Equivalence Checking Using Cuts and Heaps

Andreas Kuehlmann Florian Krohm
IBM Thomas J. Watson Research Center
Yorktown Heights, NY, U.S.A.

Abstract

This paper presents a verification technique which is specifically targeted to formally comparing large combinational circuits with some structural similarities. The approach combines the application of BDDs with circuit graph hashing, automatic insertion of multiple cut frontiers, and a controlled elimination of false negative verification results caused by the cuts. Two ideas fundamentally distinguish the presented technique from previous approaches. First, originating from the cut frontiers, multiple BDDs are computed for the internal nets of the circuit, and second, the BDD propagation is prioritized by size and discontinued once a given limit is exceeded.

1 Introduction

In recent years, formal techniques have become widely accepted in practical design methodologies to verify properties of complex systems. The computational complexity of the corresponding algorithms results in a fundamental trade-off between the generality of the verification model and the size of the designs that can be handled in practice. For example, verifying complex temporal properties using model checking is relatively expensive and often not scalable to designs with a large number of storage elements. Conversely, a combinational verification model significantly limits the expressiveness of the properties to be verified, but is practically applicable to large designs. Today, combinational models are commonly used to prove functional equivalence of two design representations modeled on different levels of abstraction [1, 2]. The matching state encoding of the two models is enforced by the overall design methodology.

An important category of approaches to combinational verification is based on canonical representations of Boolean functions, typically binary decision diagrams (BDDs) or their derivatives. The functions of the two circuits to be compared are converted into canonical forms which are then structurally compared. The major advantage of BDDs is their efficiency for a wide variety of practically relevant combinational circuits. If the BDD size does not grow too large, this type of Boolean reasoning is fast and independent of the actual circuit structure. Moreover, if structural similarities of the two designs are exploited, BDDs can effectively find implications between nets even if they are farther away from the primary inputs [3].

The main problem of BDDs is their exponential memory complexity. If the BDD structure grows too large, their storage and manipulation effort becomes very expensive. Various approaches have been proposed to reduce the complexity of BDD-based equivalence checking by exploiting structural similarities [4, 5, 6, 7]. These techniques have had success because the vast majority of industrial designs contain many intermediate functions that occur in the specification and in the implementation. These nets can be used as cutpoints to partition a complex equivalence check into a set of smaller, simpler comparisons.

Most cutpoint-based verification methods consist of three phases. First, a set of potential cutpoints is identified by using random simulation, ATPG techniques, or BDDs. From these candidates the final cutpoints are chosen by specific selection criteria which are typically difficult to tune to a wider set of applications. Second, the overall verification task is partitioned along these cutpoints into a set of smaller verification problems which are solved independently. And third, in case of miscompares, false negatives due to functional constraints at the partition boundaries are eliminated. There are basically three methods to handle false negatives, all of which have fundamental limitations that are mentioned only briefly or not discussed at all in the corresponding publications.

The first method is based on resubstitution of the cutpoint variables by their incoming functions using the BDD compose operation [5]. This methods is extremely sensitive to the order in which the cutpoints are handled. In the worst case, a bad order might cause the elimination of all cutpoints including the ones which do not cause false negatives. Practically, the problem is aggravated with more cutpoints since the likelihood of false negatives increases. This causes a dilemma for selecting the right number of cutpoints: choosing too few results in a blow up of the forward BDD construction and choosing too many leads to an explosion of the resubstitution.

The second method is based on cut frontiers, defined by the topological order of the cutpoints in the two networks. False negatives are eliminated by succes-

DAC 97, Anaheim, California
(c) 1997 ACM 0-89791-920-3/97/06..$3.50

sively applying the image of the previous cut frontier to constrain the miscompare function of the current cut frontier [6]. This method is basically identical to the backward traversal technique for sequential FSM verification and therefore has similar limitations: the size of the BDD representation of the image tends to blow up in many practical cases.

The third method for eliminating false negatives is based on ATPG techniques to disprove each counter example [7] individually. Here the complexity has shifted into the time domain which makes the approach impractical if the set of miscomparing patterns is large. In addition, if the cutpoints are farther away from the primary inputs and no other functional implications between the two circuits are known, this technique might timeout even on individual counter examples.

