ABC: An Academic Industrial-Strength Verification Tool

Robert Brayton Alan Mishchenko

EECS Department, University of California, Berkeley, CA 94720, USA
{brayton, alanmi}@eecs.berkeley.edu

Abstract. ABC is a public-domain system for logic synthesis and formal verification of binary logic circuits appearing in synchronous hardware designs. ABC combines scalable logic transformations based on And-Inverter Graphs (AIGs), with a variety of innovative algorithms. A focus on the synergy of sequential synthesis and sequential verification leads to improvements in both domains. This paper introduces ABC, motivates its development, and illustrates its use in formal verification.

Keywords: Model checking, equivalence checking, logic synthesis, simulation, integrated sequential verification flow.

1 Introduction

Progress in both academic research and industrial products critically depends on the availability of cutting-edge open-source tools in the given domain of EDA. Such tools can be used for benchmarking, comparison, and education. They provide a shared platform for experiments and can help simplify the development of new algorithms. Equally important for progress is access to real industrial-sized benchmarks.

For many years, the common base for research in logic synthesis has been SIS, a synthesis system developed by our research group at UC Berkeley in 1987-1991. Both SIS [35] and its predecessor MIS [8], pioneered multi-level combinational logic synthesis and became trend-setting prototypes for a large number of synthesis tools developed by industry.

In the domain of formal verification, a similar public system has been VIS [9], started at UC Berkeley around 1995 and continued at the University of Colorado, Boulder, and University of Texas, Austin. In particular, VIS features the latest algorithms for implicit state enumeration [15] with BDDs [11] using the CUDD package [36].

While SIS reached a plateau in its development in the middle 90's, logic representation and manipulation methods continued to be improved. In the early 2000s, And-Inverter Graphs (AIGs) emerged as a new efficient representation for problems arising in formal verification [22], largely due to the published work of A. Kuehlmann and his colleagues at IBM.

In that same period, our research group worked on a multi-valued logic synthesis system, MVSIS [13]. Aiming to find better ways to manipulate multi-valued relations, we experimented with new logic representations, such as AIGs, and found that, in addition to their use in formal verification, they can replace more-traditional representations in logic synthesis. As a result of our experiments with MVSIS, we developed a methodology for tackling problems, which are traditionally solved with SOPs [35] and BDDs [37], using a combination of random/guided simulation of AIGs and Boolean satisfiability (SAT) [25].

Based on AIGs as a new efficient representation for large logic cones, and SAT as a new way of solving Boolean problems, in the summer 2005, we switched from multi-valued methods in MVSIS to binary AIG-based synthesis methods. The resulting CAD system, ABC, incorporates the best algorithmic features of MVSIS, while supplementing them with new findings.

One such finding is a novel method for AIG-based logic synthesis that replaced the traditional SIS logic synthesis flow, which was based on iterating elimination, substitution, kerneling, don't-care-based simplification, as exemplified by SIS scripts, *script.algebraic* and *script.rugged*. Our work on AIGs was motivated by fast compression of Boolean networks in formal verification [5]. We extended this method to work in synthesis, by making it delay-aware and replacing two-level structural matching of AIG subgraphs with functional matching of the subgraphs based on enumeration of 4-input cuts [26].

It turned out that the fast AIG-based optimizations could be made even more efficient by applying them to the network many times. Doing so with different parameter settings led to results in synthesis comparable or better than those of SIS, while requiring much less memory and runtime. Also this method is conceptually simpler than the SIS optimization flow, saving months of human-effort in code development and tuning. The savings in runtime/memory led to the increased scalability of ABC, compared to SIS. As a result, ABC can work on designs with millions of nodes, while SIS does not finish on these designs after many hours, and even if it finishes, the results are often inferior to those obtained by the fast iterative computations in ABC.

The next step in developing ABC was to add an equivalence checker for verifying the results of synthesis, both combinational and sequential [29]. Successful equivalence checking motivated experiments with model checking, because both types of verification work on a miter circuit and have the common goal of reducing it to the constant 0. To test this out, we submitted an equivalence checker in ABC to the hardware model checking competition at CAV 2008, winning in two out of three categories.

Working on both sequential synthesis and verification has allowed us to leverage the latest results in both domains and observe their growing synergy. For example, the ability to synthesize large problems and show impressive gains spurs development of equally scalable equivalence checking methods, while the ability to scalably verify sequential equivalence problems spurs the development, use, and acceptance of aggressive sequential synthesis. In ABC, similar concepts are used in both synthesis and verification: AIGs, rewriting, SAT, sequential SAT sweeping, retiming, interpolation, etc.

This paper provides an overview of ABC, lists some of the ways in which verification ideas have enriched synthesis methods, shows how verification is helped by constraining or augmenting sequential synthesis, and details how various algorithms have been put together to create a fairly powerful model checking engine that can rival some commercial offerings. We give an example of the verification flow applied to an industrial model checking problem.

The rest of the paper is organized as follows. Section 2 introduces the basic terminology used in logic synthesis and verification. Section 3 describes combinational and sequential AIGs and their advantages over traditional representations. Section 4 discusses the duality of synthesis and verification. Section 5 gives a case study of an efficient AIG implementation, complete with experimental results. Section 6 describes both the synthesis and verification flows in ABC and provides an example of the verification flow applied to an industrial model checking problem. Section 7 concludes the paper and sketches some on-going research.

2 Background

2.1 Boolean network

A *Boolean network* is a directed acyclic graph (DAG) with nodes corresponding to logic gates and directed edges corresponding to wires connecting the gates. The terms Boolean network, netlist, and circuit are used interchangeably in this paper. If the network is sequential, the memory elements are assumed to be D flip-flops with initial states.

