Robust Boolean Reasoning for Equivalence Checking and Functional Property Verification

Andreas Kuehlmann, *Senior Member, IEEE*, Viresh Paruthi, Florian Krohm, and Malay K. Ganai, *Member, IEEE*

***Abstract*—Many tasks in computer-aided design (CAD), such as equivalence checking, property checking, logic synthesis, and false paths analysis, require efficient Boolean reasoning for problems derived from circuits. Traditionally, canonical representations, e.g., binary decision diagrams (BDDs), or structural satisfiability (SAT) methods, are used to solve different problem instances. Each of these techniques offer specific strengths that make them efficient for particular problem structures. However, neither structural techniques based on SAT, nor functional methods using BDDs offer an overall robust reasoning mechanism that works reliably for a broad set of applications. The authors present a combination of techniques for Boolean reasoning based on BDDs, structural transformations, an SAT procedure, and random simulation natively working on a shared graph representation of the problem. The described intertwined integration of the four techniques results in a powerful summation of their orthogonal strengths. The presented reasoning technique was mainly developed for formal equivalence checking and property verification but can equally be used in other CAD applications. The authors' experiments demonstrate the effectiveness of the approach for a broad set of applications.**

***Index Terms*—BDD, Boolean reasoning, equivalence checking, formal verification, property checking, SAT.**

I. Introduction

MANY tasks in computer-aided design (CAD) such as equivalence or property checking, logic synthesis, timing analysis, and automatic test-pattern generation, require Boolean reasoning on problems derived from circuit structures. There are two main approaches used alternatively for such applications. First, by converting the problem into a functionally canonical form such as binary decision diagrams (BDDs), the solution can be obtained from the resulting diagram. Second, structural satisfiability (SAT) procedures perform a systematic search for a consistent assignment on the circuit representation. The search either encounters a solution or, if all cases have been enumerated, concludes that no solution exists. Both approaches generally suffer from exponential worst case complexity. However, they have distinct strengths and weaknesses which make them applicable to different classes of practical problems. A monolithic integration of SAT and BDD-based techniques could combine their individual strengths and result in a powerful solution for a wider range of applications. Additionally, by including random simulation its efficiency can be further improved for problems with many satisfying solutions.

A large fraction of practical problems derived from the above-mentioned applications have a high degree of structural redundancy. There are three main sources for this redundancy: first, the primary netlist produced from a register transfer level (RTL) specification contains redundancies generated by language parsing and processing. For example, in industrial designs, between 30% and 50% of generated netlist gates are redundant [1]. A second source of structural redundancy is inherent to the actual problem formulation. For example, a miter structure [2], built for equivalence checking, is globally redundant. It also contains many local redundancies in terms of identical substructures used in both designs to be compared. A third source of structural redundancy originates from repeated invocations of Boolean reasoning on similar problems derived from overlapping parts of the design. For example, the individual paths checked during false paths analysis are composed of shared subpaths which get repeatedly included in subsequent checks. Similarly, a combinational equivalence check of large designs is decomposed into a series of individual checks of output and next-state functions which often share a large part of their structure. An approach that detects and reuses structural and local functional redundancies during problem construction could significantly reduce the overhead of repeated processing of identical structures. Further, a tight integration with the actual reasoning process can increase its performance by providing a mechanism to efficiently handle local decisions.

In this paper, we present an incremental Boolean reasoning approach that integrates structural circuit transformation, BDD sweeping [3], a circuit-based SAT procedure, and random simulation in one framework. All four techniques work on a shared AND/INVERTER graph [3] representation of the problem. BDD sweeping and SAT search are applied in an intertwined manner both controlled by resource limits that are increased during each iteration [4]. BDD sweeping incrementally simplifies the graph structure, which effectively reduces the search space of the SAT solver until the problem can be solved. The set of circuit transformations get invoked when the sweeping causes a structural change, potentially solving the problem or further simplifying the graph for the SAT search. Furthermore, random simulation can efficiently handle problems with dense solution spaces.

This paper is structured as follows. Section II summarizes previous work in the area and contrasts it to our contributions. Section III presents the AND/INVERTER graph representation,

Manuscript received December 28, 2001; revised April 18, 2002. This paper was recommended by Associate Editor J. H. Kukula.

A. Kuehlmann is with the Cadence Berkeley Labs, Berkeley, CA 94704 USA (e-mail: kuehl@cadence.com).

V. Paruthi is with the IBM Enterprise Systems Group, Austin, TX 78758 USA (e-mail: vparuthi@us.ibm.com).

F. Krohm is with the IBM Microelectronic Division, Hopewell Junction, NY 12533 USA (e-mail: florian@edamail.fishkill.ibm.com).

M. K. Ganai is with the NEC C&C Research Labs, Princeton, NJ 08540 USA (e-mail: malay@nec-lab.com).

Digital Object Identifier 10.1109/TCAD.2002.804386

which is shared among all reasoning mechanisms and outlines the set of transformations that are applied for its simplification. Section IV presents the BDD sweeping algorithm and Section V outlines the details of the circuit-based SAT procedure. The random simulation algorithm and overall reasoning flow are described in Sections VI and VIII, respectively. The last two sections present experimental results and conclusions.

II. Previous Work

SAT search has been extensively researched in multiple communities. Many of the published approaches are based on the Davis–Putnam procedure [5], [6], which executes a systematic case split to exhaustively search the solution space. Over the years, many search tactics improvements have been published. The most notable implementations of CNF-based SAT solvers are GRASP [7] and Chaff [8]. Classical CNF-based SAT solvers are difficult to integrate with BDD methods and dynamically applied circuit transformations because they use a clause-based representation of the problem. In this paper, we describe an implementation of an SAT procedure that works directly on an AND/INVERTER graph allowing a tight interaction with BDD sweeping, local circuit graph transformations, and random simulation. We describe a modified implementation of nonchronological backtracking and conflict-based learning and present an efficient means to statically learn implications.

Trading-off compactness of Boolean function representations with canonicity for efficient reasoning in computer-aided design (CAD) applications has been the subject of many publications. BDDs [9], [10] map Boolean functions onto canonical graph representations and thus are one extreme of the spectrum. Deciding whether a function is a tautology can be done in constant time, at the possible expense of an exponential graph size. XBDDs [11] propose to divert from the strict functional canonicity by adding function nodes to the graph. The node function is controlled by an attribute on the referencing arc and can represent an AND or OR operation. Similar to BDDs, the functional complement is expressed by a second arc attribute and structural hashing identifies isomorphic subgraphs on the fly. The proposed tautology check is similar to a technique presented in [12] and is based on recursive inspection of all cofactors. This scheme effectively checks the corresponding BDD branching structure sequentially, resulting in exponential runtime for problems for which BDDs are excessively large.

Another form of a noncanonical function graph representation are BEDs [13]. BEDs use a circuit graph with six possible vertex operations. The innovative component of BEDs is the application of local functional hashing, which maps any four-input substructure onto a canonical representation. Tautology checking is based on converting the BED structure into a BDD by moving the variables from the bottom of the diagram to the top. Similar to many pure cutpoint-based methods, this approach is highly sensitive to the ordering in which the variables are pushed up. In our approach, we apply an extended functional hashing scheme to an AND/INVERTER graph representation. Since our graph preserves the AND clustering, the hashing can take advantage of its commutativity which makes it less sensitive to the order in which the structure is built. If the structural method fails, we apply BDD sweeping on the circuit graph for checking tautology. Due to the multiple frontier approach described later, it is significantly more robust than the BED to BDD conversion process.

There are numerous publications that proposed the application of multiple methods to solve difficult reasoning instances. For example, in [14] and [3] the authors presented a random simulation algorithm and the mentioned method based on structural hashing and BDDs, respectively, and suggested their application in a multiengine setting. In [15], a comprehensive filter-based approach is described that successively applies multiple engines including structural decomposition, BDDs, and ATPG to solve combinational equivalence checking problems. All these techniques have in common that they apply multiple specialized techniques in a sequential independent manner. In contrast, the presented approach tightly intertwines the use of structural methods, BDD-based techniques, and an SAT search and applies them on a single uniform data representation. The proposed setting allows an automatic adaptation of the combined algorithm to match a given problem structure that results in a significant increase in the overall reasoning power.

Several publications have suggested an integration of SAT and BDD techniques for Boolean reasoning. Cutpoint-based equivalence checking uses a spatial problem partitioning and can be employed as a base to apply SAT and BDDs in distinct parts of the miter structure. A particular approach [16] first builds a partial output BDD starting from the cutset where auxiliary variables are introduced. It then enumerates the onset cubes of this BDD and applies an SAT search for justifying those cubes from the primary inputs. This method becomes intractable if the BDD includes many cubes in its onset. Further, the actual justification of individual cubes may timeout if the cutset is chosen unwisely. A modification of this approach suggests searching through all cofactors of the BDD instead of enumerating all cubes [17]. Another proposal to combine BDD and SAT is based on partitioning the circuit structure into a set of components [18]. As most cutpoint-based methods, all these approaches are highly sensitive to the chosen partitioning.

A common problem with the mentioned integration approaches is the insertion of BDD operations into the inner loop of a structural SAT search. Structural SAT is efficient if the underlying problem structure can be exploited for effective local search heuristics. BDDs work well if redundancy of the problem structure eludes an exponential growth during construction. A spatial partitioning of the application space for BDDs and SAT blurs their individual global scope and separates the application of their orthogonal strengths to different parts. In this paper, we apply BDD sweeping and structural SAT search, both working in an interleaved manner on the entire problem representation. This keeps both mechanisms focused on the global structure without being constrained by an arbitrary prepartitioning. In this setting, BDD sweeping incrementally reduces the search space for the SAT solver until the problem is solved or the resource limits are exhausted. Structural transformations are used to facilitate local decisions.

```
Algorithm create_and2(p1,p2) {
   /* constant folding */
   if (p1 == CONST_0)  return CONST_0;
   if (p2 == CONST_0)  return CONST_0;
   if (p1 == CONST_1)  return p2;
   if (p2 == CONST_1)  return p1;
   if (p1 == p2)       return p1;
   if (p1 == ¬p2)      return CONST_0;
   /* rank order inputs to catch commutativity */
   if (rank(p1) > rank(p2)) swap(p1,p2);
   /* check for isomorphic entry in hash table */
   if ((p = hash_lookup(p1,p2)) == NULL)
      p = new_and_vertex(p1,p2);
   return p;
}
```

Fig. 1. Algorithm **create_and2** for the AND constructor.

```
Algorithm new_and_vertex(p1,p2) {
   /* reschedule vertex for BDD sweeping */
   put_on_heap(bdd_from_vertex(p1),upper_size_limit);
   put_on_heap(bdd_from_vertex(p2),upper_size_limit);
   /* learn implication shortcuts for SAT */
   if (hash_lookup(¬p1,p2))  learn(p1,p2);
   if (hash_lookup(p1,¬p2))  learn(p2,p1);
   p = alloc_vertex(p1,p2);
   add_to_hash_table(p,p1,p2);
   return p;
}
```

Fig. 2. Algorithm **new_and_vertex** for allocating a new graph vertex, including restarting of BDD sweeping (Section IV) and static learning (Section V).

III. Problem Representation and Structural Transformations

In this section, we describe the basic AND/INVERTER graph representation that is employed as an underlying data structure for all Boolean reasoning algorithms described in the following sections. We also present several hashing schemes that remove structural and local functional redundancies during graph construction.

A. AND/INVERTER Graph Representation and Structural Hashing

A directed acyclic graph is used as a structural representation of the functions to be reasoned about. There are three types of graph vertices: a unique terminal vertex represents the constant "0" ("1") value when it is referenced by a noncomplemented (complemented) arc. A second type of vertex has no incoming arcs and models primary inputs. The third vertex type has two incoming arcs and represents the AND of the vertex functions referenced by the two arcs. INVERTER attributes on the graph arcs indicate Boolean complementation. Using this graph representation, a reasoning problem is expressed as an obligation to prove a particular graph vertex to be constant "0" or "1".