This paper describes a verification technique which utilizes BDDs, circuit graph hashing, cutpoint guessing, and false negative elimination. Although the individual ingredients are known, the innovation of the presented method is based on their unique combination and two new ideas. First, the processing of BDDs is prioritized by their size and limited to an upper bound which avoids expensive and often unfruitful attempts to verify intractable problems. Second, the BDD construction is not stopped at cutpoints. Instead, the cutpoint variables start additional BDD frontiers, resulting in multiple, overlapping BDD layers. In case of false negatives, these BDD layers provide multiple candidates for the elimination process. Similar to the BDD construction, the resubstitution of cutpoint variables is prioritized by size and discontinued once a given limit is exceeded.

The resulting verification engine works well for a wide variety of practical circuits. If the number of cutpoints found in the two functions to be compared is insufficient to verify them successfully, the runtime or memory usage will not grow out of hand. Controlled by the size limit, the BDD processing will stop and the problem can be passed to another verification engine, e.g. one based on permissible functions [8]. Furthermore, if the two functions are not equivalent, the presented method will fail to prove it since this generally requires the resubstitution of all cutpoint variables. Instead, the problem is also forwarded to an engine which is specialized in identifying miscomparing functions, e.g. random simulation.

2 Overview of the Presented Approach

Similar to a BDD package, the presented verification technique is implemented as a Boolean reasoning engine providing operations for Boolean function manipulation and comparison. In fact, the interface of the software implementation includes all essential BDD operations and is plug-compatible with a basic BDD package. A non-canonical circuit graph structure is used to store and manipulate functions. During the construction phase, all Boolean operations are mapped onto two-input AND gates (represented by vertices) and inverters (represented by edge attributes). Comparable to the unique table in implementations of BDD packages, a hash table is applied to uniquely identify and merge structurally equivalent parts of the circuit graph.

A request for comparing two Boolean functions is handled in several phases. The overall goal is to merge the two corresponding subgraphs such that both functions are represented by the same vertex. If this is not achieved, it is generally undecidable whether the merging failed due to resource limitations or functional miscomparison. Therefore, the comparison remains undecided and the problem is forwarded to the next verification engine. The algorithm to merge two subgraphs can be summarized as follows (see subsequent sections for further details):

- To identify functionally identical vertices of the circuit graph which are not found by hashing, BDDs are computed starting at the primary inputs. Contrary to traditional BDD processing based on depth-first or breadth-first traversal, a sorted heap similar to event heaps of event-driven simulators is used to control the propagation of the BDDs. If during propagation, the BDD size exceeds a given limit, its processing is cancelled.
- By using cross-references between BDD nodes and the corresponding circuit graph vertices, subgraphs with identical functions can easily be identified. If found, functionally equivalent vertices are merged and the subsequent part of the circuit graph is rehashed. The merged vertices are marked as potential cutpoints for the next phase.
- If the heap is empty and the two outputs have not been proven identical, any previously merged vertices are used as cutpoints to inject new BDD events onto the heap. The cutpoints are levelized according to their topological depths. All cutpoints at a given level establish a cut frontier which initiates another layer of BDD construction throughout the circuit graph.
- After processing of all cut-frontiers, the resulting BDDs at the output vertices are checked for false negatives. Similar to the forward propagation, this step uses a heap to resubstitute the BDDs with the smallest size first. If all cutpoint variables are composed or all resulting BDDs exceed the given size limit, the process stops and the verification problem remains unsolved.

There are few peculiarities which ensure the effectiveness of the presented technique for a wide variety of practical applications. First, using AND/INVERTER structures as a base system to represent Boolean functions in conjunction with vertex hashing results in a very efficient verification engine for circuits which are structurally identical except for changes caused by simple technology mapping steps. This kind of functional comparison is frequently needed to verify trivial post-synthesis transformations such as scan-path insertion, buffer optimizations etc. Second, the chosen prioritized BDD processing implements a propagation scheme which follows the size of the BDDs in the two circuits. This ensures that functionally identical vertices are found as early as possible, avoiding the processing of unnecessary BDDs. Third, after two vertices

are identified as functionally identical, the merging of the subsequent parts of the circuit graph by rehashing can often be processed all the way to the outputs. This effectively saves the construction of BDDs for several levels in the graph. And last, since the propagation is not stopped at the cuts, the BDD layers generally overlap significantly, which considerably reduces the likelihood of false negatives.