A node *n* has zero or more *fanins*, i.e. nodes driving *n*, and zero or more *fanouts*, i.e. nodes driven by *n*. The *primary inputs* (PIs) are nodes without fanins in the current network. The *primary outputs* (POs) are a subset of nodes of the network. A *fanin (fanout) cone* of node *n* is a subset of all nodes of the network, reachable through the fanin (fanout) edges of the node.

2.2 Logic synthesis

Logic synthesis transforms a Boolean network to satisfy some criteria, for example, reduce the number of nodes, logic levels, switching activity. *Technology mapping* deals with representing the logic in terms of a given set of primitives, such as standard cells or lookup tables.

Combinational logic synthesis involves changing the combinational logic of the circuit with no knowledge of its reachable states. As a result, the Boolean functions of the POs and register inputs are preserved for any state of the registers. In contrast, *sequential logic synthesis* preserves behavior on the reachable states and allows arbitrary changes on the unreachable states. Thus, after sequential synthesis, the combinational functions of the POs and register inputs may have changed, but the resulting circuit is sequentially-equivalent to the original.

2.3 Formal verification

Formal verification tries to prove that the design is correct in some sense.

The two most common forms of formal verification are model checking and equivalence checking. *Model checking* of safety properties considers the design along with one or more properties and checks if the properties hold on all reachable states. *Equivalence checking* checks if the design after synthesis is equal to its initial version, called the *golden* model.

In modern verification flows, the circuit to be model-checked is transformed into a circuit called a *model checking miter* by supplementing the logic of the design with a monitor logic, which checks the correctness of the property. Similarly, in equivalence checking, the two circuits to be verified are transformed into an *equivalence checking miter* [7] derived by combining the pairs of inputs with the same names and feeding the pairs of outputs with the same names into EXOR gates, which are ORed to produce the single output of the miter.

In both property and equivalence checking, the miter is a circuit with the inputs of the original circuit and an output that produces value 0, if and only if the original circuit satisfies the property or if the two circuits produce identical output values under any input assignment (or, in sequential verification, under any sequence of input assignments, starting from the initial state).

The task of formal verification is to prove that the constructed miter always produces value 0. If synthesis alone does not solve the miter, the output can be asserted to be constant 1 and a SAT solver can be run on the resulting problem. If the solver returns "unsatisfiable", the miter is proved constant 0 and the property holds, or the original circuits are equivalent. If the solver returns "satisfiable", an assignment of the PIs leading to 1 at the output of the miter, called a *counter-example*, is produced, which is useful for debugging the circuit.

2.4 Verifiable synthesis

An ultimate goal of a synthesis system is to produce good results in terms of area, power, speed, capability for physical implementation etc, while allowing an unbiased (independent) verification tool to prove that functionality is preserved. Developing verifiable synthesis methods is difficult because of the inherent complexity of the sequential verification problem [20].

One *verifiable sequential synthesis* is described in [29]. This is based on identifying pairs of sequentially-equivalent nodes/registers, that is groups of signals having the same or opposite values in all reachable states. Such equivalent nodes/registers can be merged without changing the sequential behavior of the circuit, often leading to substantial reductions, e.g. some parts of the logic can be discarded because they no longer affect the POs. This sequential synthesis technique is used extensively in ABC to reduce both designs and sequential miters.

3 And-Inverter Graphs

3.1 Combinational AIGs

A combinational *And-Invertor Graph* (AIG) is a Boolean network composed of two-input ANDs and inverters. To derive an AIG, the SOPs of the nodes in a logic network are factored, the AND and OR gates of the factored forms are converted into two-input ANDs and inverters using DeMorgan's rule, and these nodes are added to the AIG manager in a topological order. The *size* (*area*) of an AIG is the number of its nodes; the *depth* (*delay*) is the number of nodes on the longest path from the PIs to the POs. The goal of optimization by local transformations of an AIG is to reduce both area and delay.

Structural hashing of AIGs ensures that all constants are propagated and, for each pair of nodes, there is at most one AND node having them as fanins (up to a permutation). Structural hashing is performed by hash-table lookups when AND nodes are created and added to an AIG manager. Structural hashing was originally introduced for netlists of arbitrary gates in early IBM CAD tools [15] and was extensively used in formal verification [22]. Structural hashing can be applied on-the-fly during AIG construction, which reduces the AIG size. To reduce the number of AIG levels, the AIG is often *balanced* by applying the associative transform, $a(bc) = (ab)c$. Both structural hashing and balancing are performed in one topological traversal from the PIs and have linear complexity in the number of AIG nodes.

A *cut C* of a node *n* is a set of nodes of the network, called *leaves* of the cut, such that each path from a PI to *n* passes through at least one leaf. Node *n* is called the *root* of cut *C*. The cut *size* is the number of its leaves. A *trivial cut* of a node is the cut composed of the node itself. A cut is *K-feasible* if the number of nodes in the cut does not exceed *K*. A cut is *dominated* if there is another cut of the same node, which is contained, set-theoretically, in the given cut.

A *local* function of an AIG node *n*, denoted $f_n(x)$, is a Boolean function of the logic cone rooted in *n* and expressed in terms of the leaves, *x*, of a cut of *n*. The *global function* of an AIG node is its function in terms of the PIs of the network.

3.2 Sequential AIGs

Sequential AIGs extend combinational AIGs with technology-independent D-flip-flops with one input and one output, controlled by the same clock, omitted in the AIG representations.

