5. REFERENCES

[1] "Digital Systems Testing and Testable Design", M. Abramovici, M.A. Breuer, A. D. Friedman, *Computer Science Press*, 1990

[2] "A Practical and Efficient Method for Compare-Point Matching", D. Anastasakis, R. Damiano, T. Ma, T. Stanion, *Proc. Design Automation Conf.*, pages 305-310, 2002

[3] "Verification of Large Synthesized Designs", D. Brand, *Proc. Intl. Conf. on Computer-Aided Design*, pages 534-537, 1993

[4] "Graph-based algorithms for Boolean function manipulation", R.E. Bryant, *IEEE Trans. on CAD,* 1986

[5] "Robust Latch Mapping for Combinational Equivalence Checking", J. R. Burch, V. Singhal, *Proc. Intl. Conf. on Computer-Aided Design*, pages 563–569, 1998

[6] "Tight Integration of Combinational Verification Methods", J. R. Burch, V. Singhal, *Proc. Intl. Conf. on Computer-Aided Design* , pages 570-576, 1998

[7] "Apparatus and method for deriving correspondence between storage elements of a first circuit model and storage elements of a second circuit model", H. Cho, C. Pixley, U. S. Patent 5,638,381, June 1997

[8] "Equivalence Checking Using Cuts and Heaps", A. Kuehlman, F. Krohm,. *Proc. Design Automation Conf.*, pages 263-268, 1997

[9] "An Efficient Equivalence Checker for Combinational Circuits", Y. Matsunaga, *Proc. Design Automation Conf.*, pages 629-634, 1996

[10] "Hierarchical Verification for Equivalence Checking of Designs", L. McIlwain, D. Anastasakis, S. Pilarski, U.S. Patent 6,668,362, December, 2003

[11] "CLEVER: Divide and Conquer Combinational Logic Equivalence VERification with False Negative Elimination", J. Moondanos, Carl Seger, Ziyad Hanna, Daher Kaiss, *13th Conf. on Computer-Aided Verification*, Jul. 2001

[12] "An Efficient Filter-based Approach for Combinational Verification", R. Mukherjee, J. Jain, K. Takayama, M. Fujita, J. A. Abraham, D. S. Fussell, IEEE Trans. On *CAD*, pages 1542-1557, 1999

[13] "A Verification Algorithm for Logic Circuits with Internal Variables", T. Nakaoka, S. Wakabayashi, T. Koide, N. Yoshida, *ISCAS 1995*, pages 1920-1923

[14] "Novel Verification Framework Combining Structural and OBDD Methods in a Synthesis Environment," S. M. Reddy, W. Kunz and D. K. Pradhan, *Proc. Design Automation Conf.*, pp. 414–419, 1995

[15] "Circuit synthesis verification method and apparatus", T. Stanion, U. S. Patent 6,056,784, May 2000

[16] "Equivalence Checking of Hierarchical Combinational Circuits", P. F. Williams, H. Hulgaard, H.R. Andersen, *6th IEEE Intl. Conf. Electronics, Circuits and Systems*, Sep. 1999