Similar to the construction of BDDs, the AND/INVERTER graph is built from the inputs toward the outputs using a set of construction operators. There are three basic constructors: (1) **create_input**, (2) **create_and2**, and (3) **create_inverter**. Other operators for alternative or more complex operations are composed of these basic constructors. Intermediate functions are passed between constructors by arc handles, which consist of a reference to the source vertex and a possible INVERTER attribute. The same handles are applied by the reasoning application to refer to functions that are stored by the graph.

The implementation of the construction operation **create_input** is straightforward. It allocates and initializes a corresponding vertex data structure and returns a handle pointing to it. Similarly, the operation **create_inverter** simply toggles the attribute of the handle. Fig. 1 shows the pseudo-code for the operati2pon **create_and2**. The algorithm takes two arc handles as input parameters and returns an arc handle that represents the output of the AND operation. In the code, the symbol "¬" denotes Boolean complemention using the procedure **create_inverter**. The first part of the algorithm performs constant folding, which automatically simplifies redundant and trivial expressions in the graph structure. Next a hash-lookup identifies isomorphic graph structures and eliminates them during construction. For this the procedure **hash_lookup** checks whether an AND vertex with the requested input arcs has been created before. If found the existing ver2ptex is reused, otherwise a new vertex is created using the function **new_and_vertex**. Before applying the hash-lookup the two operators p_1 and p_2 are ordered using a unique ranking criteria. This assures that commutative expressions, such as $p_1 \wedge p_2$ and $p_2 \wedge p_1$, are merged onto the same graph vertex.

The algorithm **new_and_vertex** is shown in Fig. 2. It is used to allocate a new graph vertex and add a corresponding entry to the hash table. This procedure also handles the reactivation of the BDD sweeping algorithm and static learning as described in Sections IV and V, respectively.

The construction of the AND/INVERTER graph for a simple example is illustrated in Fig. 3. Fig. 3(a) represents a circuit built for proving equivalence of nets x and y, which are functionally identical but have different structural implementations. Functionally equivalent nets are labeled using identical numbers with one or more apostrophes. Fig. 3(b) shows the result of the graph construction using the algorithm **create_and2** of Fig. 1. The vertices of the graphs represent AND functions and the filled dots on the arcs symbolize the INVERTER attributes. Note that in several cases structurally isomorphic nets are mapped onto the same graph vertices. For example, the functions $\overline{a \vee b}$ (net 1 of the upper circuit) and $\overline{a} \wedge \overline{b}$ (net 1" of the lower circuit) are identified as structurally equivalent and represented by a single vertex.

B. Functional Hashing

The simple two-level hashing scheme of algorithm **create_and2** can eliminate structurally isomorphic graph vertices but cannot handle functionally identical vertices that are implemented by different structures. For example, the equivalence of vertices x and y of the circuit in Fig. 3 cannot be shown by simple hashing. In this section, we present a generalized hashing scheme that identifies functionally identical subcircuits of bounded size independent of their actual structural implementation.

A natural way to increase the scope of structural hashing would be to divert from the two-input graph model and use vertices with higher fanin degree. The set of possible functions of a vertex with more than two inputs cannot be encoded efficiently

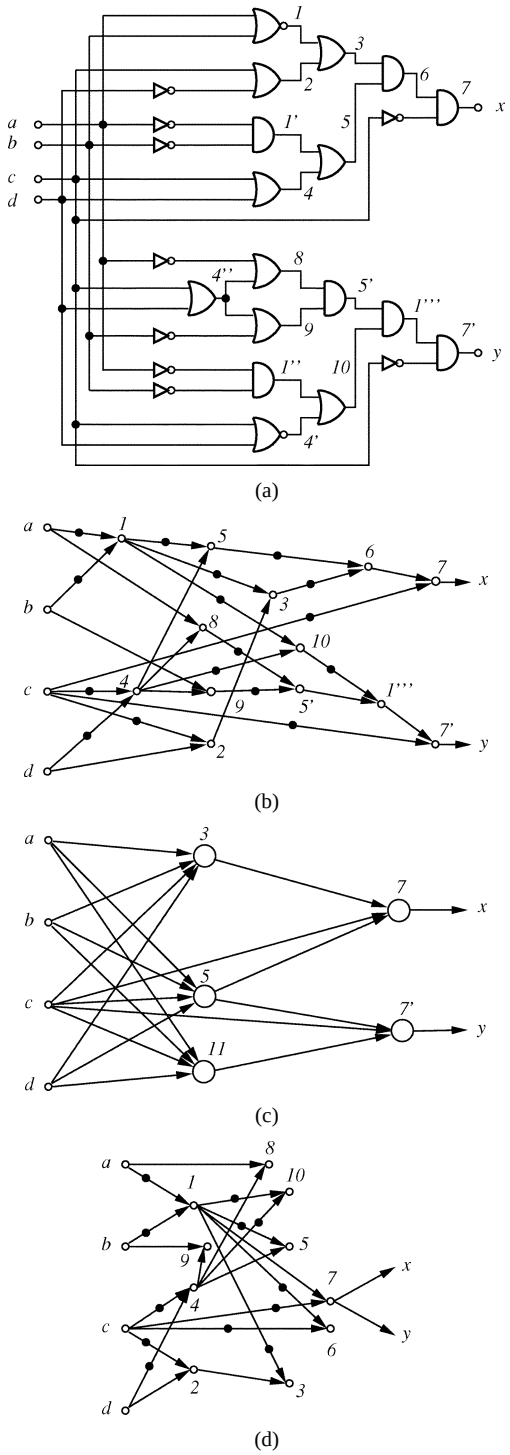


Fig. 3. Example for the construction of an AND/INVERTER graph: (a) functionally redundant structure generated to check functional equivalence of outputs x and y; (b) corresponding two-input AND/INVERTER graph built by algorithm **create_and2** of Fig. 1; (c) alternative graph representation with four-input vertices; and (d) resulting two-input graph after functional hashing using the algorithm **create_and** of Fig. 4.

using uniform vertex operations and arc attributes only. Instead, the vertex function should be represented by an attribute which is hashed in conjunction with the input references to find structurally identical circuit parts. Since the number of possible vertex functions grows exponentially, this method is only practical for vertices with up to four inputs. For the circuit example of Fig. 3(a), part (c) shows the graph model based on

```
Algorithm create_and(p1,p2) {
   /* constant folding */
   if (p1 == CONST_0)  return CONST_0;
   if (p2 == CONST_0)  return CONST_0;
   if (p1 == CONST_1)  return p2;
   if (p2 == CONST_1)  return p1;
   if (p1 == p2)       return p1;
   if (p1 == ¬p2)      return CONST_0;
   /* rank order inputs */
   if (rank(p1) > rank(p2)) swap(p1,p2);
   /* check for isomorphic entry in hash table */
   if ((p = hash_lookup(p1,p2)) == NULL) {
   /* 3 cases depending on position in graph  */
      if (is_var(p1) && is_var(p2))
                          p = new_and_vertex(p1,p2);
      else if (is_var(p1)) p = create_and3(p1,p2);
      else if (is_var(p2)) p = create_and3(p2,p1);
      else                 p = create_and4(p1,p2);
   }
   return p;
}
```

Fig. 4. Algorithm **create_and** for an AND constructor that includes local functional hashing.

vertices with a maximum fanin degree of four. Note that this method can identify the equivalence of the net pair (5, 5') but still fails to show the same for pair (7, 7'), and therefore for x and y.

A more comprehensive approach denoted as *functional hashing* [1] is based on the presented two-input graph and an extension of the structural analysis that includes the two graph levels preceding a vertex. As a result, the granularity of identifying functionally identical vertices is comparable to the granularity of the hashing technique based on four-input vertices. Moreover, by applying this method on all intermediate vertices in an overlapping manner, this approach can take advantage of additional structural similarities that otherwise remain internal to four-input vertices.

Fig. 4 outlines the overall flow of the functional hashing scheme. The first part, which performs constant folding and structural hashing, is identical to the algorithm **create_and2** of Fig. 1. In case of a hash miss, the second part includes an extended two-level lookup scheme, which converts the local function of the four grandchildren into a canonical representation. During graph construction from the primary inputs, the first level of vertices does not have four grandchildren and, thus, must be treated specially. If both immediate children are primary inputs, the algorithm creates a new vertex using the procedure **new_and_vertex**, which is shown in Fig. 2. If only one of the children is a primary input, a canonical three-input substructure is created by applying the procedure **create_and3**, and for the remaining case the procedure **create_and4** is called. Since the algorithms of the procedures **create_and3** and **create_and4** are fairly similar, we limit the description to the latter. Its pseudo-code is given in Fig. 5. The procedure **create_and3** simply implements a subset of the shown cases.

The algorithm **create_and4** first analyzes the local substructure using the procedure **analyze_case**. It computes a signature which reflects: 1) the equality relationship of the four grandchildren and 2) the inverter attributes of the six arcs. This signature is mapped to one of 235 different cases (for the algorithm **create_and3**, the signature is mapped to one of

```
Algorithm create_and4(l,r) {
  ll    = l->left;
  lr    = l->right;
  rl    = r->left;
  rr    = r->right;
  index = analyze_case(l,r);
  switch(index) {
    ...
    case 98:  /* function g in Fig. 6 */
      p1 = ¬create_and(ll,lr);      /* recursive     */
      p2 = ¬create_and(¬ll,¬lr);
      return create_and2(p1,p2); /* not recursive */
    ...
    case 123: /* function h in Fig. 6 */
      p1 = ¬create_and(ll,lr);
      p2 = ¬create_and(¬ll,¬lr);
      return ¬create_and2(p1,p2);
    ...
    case 144: /* example of Fig. 7    */
      return create_and(rl,rr);
    ...
  }
  /* remaining cases with no reduction */
  return new_and_vertex(l,r);
}
```

Fig. 5. Sketch of the algorithm **create_and4** which handles vertex construction for substructures with four grandchildren.

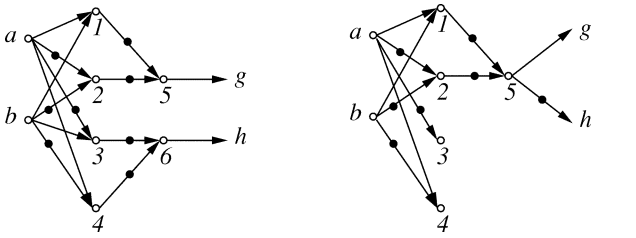


Fig. 6. Example for merging the structures of functions $g = \mathrm{XOR}(a, b)$ and $h = \mathrm{XNOR}(a, b)$ using the algorithm **create_and4**. (a) Resulting structure without functional hashing. (b) Structure with functional hashing.

44 cases). By construction, the topology of the substructure is uniquely identified by this signature. Its value is then mapped onto an implementation index such that all structures with identical functions get projected onto the same index. For each distinct index a new canonical implementation is then generated. Because of this canonicity and the applied vertex hashing this method merges all functionally equivalent substructures, effectively removing local functional redundancies. Fig. 5 provides pseudo-code examples to handle cases 98 and 123, which represent the structures of an XOR and XNOR function as shown in Fig. 6(a). Fig. 6(b) demonstrates how functional hashing maps both functions onto the same vertex referenced by complemented arcs.

Note that functional hashing is applied recursively as shown in the implementation of the intermediate functions p_1 and p_2 of cases 98 and 123 of Fig. 5. However, to ensure termination, the final vertex must be constructed with the nonrecursive procedure **create_and2** shown in Fig. 2. The recursive application of functional hashing often results in a significant graph reduction. For example, the two outputs in Fig. 3(a) can be merged by functional hashing resulting in the graph shown in (d).

