# Verified AIG Algorithms in ACL2

Jared Davis Sol Swords
Centaur Technology Inc.
7600-C N. Capital of Texas Hwy, Suite 300
Austin, TX 78731
{jared,sswords}@centtech.com

And-Inverter Graphs (AIGs) are a popular way to represent Boolean functions (like circuits). AIG simplification algorithms can dramatically reduce an AIG, and play an important role in modern hardware verification tools like equivalence checkers. In practice, these tricky algorithms are implemented with optimized C or C++ routines with no guarantee of correctness. Meanwhile, many interactive theorem provers can now employ SAT or SMT solvers to automatically solve finite goals, but no theorem prover makes use of these advanced, AIG-based approaches.

We have developed two ways to represent AIGs within the ACL2 theorem prover. One representation, Hons-AIGs, is especially convenient to use and reason about. The other, Aignet, is the opposite; it is styled after modern AIG packages and allows for efficient algorithms. We have implemented functions for converting between these representations, random vector simulation, conversion to CNF, etc., and developed reasoning strategies for verifying these algorithms.

Aside from these contributions towards verifying AIG algorithms, this work has an immediate, practical benefit for ACL2 users who are using GL to bit-blast finite ACL2 theorems: they can now optionally trust an off-the-shelf SAT solver to carry out the proof, instead of using the built-in BDD package. Looking to the future, it is a first step toward implementing verified AIG simplification algorithms that might further improve GL performance.

## 1 Introduction

Fully automatic tools like SAT and SMT solvers are now commonly used by interactive theorem provers to carry out certain proofs. For instance, in previous work [28] we developed a way to solve finite ACL2 problems using a BDD package or a SAT solver. Similarly, Fox [17] implemented a translation from bit-vector problems in HOL4 into SAT problems which—thanks to integration work by Weber and Amjad [31]—can be solved by zChaff or MiniSat. Going further, Böhme, et al. [4] present a method to solve bit-vector problems from Isabelle/HOL and HOL4 with the Z3 SMT solver.

State of the art SAT and SMT solvers are heavily optimized C or C++ programs that are not intrinsically trustworthy. For instance, Brummayer and Biere [10] report bugs in many SMT solvers that lead to crashes or—much worse—wrong answers! For SAT solvers we have some options for dealing with this:

- We could simply trust the solver. This is pragmatic and allows us to use the fastest tools with minimal overhead, at the risk of getting a wrong answer.
- We could check the solver's work. In the HOL family of theorem provers this checking is done in the usual LCF [18] style, which is quite satisfying but is also expensive. Faster checking is possible for provers that support reflection, e.g., Darbari, et al. [12] have implemented and reflectively verified a checker for SAT proofs in Coq. Some important SAT techniques cannot be checked with these methods, but Wetzler et al. [32] describe promising work to address this.
- We could verify the solver itself. Marić [23] has formally verified many algorithms used in SAT solvers. Oe, et al. [27] have developed and verified a solver in Guru, whose implementation

R. Gamboa and J. Davis (Eds.): ACL2 Workshop 2013 (ACL2 '13).
EPTCS 114, 2013, pp. 95–110, doi:10.4204/EPTCS.114.8

involves efficient, low-level structures like pointers, mutable arrays, and machine arithmetic. These efforts are inspiring, although performance is still not competitive with modern solvers.

For SMT solvers there are similar approaches. For instance, the HOL connection to Z3's bit-vector reasoning checks the solver's work in the LCF style, and Armand et al. [1] have developed a reflectively verified Coq checker for certain SMT theories.

This is a good start, but many hardware verification techniques go well beyond SAT. For instance, in Brayton and Mishchenko's [8] ABC system, circuits and specifications are typically represented as And-Inverter Graphs (AIGs). Algorithms like AIG rewriting [25] can significantly reduce the size of these graphs by analyzing their structure. Algorithms like fraiging [26] can simplify AIGs by using a combination of random simulation and SAT to find nodes that are equivalent and can be merged. These, and many other AIG algorithms, can greatly improve equivalence checking.

Much like SMT solvers, these algorithms are implemented as optimized, tricky C++ routines that might easily have errors. We are not aware of any work to formally verify these tools. Although ways have been proposed [11] to emit proofs from some of these algorithms, existing tools do not generate such proofs. This leaves us with the choice of either (1) pragmatically using these tools, while accepting that they may not be correct, or (2) conservatively rejecting these tools as too risky, forgoing their benefits.