We represent flip-flops in the AIG explicitly as additional PI/POs pairs. The PIs and register outputs are called *combinational inputs* (CIs) and the POs and register inputs are called *combinational outputs* (COs). The additional pairs of CI/CO nodes follow the regular PIs/POs, and are in one to one correspondence with each other. This representation of sequential AIGs differs from that used in [1] where latches are represented as attributes on AIG edges, similar to the work of Leiserson and Saxe [23].

The chosen representation of sequential AIGs allows us to work with the AIG manager as if it was a combinational AIG, and only utilize its sequential properties when sequential transformations are applied. For example, combinational AIG rewriting works uniformly on combinational and sequential AIGs, while sequential cleanup, which removes structurally equivalent flip-flops, exploits the fact that they are represented as additional PIs and POs. Sequential transformation, such as retiming, can add and remove latches as needed.

3.3 Distinctive features of AIGs

Representing logic using networks containing two-input nodes is not new. In SIS, there is a command *tech_decomp* [35] generating a two-input AND/OR decomposition of the network. However, there are several important differences that make two-input node representation in the form of AIGs much more efficient that its predecessors in SIS:

- Structural hashing ensures that AIGs do not contain structurally identical nodes. For example, node $a \wedge b$ can only exist in one copy. When a new node is being created, the hash table is checked, and if such node already exists, the old node is returned. The on-the-fly structural hashing is very important in synthesis applications because, by giving a global view of the AIG, it finds, in constant time, simple logic sharing across the network.
- Representing inverters as edge attributes. This feature is borrowed from the efficient implementation of BDDs using complemented edges [36]. As a result, single-input nodes representing invertors and buffers do not have to be created. This saves memory and allows for applying DeMorgan's rule on-the-fly, which increases logic sharing.
- The AIG representation is uniform and fine-grain, resulting in a small, fixed amount of memory per node. The nodes are stored in one memory array in a topological order, resulting in fast, CPU-cache-friendly traversals. To further save memory, our AIG packages compute fanout information on demand, resulting in 50% memory reduction in most applications. Similar to the AIG itself, fanout information for arbitrary AIG structures can be represented efficiently using a constant amount of memory per node.

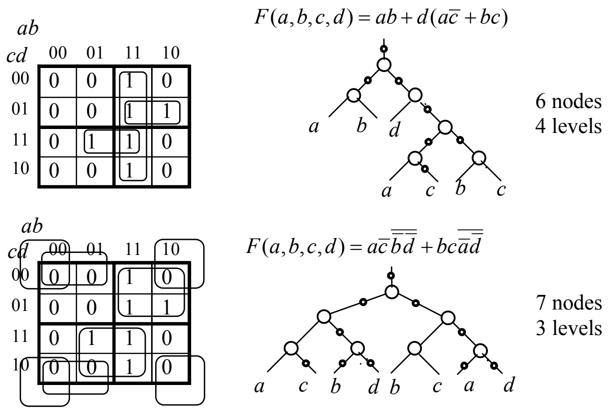
Fig. 1 shows a Boolean function and two of its structurally-different AIGs. The nodes in the graphs denote AND-gates, while the bubbles stand for complemented edges. The figure shows that the same completely-specified Boolean function can be represented by two structurally different AIGs, one with smaller size and larger depth, the other vice versa.

3.4 Comparing logic synthesis in SIS and in ABC

In terms of logic representation, the main difference between SIS and ABC, is that SIS works on a logic network whose nodes are represented using SOPs, while ABC works on an AIG whose nodes are two-input AND gates. A SIS network can be converted into an AIG by decomposing each node into two-input AND gates. For a deterministic decomposition algorithm, the resulting AIG is unique. However, the reverse transformation is not unique, because many logic networks can be derived from the same AIG by grouping AND gates in different ways. This constitutes the main difference between SIS and ABC.

SIS works on one copy of a logic network, defined by the current boundaries of its logic nodes, while ABC works on an AIG. A cut computed for an AND node in the AIG can be seen as a logic node. Since there are many cuts per logic node, the AIG can be seen as an implicit representation of many logic networks. When AIG rewriting is performed in ABC, a minimal representation is found among all decompositions of all structural cuts in the AIG, while global logic sharing is captured using a structural hashing table. Thus, ABC is more likely to find a smaller representation in terms of AIG nodes than SIS, which works on one copy of the logic network and performs only those transformations that are allowed by this network.

SIS and ABC use different heuristics for logic manipulation, so it is still possible that, for a particular network, SIS finds a better solution than ABC.



ab
cd
$F(a,b,c,d) = ab + d(a\overline{c} + bc)$
6 nodes
4 levels
$F(a,b,c,d) = a\overline{c}\,\overline{\overline{b}\overline{d}} + bc\overline{\overline{a}\overline{d}}$
7 nodes
3 levels

Fig. 1. Two different AIGs for a Boolean function.

3.5 Advantages of AIGs summarized

The following properties of AIGs fascilitate development of robust applications in synthesis, mapping, and formal verification:

- AIGs unify the synthesis/mapping/verification by representing logic compactly and uniformly. The results of technology-independent synthesis are naturally expressed as an AIG. During technology mapping, the AIG is used as a subject graph annotated with cuts that are matched with LUTs or gates. At any time, verification can be performed by contructing a miter of the two synthesis snapshots represented as one AIG, handled by a complex AIG-based verification flow.
- Although AIG transformations are local, they are performed with a global view afforded by the structural hashing table. Because these computations are memory/runtime efficient, they can be iterated, leading to superior results, unmatched by a single application of a more global transform.
- An AIG can be efficiently duplicated, stored, and passed between calling applications as a memory buffer or compactly stored on disk in the AIGER format [4].