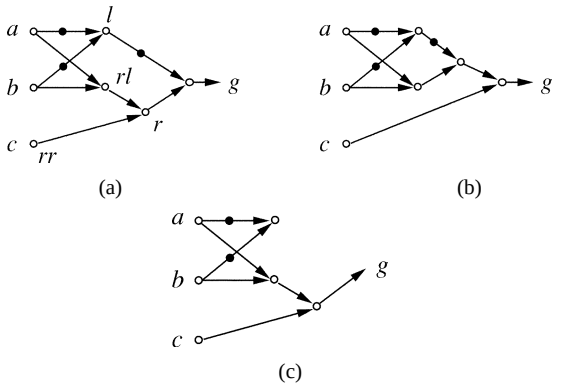


Fig. 7. Example of local rewriting: (a) original graph that cannot be reduced by functional hashing, (b) result of rewriting using case 236 of algorithm **create_and4** (Fig. 8), and (c) result after recursive application of case 144 of algorithm **create_and4** (Fig. 5).

C. Local Rewriting

Functional hashing, as described in the previous subsection, has the potential to compact graph representations for structures with shared grandchildren. However, if all four grandchildren of the two operands are distinct, the hashing does not result in any structural reduction. Still, in some cases where the operand's grand-grandchildren are shared, the local structure can be rearranged such that they share at least one grandchild. This rearrangement will enable a following functional hashing step. For example, the expression $g = (a \vee b) \wedge ((a \wedge b) \wedge c)$ cannot be simplified with functional hashing because it has four distinct grandchildren, $\{a, b, a \wedge b, c\}$. However, after rewriting the expression into $g = ((a \vee b) \wedge (a \wedge b)) \wedge c)$ functional hashing can simplify the structure to $g = a \wedge b \wedge c$. The corresponding step-wise graph transformations are illustrated in Fig. 7.

To handle local rewriting, the algorithm **create_and4** is enhanced by recognizing more cases in the procedure **analyze_case** and adding the corresponding indexes to the implementation cases. In essence, the mentioned rewriting mechanism is applicable if: 1) at least one operand of the AND operation is complemented and 2) the grandchildren and/or children of the operands are shared. Fig. 8 shows a modified version of the algorithm **create_and4** that includes the additional three cases for local rewriting. The procedure **share** tests whether two given vertices share any children. In the given example of Fig. 7, the left child of vertex g is inverted and shares its children with the right–left grandchild of g. Here the depicted case 236 of the modified algorithm of Fig. 8 is applied for rewriting. During the implementation step the procedure **create_and** is called recursively after which case 144 shown in Fig. 5 is applicable and simplifies the graph to the structure given in Fig. 7(c).

D. Symmetric Cluster Hashing

The previously described methods, including structural hashing, functional hashing, and local rewriting, restructure the AND/INVERTER graph only locally by examining a limited number of fanin vertices. A further compression of the graph can be achieved by analyzing larger symmetric graph clusters. The idea is that expression trees utilizing a uniform symmetric vertex function (e.g., AND or XOR) represent the same Boolean

```
Algorithm create_and4(l,r) {
  ll    = l->left;
  lr    = l->right;
  rl    = r->left;
  rr    = r->right;
  index = analyze_case(l,r);
  switch(index) {
    ...
  /* rewriting cases */
    case 235: /* l and r are inverted */
    ...
    case 236: /* only l is inverted */
      if (share(l,rl) || share(ll,rl) || share(lr,rl)) {
        return create_and(create_and(l,rl),rr));
      }
      if (share(l,rr) || share(ll,rr) || share(lr,rr)) {
        return create_and(create_and(l,rr),rl));
      }
      if (share(ll,r) || share(lr,r)) {
        return ¬create_and(¬create_and(¬ll,r),
                           ¬create_and(¬lr,r));
      }
      break;
    case 237: /* only r is inverted */
    ...
  }
  /* no reduction or rewriting */
  return new_and_vertex(l,r);
}
```

Fig. 8. Sketch of improved algorithm **create_and4** with local rewriting.

function if they have identical sets of source vertices. This equivalence is independent of the actual tree structure and permutation of sources.

For identifying identical symmetric cluster functions, a special cluster hashing algorithm is applied whenever a new AND vertex is built. The algorithm traverses the transitive fanin of the vertex, determines the boundaries of the symmetric expression tree, and collects the set of source vertices. This set is then hashed using a special cluster hash table. If identical entries are found, the corresponding vertices are merged and their fanout structures are rebuilt.

The symmetric cluster hashing is done for two function types. AND clusters are simple to identify by just traversing the fanin structure until inverted arcs are encountered. Due to their duality, OR clusters are automatically handled by the same algorithm. XOR/XNOR clusters are found by recursively searching for the canonical XOR structure (shown in Fig. 6). Only one of the two possible XOR/XNOR needs to be identified, since the other structure gets rewritten by functional hashing.

Note that an alternative approach to handle symmetric clusters would be to build AND and XOR expression trees in a canonical manner, for example, by always building a balanced tree structure using the source vertices in some lexicographical order. However, our experience is that such an approach is inferior to the presented method since it destroys up-font large parts of the existing (empirically useful) circuit structure and as a result prevents many matchings that are otherwise possible.

IV. BDD Sweeping

In this section, we describe the BDD sweeping algorithm, a method that systematically identifies and merges functionally equivalent AND/INVERTER graph vertices that are not found to be equivalent by the previously described structural methods.

The sweeping method builds BDDs for the individual graph vertices starting from inputs and multiple cut frontiers toward the outputs. By maintaining cross references between the graph vertices and its BDD nodes, functionally identical vertices can be found constructively during the sweep. There are several key ideas that make BDD sweeping robust and efficient.

- As soon as two functionally equivalent vertices are identified, their output structures are merged and rehashed using the algorithms described in the previous section. The instantaneous application of structural simplification can solve reasoning problems without building BDDs for the entire problem structure, resulting in a significant increase in the overall reasoning power and performance.
- The BDD propagation is prioritized by the actual size of the input BDDs using a heap as processing queue. As a result, the sweeping algorithm focuses first on inexpensive BDD operations and avoids the construction of large BDDs unless they are needed for solving a problem.
- The maximum size of the processed BDDs is limited by a threshold, which effectively controls the computing resources and reasoning power. BDDs that exceed the size of the threshold are "hidden" in the processing heap and will reappear when the sweeping is restarted with a sufficiently large limit. This mechanism is used to interleave BDD sweeping with structural SAT search. By incrementally increasing the resources of the individual algorithms during each iteration, their reasoning power continues to grow until the problem can be solved by either one of them.
- Multiple BDD frontiers are concurrently propagated in the heap controlled manner. This approach effectively handles local redundancies without the need to always build large BDDs from the graph inputs.
- When the BDD processing reaches any of the target vertices that represent a proof obligation (i.e., it must be shown to be constant, or not) one of the following steps is applied: if the corresponding BDD represents a constant, the vertex gets merged with the constant graph vertex and the reasoning result is obvious (depending on the problem either SAT or UNSAT). Otherwise, if the support of the BDD contains only primary input variables, satisfiability is proven and any paths from the BDD root to the corresponding constant BDD node can serve as counterexample. If the support contains variables from intermediate cutsets, false negative resolution is applied.

A. Basic Sweeping Algorithm

Fig. 9 shows the self-explanatory pseudo-code for the basic BDD sweeping algorithm **bdd_sweep**. It does not include the processing of multiple BDD frontiers, which is described in the next subsection. The heap structure is initialized in the overall procedure (see Section VIII). For this, primary inputs are initialized at the beginning of the reasoning flow using the procedure **sweep_init**, whereas cutset vertices are declared and initialized between the individual sweeping iterations.

The invocation of the sweeping algorithm processes all heap BDDs that have a smaller size than the given threshold

```
Algorithm sweep_init(vertex) {
  heap = new_heap();
  for all input vertices v_in do {
    bdd = create_bdd_variable();
    store_vertex_at_bdd(bdd, v_in);
    put_on_heap(heap, bdd, bdd_upper_size_limit);
  }
  return heap;
}

Algorithm bdd_sweep(heap, vertex) {
  /* check if there are any BDDs on heap with
     size(bdd) ≤ bdd_lower_size_limit */
  while (!is_heap_empty(heap, bdd_lower_size_limit)) do {
    bdd  = get_smallest_bdd(heap);
    v    = get_vertex_from_bdd(bdd);
    /* check if previously encountered */
    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);
        /* return if problem solved */
        if (vertex == CONST_1) return SAT;
        if (vertex == CONST_0) return UNSAT;
      } else {
        store_vertex_at_bdd(bdd_res, v_out);
      }
      /* BDD for v_res cannot be constant */
      if (v_out == vertex) return SAT;
      put_on_heap(heap, bdd_res, bdd_upper_size_limit);
    }
  }
  return UNDECIDED;
}
```

Fig. 9. Initialization procedure **sweep_init** and basic BDD sweeping algorithm **bdd_sweep** for deciding *SAT*(*vertex*).

bdd_lower_size_limit. All larger BDDs remain hidden in the heap and get processed when the algorithm is called again with a sufficiently large threshold. During each iteration of the inner sweeping loop, the algorithm removes the smallest BDD from the heap, processes the Boolean operations for the immediate fanout structure of the corresponding circuit graph vertex, and reenters the resulting BDDs onto the heap, if their size is below the threshold *bdd_upper_size_limit*.

Using cross referencing between graph vertices and the corresponding BDD nodes, functionally equivalent vertices can be identified. An equivalent vertex pair is found if the result v_{res} of a BDD operation already refers to another vertex that was processed before. In this case, both vertices are merged immediately and their subsequent parts of the graph are rehashed by the procedure **merge_vertices**. The rehashing is applied in depth-first order starting from the merged vertices toward the primary outputs and stops if no further reconvergency occurs. As a result, the forward rehashing may merge the reasoning target vertex *vertex* with the constant vertex, effectively deciding the problem. The corresponding two checks in the inner loop test for these cases. Before the BDD is reentered onto the heap, another check tests whether the target vertex was reached. In this case, the target vertex must be nonconstant, otherwise it would have been merged with a constant vertex and one of the previous tests would have succeeded. Therefore, the problem is satisfiable since the BDD support includes only primary input variables.

The following remarks further explain particular details of the sweeping algorithm.

- The sequence by which the cross referencing is performed and checked using the procedure **get_bdd_from_vertex**, **store_bdd_at_vertex**, **get_vertex_from_bdd**, and **store_vertex_at_bdd**, ensures that all vertices are handled exactly once, unless new graph vertices are added to the fanout of an already processed vertex. In this case, the procedure **new_and_vertex** restarts the sweeping process for these vertices.
- If one of the two BDD operands is missing, the BDD operation is skipped and processing continues with the next BDD from the heap. Note that as soon as this operand is available, the same BDD operation will be reinvoked.
- The merging of two vertices is done in a forward manner, i.e., the fanouts of the vertex, which is topologically farther from the primary inputs, must be reconnected to the vertex that is topologically closer to the inputs. Otherwise, the merge operation may cause structural loops in the graph, which would invalidate its semantic.

Fig. 10 illustrates the mechanism of BDD sweeping for proving equivalence of two functionally identical but structurally different circuit cones. Fig. 10(a) and (b) show the miter structure of the two cones 5 and 5' to be compared with an XNOR gate and the corresponding AND/INVERTER graph, respectively. The following figures show the progress of the sweeping until equivalence is proven. It is assumed that the BDDs are processed in the order of their corresponding vertices 1,2,3,4,3', and 2'. The first four iterations create the BDDs for vertices 1,2,3, and 4. In the next iteration, the resulting BDD node for vertex 3' already points to the functionally equivalent vertex 3. Therefore, vertices 3' and 3 are merged as depicted in Fig. 10(c). The next figure shows the graph after vertex 2' has been processed and merged with vertex 2. The subsequent forward rehashing identifies that 5 and 5' are isomorphic and merges them, which further causes 6 to be merged with the constant vertex. Note that for simplicity we used only structural hashing in this example. The resulting graph structure is shown in Fig. 10(e). At this point functional equivalence is proven and the algorithm terminates without having to build BDDs for the entire miter structure.