This paper presents some results toward verifying AIG algorithms and incorporating these algorithms into ACL2. We have actually developed two complementary AIG representations:

- A Hons-based representation that is especially simple and easy to reason about. The "graph" part of the And-Inverter Graph is kept outside of the logic by using the hash-consing features of ACL2(h), which avoids the sort of invariant-preservation theorems that you normally need when dealing with imperative structures. During proofs, we abstract away the details of the Hons-AIG representation and instead focus on semantic equivalence, a relation that is amenable to Greve-like [19] quantifier automation. We have implemented and verified algorithms on Hons-AIGs such as random vector simulation and conversion to BDDs. (Section 2)

- A stobj [6] based representation that is styled after ABC's and aimed squarely at efficiency. Here, the graph is an explicit stobj array. This is much harder to reason about: the graph is destructively updated as nodes are added, so we need invariant- and semantics-preservation theorems. To deal with this, we develop an interesting *extension* relationship between AIG networks, and a `bind-free`-based strategy for using this relation. We have implemented and verified algorithms like random vector simulation, and also an efficient, somewhat "smart" conversion to Conjunctive Normal Form (CNF), which allows us to export AIGs to SAT. (Section 3)

Besides these contributions to reasoning about AIG algorithms, we have integrated our stobj based representation and its CNF conversion algorithm into the GL [28] framework for bit-blasting finite ACL2 theorems. This allows GL to solve more theorems automatically, and opens up interesting possibilities for further scaling. (Section 4).

## 2 A Hons-Based AIGs Representation

ACL2(h) is an extension of ACL2 that was originally developed by Boyer and Hunt [5]. It extends ACL2 with hash-consing and memoization features that make it quite easy to develop a Hons-based AIG library that is flexible, quite easy to prove theorems about, and reasonably efficient.

## 2.1 Representation and Semantics

(Note: our Hons-AIG representation and semantics are briefly described in previous work by Swords and Hunt [29], we repeat some details here to make this paper more self-contained.)

In the Hons-AIG library, an AIG is an object that represents a single Boolean function as a tree of AND and NOT nodes with constants and input variables as the leaves. We use `t` and `nil` to represent the constants true and false, and treat any other atom as a variable. We encode NOT nodes as conses of the form `(`$a$ `. nil)`, and AND nodes with any other cons, `(`$a$ `. `$b$`)`. This special-casing of `nil` may seem awkward, but it lets us represent either kind of node with just one cons and, at any rate, it wouldn't be useful to AND together `nil` with another AIG anyway.

The semantics of a Hons-AIG are given by an evaluation function, AIGEVAL, that uses an *environment* to give values to the variables:

$$\textsc{AigEval}(x, env) \triangleq \begin{cases} \texttt{nil} & \text{when } x = \texttt{nil}, \\ \texttt{t} & \text{when } x = \texttt{t}, \\ \textsc{AigEnvLookup}(x, env) & \text{when } x \text{ is an atom}, \\ \neg\textsc{AigEval}(a, env) & \text{when } x = (a \texttt{ . nil}), \text{ or} \\ \textsc{AigEval}(a, env) \wedge \textsc{AigEval}(b, env) & \text{where } x = (a \texttt{ . } b). \end{cases}$$

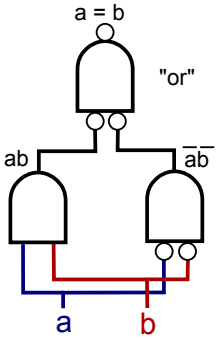
Is it fair to call these AIGs? Implicit in the very name "And-Inverter *Graph*" is the idea of using a directed, acyclic graph (DAG) to represent a collection of Boolean functions. Where's the DAG here? We construct Hons-AIGs using HONS, the hash-consing constructor from ACL2(h). Logically, HONS is nothing more than CONS. But in the execution, $\textsc{Hons}(a, b)$ checks whether there is already a hons pair corresponding to `(`$a$ `. `$b$`)` and, if so, returns the existing pair instead of making a new one. The bookkeeping that keeps track of honses is done in ACL2(h) via Common Lisp hash tables. These tables are extended when we build new Hons-AIGs, and, in some sense, contain the DAG for all Hons-AIGs.