4 Synthesis-verification duality

Recent advances in formal verification and logic synthesis have made these fields increasingly interdependent, especially in the sequential domain [10].

In addition to algorithm migration (for example, AIG rewriting, SAT solving, interpolation came to synthesis from verification), hard verification problems challenge synthesis methods that are used to simplify them, while robust verification solutions enable more aggressive synthesis. For example, bold moves can be made in sequential synthesis by assuming something that seems likely to hold but cannot be proved easily. If the result can be verified (provided that sequential verification is powerful enough), synthesis is over. Otherwise, different types of synthesis can be tried, for example, traditional or other risk-free synthesis. Preliminary experiments show potentially large gains in synthesis for industrial problems.

4.1 Known synergies

In this section, we outline several aspects of combinational and sequential verification that benefit from synthesis.

Combinational equivalence checking (CEC) proves equivalence of primary outputs and register inputs after combinational synthesis. To this end, a combinational miter is constructed and solved using a set of integrated methods, including simulation, SAT solving, and BDD or SAT sweeping [22]. Running combinational synthesis on a miter during verification substantially improves the CEC runtime [27]. This is because synthesis quickly merges shallow equivalences and reduces the size of the miter, allowing difficult SAT calls go through faster.

A similar observation can be made about *retiming* [23]. If retiming has been applied during sequential synthesis, it is advantageous to apply *most-forward* retiming as one of the preprocessing steps during sequential verification. It can be shown that if during sequential synthesis only retiming was applied without changing the logic structure, then most forward retiming followed by an inductive register correspondence computation is guaranteed to prove sequential equivalence [21]. This observation is used in our verification tool, which allows the user to enable retiming as an intermediate step during sequential verification [29].

Yet another synthesis/verification synergy holds when induction is used to detect and merge sequentially equivalent nodes. The following result was obtained in [29]: if a circuit was synthesized using only k-step induction to find equivalent signals, then equivalence between the original and final circuits is guaranteed provable using k-step induction with the same k.

These results lead to the following rule of thumb which is used in our verification flow: if a transformation is applied during synthesis, it is often helpful (and necessary) to apply the same or more powerful transformations during verification.

5 Case study: Developing a fast sequential simulator for AIGs

Several applications suffer from the prohibitive runtime of a sequential gate-level simulator. For example, in formal verification, the simulator is used to quickly detect easy-to-disprove properties or as a way to compute simulation signatures of internal nodes proving their equivalence. The same sequential simulator is useful to estimate switching activity of registers and internal nodes. The pre-computed switching activity can direct transformations that reduce dynamic power dissipation in low-power synthesis. In this case study, based on [19], we discuss how to develop a fast sequential simulator using AIGs.

5.1 Problem formulation

The design is sequentially simulated for a fixed number of time-frames. A sequential simulator applies, at each time step, a set of values to the PIs. In the simplest case, random PI patterns are generated to have a 0.5 probability of the node changing its value (fixed toggle rate). In other scenarios, the probability of an input transitions is given by the user, or produced by another tool. For example, if an input trace is known, it may be used for simulating the design. It is assumed that the initial state is known and initializes the sequential elements at the first time-frame. In subsequent time frames, the state derived at the previous iteration is used.

The runtime of sequential simulation can be reduced by minimizing the memory footprint. This is because most CPUs have a local cache ranging in size from 2Mb to 16Mb. If an application requires more memory than this, repeated cache misses cause the runtime to degrade. Therefore, the challenge is to design a simulator that uses a minimalistic data-structure without compromising the computation speed.

We found three orthogonal ways of reducing the memory requirements of the simulator, which in concert greatly improve its performance.

Compacting logic representation. Sequential designs are represented as AIGs. A typical AIG package uses 32 or more bytes to represent each AIG object (an internal AND node, an PI/PO, or a flop outputs/inputs). However, a minimalistic AIG package requires only 8 bytes per object. For an internal node, two integer fields, four bytes each, are used to store the fanin IDs. Other data structures may be temporarily allocated, for example, a hash-table for structural hashing may be used during AIG construction and deallocated before simulation begins.

Recycling simulation memory. When simulation is applied to a large sequential design, storing simulated values for all nodes in each timeframe requires a lot of memory. One way of avoid this, is to use the simulation information as soon as it is computed and to recycle the memory when it is not needed. For example, to estimate switching activity, we are only interested in counting the number of transitions seen at each node. For this, an integer counter can be used, thereby adding four bytes per object to the AIG package memory requirements, while the simulation information does not have to be stored.

Additionally, there is no need to allocate simulation memory for each object in the AIG. At any time during simulation, we only need to store simulation values for each combinational input/output and the nodes on the simulation frontier. These are all the nodes whose fanins are already simulated but at least one fanout is not yet simulated. For industrial designs, the number of internal nodes where simulation information should be stored is typically very small. For example, large industrial designs tend to have simulation frontier that is less than 1% of the total number of AIG nodes. The notion of a simulation frontier has also been useful to reduce memory requirements for the representation of priority cuts [28].

Bit-parallel simulation of two time-frames at the same time. A naïve approach to estimate the transition probability for each AIG node would be to store simulation patterns in two consecutive timeframes. Then, this information is compared (using bitwise XOR), and the number of ones in the bitwise representation is accumulated while simulating the timeframes. However, saving simulation information at each node for two consecutive timeframes leads to a large memory footprint. For example, an AIG with 1M objects requires 80Mb to store the simulation information for two timeframes, assuming 10 machine words (40 bytes) per object.