3 Basic Algorithm for Equivalence Checking

Generally, verifying functional equivalence of two circuits is performed in two steps. First, a circuit model is constructed based on primitive Boolean operations and then the actual comparison is performed on that model. In the presented approach, a circuit graph is built by converting all Boolean operations into a structure using two-input AND gates and inverters. During the graph construction, each vertex is entered into a hash table using the vertices of the two input operands and their polarities as key. Since identical vertex keys are a sufficient condition for structural equivalence, the hash table can be used during graph construction to map isomorphic parts of the two circuits onto the same subgraph.

```
Algorithm Check_Equivalence (v1, v2) {
  if (v1 == v2)          return equal;
  if (v1 == NOT(v2))     return not_equal;
  for all primary inputs i do {
     bdd_i = create_bdd_variable ();
     store_vertex_at_bdd (bdd_i, i);
     put_on_heap (heap, bdd_i);
  }
  while (heap != empty) do {
     bdd = get_smallest_bdd (heap);
     v   = get_vertex_from_bdd (bdd);
         /* check if handled before */
     if (get_bdd_from_vertex (v)) continue;
     store_bdd_at_vertex (v, bdd);
     for all fanout vertices v_out of v do {
         bdd_left  = get_bdd_from_vertex (v_out->left);
         bdd_right = get_bdd_from_vertex (v_out->right);
         bdd_res   = bdd_and (bdd_left, bdd_right);
         v_res     = get_vertex_from_bdd (bdd_res);
         if (v_res) {
             merge_vertices (v_res, v_out);
             if (v1 == v2)          return equal;
             if (v1 == NOT(v2))     return not_equal;
         } else {
             store_vertex_at_bdd (bdd_res, v_out);
         }
         put_on_heap (heap, bdd_res);
     }
  }
  return undecided;
}
```

Figure 1: Algorithm for heap-based BDD processing.

Figure 2 illustrates the construction of the circuit graph for a simple example. The two circuits in Figure 2(a) are structurally different but implement the same function. Figure 2(b) shows the result of the graph construction after the first phase. The vertices of the graphs represent AND functions. Similarly, the filled dots at the edges symbolize the inverters. Note, that the functions $a \wedge b$ of the first circuit and $\overline{a} \vee \overline{b}$ of the second circuit are identified as structurally equivalent (modulo inversion) and mapped to the same vertex 1 in the graph model. No other parts of the two circuits could be merged by the initial hashing process.