We usually build Hons-AIGs with constructors named AIGAND, AIGOR, etc., that apply basic simplifications like constant folding and reduce expressions like $a \wedge a$ and $a \wedge \neg a$. Identifying expressions like $a \wedge a$ would be very expensive if we were to use a deep structural-equality check, but ACL2(h) has a HONS-EQUAL function that boils down to pointer-equality for honses. Correctness theorems for these constructors are trivial and are stated in terms of AIGEVAL, e.g., for AIGAND we prove

$$\textsc{AigEval}(\textsc{AigAnd}(a, b), env) = \textsc{AigEval}(a, env) \wedge \textsc{AigEval}(b, env).$$

## 2.2 AIG Traversal and Memoization

Almost any algorithm that operates on Hons-AIGs needs a way to avoid repeatedly visiting the same node. For instance, to the right is an AIG that represents $a = b$. Notice in particular how $a$ and $b$ are reused as *fanins*—inputs to AND gates—shared by both $ab$ and $\overline{ab}$. Because of this sharing, if we evaluate this AIG with AIGEVAL as above, we will end up evaluating $a$ **twice**: once because it is a fanin of $ab$, and once because it is a fanin of $\overline{ab}$. We will evaluate $b$ twice for the same reason. As we generate larger AIGs with lots of shared structure, the cost of this recomputation grows exponentially.




The ACL2(h) system has a function memoization capability that we can use to avoid this recomputation. This mechanism is especially convenient: we simply

say, "memoize AIGEVAL," and we are done. The logical definition of AIGEVAL is unchanged, but its executable definition is extended with a Common Lisp hash table that records its return values: when AIGEVAL reaches a node whose value has already been computed, this cached return value is returned, making the computation linear in time and space in the number of unique AIG nodes.

Automatic memoization works well for AIGEVAL, but is not suitable for some other AIG algorithms. Notice that as AIGEVAL recurs, its *env* argument remains fixed. In many other algorithms, there are additional parameters that change as we recur over the AIG. These changes ruin simple memoization, because, e.g., when we encounter $a$ for the second time, these arguments may now be different. In these cases, we generally represent the memo table explicitly as a "fast" association list. This is roughly as efficient as automatic memoization; fast alists provide hash-table speeds for honsed keys. Unfortunately, it is far less convenient. When we verify these kind of algorithms, we typically need invariants that say the memo tables have correct entries.

## 2.3 Reasoning about Hons-AIGs

Hons-AIGs are a *non-canonical* representation, meaning that there are different ways to represent the same Boolean function. For instance, we could represent the function $\mathtt{a} \wedge \mathtt{b}$ as `(a . b)`, or as `(b . a)`, or `(a . (b . t))`, and so on. Because of this, when we reason about algorithms that manipulate AIGs, we typically do not want to deal with structural equality, but rather semantic equivalence,

$$\text{AIGEQUIV}(a,b) \triangleq \forall \mathit{env} : \text{AIGEVAL}(a,\mathit{env}) = \text{AIGEVAL}(b,\mathit{env}).$$

This equivalence relation has a strong analogue in set theory. Instead of thinking about an AIG as some particular cons tree, think about it as a set whose elements are the environments that satisfy it. That is, for the AIG $a$, think of the set $\{\mathit{env} : \text{AIGEVAL}(a,\mathit{env}) = \mathtt{t}\}$. Seen this way, $\text{AIGEQUIV}(a,b)$ is nothing more than set equality; AIGAND is set intersection, AIGOR is union, etc.

To reason about AIGEQUIV in ACL2, we use the *witness*[1] clause processor, which is much like Greve's `quant` system [19] for automatically Skolemizing and instantiating quantified formulas. When ACL2 tries to prove a conclusion of the form $\text{AIGEQUIV}(a,b)$, this clause processor reduces the problem to showing that $a$ and $b$ have the same evaluation under a particular *env*, similar to the pick-a-point strategy used in Davis' osets library [13]. Going further, when we have a hypothesis of the form $\text{AIGEQUIV}(a,b)$, the clause processor will add a witnessing environments *env* for which $a$ and $b$ are known to produce the same evaluation.

We have implemented and verified several algorithms on Hons-AIGs. As basic building blocks, we have routines for collecting the variables in an AIG and composing AIGs with other AIGs. We have also developed routines for random vector simulation; this is like AIGEVAL, except that we do $n$ simulations at a time using $n$-bit integers by replacing Boolean AND and NOT with bitwise AND and NOT. We also developed a basic way to partition AIGs into equivalence classes by first using random vector simulation as a coarse filter, then use SAT to differentiate the AIGs that were the same across random simulations. The most sophisticated algorithm that has been verified atop Hons-AIGs is the AIG to BDD conversion algorithm of Swords and Hunt [29]. This algorithm avoids converting irrelevant parts of the AIG, which is important since constructing BDDs is often prohibitively expensive.

[1] See `clause-processors/witness-cp` in the ACL2 Books.