This increase in memory can be avoided by simultaneously simulating data belonging to two consecutive timeframes. In this case, comparison across the timeframes is made immediately, without memorizing previously computed results. This leads to duplicating the computation effort by simulating every pattern twice, one with the previous state value and the other with the current state value. However, the speedup due to not having to traverse the additional memory (causing excessive cash misses) outweighs the disadvantage of the re-computation.

5.2 Experimental results

This section summarizes two experiments performed to evaluate the new simulator.

The first experiment, was designed to show that the new sequential simulator, called *SimSwitch,* has affordable runtimes for large designs. Four industrial designs ranging from 304K to 1.3M AIG nodes were simulated with different numbers of simulation patterns, ranging from 2,560 to 20,480. The input toggle rate was assumed to be 0.5. The results are shown in Table 1. Columns "AIG" and "FF" show the numbers of AIG nodes and registers. The runtimes for different amounts of input patterns are shown in the last columns. Note that the runtimes are quite affordable even for the design with 1.3M AIG nodes. In all four cases, the 2,560 patterns were sufficient for node switching activity rates to converge to a steady state.

Table 1. Runtime of *SimSwitch*.

Design	**AIG**	**FF**	**Runtime for inputs patterns (seconds)**			
			2560	**5120**	**10240**	**20480**
C1	304K	1585	0.1	0.2	0.2	0.4
C2	362K	27514	2.7	2.9	4.1	6.6
C3	842K	58322	7.4	7.6	10.2	18.2
C4	1306K	87157	12.1	15.4	15.7	24.2

In the second experiment, we compare the runtime of S*imSwitch* vs. ACE-2.0 on 14 industry designs and 12 large academic benchmarks. The input toggle rate is assumed to be 0.5 for both tools. The number of input patterns is assumed to be 5,000 for both runs. All circuits are decomposed into AIG netlists before performing the switching estimation. The table of results can be found in [19]. The summary of results are as follows:

- For industry designs, *SimSwitch* is 149+ times faster than ACE-2.0.
- For academic benchmarks, SimSwitch is 85+ times faster than ACE-2.0.
- *SimSwitch* finished all testcases while ACE-2.0 times out on four industrial designs.

6 Optimization and verification flows

This section describes integrated sequences of transformations applied in ABC.

6.1 Integration of synthesis

The optimization algorithms of ABC are integrated into a system called Magic [31] and interfaced with a design database developed to store realistic industrial designs. For instance, Magic handles multiple clock domains, flip-flops with complex controls, and special objects such as adder chains, RAMs, DSP modules, etc. Magic was developed to work with hierarchical designs whose sequential logic cones, when represented as a monolithic AIG, contain more than 1M nodes. The algorithms are described in the following publications:

Synthesis
Scalable sequential synthesis [29] and retiming [34].
Combinational synthesis using AIG rewriting [26].
Combinational restructuring for delay optimization [30].

Mapping
Mapping with structural choices [14].
Mapping with global view and priority cuts [28].
Mapping to optimize application-specific metrics [18][19].

Verification
Fast sequential simulation [19]
Improved combinational equivalence checking [27].
Improved sequential equivalence checking [29][33].

The integration of components inside Magic is shown in Fig. 2. The design database is the central component interfacing the application packages. The design entry into Magic is performed through a file or via programmable APIs.

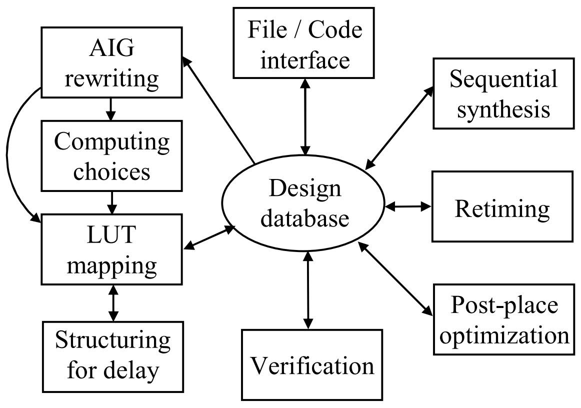


Fig. 2. Interaction of application packages in Magic.

Shown on the right of Fig. 2, is sequential synthesis based on detecting, proving, and merging sequentially equivalent nodes. This transformation can be applied at the beginning of the flow, before combinational synthesis and mapping. Another optional transform is retiming that reduces the total number of logic levels in the AIG or in the mapped network. Reducing the number of logic levels correlates with but does not always lead to an improvement in the clock frequency after place-and-route. The sequential transforms can be verified by sequential simulation and sequential equivalence checking.

Shown on the left of Fig. 2, is the combinational synthesis flow, which includes AIG rewriting, computing structural choices, and FPGA look-up-table (LUT) mapping. Computation of structural choices can be skipped if fast low-effort synthesis is desired. The result of mapping is returned to the design database or passed on to restructuring for delay optimization. After combinational synthesis, the design can be verified using combinational equivalence checking.

Finally, the box in the bottom right corner represents post-placement resynthesis, which includes incremental restructuring and retiming with wire-delay information.

6.2 Integration of verification

Similar to IBM's tool SixthSense [2], the verification subsystem of ABC is an integrated set of applications, divided into several categories: miter simplifiers (i.e. sequential synthesis), bug-hunters (i.e. bounded model checking), and provers (i.e. interpolation). The high-level interface coded in Python orchestrates the applications and determines the resource limits used. An embedded Python interpreter allows for defining new procedures in addition to those included.