B. Enhanced Sweeping Algorithm With Multiple BDD Frontiers

The basic sweeping algorithm as described in the previous section starts the BDD propagation from the input vertices only. As a result, the size threshold of the BDD processing precludes a full penetration of deeper AND/INVERTER graphs. An enhanced sweeping approach is based on a multilayered propagation of BDDs that start from the primary inputs as well as intermediate cut frontiers. Using this scheme, the graph vertices are generally associated with multiple BDDs that represent their function from different cuts of their fanin logic.

For the multilayer BDD propagation, the overall algorithm, which is described in Section VIII, declares cutpoints between

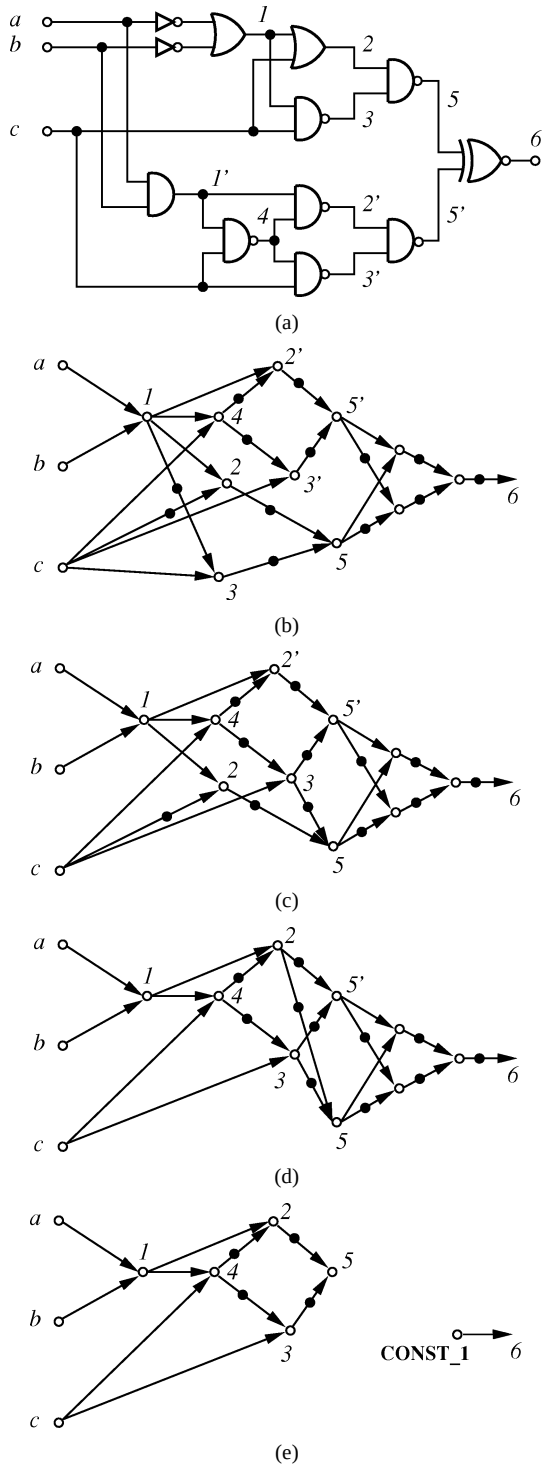


Fig. 10. Example for BDD sweeping: (a) miter for two functionally identical circuit cones, (b) original AND/INVERTER graph, (c) BDDs are computed for vertices 1,2,3,4,3', which causes 3' and 3 to be merged, (d) BDD is computed for 2' which causes 2' and 2 to be merged, and (e) forward hashing causes 5 and 5' to be merged and 6 be merged with the constant vertex thus solving the reasoning problem.

```
Algorithm bdd_sweep(heap, vertex) {
  while (!is_heap_empty(heap, bdd_lower_size_limit)) do {
    bdd   = get_smallest_bdd(heap);
    v     = get_vertex_from_bdd(bdd);
    level = get_level_from_bdd(bdd);
    if (get_bdd_from_vertex(v, level)) 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 (vertex == CONST_1) return SAT;
        if (vertex == CONST_0) return UNSAT;
      } else {
        store_vertex_at_bdd(bdd_res, v_out);
        store_level_at_bdd(bdd_res, level);
      }
      if (v_out == vertex) {
        resolve_false_negative(v);
        if (vertex == CONST_1) return SAT;
        if (vertex == CONST_0) return UNSAT;
        /* non-constant level-0 BDD */
        if (get_bdd_from_vertex(vertex, 0)) return SAT;
      }
      put_on_heap(heap, bdd_res, bdd_upper_size_limit);
    }
  }
  return UNDECIDED;
}
```

Fig. 11. Enhanced BDD sweeping algorithm with multilayered BDD frontiers for deciding *SAT*(*vertex*).

individual sweeping steps. There are several heuristics to identify vertices that represent effective cutpoints, including the use of:

- vertices that have a large fanout;
- vertices that have multiple paths to the reasoning vertex;
- in case of equivalence checking, vertices that are on the border of the intersection of the two cones that form the miter structure.

Based on the declared cutpoints, the cut level $\texttt{c\_level}(v)$ of a circuit graph vertex v is recursively defined as shown in the equation at the bottom of the page.

The cut level is used to define cut frontiers and to align the BDD propagation with these frontiers. Fig. 11 shows the modified sweeping algorithm that supports the handling of multiple BDD frontiers. As shown, the additions mainly involve a level-specific handling of BDDs. The procedure **get_bdd_from_vertex** returns the BDD stored for a specified level at a vertex. If the given level exceeds the cut level of the vertex, the BDD of the maximally available cut level is taken.

Similar to the basic algorithm, special checks are applied when the BDD processing reaches the target vertex *vertex*. However, since the BDDs of the vertices do not necessarily originate from the primary inputs, it cannot be decided immediately if the

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

```
Algorithm resolve_false_negatives (v1) {
  for level = 0 to c_level (v1) do {
      bdd = get_bdd_from_vertex(v1, level);
      put_on_heap(compose_heap, bdd);
  }
  limit = bdd_lower_size_limit;
  while (!is_heap_empty(compose_heap,limit)) 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);
        } else {
            store_vertex_at_bdd (bdd_res, v);
            put_on_heap(compose_heap, bdd_res);
        }
      }
  }
}
```

Fig. 12. Algorithm to eliminate false negatives.

problem is solved. First, false negative resolution attempts to resubstitute cutpoint variables of the BDD with the BDDs driving the corresponding vertices (see Section IV-C). This may cause the target vertex to get merged with a constant vertex in which case the problem is satisfiable or unsatisfiable for a merger with the constant "1" or constant "0", respectively. Furthermore, if all cutpoint variables have been resubstituted the resulting BDD originates only from the primary inputs. The problem must then be satisfiable.

C. False Negative Resolution

The algorithm to resolve false negatives in shown in Fig. 12. To fully explore BDDs constructed for the different levels of the target vertex without memory explosion, the elimination process is also controlled by a heap. In each iteration, the smallest BDD is taken and its topmost cut variable resubstituted by the corresponding driving function. The resulting BDD is 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.

V. Structural SAT Solver

A. Basic SAT Procedure

The structural SAT solver is based on the Davis–Putnam procedure working on the presented AND/INVERTER graph. It attempts to find a set of consistent value assignments for the vertices such that the target vertex evaluates to a logical "1". Unsatisfiability is proven if an exhaustive enumeration does not uncover such an assignment.

Fig. 13 provides the top level view of the SAT algorithm consisting of two routines, the procedure **sat_init**, and the procedure **justify**, which handles the case splitting and backtracking. The overall SAT search is based on a processing queue *justification_queue* that contains all vertices for which a consistent

```
Algorithm sat_init(vertex) {
  stack = new_stack();
  assign(vertex, 1);
  if (imply(vertex)) {
    d_level        = push_on_stack(stack,NULL);
    d_level->mark  = 0;
    d_level->queue = justification_queue;
    d_level->v     = dequeue_vertex(d_level->queue);
  }
  return stack;
}

Algorithm justify(stack) {
  while (1) {
    if (backtracks++ > sat_backtrack_limit)
                    return UNDECIDED;
    d_level = pop_from_stack(stack);
    if (!d_level)   return UNSAT; /* exhausted  */
label:
    if (!d_level->v) return SAT;   /* justified  */
    /* try all values one by one;
       continue from last value after
       returning from higher decision level */
    for all values for d_level->value do {
      assign(d_level->v->left, d_level->value);
      if (imply(d_level->v->left)) {
        d_level        = push_on_stack(stack, d_level);
        d_level->mark  = tail_pointer(assignment_list);
        d_level->queue = justification_queue;
        d_level->v     = dequeue_vertex(d_level->queue);
        goto label;
      }
      /* failed undo assignments and reset queue */
      undo_assignments(d_level->mark);
      justification_queue = d_level->queue;
    }
  }
}
```

Fig. 13. General Davis–Putnum SAT procedure for deciding *SAT*(*vertex*).

assignment must be found. The algorithm attempts to sequentially justify these vertices using a branch-and-bound case enumeration. Note that due to their uniform AND functionality, only vertices that are to be justified to "0" need to be scheduled on that queue. A required logical "1" at a vertex output implies a "1" at both of its inputs and is handled directly by the procedure **imply**. Further, if the value of a vertex output is not yet specified ("X") it does not need to be justified since any value setting at its inputs will lead to a consistent setting.

The procedure **sat_init** first assigns the target vertex to "1" and propagates all implications using the procedure **imply**. Unless the target assignment results in an immediate conflict, it creates the first stack entry for the procedure **justify**. This entry contains all "to-be-justified" vertices that have been collected by **imply**. The following call of the procedure **justify** then performs a systematic case search by recursively processing all queue vertices and enumerating for them all valid input assignments (two for Boolean logic). In the case that the assignments of a search subtree result in a conflict, a marking mechanism allows undoing all assignments up to that decision level.

The tight integration of the SAT solver into the overall framework requires an execution control by providing resource limits such as the number of backtracks. If during the current application of **justify** this number exceeds a given threshold, the SAT solver interrupts its search and returns control to the calling procedure. This supports an interleaved application

```
Algorithm imply(vertex) {
  for all fanout_vertices v_out of vertex do {
    if (! imply_aux(v_out)) return 0;
  }
  return imply_aux(vertex);
}

Algorithm imply_aux(vertex) {
  value         = get_value(vertex);
  lvalue        = get_value(vertex->left);
  rvalue        = get_value(vertex->right);
  current_state = (value, lvalue, rvalue);
  next_state    = lookup(current_state);
  switch (action) {
    case STOP:
      return 1;
    case CONFLICT:
      return 0;
    case CASE_SPLIT:
      enqueue_vertex(vertex, justification_queue);
      return 1;
    case PROP_FORWARD:
      assign(vertex, next_state->value);
      for all fanout_vertices v_out of vertex do {
        if (!imply_aux(v_out))      return 0;
      }
      return 1;
    ...
    case PROP_LEFT_RIGHT:
      assign(vertex->left, next_state->lvalue);
      assign(vertex->right, next_state->rvalue);
      if (!imply(vertex->left))     return 0;
      if (!imply(vertex->right))    return 0;
      return 1;
    ...
  }
  return 1;
}
```

Fig. 14. Implication procedure to the AND/INVERTER graph.

of the SAT search with BDD sweeping as described in Section VIII. Furthermore, by preserving the state of the decision stack between subsequent invocations, the SAT algorithm can continue its search from the point it stopped earlier without repeatedly searching previously handled subtrees. This reentrant functionality is implemented in the procedure **justify** by checking the backtrack limit each time the search returns from a higher decision level. If the limit is exceeded, the control is returned to the calling, overall process and the backtracking is postponed until **justify** is called again. Note that the setup of **sat_init** ensures a correct initialization of the first stack entry.

The details of the algorithm **imply** for implication processing are shown in Fig. 14. Its implementation takes specific advantage of the underlying AND/INVERTER graph structure by applying an efficient table-lookup scheme for propagating logic implications. The routines **imply** and **imply_aux** iterate over the AND/INVERTER graph and determine at each vertex all implied values and the directions for further processing.