### 2.4 Critique of Hons-AIGs

Hons-AIGs have a lot of redeeming qualities. By hiding the details of their DAG representation beneath ACL2(h)'s implementation of hash consing, we can think of AIG "nodes" as if they are ordinary ACL2 objects. An important consequence is that we can embed Hons-AIGs within other ACL2 objects, e.g., we often deal with lists or alists of Hons-AIGs. Hiding the graph from the logic means we don't need to prove any invariant-preservation theorems, and even the memoization needed to avoid repeating computations can often be hidden from the logic, thanks to ACL2(h)'s memoize command.

Unfortunately, Hons-AIGs are not especially efficient. Conventional AIG representations use topologically ordered arrays of nodes. By comparison, the nodes of Hons-AIGs, being conses, have worse memory locality and must be traversed during garbage collection, reducing overall performance.

Algorithms that operate over Hons-AIGs have to do a lot of hashing. Even for something as simple as AIGEVAL, for instance, we look up each AND node in a memoization table. This means hashing on the address of a cons regardless of whether we use ACL2(h) memoization or fast-alists for the table. In contrast, in an array-based representation, each node has a successive index and so, e.g., AIGEVAL could simply allocate an array and use array-indexing, instead of hashing, to do the memoization. In packages like ABC, some node representations include scratch space that can be used for such purposes, giving even better memory locality. Our memoization schemes also require allocation, hash-table growth, etc., and ultimately impact garbage collection performance.

Hons-AIGs also have one node for every AND and NOT operation, whereas most AIG representations avoid NOT nodes by encoding negation information in the fanins of AND nodes. For rough perspective, we measured the Hons-AIG for a GL proof that shows our media unit correctly implements the x86 PSADBW instruction. We found 53.3% AND nodes and 46.7% NOT nodes. In short, the Hons-AIG representation uses about twice as many nodes as a typical representation uses. This is unfortunate for memory locality and makes AIG construction more expensive since we have to hash to create NOT nodes.

## 3 Aignet Library

In contrast to the Hons-AIG library, our Aignet library is designed for better execution efficiency at the expense of logical simplicity. Its implementation is similar to those in high-performance AIG packages such as ABC [8].

### 3.1 Representation

In Aignet, an AIG is represented in a single-threaded object (stobj) [6]. The Aignet stobj may be thought of as a medium for storing a network of inter-related (combinational) Boolean formulas. Unlike Hons-AIGs, it also directly supports viewing these formulas as a (sequential) finite state machine.

The primary content of the Aignet stobj is an array of nodes. Each node has an *ID* which is its index in the array. We usually refer to stored formulas via a *literal*, which is a combination of an ID with a bit saying whether it is negated or not. A literal is represented as a natural number using the construction

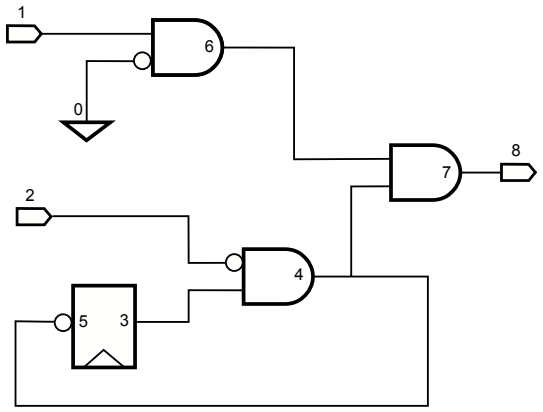
$$\textsc{MkLit}(id, neg) \triangleq 2\ id + neg.$$

Extracting the parts of a literal can be implemented efficiently; in C-like notation,

$$\textsc{LitId}(lit) \triangleq lit >> 1$$
$$\textsc{LitNeg}(lit) \triangleq lit\ \&\ 1.$$

Every Aignet has a constant-false node with ID 0. Hence, the literal 0 means constant-false, and the literal 1 means constant-true.

Here is an example circuit and its node array representation:




| ID | Node Type | Fanins | Other |
|---|---|---|---|
| 0 | Constant False | | |
| 1 | Primary Input | | |
| 2 | Primary Input | | |
| 3 | Register Output | | |
| 4 | AND gate | 5, 6 | |
| 5 | Register Input | 9 | RO: 3 |
| 6 | AND gate | 1, 2 | |
| 7 | AND gate | 8, 12 | |
| 8 | Primary Output | 14 | |

Each node in the array has a type. This circuit contains nodes of all six types: constant false, primary input (PI), primary output (PO), AND gate, register input (RI), and register output (RO). Unlike Hons-AIGs, there are no NOT gates in the node array. Instead, the fanins of each node are encoded as literals, which may or may not be negated. Negations are shown as bubbles in the picture. For instance: the fanin of node 5 is the negation of node 4, so its fanin literal is $2 \cdot 4 + 1 = 9$; the fanin of node 8 is node 7, without negation, so its fanin literal is 14.