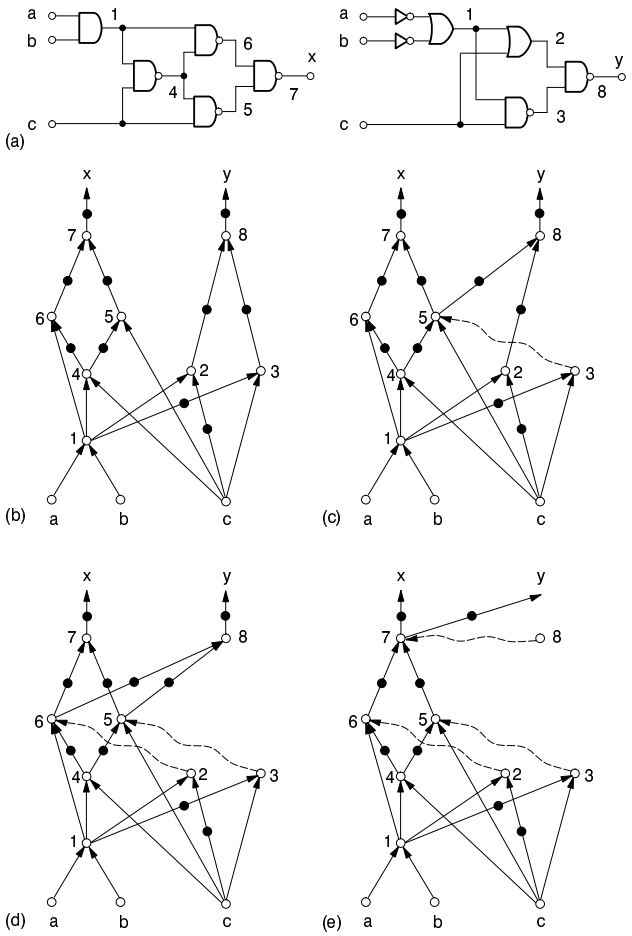

Figure 2: Example for circuit graph manipulation: (a) two functionally identical circuits, (b) original graph for both circuits, (c) BDDs are computed for vertices 1,2,3,4,5 which causes 5 and 3 to merge, (d) BDD is computed for 6 which causes 6 and 2 to merge, (e) forward hashing causes 7 and 8 to merge and solves the verification problem.

After the graph construction is finished, the equivalence check is performed by the algorithm shown in Figure 1. The following remarks are given for further clarification: procedure `put_on_heap` stores a BDD on the heap only if its size is smaller than the given limit; otherwise the BDD node is freed and disregarded. The notations v`->left` and v`->right` refer to the two incoming operands of vertex v. Procedures `get_bdd_from_vertex` and

get_vertex_from_bdd provide cross-references between BDD nodes and the corresponding circuit graph vertices, and handle inverted edges and inverted BDD nodes internally.

The overall idea of the algorithm is to merge the subgraphs of the two output vertices using BDDs to prove functional equivalence of intermediate vertices. The BDD propagation is controlled by a sorted heap. First, for each primary input a BDD variable is created and entered onto the heap. Then an iterative process removes the smallest BDD from the heap, processes the Boolean operation for the immediate fanout of the corresponding circuit graph vertex, and reenters the resulting BDDs onto the heap. Functionally equivalent vertices found during that process are immediately merged and the subsequent parts of the circuit graph are rehashed by the routine merge_vertices. The rehashing is applied in depth-first order starting from the merged vertex toward the primary outputs and stops if no further reconvergency is found.

Figures 2(c-e) illustrate the results of the equivalence checking algorithm for the circuit graph of Figure 2(b). It is assumed that the BDDs are processed in the order of their corresponding vertices $1, 2, 3, 4, 5$, and 6. The first four iterations create the BDDs for vertices $1, 2, 3$, and 4. In the next iteration, the computed BDD for vertex 5 points to the functionally equivalent vertex 3. Therefore, vertices 5 and 3 are merged as indicated in Figure 2(c). The next figure shows the graph after vertex 6 has been processed and merged with vertex 2. The subsequent forward rehashing identifies vertices 7 and 8 as structurally identical and merges them which yields the graph structure of Figure 2(e). At this point the equivalence of both outputs is proven and the algorithm terminates without building BDDs for the last level of the two circuits.

4 Advanced Algorithm Using Cut Frontiers

The algorithm of Figure 2 handles BDDs up to a maximum size only. Therefore, the heap processing potentially terminates without succeeding to merge the two output vertices, even if they are functionally equivalent. In order to exploit the structural similarities found in the previous phase, all vertices that have been merged are now used as cutpoints to inject new BDD variables onto the heap.

The cut level c_level(v) of circuit graph vertex v is defined as follows:

$$\texttt{c\_level}(v) = \begin{cases} 0 & \textbf{if } v \textit{ is primary input} \\ max(\texttt{c\_level}(v\texttt{->left}), \texttt{c\_level}(v\texttt{->right}))+1 & \\ \quad \textbf{if } v \textit{ is cut point} & \\ max(\texttt{c\_level}(v\texttt{->left}), \texttt{c\_level}(v\texttt{->right})) & \\ \quad \textbf{otherwise.} & \end{cases}$$