Fig. 15 gives an excerpt from the implication lookup table. As described above, for Boolean logic only one case, a justification request for a logical "0" at the output of an AND vertex requires scheduling a new vertex on the *justification_queue*. All other assignments result in one of three cases: 1) a conflict occurred, in which case the algorithm returns and backtracks; 2) further implications are triggered, which are processed recursively; or 3) the vertex is fully justified, in which case the procedure returns for processing the next element from the *justification_queue*.

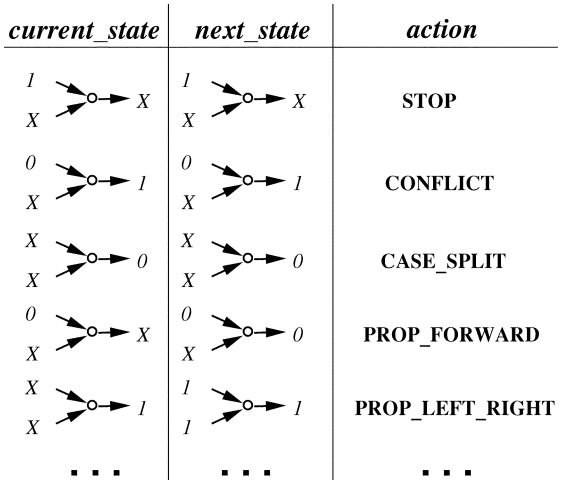


Fig. 15. Excerpt of the lookup table for fast implication propagation applied in the procedure **imply** of Fig. 14.

The lookup table is programmable for different logics. For example, using a different table the procedure **imply** can equally be applied to implement a parallel, one-level recursive learning scheme using nine-valued logic [19]. Due to its uniformity and low overhead, the presented implication algorithm is highly efficient. As an indication, on a Pentium III class machine it can execute several hundred thousand backtracks per second on typical circuit structures. For being beneficial for the overall performance, any gain that is potentially achieved through additional structural analysis must offset the resulting slowdown of the **imply** function. In [8] and [20], a similar reasoning is given for efficient implementations of SAT and ATPG algorithms, respectively.

B. Improvements to the SAT Procedure

1) Conflict Analysis: Advanced SAT solvers use conflict analysis to skip the evaluation of assignments which are symmetric to previously encountered conflicts [7]. Two mechanisms are used for this purpose: first, nonchronological backtracking skips the evaluation of case alternatives if the corresponding case splitting vertex was not involved in any lower level conflict. Second, conflict-based learning creates additional implication shortcuts, which reflect the assignments that caused a conflict. These redundant structures result in additional implications, which detect subsequent, symmetric conflicts earlier.

Conflict analysis requires tracking the logical impact of case split assignments on the conflict points. Other implementations (e.g., [7]) apply an implication graph for which the nodes correspond to variables and edges reflect single implication steps. In the given setting, the conflict graph manipulation during each step of the **imply** routine would severely impact its performance. To reduce this penalty, we apply a scheme that directly collects the responsible case assignments as a side function of the implication process. This mechanism uses a conflict bit-vector where each bit represents a case vertex in the decision tree. In other words, the bits of this vector represent the source vertices from which implication sequences were started. The table lookup in function **imply** is expanded to also determine the controlling

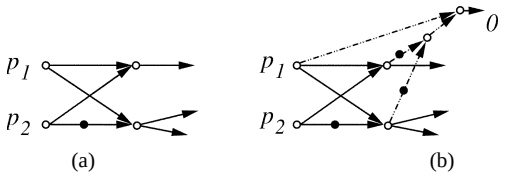


Fig. 16. Learning for $(p_1p_2 = 0) \wedge (p_1\overline{p_2} = 0) \Rightarrow (p_1 = 0)$: (a) original structure and (b) structure with learned implication shortcut.

sources for the propagation of the conflict bit-vectors. The actual propagation is done by word-wide OR operations of these vectors during the implication sequence. As a result, when a conflict is reached, the active positions of the bit-vector reflect the decisions vertices that are responsible for this assignment. This scheme reduces the speed of the implication process by typically less than 50%, a penalty that is easily offset by the average gain.

The resulting conflict bit-vectors are used to control the backtracking mechanism. If all choices of a decision level result in a conflict, the backtrack level is determined by the lowest level that was involved in a conflict. This is implemented efficiently by bit-vector operations. The combined conflict bit-vector reflects all responsible assignments for that part of the decision tree and is passed upward to the next backtracking level.

Further, the conflict bit-vector is examined for compact clauses to learn. Its function is directly mapped onto a AND/INVERTER graph structure representing that conflict. To avoid excessively large learned structures, we apply a variable limitation similar to [7]. However, instead of just restricting the maximum number of conflict variables, we also take their assignment into account, effectively estimating the size of the eliminated decision subtrees.

2) Static Learning: An effective mechanism to exploit the structure of the AND/INVERTER graph is illustrated in Fig. 16. By reusing the vertex hash table applied during graph construction, a pair of vertices that implement the functions p_1p_2 and $p_1\overline{p_2}$ can be detected using two hash lookups. This configuration occurs often in practical designs, for example in multiplexer-based data exchange circuits that switch data streams between two sources and two destinations. By adding two additional vertices to the graph, an implication shortcut can utilize the existing **imply** function. If a logical 0 is scheduled for both output vertices, the implication procedure can immediately justify the entire structure and bypass the two case splits. This learning structure is created statically and integrated into the vertex allocation algorithm **new_and_vertex**, which is shown in Fig. 2. Note that the learned vertices are built using the regular **create_and** routine which may cause additional circuit restructuring or learning events.

VI. Random Simulation

Many problem instances of Boolean reasoning are satisfiable and have dense solution spaces. The most effective approach for finding a satisfying assignment for such problem is often based on pure random simulation. The design of a random simulation algorithm is straightforward. In this section, we briefly describe the details of its implementation in the given setting.

```
Algorithm simulate(vertex) {
  for all input vertices v_in do {
    set_value(v_in, random_word());
  }
  simulate_aux();
  if (get_value(vertex) != 0) return SAT;
  else                        return UNDECIDED;
}

Algorithm simulate_aux() {
  for all vertices v in topological order do {
    lvalue = get_value(v->left);
    rvalue = get_value(v->right);
    if (is_inverted(v->left)) lvalue = word_not(lvalue);
    if (is_inverted(v->right)) rvalue = word_not(rvalue);
    value = word_and(lvalue, rvalue);
    set_value(v, value);
  }
}
```

Fig. 17. Random simulation algorithm **simulate**.

The presented AND/INVERTER is highly suitable for an efficient word-parallel implementation of random simulation. The pseudo-code for the corresponding algorithmic flow is shown in Fig. 17. After assigning random values to the primary input vertices, a levelized processing using word-wide AND and NOT instructions propagates the resulting assignments toward the target vertices. A check for satisfiability of a target vertex is simply done by a parallel comparison of its value with the zero word. Note that by applying proper reference counting, only a single value frontier needs to be stored during their propagation, which results in a sublinear memory complexity [14].

VII. Input Constraints

Many Boolean reasoning problems require an efficient handling of input constraints, typically referred to as "don't cares." For example, in combinational equivalence checking, input constraints express the Boolean subspace for which the two designs under comparison have to be functionally identical. The remaining input combinations represent "don't cares" for which the functions may differ. Other problems that require efficient processing of input constraints occur during synthesis and verification of incompletely specified functions.

Input constraints split the set of values at the primary inputs into two parts, the "valid" or "care" set and the "invalid" or "don't care" set. The problem of Boolean reasoning under input constraints is to find a consistent assignment within the care set. A convenient method for expressing and storing input constraints in the given setting is based on characteristic functions that can be stored and manipulated as part of the overall AND/INVERTER graph. The graph representation for the characteristic function can be built using the standard constructor operations, which are described in Section III. Its result is then asserted to be logical "1," meaning that all input values that evaluate this function to "1" are considered to be the care set.

The simplest method for handling input constraints is based on a scheme that first conjoins the constraint vertices with the target vertex and then applies the reasoning algorithms on the resulting AND vertex. However, for structural simplification, BDD sweeping, and random simulation, this approach would result in

```
Algorithm assert_to_1(vertex) {
  if (is_var(vertex) { /* leaf vertex */
    merge_vertices(vertex,CONST_1);
  } else if(is_inverted(vertex->left) &&
            is_inverted(vertex->right)) {
    /* search for XNOR and XOR structures */
    if (vertex->left->left == ¬vertex->right->left &&
        (vertex->left->right == ¬vertex->right->right) {
      if (is_inverted(vertex)) {     /* XNOR */
        merge_vertices(vertex->left->left,
                       vertex->left->right);
      }
      else {                         /* XOR */
        merge_vertices(vertex->left->left,
                       ¬vertex->left->right);
      }
    }
    return;
  }
  /* recursive conjunctive decomposition */
  if (!is_inverted(vertex)) {
    assert_to_1(vertex->left);
    assert_to_1(vertex->right);
  } else { /* no further decomposition */
    merge_vertices(vertex,CONST_1);
  }
}
```

Fig. 18. Algorithm **assert_to_1** for asserting a vertex to "1" that represents an input constraint.

a significant performance degradation, especially if the fraction of valid assignments is very small. A more efficient method is to keep the characteristic functions of the input constraints separate and to handle them specifically in each reasoning algorithm. In the following sections, we elaborate on the mechanisms to handle input constraints by the individual reasoning algorithms.

A. Structural Representation

As mentioned before, input constraints are simply expressed as characteristic functions and represented as vertices in the AND/INVERTER graph. The resulting constraint vertices are marked for special handling for the SAT solver, BDD sweeping, and random simulation. Fig. 18 gives the pseudo-code for the algorithm that asserts a graph vertex to constant "1." As shown, the algorithm consists of two parts. First, a local analysis of the asserted vertex searches for XOR and XNOR structures. If found, the input vertices of these functions are asserted to be equal by structurally merging them with the same merge function applied for BDD sweeping (see Section IV). Second, if no XOR or XNOR is found, a structural conjunctive decomposition of the assertion function is attempted. This is done by recursively traversing the AND tree driving the asserted vertex. The resulting individual conjuncts are then separately merged with the constant "1" vertex. This merge has the advantage that structurally isomorphic functions, which are part of the actual reasoning problem, can be identified as constant without any further processing. The forward rehashing, which is applied when this vertex is merged with the constant vertex, automatically simplifies the subsequent graph structure.

B. BDD Sweeping

Input constraints can be used during BDD sweeping to identify additional sets of vertices that are functionally equivalent for the care set only. This is accomplished by restricting the vertex BDD to the care set before it is checked for a pre-existing vertex references and put onto the heap. The BDD restriction is done by ANDing it with the set of BDDs generated for the constraint vertices, which are asserted to "1." Since the constraint handling is conservative and cannot produce false negatives, this restriction can be done dynamically. As soon as the BDDs for the individual constraint vertices become available through the heap controlled processing, they can be used to restrict all existing and future vertex BDDs.

C. Structural SAT Solver

The existence of input constraints implies for the structural SAT search that the values of all asserted constraint vertices must be preset to constant "1." Furthermore, these values need to be fully justified, which is accomplished by adding the corresponding vertices to the *justification_queue*. Both requirements are implemented in a preprocessing step before the actual SAT search starts. Note that for structural SAT this approach is identical to the method in which the asserted vertices are simply combined with the target vertex by conjunction. For that method the first application of the **imply** function would immediately schedule all asserted vertices to be justified to "1." The resulting search flow would then be identical to the flow produced by the presented approach using a separate preprocessing step.

D. Random Simulation

To achieve high coverage in random simulation, it is essential to avoid simulating input values that are don't cares. In particular, for sparse care sets, a pure random value selection from the entire Boolean space may result in no coverage at all. For generating valid input combinations in the presented random simulation approach, the SAT solver is applied to search for satisfying assignments for all vertices that are asserted to "1." However, instead of stopping the search once a solution is found, the SAT procedure continues to traverse the search tree. For each encountered solution, the input values for the satisfying assignments are recorded and later simulated in the word-parallel manner presented in Section VI.