Register input and output nodes allow an Aignet to represent a sequential circuit or finite state machine. However, in many algorithms, the circuit is viewed as purely combinational. In these cases, register outputs are treated similarly to primary inputs, both being essentially free variables; register inputs are treated like to primary outputs, both being essentially a way to label certain formulas. Thus, we sometimes refer to primary inputs and register outputs collectively as *combinational inputs* (CIs), and primary outputs and register inputs collectively as *combinational outputs* (COs).

Primary inputs are easy to understand: they are like the variables in a Hons-AIG. But what are primary outputs? Any AIG simplification tends to affect the numbering of nodes: nodes may be reordered, eliminated, etc. But by convention, we preserve the ordering among primary outputs. The sole purpose of primary outputs is to maintain an order, so that before and after a transformation, the *n*th primary output always refers to the same formula. For similar reasons, we also generally expect that the order of primary inputs and (for combinational simplification, at least) register outputs will remain the same.

The Aignet stobj supports this convention by maintaining arrays that allow us to look up the ID of, e.g., the *n*th primary input, and also to do reverse lookups. These extra mappings are not material to the logical meaning of the stobj, which is completely expressed by the node table; e.g., looking up the *n*th input in the input array is always the same as scanning the node table for the *n*th PI node.

## 3.2 Semantics

Since the Aignet stobj can be viewed as both a medium for storing (combinational) Boolean formulas and (sequential) finite state machines, we have both combinational and sequential evaluation semantics; we describe combinational evaluation first, since sequential evaluation builds on it.

### 3.2.1 Combinational Semantics

An Aignet node or literal is evaluated under an assignment of bits to the combinational inputs. This assignment is given by a pair of arrays $V_{in}, V_{reg}$, where, e.g., $V_{in}[n]$ is the value to use for the $n$th primary input, and $V_{reg}[n]$ is the value for the $n$th register output. Evaluation is defined as follows, except that we hide the stobj argument for simpler presentation:

$$\textsc{EvalLit}(lit, V_{in}, V_{reg}) \triangleq \textsc{EvalID}(\textsc{LitId}(lit), V_{in}, V_{reg}) \ \texttt{xor} \ \textsc{LitNeg}(lit)$$

$$\textsc{EvalID}(id, V_{in}, V_{reg}) \triangleq \begin{cases} 0 & \text{if } \textsc{Type}(id) = \textsc{Const} \\ V_{in}[\textsc{InIdx}(id)] & \text{if } \textsc{Type}(id) = \text{PI} \\ V_{reg}[\textsc{RegIdx}(id)] & \text{if } \textsc{Type}(id) = \text{RO} \\ \textsc{EvalLit}(\textsc{Fanin1}(id), V_{in}, V_{reg}) & \\ \quad \& \ \textsc{EvalLit}(\textsc{Fanin2}(id), V_{in}, V_{reg}) & \text{if } \textsc{Type}(id) = \textsc{And} \\ \textsc{EvalLit}(\textsc{Fanin}(id), V_{in}, V_{reg}) & \text{if } \textsc{Type}(id) \in \{\text{PO}, \text{RI}\} \end{cases}$$

This evaluation function is nice for reasoning but not practical for execution since it does no memoization. A more practical evaluation function simply does a linear sweep over the node array, storing each node's value in a result array. We require the node array to be topologically ordered, so that each fanin's value is already computed. This makes evaluation linear in the size of the graph. (Topological ordering is also important to show that EVALLIT terminates!)

Evaluation is a good example of the performance advantage of the Aignet representation over the Hons-AIG representation. We benchmarked evaluating the functions for a hardware module with 3,664 input variables, 131,605 AND nodes, and (in the Hons-AIG representation) 121,490 NOT nodes. We also include a comparison with ABC, compiled with and without GCC's "-O" optimization flag. In each case, we evaluated the network under the all-false environment 1000 times. For the Hons-AIG test, we cleared the AIGEVAL memoization hash table at each iteration, and used a large enough Lisp heap to avoid garbage collection, so that GC time was not counted. For the Aignet and ABC tests, we used a bit array to remember node values, re-allocating and clearing this array at each iteration.