All cutpoints with identical cut levels are assigned to a cut frontier which initiates an independent layer of BDD propagation through the circuit graph. Since the layers generally overlap, this scheme effectively generates multiple BDDs for each graph vertex. Intuitively, this diversity increases the chance of merging subgraphs and decreases the likelihood of false negatives.

```
Algorithm Check_Equivalence_with_Cuts (v1,v2) {
  for all vertices c that have been merged before do {
     bdd_c     = create_bdd_variable ();
     level_c   = c_level (c);
     store_level_at_bdd (bdd_c, level_c);
     put_on_heap (heap, bdd_c);
  }
  while (heap != empty) do {
     bdd       = get_smallest_bdd (heap);
     v         = get_vertex_from_bdd (bdd);
     level     = get_level_from_bdd (bdd);
     if (get_bdd_from_vertex (v)) continue;
     store_bdd_at_vertex (v, bdd, level);
     for all fanout vertices v_out of v do {
         bdd_left  = get_bdd_from_vertex (v_out->left, level);
         bdd_right = get_bdd_from_vertex (v_out->right, level);
         bdd_res   = bdd_and (bdd_left, bdd_right);
         v_res     = get_vertex_from_bdd (bdd_res);
         if (v_res) {
             merge_vertices (v_res, v_out);
             if (v1 == v2)          return equal;
             if (v1 == NOT(v2))     return not_equal;
         } else {
             store_vertex_at_bdd (bdd_res, v_res);
             store_level_at_bdd (bdd_res, level);
         }
         put_on_heap (heap, bdd_res);
     }
  }
  return undecided;
}
```

Figure 3: Heap-based BDD propagation with cut frontiers.

An extended version of the algorithm of Figure 1 is used to implement the multi-layer propagation of BDDs. As shown in Figure 3, the additions mainly involve a level-oriented handling of BDDs. The procedure get_bdd_from_vertex returns the BDD stored for the specified level at the vertex. If the given level exceeds the cut level of the vertex, the BDD of the cut level itself is taken. The algorithm of Figure 3 is called repeatedly until no new cut frontiers are found or equivalence of the two outputs is proven.

5 Elimination of False Negatives

As discussed in the previous section, the application of cutpoints can potentially introduce false negative verification results. This occurs, if the insertion of cut frontiers produces different BDDs for two functionally equivalent output vertices. To prove equivalence for those cases, the cutpoint variables that support these BDDs need to be resubstituted by their driving functions. As discussed above, this resubstitution process potentially results in a blow-up of the BDDs. In order to fully explore all BDD pairs constructed for the two outputs without running into memory explosion, the elimination process is also controlled by a heap.

Figure 4 shows the corresponding algorithm to eliminate false negatives. First, the heap is initialized with all BDDs computed for the two output vertices. Then, in each iteration, the BDD with the smallest size is taken and its topmost cut variable resubstituted by the corresponding driving function. The resulting BDD is

```
Algorithm Eliminate_False_Negatives (v1,v2) {
  for level = 0 to c_level (v1) do {
     bdd = get_bdd_from_vertex (v1, level);
     put_on_heap (compose_heap, bdd);
  }
  for level = 0 to c_level (v2) do {
     bdd = get_bdd_from_vertex (v2, level);
     put_on_heap (compose_heap, bdd);
  }
  while (compose_heap != empty) do {
     bdd     = get_smallest_bdd (compose_heap);
     v       = get_vertex_from_bdd (bdd);
     bdd_var = get_cutvar_from_bdd (bdd);
     if (bdd_var) {
         v_var   = get_vertex_from_bdd (bdd_var);
         level   = c_level (v_var);
         bdd_func = get_bdd_from_vertex (v_var, level - 1);
         bdd_res  = bdd_compose (bdd, bdd_var, bdd_func);
         v_res    = get_vertex_from_bdd (bdd_res);
         if (v_res) {
             merge_vertices (v_res, v);
             if (v1 == v2)          return equal;
             if (v1 == NOT(v2))     return not_equal;
         } else {
             store_vertex_at_bdd (bdd_res, v);
             put_on_heap (compose_heap, bdd_res);
         }
     }
  }
  return undecided;
}
```

Figure 4: Algorithm to eliminate false negatives.

then checked for a functionally equivalent vertex that has been processed before. If found, both vertices are merged and the subsequent parts of the circuit graph are rehashed. Otherwise, if the size of the resulting BDD is smaller than the given limit, it is reentered onto the heap for further processing.

6 Implementation Details

The presented verification technique is implemented in the verification tool Verity [9] which is in practical use for various microprocessor projects within IBM. The approach is embedded in a scenario which starts with randomly simulating 32 patterns to uncover the majority of miscomparing outputs quickly and to simulate trivial functions [10] exhaustively. Then the described engine is applied in several iterations with increasing limits for the BDD size. After that, the remaining problems are forwarded to other BDD- and ATPG-based reasoning engines.