VIII. Overall Algorithm

The overall algorithm that combines structural transformations, BDD sweeping, SAT search, and random simulation is outlined in Fig. 19. For each reasoning query the algorithm first checks if the structural hashing algorithm solved the problem. Interestingly, for a large number of queries in practical applications the structural test is successful and immediately solves the problem. For example, in a typical ASIC methodology, equivalence checking is used to compare the logic before and after insertion of the test logic. Since no logic transformations have actually changed the circuit, a simple structural check suffices to prove equivalence.

Next random simulation is applied to quickly check for a satisfying assignment. If simulation cannot solve the problem, SAT search and BDD sweeping are first initialized and then invoked in an intertwined manner [4]. In the inner loop, a call to the justification procedure **justify** is alternated with an invocation of multiple sweeping iterations. After each BDD sweeping

```
Algorithm check_SAT(v) {
  if (v == CONST_1) return SAT;
  if (v == CONST_0) return UNSAT;
  /* random simulation */
  res = simulate(v);
  if (res != UNDECIDED) return res;
  /* initialize BDD sweeping */
  heap = sweep_init(v);
  /* initialize SAT search */
  stack = sat_init(v);
  while (!is_heap_empty(heap, bdd_upper_size_limit)) do {
    /* try to justify */
    res = justify(stack);
    if (res != UNDECIDED) return res;
    /* BDD sweeping till no more cutpoints found */
    do {
      res = bdd_sweep(heap, v);
      if (res != UNDECIDED) return res;
      found = find_and_init_cutpoints(heap, v);
    } while (found);
    bdd_lower_size_limit    += delta_bdd_limit;
    sat_backtrack_limit     += delta_sat_limit;
  }
  sat_backtrack_limit = max_sat_backtrack_limit;
  return justify(stack);
}
```

Fig. 19. Overall reasoning algorithm integrating BDD sweeping, SAT search and random simulation.

step, the cutpoint selection heuristic implemented in procedure **find_and_init_cutpoints** is applied to search for promising cutpoints. Newly found cutpoints are initialized with fresh BDD variables, which are then added to the processing heap. As long as new cutpoints are detected, BDD sweeping is restarted until the problem is solved or the cutpoint selection is exhausted. In the latter case, the reasoning algorithm returns to the SAT solver to search for a satisfying assignment. Note that BDD sweeping may merge vertices that are on the justification queue of the SAT solver. This artifact is handled by preserving the merge information and explicitly processing the SAT implications for all merged vertices.

During each iteration of BDD sweeping and SAT search, the size limit for sweeping and the backtrack limit for the SAT solver are increased. In this setting, these algorithms do not just independently attempt to solve the problem. Each BDD sweeping iteration incrementally compresses the AND/INVERTER graph structure from the inputs toward the target vertex, which effectively reduces the search space for the SAT solver. This interleaved scheme dynamically determines the minimum effort needed by the sweeping algorithm to make the SAT search successful. If the iterative invocation of BDD sweeping and SAT search was not able to solve the problem, the algorithm applies the SAT solver with a maximum backtracking limit as a final attempt to find a solution in a brute-force manner. Note that in this case the sweeping process is stopped by the limit *bdd_upper_size_limit*. This limit prevents the processing of excessively large BDDs.

IX. Experiments

In order to evaluate the effectiveness of the presented approach we performed extensive experiments using 488 circuits randomly selected from a number of microprocessors designs.

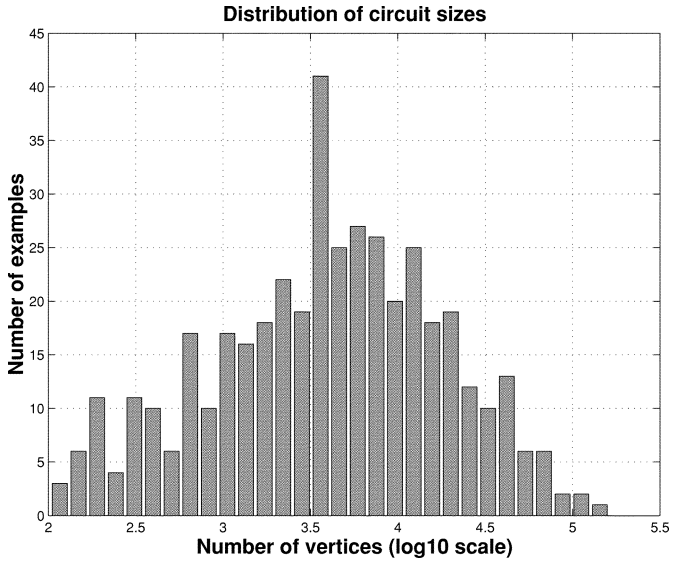


Fig. 20. Distribution of circuit sizes for the experiments.

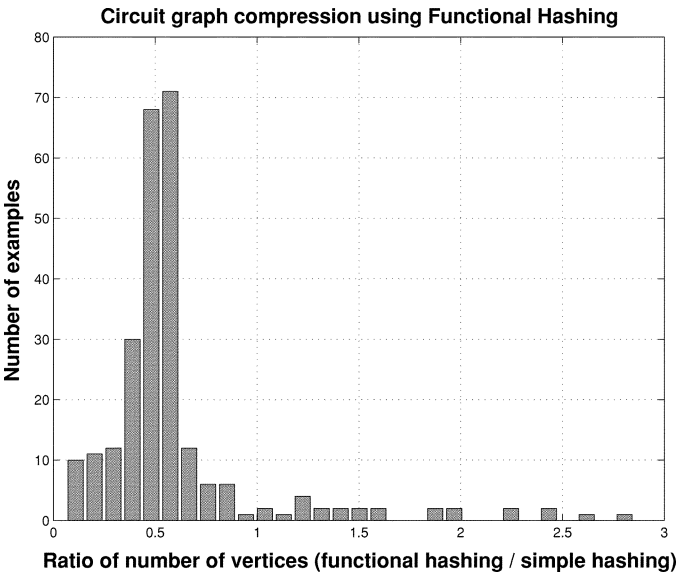


Fig. 21. Comparison of graph reduction of simple versus functional hashing.

The circuits range in size from a few 100 to 100 *K* gates with a size distribution given in Fig. 20. The number of outputs and inputs per circuit range from a few 100 to more than 10 000. The experiments were performed on a RS/6000 model 270 with a 64-bit two-way Power3 processor running at 375 MHz and 8 GBytes of main memory.

A. Functional Hashing

In the first experiment, we evaluated the effectiveness of the AND/INVERTER graph structure and functional hashing. For this we constructed the circuit graphs for the design specifications and compared the sizes generated by simple hashing described in Section III-A with the results of the functional method presented in Section III-B. The histogram for the size reduction of the circuit graphs is plotted in Fig. 21. As shown, on average the given sample of circuit representations can be reduced by 50%, the runtime overhead for all runs was negligible. Since it is not

TABLE I
PERFORMANCE OF BDD SWEEPING AND SAT SEARCH FOR VARIOUS BDD SIZE LIMITS (HIGHLIGHTED ENTRIES CORRESPOND TO GRAPHS SHOWN IN FIG. 22)

BDD Size Limit	BDD Sweeping Memory [kB] / Time [sec]	SAT Search # Backtracks / Time [sec]	Total Memory [kB] / Time [sec]
2^0	**342 / 0.00**	**2407939 / 347.17**	**342 / 347.17**
2^1	347 / 0.64	115 / 7.40	347 / 8.04
2^2	349 / 0.60	115 / 7.47	349 / 8.07
2^4	**358 / 0.66**	**87 / 6.93**	**358 / 7.59**
2^6	372 / 0.78	43 / 6.88	372 / 7.66
2^8	396 / 1.18	43 / 7.03	396 / 8.21
2^{10}	791 / 1.67	43 / 6.57	791 / 8.24
2^{12}	2212 / 4.22	43 / 6.30	2212 / 10.52
2^{14}	2219 / 6.98	43 / 6.15	2219 / 13.13
2^{16}	8381 / 12.14	27 / 5.27	8381 / 17.41
2^{17}	**8540 / 19.16**	**0 / 0.00**	**8540 / 19.16**

clear which choice of recursive branch will lead to more functional mapping, we do observe a few cases where enlargement in the circuit takes place compared to structural hashing. However, this increase is easily offset by savings in other parts of the circuit. The results suggest that the presented hashing method is not only useful for Boolean reasoning but can also be applied for general netlist compression.

B. Formal Equivalence Checking

1) Interleaved Invocation of BDD Sweeping and SAT: First, to demonstrate the effect of the interleaved application of BDD sweeping and structural SAT search, we chose a miter structure from a particular equivalence checking problem. From the above-mentioned circuits, we selected an output pair which has 97 inputs, 1322 gates for the specification, and 2782 gates for the implementation.

In a series of experiments the BDD sweeping algorithm was applied to the original miter circuit with varying limits for the BDD size. After sweeping, the SAT solver was invoked on the compressed miter structure and run until equivalence was proven. Table I gives the results for different limits on the BDD size. As shown, there is a clear tradeoff between the effort spent in BDD sweeping and SAT search. For this example, the optimal performance was achieved with a BDD size limit of 2^4. The use of BDD sweeping and SAT search in the described incremental and intertwined manner heuristically adjusts the effort spent by each algorithm to the difficulty of the problem.

Fig. 22 shows the two outputs forming the miter structure for three selected runs for which the corresponding entries are highlighted in Table I. In the drawings, all inputs are positioned at the bottom. The placement of the AND vertices is done based on their connectivity to the two outputs which are located at the top. AND vertices that feed only one of the two outputs are aligned on the left and right side of the picture. Vertices that are shared between both cones are placed in the middle. Further, filled circles and open circles are used to distinguish between vertices with and without BDDs, respectively.

Part (a) of the picture illustrates the initial miter structure without performing any BDD sweeping. As shown, a number of vertices are shared as a result of structural and functional hashing. In order to prove equivalence at this stage, the SAT solver would need about 2.4 million backtracks. Fig. 22(b) shows the miter structure after performing a modest BDD sweep with a size limit of 16 BDD nodes. It is clear that many more vertices are shared at this point. The SAT solver