| Representation | Time | Allocation |
|---|---|---|
| Hons-AIG | 57.6 sec | 2.62 GB |
| Aignet | 6.85 sec | 16.2 MB |
| ABC (no opt.) | 4.03 sec | 16.0 MB |
| ABC (opt.) | 1.21 sec | 16.0 MB |

### 3.2.2 Sequential Semantics

Sequential evaluation depends on an initial assignment to the registers and a series of assignments to the primary inputs. A common convention is to assume that each register's initial value is 0, but the Aignet library does not enforce this convention. Letting $V_{ini}$ be the initial state values (indexed by register number) and $V_{fr}$ be a two-dimensional array giving the input assignments at each frame (indexed first by

frame, then by input number), the evaluation of a node or literal at frame $k$ is given by:

$$\textsc{SeqEvLit}(k, lit, V_{fr}, V_{ini}) \triangleq \textsc{SeqEvID}(k, \textsc{LitID}(lit), V_{fr}, V_{ini}) \texttt{ xor } \textsc{LitNeg}(lit)$$

$$\textsc{SeqEvID}(k, id, V_{fr}, V_{ini}) \triangleq \begin{cases} 0 & \text{if } \textsc{Type}(id) = \textsc{Const} \\ V_{fr}[k][\textsc{InIdx}(id)] & \text{if } \textsc{Type}(id) = \text{PI} \\ V_{ini}[\textsc{RegIdx}(id)] & \text{if } \textsc{Type}(id) = \text{RO and } k = 0 \\ \textsc{SeqEvID}(k-1, \textsc{RegIn}(id), V_{fr}, V_{ini}) & \text{if } \textsc{Type}(id) = \text{RO and } k > 0 \\ \textsc{SeqEvLit}(k, \textsc{Fanin1}(id), V_{fr}, V_{ini}) & \\ \quad \& \ \textsc{SeqEvLit}(k, \textsc{Fanin2}(id), V_{fr}, V_{ini}) & \text{if } \textsc{Type}(id) = \textsc{And} \\ \textsc{SeqEvLit}(k, \textsc{Fanin}(id), V_{fr}, V_{ini}) & \text{if } \textsc{Type}(id) \in \{\text{PO}, \text{RI}\}. \end{cases}$$

Alternatively, we can define SEQEVID in terms of the combinational semantics EVALID as follows:

$$\textsc{SeqEvID}(k, id, V_{fr}, V_{ini}) \triangleq \textsc{EvalID}\left(id, V_{fr}[k], \textsc{RegFrame}\left(k, V_{fr}, V_{ini}\right)\right)$$

$$\textsc{RegFrame}(k, V_{fr}, V_{ini})[n] \triangleq \begin{cases} V_{ini}[n] & \text{if } k = 0 \\ \textsc{SeqEvID}\left(k-1, \textsc{RegIn}\left(\textsc{NthReg}\left(n\right)\right), V_{fr}, V_{ini}\right) & \text{if } k > 0. \end{cases}$$

Here, we capture the values on the register outputs at each frame $k$ as a bit array, $\textsc{RegFrame}(k, V_{fr}, V_{ini})$. This takes the initial values at frame 0, and for later frames collects the register input values computed for the previous frame. Then the sequential evaluation of any node in frame $k$ reduces to a combinational evaluation of the inputs and register values for that frame.

Either definition is well suited for reasoning but performs poorly on real examples since it does no memoization. An efficient implementation that linearly computes each frame in sequence by storing the current values in a bit array can be proven to compute these values.

### 3.3 Constructing Aignets

The following basic functions are used to add nodes to an Aignet stobj. The stobj is both an input and output of all functions below, but is omitted for brevity:

- ADDINPUT() $\rightarrow$ *lit*: Adds a new primary input node and returns its non-negated literal.
- ADDREG() $\rightarrow$ *lit*: Adds a new register output node and returns its non-negated literal.
- ADDAND(*lit1*, *lit2*) $\rightarrow$ *lit*: Adds a new AND gate of *lit1*, *lit2*; returns its non-negated literal.
- ADDOUTPUT(*lit*): Adds a new primary output with fanin *lit*.
- ADDREGIN(*lit*, *ro*): Adds a new register input node with fanin *lit* and register output ID *ro*.

ADDAND is a very low-level way to add a gate to an Aignet stobj. It unconditionally adds an AND node with no simplification and with no regard to whether such a node already exists. Higher-level functions called HASHAND, HASHOR, HASHXOR, and HASHMUX can add logic in a smarter way:

- They may optionally propagate constants, eliminate certain tautologies and contradictions, detect cases where an AND is logically equivalent to one of its fanins, etc., to produce equivalent formulas with fewer nodes. These reductions are described in Brummayer and Biere [9].