In practice, a verification approach based on multiple engines is quite powerful since each engine can be tuned for a specific class of designs. For example, if the two designs to be compared are very similar, the presented engine works highly efficient using a small limit on the BDD size. Contrarily, with larger limits, the engine can handle design pairs with significant structural differences, but also uses more time to build the layered BDD representation.

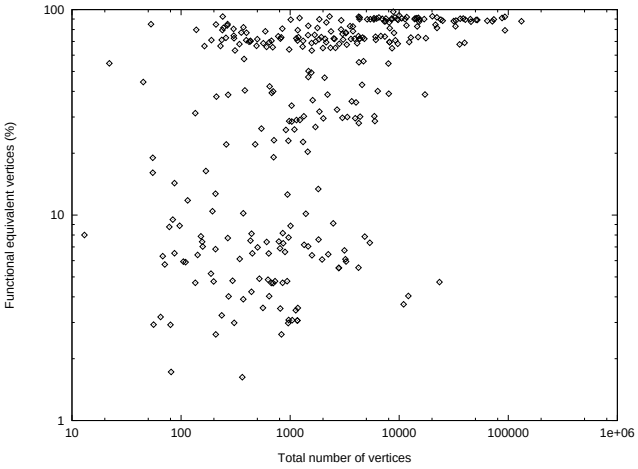

Figure 5: Number of functionally equivalent vertices versus total number of vertices in typical circuit graphs.

7 Practical Experiments

This section presents practical results to support the previous claims. The first experiment was conducted to validate the assumptions that many industrial circuits are structurally similar and that the presented method can effectively exploit this property. The test is based on a suite of approximately 300 circuits taken from several PowerPC, System/390, and AS/400 microprocessor designs within IBM. These circuits cover the whole complexity spectrum from simple data-path components to full chips.

To measure the structural similarity of the two designs to be compared, the number of vertices in the circuit graph representations were counted which have functionally equivalent counterparts. The numbers include all hash table matches during graph construction, the merge operations during BDD propagation, and the matches during the following forward rehashing step. Note that the effect of constant folding is not included, since it is highly dependent on the implementation of the switch-level extraction algorithm. The results for the 300 designs are shown in Figure 5. As displayed in the picture, for about 80 % of the circuits, more than 80 % of the graph vertices did find an equivalent counterpart vertex. This attests to the fact that equivalence checking in practice can and should heavily exploit structural similarity.

Next, the performance of the presented technique was measured for a number of IBM internal circuits. The tests were based on the verification tool Verity and performed on a RS/6000 workstation model 390, the results are shown in Table 1. The second and third column report the design complexity in terms of the number inputs, outputs, gates, and transistors. The next column shows the number of functional comparisons and consistency checks performed by Verity. The reported seven designs are taken from various microprocessor designs: D9000 is a large chip where all functional components are black-boxed. This design exercises an extreme verification case with no logic but a huge number of verification problems. As shown, the runtime and memory consumption is reasonable and

Design	Inputs/ Outputs	Gates/ Transistors	Comparisons/ Checks	CPU (sec)	Memory (MB)
D9000	202615 / 84354	- / -	406068 / 3109078	13922.4	670.3
D9001	47370 / 5802	1024763 / 14460	93236 / 687792	1785.7	552.7
D9002	2383 / 3325	28432 / 57806	7316 / 86066	3437.7	83.1
D9003	1519 / 482	29807 / -	3108 / 44765	1760.9	31.8
D9004	11661 / 2398	51759 / -	23838 / 100204	572.0	47.8
D9005	209 / 84	1923 / 7874	168 / 8192	24.0	14.0
D9006	133 / 247	3677 / 16672	594 /12130	136.2	25.1

Table 1: Verification performance for selected circuits.

expected to grow linearly with larger chips. D9001 is the flat design of a complete microprocessor. The two representations to be compared are modeled on gate- and transistor-level and are structurally very similar. In this case, the graph hashing solves the majority of the verification problems, where slight irregularities are effectively "bridged" by the BDD propagation. D9002 is a multiplier circuit for which the gate-level representation is compared against the custom-designed transistor-level implementation. The larger structural difference is clearly reflected in a larger runtime effort to compare them. D9003 and D9004 are two designs which previously could not be verified without manually partitioning them into smaller pieces. Using the presented approach, Verity can handle these circuits for the first time automatically. The last two designs are typical data-path units.