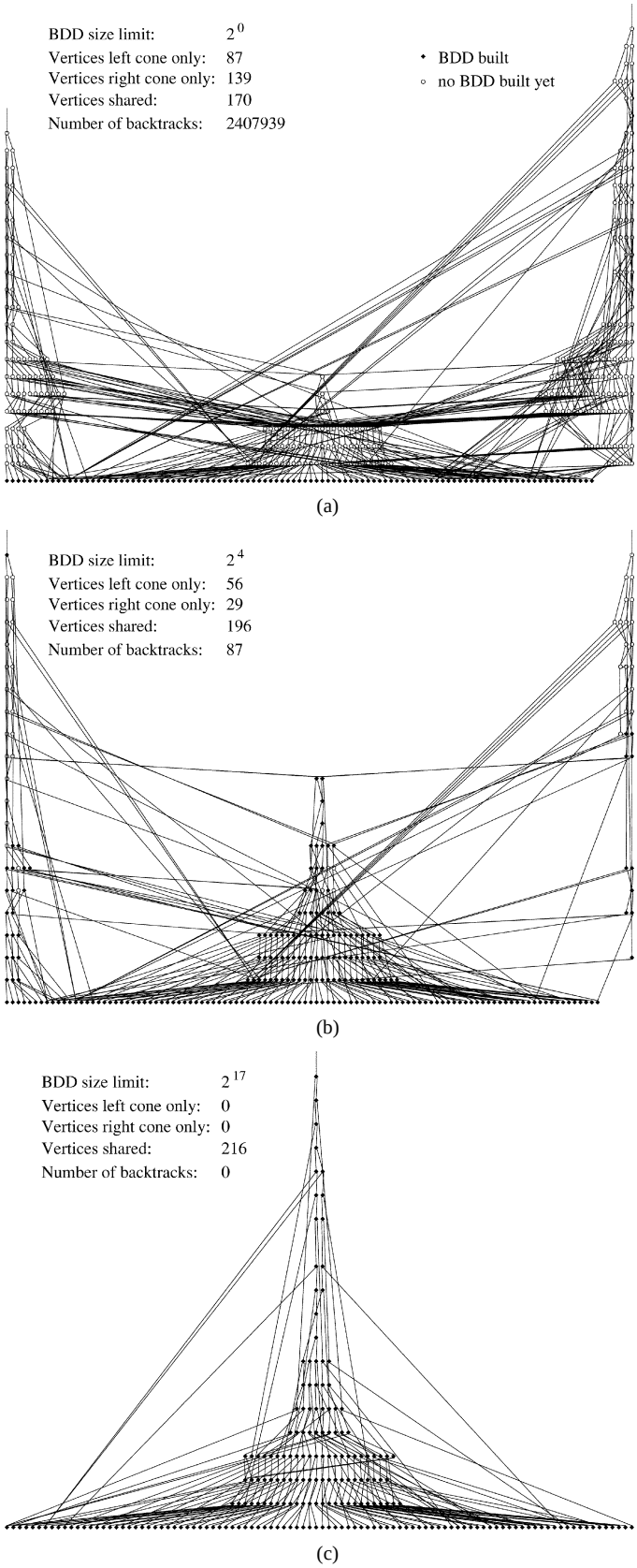
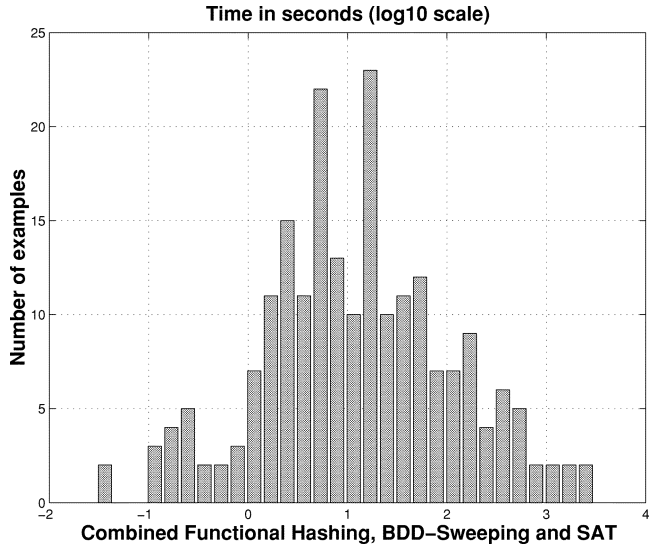
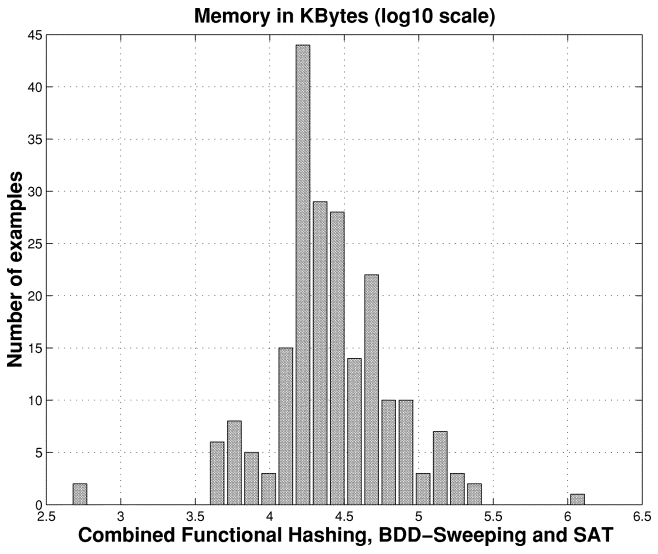


Fig. 22. Example the two outputs forming the miter structure at different stages of BDD sweeping: (a) no sweeping performed; (b) sweeping result with BDD size limit of 2^4; (c) sweeping result with BDD size limit of 2^{17}.

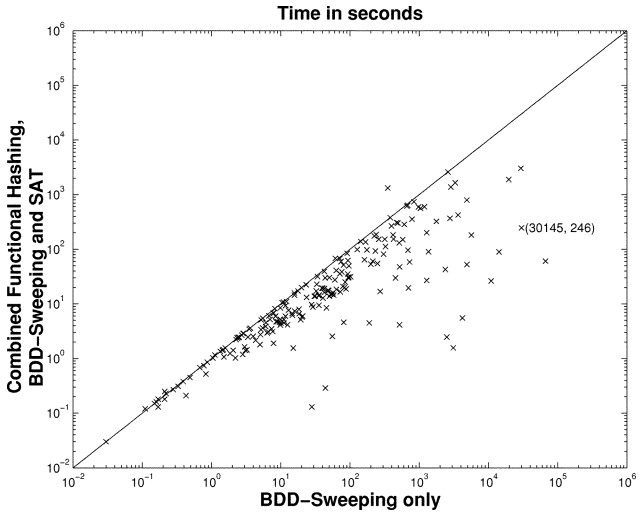


(a)

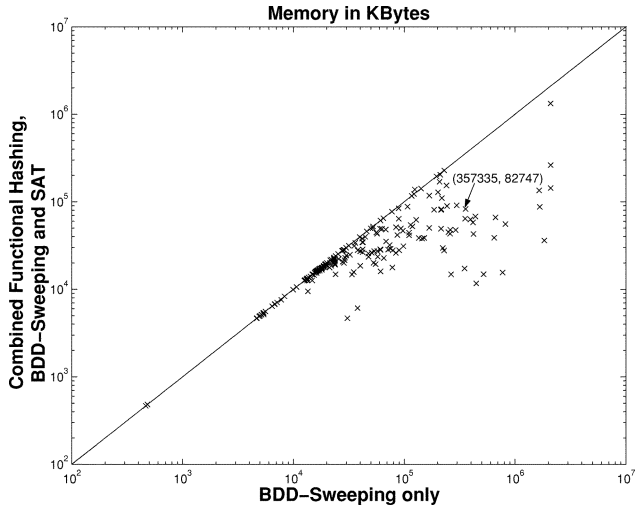


(b)

Fig. 23. Computing resources for equivalence checking of the given set of 488 circuits: (a) runtime distribution and (b) memory distribution.



(a)



(b)

Fig. 24. Comparison of the original BDD sweeping algorithm with the new algorithm for equivalence checking: (a) runtime and (b) memory.

can now prove equivalence using only 87 backtracks. The last part of the picture displays the miter structure when it is completely merged by BDD sweeping. Here, the equivalence proof required building BDDs for all miter vertices.

2) Overall Performance in an Industrial Setting: In a further experiment, we evaluated the overall effectiveness of the combination of BDD sweeping, structural transformations, structural SAT, and random simulation. First, to provide an intuition of the required computing resources in a typical industrial application project, we ran a full equivalence check for the given set of designs using the presented approach. All designs are correct, i.e., the specification and implementation are functionally equivalent. Fig. 23 provides two histograms showing the distribution of the runtimes and memory use. As shown, the majority of circuits can be compared within a few ten seconds using less than 100 MBytes of memory.

3) Comparison of Combined Approach With Simple BDD Sweeping: Next, we compare the presented comprehensive approach with the original plain BDD sweeping algorithm as described in [3]. For the former, BDD sizes were varied from a *bdd_lower_size_limit* of 2^4 to a *bdd_upper_size_limit* of 2^{28}, with a *delta_bdd_limit* of 2^1. The *sat_backtrack_limit* ranged from a low of 1000 to a high of 1 000 000 with a *delta_sat_limit* varying between 1000 and 5000. The results are given in Fig. 24. As shown, the majority of circuits could be compared using significantly less time, sometimes two orders of magnitude less. The memory consumption remained about the same. The performance for a particularly complex circuit is marked in both diagrams. This design contains 55 096 gates, 302 primary inputs, 2876 outputs, and 2200 latches. The verification run included 5076 comparisons and 231 232 consistency checks (checks for all nets, prohibiting floating, or collision condition) and could be accomplished in 246 s versus 8.3 h using 82 MBytes versus 357 MBytes for the new and old methods, respectively.

C. Formal Property Checking

For evaluating the effectiveness of the presented approach for property checking, we integrated the algorithms in a bounded

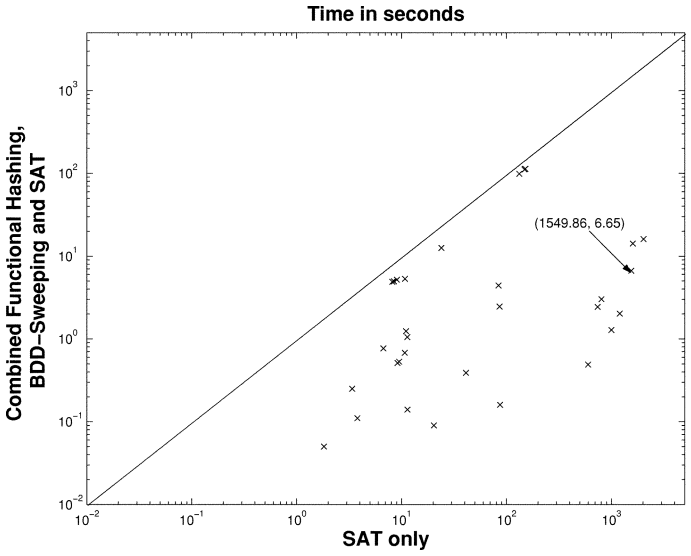


Fig. 25. Comparison of the application of plain SAT versus the presented combined approach to prove unreachability in a bounded model checking setting.

model checking setting. Bounded model checking [21] is based on a sequence of combinational property checks using a finite unfolding of the design under verification. By systematically increasing the unfolding depth from 1 to a bounded integer k, this approach checks whether the property can be disproved by a counter example of length k or less. We implemented the bounded model checking approach in a transformation-based tool setting. For proving a particular property, the design description is converted into a bit-level netlist.

In our experimental setting the netlist is first simplified by iteratively applying a sequence of reduction engines including: 1) a combinational simplification engine based on the presented algorithm and 2) a retiming engine for sequential optimization [22]. The first engine eliminates functionally equivalent circuit structures and removes redundant registers. The second engine reduces the number of registers by applying an ILP-based min-area retiming algorithm. After simplification the netlist is verified with the above mentioned bounded model checking method by checking a sequence of SAT problems.

In the first experiment, we compared the effectiveness of the presented approach, which combines structural and functional hashing, BDD sweeping, and SAT against a plain application of the SAT procedure only. Both methods work on the circuit graph that was compressed by simple hashing only (as described in Section III-A). For this experiment we used 40 properties from the given set of designs that are boundedly correct (i.e., the target states are not reachable within the given unfolding limit). In this experiment, the unfolding length varied between 6 and 25 time frames.