- They allow *structural hashing* (strashing): when preparing to add an AND node, we check to see if there is an existing AND node with the same fanins, and if so reuse it. Our structural hashing implementation currently requires a trust tag to allow stobjs to have hash table fields; we believe it is sound, and perhaps it can be integrated into ACL2 in the future.

In Hons-AIGs, structural hashing is automatic due to our use of HONS, but in our stobj-based approach the strash table is necessarily visible in the logic. Normally we would need an invariant about the strash table to know that our construction functions return nodes with the right meanings; this would mean proving invariant-preservation theorems for many algorithms. We initially tried to do this, but found it to be tedious. We then realized that it is reasonably fast[2] to check whether a hit in the strash table actually gives the correct node. Adding this runtime check to our strash lookup obviates the need for an invariant altogether! Explicit hashing also has some advantages: once we are done building a network we may throw out the strash table and reclaim its memory while keeping the AIG itself.

Each of these functions returns a literal that provably evaluates to the correct function of the evaluations of the input literals, and also provably preserves the meaning of previously existing literals. For example, in the case of HASHAND, letting

$$(\mathit{newlit}, \mathit{strash}', \mathit{aignet}') = \textsc{HashAnd}(\mathit{lit}_1, \mathit{lit}_2, \mathit{strash}, \mathit{simp\_opts}, \mathit{aignet}),$$

the new-literal correctness theorem is

$$\begin{aligned} \textsc{EvalLit}(\mathit{newlit}, V_{in}, V_{reg}, \mathit{aignet}') \quad = \quad & \textsc{EvalLit}(\mathit{lit}_1, V_{in}, V_{reg}, \mathit{aignet}) \wedge \\ & \textsc{EvalLit}(\mathit{lit}_2, V_{in}, V_{reg}, \mathit{aignet}), \end{aligned}$$

and the previous-literal preservation theorem is

$$\begin{aligned} \textsc{LitId}(\mathit{oldlit}) < \textsc{NumNodes}(\mathit{aignet}) \quad \Rightarrow \quad & \textsc{EvalLit}(\mathit{oldlit}, V_{in}, V_{reg}, \mathit{aignet}') = \\ & \textsc{EvalLit}(\mathit{oldlit}, V_{in}, V_{reg}, \mathit{aignet}). \end{aligned}$$

This preservation property is an instance of a more pervasive pattern: most functions which modify an Aignet stobj are simply adding new nodes, without modifying the existing ones. This means that many properties of existing nodes in the original network remain true in the modified network. To take advantage of this, we use a relation

$$\begin{aligned} & \textsc{AignetExtension}(\mathit{new}, \mathit{old}) \triangleq \\ & \qquad \textsc{NumNodes}(\mathit{old}) \leq \textsc{NumNodes}(\mathit{new}) \wedge \\ & \qquad \forall\, \mathit{id} < \textsc{NumNodes}(\mathit{old}) : \textsc{Nodes}(\mathit{new})[\mathit{id}] = \textsc{Nodes}(\mathit{old})[\mathit{id}], \end{aligned}$$

that is, all nodes that were in the old network are still the same in the new network.

Most functions that modify an Aignet stobj produce an output Aignet which is an extension of the input Aignet, and there are many useful preservation properties that follow when we know this is the case. For example, when we know $\textsc{AignetExtension}(\mathit{new}, \mathit{old})$, it follows that:

- an ID that is within bounds for *old* is still in-bounds for *new*
- a list of in-bounds literals for *old* is still in-bounds for *new*

[2] This runtime check costs an overhead of 30–40% in the worst case, i.e., when every strash lookup is a hit and no simplification is performed. We generally expect to see fewer strash hits and to perform simplification, which has much greater overhead. And, at any rate, Aignet construction is typically not a bottleneck.

- evaluation of an in-bounds literal in *old* is the same as its evaluation in *new*
- the *N*th output in *old* still has the same ID in *new*.

Written as rewrite rules, these preservation properties generally bind *new* in the LHS, but have *old* as a free variable. We use a `bind-free` [20] form to instruct ACL2 how to bind *old*. In particular, if *new* is bound to a stobj return value of a function call, we will try binding *old* to the corresponding stobj input of that function call. Otherwise, we fall back to searching the type-alist for a known-true term of the form AIGNETEXTENSION(*new*, *old*). This strategy works well in most cases, but in some cases it would be better to try multiple different bindings for *old*, in case the first one fails to satisfy one of the other hypotheses. This capability is not yet available to `bind-free`.