An AIG file is read in, and the objective is to prove each output unsatisfiable or find a counter-example. The top-level functions are *prove* and *prove_g_pos*. The former works for single-output properties, while the latter applies the former to each output of a multi-output miter, or to several outputs grouped together based on the group's support. The main flow is

pre_simp → *quick_verify* → *abstract* → *quick_verify* → *speculate* → *final_verify*,

with each function passing the resulting AIG to the next function. At each stage, a set of resources is selected to spend on an algorithm. These resources are: total time, limit on the number of conflicts in SAT, maximum number of timeframes to unroll, maximum number of BDD nodes, etc. The allocation of resources is guided by the state of verification and the AIG parameters (the number of PIs, POs, FFs, AIG nodes, BMC depth reached, etc), which vary when the AIG is simplified and abstracted.

A global parameter *x_factor* can be used to increase the resources. If the problem is proved UNSAT by one of the application packages, the computation stops and the result is returned. . If the problem is found SAT and no abstraction has been done, the counter-example is returned.

The function *pre_simp* tries to reduce the AIG by applying several simplification algorithms:

- Phase abstraction, trying to identify clock-like periodic behaviors and deciding to unfold the design several frames depending on the clocks found and the amount of simplification this may allow [6].
- Trimming, which eliminates PIs that have no fanouts.
- Constraint extraction, which looks for implicit constraints inductively, uses these to simplify the design, and folds them back in with a structure such that if ever a constraint is not satisfied, the output is forced to be 0 from then on [12].
- Forward retiming and sequential FF correspondence, which finds correspondences between FFs and reduces the FF count, especially in SEC examples [29].
- Strong simplification script *simplify*, which iterates AIG rewriting, retiming for minimum FF (flip-flop) count, *k*-step sequential signal correspondence with *k* selected based on problem size. Also, the effort spent in signal correspondence can be adjusted by using a dedicated circuit-based SAT solver.

If *simplify* has already been applied to an AIG, then repeating it is usually fast, so the verification flow iterates it several times when other reductions have been done.

The function *quick_verify*, performed after each significant AIG reduction, is a low resource version of *final_verify*. These functions try to prove the problem by running interpolation or, if the problem seems small enough, by attempting BDD reachability.

The algorithm *abstract* is a combination of counter-example abstraction and proof-based abstraction implemented in a single SAT instance [17]. It returns an abstracted version of the AIG (a set of registers removed and replaced by PIs) and the frame count it was able to explore. To double check that a valid abstraction is derived, BMC (or, if the problem is small enough, BDD reachability) is applied to the resulting abstraction using additional resources. If a counter-example is found, *abstract* is restarted with additional resources from the frame where the counter-example was found.

The algorithm *speculate* applies speculative reduction [32][33]. This algorithm finds candidate sequential equivalences in the AIG, and creates a speculative reduced model, by transferring the fanouts of each equivalence class to a single representative, while creating new outputs, which become additional proof obligations. This model is refined as counter-examples are produced, finally arriving at a model that has no counterexamples up to some depth explored by BMC. Then, attempts are made to prove the outputs of the speculatively reduced model. If all outputs are successfully proved, the initial verification problem is solved. If at least one of the outputs failed, the candidate equivalences have to be filtered and speculative reduction repeated.

6.3 Example of running the verification flow

Below is an example of a printout produced by ABC during verification of an industrial design. Comments follow the printout.

```
abc> Read_file  example1.aig
PIs = 532, POs = 1, FF = 2389, ANDs = 12049
abc> prove

Simplifying
Number of constraints found = 3
Forward retiming, quick_simp, scorr_comp, trm: PIs = 532, POs = 1, FF = 2342, ANDs = 11054
Simplify:    PIs = 532, POs = 1, FF = 2335, ANDs = 10607
Phase abstraction:    PIs = 283, POs = 2, FF = 1460, ANDs = 8911

Abstracting
Initial abstraction:     PIs = 1624, POs = 2, FF = 119, ANDs = 1716, max depth = 39
Testing with BMC
bmc3 -C 100000 -T 50 -F 78:    No CEX found in 51 frames
Latches reduced from 1460 to 119
Simplify:    PIs = 1624, POs = 2, FF = 119, ANDs = 1687, max depth = 51
Trimming:     PIs = 158, POs = 2, FF = 119, ANDs = 734, max depth = 51
Simplify:    PIs = 158, POs = 2, FF = 119, ANDs = 731, max depth = 51

Speculating
Initial speculation:   PIs = 158, POs = 26, FF = 119, ANDs = 578, max depth = 51
Fast interpolation:   reduced POs to 24
Testing with BMC
bmc3 -C 150000 -T 75:    No CEX found in 1999 frames
PIs = 158, POs = 24, FF = 119, ANDs = 578, max depth = 1999
Simplify:    PIs = 158, POs = 24, FF = 119, ANDs = 535, max depth = 1999
Trimming:     PIs = 86, POs = 24, FF = 119, ANDs = 513, max depth = 1999
```

```
Verifying
Running reach -v -B 1000000 -F 10000 -T 75:    BDD reachability aborted
RUNNING interpolation with 20000 conflicts, 50 sec, max 100 frames:  'UNSAT'

Elapsed time: 457.87 seconds, total: 458.52 seconds
```

NOTES:

1. The file *example1.aig* is first read in and its statistics are reported: 532 primary inputs, 1 primary output, 2389 flip-flops, and 12049 AIG nodes.
2. 3 implicit constraints were found, but they turned out to be only mildly useful in simplifying the problem.
3. Phase abstraction found a cycle of length 2 and this was useful for simplifying the problem to 1460 FF from 2335 FF. Note that the number of outputs increased to 2 because the problem was unrolled 2 time frames.
4. Abstraction was successful in reducing the FF count to 119. This was proved valid out to 39 time frames.
5. BMC verified that the abstraction produced is actually valid to 51 frames, which gives us good confidence that the abstraction is valid for all time.
6. Trimming reduced the inputs relevant to the abstraction from 1624 to 158 and *simplify* reduced the number of AIG nodes to 731.
7. Speculation produced a speculative reduced model (SRM) with 24 new outputs to be proved and low resource interpolation proved 2 of them. The SRM model is simpler and has only 578 AIG nodes. The SRM was tested with BMC and proved valid out to 1999 frames.
8. Subsequent trimming and simplification reduced the PIs to 86 and AIG size to 513.
9. The final verification step first tried BDD reachability allowing it 75 sec. and to grow to up to 1M BDD nodes. It could not converge with these resources so it was aborted. Then interpolation has returned UNSAT, and hence all 24 outputs are proved.
10. Although *quick_verify* was applied between simplification and abstraction, and between abstraction and speculation, it was not able to prove anything, so its output is not shown.
11. The total time was 457 seconds on a Lenovo X301 laptop with 1.4Gb Intel Core2 Duo CPU and 3Gb RAM.

7 Conclusions and future work

In this paper, we discussed the development of ABC and described its basic principles. Started five years ago, ABC continues to grow and gain momentum as a public-domain tool for logic synthesis and verification. New implementations, improvements, bug fixes, and performance tunings are added frequently. Even the core computations continue to improve through better implementation and exploiting the synergy between synthesis and verification. Possibly another 2-5x speedup can be obtained in these computations using the latest findings in the field. As always, a gain in runtime allows us to perform more iterations of synthesis with larger resource limits, resulting in stronger verification capabilities.

Future work will continue in the following directions:

- Improving core applications, such as AIG rewriting (by partitioning the problem and prioritizing rewriting moves) and technology mapping (by specializing the mapper to an architecture based on a given lookup-table size or a given programmable cell).
- Developing new applications (for example, a fast incremental circuit-based SAT solver or a back-end prover based on an OR-decomposition of the property cone, targetting properties not provable by known methods).
- Building industrial optimization/mapping/verification flows, such as Magic [31], targeting other implementation technologies (for example, the FPGA synthesis flow can be extended to work for standard cells).
- Disseminating the innovative principles of building efficient AIG/SAT/simulation applications and the ways of exploiting the synergy of synthesis and verification.
- Customizing ABC for users in such domains as software synthesis, cryptography, computational biology, etc.

ABC is available for free from Berkeley Verification and Synthesis Research Center [3].

Acknowledgement

This work has been supported in part by SRC contracts 1361.001 and 1444.001, NSF grant CCF-0702668, and the industrial sponsors: Abound Logic, Actel, Altera, Atrenta, Calypto, IBM, Intel, Intrinsity, Magma, Mentor Graphics, Synopsys, Synplicity (Synopsys), Tabula, Verific, and Xilinx.

References

[1] J. Baumgartner and A. Kuehlmann, “Min-area retiming on flexible circuit structures”, *Proc. ICCAD ’01*, pp. 176-182.

[2] J. Baumgartner, H. Mony, V. Paruthi, R. Kanzelman, and G. Janssen, “Scalable sequential equivalence checking across arbitrary design transformations”, *Proc. ICCD ‘06*.

[3] Berkeley Verification and Synthesis Research Center (BVSRC). http://www.bvsrc.org

[4] A. Biere, *AIGER: A format for And-Inverter Graphs*. http://fmv.jku.at/aiger/

[5] P. Bjesse and A. Boralv, "DAG-aware circuit compression for formal verification", *Proc. ICCAD '04*, pp. 42-49.

[6] P. Bjesse and J. H. Kukula, ”Automatic generalized phase abstraction for formal verification”. *Proc. ICCAD ’05*, pp. 1076-1082.

[7] D. Brand, “Verification of large synthesized designs.” *Proc. ICCAD ’93*, pp. 534 -537.

[8] R. K. Brayton, R. Rudell, A. L. Sangiovanni-Vincentelli, and A. R. Wang. “MIS: A multiple-level logic optimization system”. *IEEE Trans. CAD*, 1987, Vol. 6(6), pp. 1062-1081.

[9] R. K. Brayton, G. D. Hachtel, A. Sangiovanni-Vincentelli, F. Somenzi, A. Aziz, S.-T. Cheng, S. Edwards, S. Khatri, Y. Kukimoto, A. Pardo, S. Qadeer, R. K. Ranjan, S. Sarwary, T. R. Shiple, G. Swamy, and T. Villa, “VIS: A system for verification and synthesis”, *Proc. CAV’96*. LNCS-1102.

[10] R. Brayton, “The synergy between logic synthesis and equivalence checking”, *Keynote at FMCAD’07*. http://www.cs.utexas.edu/users/ hunt/FMCAD/2007/presentations/fmcad07_brayton.ppt

[11] R. E. Bryant, “Graph based algorithms for Boolean function manipulation”. *IEEE TC*, Vol. 35(8), 1986, pp. 677-691.

[12] G. Cabodi, P. Camurati, L. Garcia, M. Murciano, S. Nocco, and S. Quer, “Speeding up model checking by exploiting explicit and hidden verification constraints”. *Proc. DATE ‘09*, pp. 1686-1691.

[13] D. Chai, J.-H. Jiang, Y. Jiang, Y. Li, A. Mishchenko, and R. Brayton. “MVSIS 2.0 programmer’s manual”, UC Berkeley, May 2003.