The results of the comparison are depicted in Fig. 25. Each marker in the diagram represents a particular property and the position indicates the performances of the two approaches. As shown, the combined approach is vastly superior, sometimes by several orders of magnitude. This result is particularly interesting because, in contrast to an application in equivalence checking, the unfolded circuit structure does not necessarily

TABLE II
PERFORMANCE OF BOUNDED MODEL CHECKING FOR VARIOUS SWEEPING LIMITS FOR THE MARKED PROPERTY OF FIG. 25

BDD Size Limit	Graph Size # Vertices	SAT Search # Backtracks	Performance Memory [kB] / Time [sec]
2^0	1669	967887	924 / 1549.86
2^1	1228	6563	4494 / 10.21
2^2	1228	6563	4496 / 10.43
2^3	1228	6563	4497 / 10.51
2^4	1222	6221	4498 / 9.43
2^5	1211	5590	4501 / 8.59
2^6	1203	4396	4503 / 6.65
2^7	1188	4484	4504 / 6.71
2^8	1188	4484	4505 / 6.80

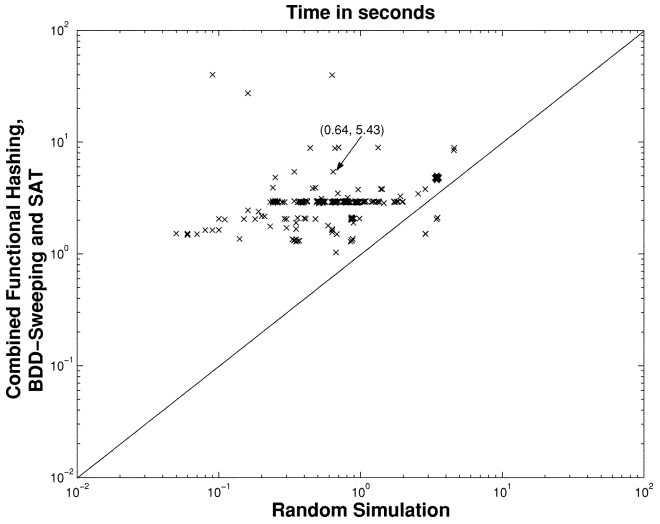


Fig. 26. Comparison of the application of random simulation versus the presented combined approach to prove reachability of easy-to-hit properties in a bounded model checking setting.

contain a large number of functionally identical nets that cannot be discovered by simple structural hashing. The efficient handling of these cases demonstrates the significant robustness and versatility of the presented approach. As an illustration, the plain application of SAT search required 1550 s for proving the property highlighted in Fig. 25; in contrast the combined approach used only 7 s. Table II shows the performances for various BDD sweeping limits and gives the corresponding compression of the graph structure.

In a second experiment, we evaluated the effectiveness of simulation in the presented setting. For this, we compared the plain application of random simulation [14] with an implementation that includes hashing, BDD sweeping, and SAT. We used 396 easy-to-hit properties from the given set of designs. In this experiment, the unfolding depth varied between 6 and 25 time frames, depending on the depth of the counterexample. We found that simulation significantly outperforms structured search techniques, such as SAT, in hitting reachable target states. Fig. 26 illustrates the run times for random simulation versus the combined approach. It reaffirms our view of using simulation to discharge easy to hit targets and utilizing more expensive but exhaustive techniques such as BDD sweeping and SAT to hit difficult targets, or to prove targets unreachable.

In a last experiment, we selected 10 deep hard-to-hit properties from the given set of designs. None of these properties

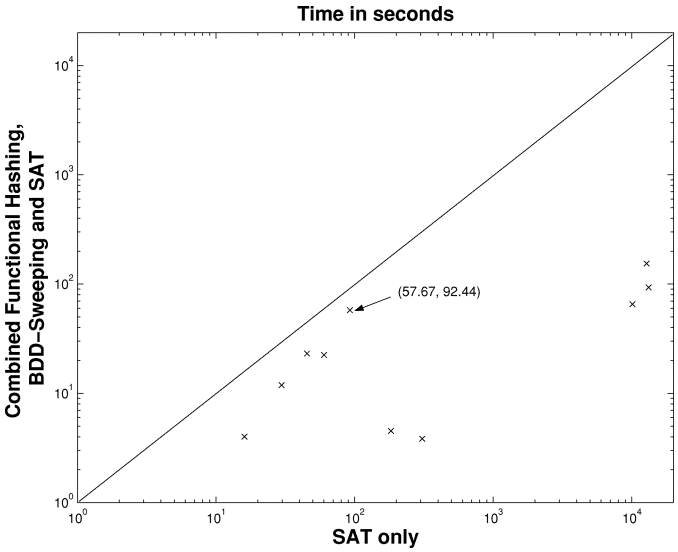


Fig. 27. Comparison of the application of plain SAT versus the presented combined approach to prove reachability of hard-to-hit properties in a bounded model checking setting.

could be handled by the simulation approach used in the previous experiment. Here the unfolding depth varied between 17 to as much as 400 for some particularly hard satisfiable properties. Fig. 27 illustrates the run times for the combined approach compared against an application of the SAT-solver alone. Again, the combined approach vastly outperforms the use of SAT alone.

X. Conclusion

In this paper, we presented a combination of techniques for Boolean reasoning using structural transformations, BDD sweeping, an SAT solver, and random simulation in a tight integration. All four methods work on a shared AND/INVERTER graph representation of the problem and are invoked in an intertwined manner. This unique integration results in a robust summation of their natively orthogonal strength. Using an extensive set of industrial problems we demonstrate the effectiveness of the presented technique for a wide range of applications.

The outlined approach is well suited for formal equivalence checking. It is currently integrated in the equivalence checking tool Verity [23], which has been used on numerous practical microprocessor and ASIC designs within IBM. Nevertheless, the presented reasoning method is equally applicable to other CAD applications, such as logic synthesis, timing analysis, or formal property checking.

References

[1] M. K. Ganai and A. Kuehlmann, "On-the-fly compression of logical circuits," in *Int. Workshop Logic Synthesis*, May 2000.
[2] D. Brand, "Verification of large synthesized designs," *Dig. Tech. Papers IEEE/ACM Int. Conf. Computer-Aided Design*, pp. 534–537, Nov. 1993.
[3] A. Kuehlmann and F. Krohm, "Equivalence checking using cuts and heaps," *Proc. 34th ACM/IEEE Design Automation Conf.*, pp. 263–268, June 1997.
[4] V. Paruthi and A. Kuehlmann, "Equivalence checking combining a structural SAT-solver, BDD's, and simulation," *Proc. IEEE Int. Conf. Computer Design*, pp. 459–464, Sept. 2000.
[5] M. Davis and H. Putnam, "A computing procedure for quantification theory," *J. Assoc. Computing Machinery*, vol. 7, pp. 102–215, 1960.
[6] M. Davis, G. Logeman, and D. Loveland, "A machine program for theorem proving," *Commun. ACM*, vol. 5, pp. 394–397, July 1962.
[7] J. P. Marques-Silva and K. A. Sakallah, "GRASP: A search algorithm for propositional satisfiability," *IEEE Trans. Comput.*, vol. 48, pp. 506–521, May 1999.
[8] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik, "Chaff: Engineering an efficient SAT solver," *Proc. 38th ACM/IEEE Design Automation Conf.*, pp. 530–535, June 2001.
[9] S. B. Akers, "Binary decision diagrams," *IEEE Trans. Comput.*, vol. 27, pp. 509–516, June 1978.
[10] R. E. Bryant, "Graph-based algorithms for Boolean function manipulation," *IEEE Trans. Comput.*, vol. 35, pp. 677–691, Aug. 1986.
[11] S.-W. Jeong, B. Plessier, G. Hachtel, and F. Somenzi, "Extended BDD's: Trading off canonicity for structure in verification algorithms," *Dig. Tech. Papers IEEE Int. Conf. Computer-Aided Design*, pp. 464–467, Nov. 1991.
[12] G. L. Smith, R. J. Bahnsen, and H. Halliwell, "Boolean comparison of hardware and flowcharts," *IBM J. Res. Development*, vol. 26, pp. 106–116, Jan. 1982.
[13] H. Hulgaard, P. F. Williams, and H. R. Andersen, "Equivalence checking of combinational circuits using Boolean expression diagrams," *IEEE Trans. Computer-Aided Design*, vol. 18, July 1999.
[14] F. Krohm, A. Kuehlmann, and A. Mets, "The use of random simulation in formal verification," *Proc. IEEE Int. Conf. Computer Design*, pp. 371–376, Oct. 1996.
[15] R. Mukherjee, J. Jain, K. Takayama, M. Fujita, J. A. Abraham, and D. S. Fussel, "An efficient filter-based approach for combinational verification," *Trans. Computer-Aided Design*, vol. 18, pp. 1542–1557, Nov. 1999.
[16] S. M. Reddy, W. Kunz, and D. K. Pradhan, "Novel verification framework combining structural and OBDD methods in a synthesis environment," *Proc. 32th ACM/IEEE Design Automation Conf.*, pp. 414–419, June 1995.
[17] A. Gupta and P. Ashar, "Integrating a Boolean satisfiability checker and BDD's for combinational equivalence checking," in *Proc. Int. Conf. VLSI Design*, 1998, pp. 222–225.
[18] J. R. Burch and V. Singhal, "Tight integration of combinational verification methods," *Dig. Tech. Papers IEEE/ACM Int. Conf. Computer-Aided Design*, pp. 570–576, Nov. 1998.
[19] W. Kunz, "HANNIBAL: An efficient tool for logic verification based on recursive learning," *Dig. Tech. Papers IEEE/ACM Int. Conf. Computer-Aided Design*, pp. 538–543, Nov. 1993.
[20] S. Kundu, L. M. Huisman, I. Nair, V. Ivengar, and L. Reddy, "A small test generator for large designs," in *Proc. Int. Test Conf.*, 1992, pp. 30–40.
[21] A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu, "Symbolic model checking without BDD's," in *Proc. 5th Int. Conf. Tools Algorithms for Construction and Analysis of Systems*, Amsterdam, The Netherlands, Mar. 1999, pp. 193–207.
[22] A. Kuehlmann and J. Baumgartner, "Transformation-based verification using generalized retiming," in *Computer Aided Verification* Paris, France, 2001, pp. 104–117.
[23] A. Kuehlmann, A. Srinivasan, and D. P. LaPotin, "Verity – A formal verification program for custom CMOS circuits," *IBM J. Res. Development*, vol. 39, no. 1/2, pp. 149–165, Jan./Mar. 1995.

Andreas Kuehlmann (SM'01) received the Dipl-Ing. degree and the Dr.-Ing. habil degree in electrical engineering from the University of Technology, Ilmenau, Germany, in 1986 and 1990, respectively.

After graduation, from 1990 to 1991, he worked at the Fraunhofer Institute of Microelectronic Circuits and Systems, Duisburg, on a project to automatically synthesize embedded microcontrollers. In 1991, he joined the IBM T. J. Watson Research Center where he worked until June 2000 on various projects in high-level and logic synthesis and hardware verification. Among others, he was the principal author and project leader of Verity, IBM's standard equivalence checking tool. From January 1998 until May 1999, he visited the Department of Electrical Engineering and Computer Science, University of California, Berkeley. In July 2000, he joined the Cadence Berkeley Labs where he continues to work on synthesis and verification problems.

Viresh Paruthi received the B.Tech(H) degree in computer science and engineering from the Indian Institute of Technology, Kharagpur, in 1995, and the M.S. degree in computer engineering from the University of Cincinnati, in 1997.

In 1997, he joined IBM's Server Group where he supported and applied the Verity equivalence checker to the Gigahertz Processor project. Later, he transitioned into a development role contributing to the enhancement of Verity's core algorithms. His research interests include functional formal and semiformal verification, abstractions, and equivalence checking.

Florian Krohm received the Dipl-Ing. degree in electrical engineering from the University of Dortmund, Germany, in 1985. In 1992, he received the Dr.-Ing. degree from the University of Duisburg, Germany.

In 1985, he joined the Fraunhofer Institute of Microelectronic Circuits and Systems, Duisburg, where he worked on a project to automatically synthesize software development environments for embedded microcontrollers. His research focused on methods of retargetable code generation. In 1994, he joined the IBM T. J. Watson Research Center where he worked on topics of formal verification. In 1997, he took a management position in IBM's Microelectronic Division. He decided to return to technical work in 1999. His current research interests include static analysis of real world programs.

Malay K. Ganai (M'00) received the M.S. and Ph.D. degrees in computer engineering from the University of Texas at Austin, in 1998 and 2001, respectively. He received the B.Tech. degree in electrical engineering from IIT Kanpur, India, in 1992.

He worked with Larsen and Toubro, India, between 1992 and 1995 and with Cadence Design Systems, India, between 1995 and 1997. He is currently working with NEC Research Lab, USA, on next-generation formal verification tools. His research interests include CAD tools for VLSI, combining formal and informal verification, logic synthesis, and formal verification.