Overall, the proposed engine greatly extends the class of designs which can be handled automatically. For example, with the exception of D9000, none of the industrial designs of Table 1 could be verified in a reasonable amount of time using a BDD engine only.

8 Conclusions

The paper presents a new method to perform functional comparison of combinational circuits using BDDs, circuit graph hashing, cutpoint guessing, and false negative elimination. To generally avoid a memory blow-up, the BDD construction is limited to a maximum size. A BDD propagation scheme controlled by a sorted heap in combination with vertex hashing of the circuit graph effectively exploits structural similarities of the two circuits to be compared. In addition, functionally identical internal nets are used as cutpoints to partition the verification problems into a set of smaller, possibly easier tasks. These cutpoints are combined to start new overlapping BDD propagation layers which potentially get closer to the outputs. Similar to the construction process, the elimination of false negatives by resubstituting the cutpoint variables is controlled by a sorted heap.

Overall, the presented approach performs efficiently for a wide variety of designs with some degree of structural similarity. Compared to a pure BDD approach, many practical designs can now be verified without manual partitioning; others run significantly faster. The combination of the presented technique with alternative verification engines specialized in other classes of designs results in a powerful practical verification tool which is robust and efficient for most applications.

9 Acknowledgments

The authors would like to thank Arjen Mets, Mark Williams, and Jiazhao Xu from IBM Fishkill, Victor Rodriguez from IBM Austin, Robert Kanzelman from IBM Rochester, and Ee Cho from IBM Poughkeepsie for their significant contributions to develop and support the verification tool Verity. They also wish to thank Geert Janssen from the Technical University Eindhoven for providing his BDD-package and for valuable technical discussions.

References

[1] G. L. Smith, R. J. Bahnsen, and H. Halliwell, "Boolean comparison of hardware and flowcharts," *IBM Journal of Research and Development*, vol. 26, pp. 106–116, January 1982.

[2] D. P. Appenzeller and A. Kuehlmann, "Formal verification of a PowerPC microprocessor," in *Proceedings of the IEEE International Conference on Computer Design*, (Austin, TX), pp. 79–84, IEEE, October 1995.

[3] J. Jain, R. Mukherjee, and M. Fujita, "Advanced verification technique based on learning," in *Proceedings of the 32th ACM/IEEE Design Automation Conference*, (San Francisco), pp. 420–426, ACM/IEEE, June 1995.

[4] C. L. Berman and L. H. Trevillyan, "Functional comparison of logic designs for VLSI circuits," in *Digest of Technical Papers of the IEEE International Conference on Computer-Aided Design*, pp. 456–459, IEEE, November 1989.

[5] C. van Eijk and G. Janssen, "Exploiting structural similarities in a BDD-based verification method," in *Proceedings of the 2nd International Conference on Theorem Provers in Circuit Design*, pp. 110–125, 1995.

[6] Y. Matsunaga, "An efficient equivalence checker for combinatorial circuits," in *Proceedings of the 33th ACM/IEEE Design Automation Conference*, (Las Vegas), pp. 629–634, ACM/IEEE, June 1996.

[7] S. M. Reddy, W. Kunz, and D. K. Pradhan, "Novel verification framework combining structural and OBDD methods in a synthesis environment," in *Proceedings of the 32th ACM/IEEE Design Automation Conference*, (San Francisco), pp. 414–419, ACM/IEEE, June 1995.

[8] D. Brand, "Verification of large synthesized designs," in *Digest of Technical Papers of the IEEE International Conference on Computer-Aided Design*, (Santa Clara, CA), pp. 534–537, IEEE, November 1993.

[9] A. Kuehlmann, A. Srinivasan, and D. P. LaPotin, "Verity - a formal verification program for custom CMOS circuits," *IBM Journal of Research and Development*, vol. 39, pp. 149–165, January/March 1995.

[10] F. Krohm, A. Kuehlmann, and A. Mets, "The use of random simulation in formal verification," in *Proceedings of the IEEE International Conference on Computer Design*, (Austin, TX), pp. 371–376, IEEE, October 1996.