[14] S. Chatterjee, A. Mishchenko, R. Brayton, X. Wang, and T. Kam, "Reducing structural bias in technology mapping", *Proc. ICCAD ’05*, pp. 519-526. http://www.eecs.berkeley.edu/~alanmi/publications/2005/iccad05_map.pdf

[15] O. Coudert, C. Berthet, and J. C. Madre, “Verification of sequential machines based on symbolic execution”, *Proc. Workshop on Automatic Verification Methods for Finite State Systems*, Grenoble, France, 1989.

[16] A. Darringer, W. H. Joyner, Jr., C. L. Berman, L. Trevillyan, "Logic synthesis through local transformations," *IBM J. of Research and Development*, Vol. 25(4), 1981, pp 272-280.

[17] N. Een, A. Mishchenko, and N. Amla, "A single-instance incremental SAT formulation of proof- and counterexample-based abstraction". *Proc. IWLS'10.*

[18] S. Jang, B. Chan, K. Chung, and A. Mishchenko, "WireMap: FGPA technology mapping for improved routability". *Proc. FPGA '08*, pp. 47-55.

[19] S. Jang, K. Chung, A. Mishchenko, and R. Brayton, "A power optimization toolbox for logic synthesis and mapping", *Proc. IWLS '09*, pp. 1-8.

[20] J-H. R. Jiang and R. Brayton, “Retiming and resynthesis: A complexity perspective”, *IEEE Trans. CAD,* Vol. 25(12), Dec 2006, pp. 2674-2686. http://www.eecs.berkeley.edu/~brayton/publications/2006/tcad06_r&r.pdf

[21] J.-H. R. Jiang and W.-L. Hung, “Inductive equivalence checking under retiming and resynthesis”, *Proc. ICCAD’07*, pp. 326-333.

[22] A. Kuehlmann, V. Paruthi, F. Krohm, and M. K. Ganai, “Robust Boolean reasoning for equivalence checking and functional property verification”. IEEE Trans. CAD. Vol. 21(12), 2002, pp. 1377-1394.

[23] C. E. Leiserson and J. B. Saxe. “Retiming synchronous circuitry”, *Algorithmica*, 1991, Vol.6, pp. 5-35.

[24] J. Lamoureux and S.J.E. Wilton, “Activity estimation for Field-Programmable Gate Arrays”, *Proc. Intl Conf. Field-Prog. Logic and Applications (FPL)*, 2006, pp. 87-94.

[25] A. Mishchenko, J. S. Zhang, S. Sinha, J. R. Burch, R. Brayton, and M. Chrzanowska-Jeske, "Using simulation and satisfiability to compute flexibilities in Boolean networks", *IEEE Trans. CAD*, Vol. 25(5), May 2006, pp. 743-755. (Donald O. Pederson Best Paper Award for papers published in IEEE Trans. CAD in 2006-2008.)

[26] A. Mishchenko, S. Chatterjee, and R. Brayton, "DAG-aware AIG rewriting: A fresh look at combinational logic synthesis", *Proc. DAC '06*, pp. 532-536. http://www.eecs.berkeley.edu/~alanmi/publications/ 2006/dac06_rwr.pdf

[27] A. Mishchenko, S. Chatterjee, R. Brayton, and N. Een, "Improvements to combinational equivalence checking", *Proc. ICCAD '06*, pp. 836-843.

[28] A. Mishchenko, S. Cho, S. Chatterjee, and R. Brayton, “Combinational and sequential mapping with priority cuts”, *Proc. ICCAD ’07*, pp. 354-361.

[29] A. Mishchenko, M. L. Case, R. K. Brayton, and S. Jang, "Scalable and scalably-verifiable sequential synthesis", *Proc. ICCAD'08*, pp. 234-241.

[30] A. Mishchenko, R. Brayton, and S. Jang, "Global delay optimization using structural choices", *Proc. FPGA'10*, pp. 181-184.

[31] A. Mishchenko, N. Een, R. K. Brayton, S. Jang, M. Ciesielski, and T. Daniel, "Magic: An industrial-strength logic optimization, technology mapping, and formal verification tool". *Proc. IWLS'10.*

[32] H. Mony, J. Baumgartner, V. Paruthi, and R. Kanzelman. ”Exploiting suspected redundancy without proving it”. *Proc. DAC’05*, pp. 463-466.

[33] H. Mony, J. Baumgartner, A. Mishchenko, and R. Brayton, "Speculative reduction-based scalable redundancy identification", *Proc. DATE'09*, pp. 1674-1679.

[34] S. Ray, A. Mishchenko, R. K. Brayton, S. Jang, and T. Daniel. “Minimum-perturbation retiming for delay optimization”. *Proc. IWLS’10.*

[35] E. M. Sentovich, K. J. Singh, L. Lavagno, C. Moon, R. Murgai, A. Saldanha, H. Savoj, P. R. Stephan, R. K. Brayton, and A. Sangiovanni-vincentelli. “SIS: A system for sequential circuit synthesis.” *Technical Report, UCB/ERI, M92/41*, ERL, Dept. of EECS, UC Berkeley, 1992.

[36] F. Somenzi, BDD package “CUDD v. 2.3.1.” http://vlsi.colorado.edu/~fabio/CUDD/cuddIntro.html

[37] C. Yang, M. Ciesielski, and V. Singhal. “BDS: a BDD-based logic optimization system“, *Proc. DAC’00*, pp. 92-97. (http://www.ecs.umass.edu/ece/labs/vlsicad/bds/bds.html)