In the Aignet library sources, we have identified about 40 such preservation properties. Furthermore, we have proven that about 25 Aignet-modifying functions return an Aignet that is an extension of their input. If we tried to do without the AIGNETEXTENSION property and instead prove each of the 40 preservation properties for all 25 functions then we would need 1000 rules, whereas we now accomplish the same thing with only 65 rules.

### 3.4 CNF Generation

We have implemented and proven correct a CNF generation algorithm for Aignet stobjs. Today, this algorithm allows us to export AIG-derived problems to an external SAT solver. For the future, it is also groundwork toward implementing and verifying many AIG algorithms that are based on *incremental SAT* [16], a technique for efficiently checking multiple related satisfiability queries. For instance, modern model- and equivalence-checking packages make use of:

- *Fraiging* or *SAT sweeping* [26] is a combinational simplification algorithm designed to eliminate circuit nodes that are redundant; that is, any nodes whose functionality is provably the same as another. It finds candidate combinational equivalences among circuit nodes using random simulation, then attempts to prove or disprove each equivalence using SAT.
- *Signal correlation* [24] is a sequential simplification algorithm that somewhat resembles fraiging, but allows elimination of nodes that are sequentially equivalent, even if they are not combinationally equivalent. It finds candidate sequential equivalences using random simulation, then uses SAT to attempt to prove all equivalences by induction over one or more frames.
- *IC3* or *property directed reachability* [7, 14] is an algorithm for checking safety properties. It operates by repeatedly using SAT to refine an overapproximation of the reachable state space until it is shown that no bad states are reachable or a counterexample is found.

An incremental SAT solver maintains a database of clauses, of which some are given and some are learned. Typically the CNF database as a whole—the conjunction of all the clauses—is known to be satisfiable. The problems to solve are given by providing a *cube* (a conjunction of literals) as an additional constraint. The solver checks the satisfiability of the cube together with the clause database, but only learns clauses relative to the database, not assuming the cube. The point is: subsequent SAT checks may reuse heuristic information and learned clauses even though the constraints change.

An incremental SAT interface is useful for solving problems generated by AIG-based algorithms. The idea here is to generate the clause database from the circuit by adding structural constraints as in the Tseitin [30] transform. For instance, if $c$ is an AND node with fanins $a$ and $b$, we add the clauses for $\neg a \Rightarrow \neg c$, $\neg b \Rightarrow \neg c$, and $(a \wedge b) \Rightarrow c$. These clauses fully and exactly represent the logical meaning of

the AND node: any possible configuration of the AND gate and its fanins satisfies the clauses, and any satisfying assignment of the clauses is a possible configuration of the AND gate.

When we generate a clause database in this way, any evaluation of the AIG induces a satisfying assignment of the database by assigning each literal its value under the evaluation. This allows us to prove facts about the network by checking satisfiability of the database together with some additional constraints as follows. Suppose we generate a clause database from part of a circuit. Suppose we prove that the database together with a certain constraint—say, some literal—is unsatisfiable. This proves that this literal may not be true under any evaluation of the circuit. Why? Suppose we have an assignment to the combinational inputs of the network that makes this literal true. This produces a satisfying assignment of the database, and the literal is true, so the constraint is satisfied, which we have proved impossible. This generalizes to any additional constraint not in the database, most commonly a cube of literals.

Conversely, if the database contains sufficient structural constraints relating some set of nodes to their fanins, and also transitively for their fanins reaching back to the combinational inputs, then any satisfying assignment of the database induces an assignment to the CIs of the circuit under which each node in the set evaluates to its ID's value under the satisfying assignment. This allows us to transform a satisfying assignment produced by a SAT solver into an assignment to the combinational inputs that satisfies whatever constraint we added to the clause database.

Our CNF generation algorithm is a commonly-implemented [15] variation on the standard Tseitin transform. This variation has two optimizations that reduce the number of variables and clauses that will be given to the solver, which tends to speed up SAT checks. First, it recognizes *supergates*: trees of multiple AND gates, without negations, where only the root has multiple fanouts. For each supergate it generates $n$ binary clauses and one clause of length $n+1$:

$$o = i_0 \wedge i_1 \wedge \ldots \wedge i_n \quad \longrightarrow \quad \begin{array}{rcl} \neg i_0 & \Rightarrow & \neg o \\ \ldots & & \\ \neg i_n & \Rightarrow & \neg o \\ i_0 \wedge \ldots \wedge i_n & \Rightarrow & o \end{array}$$

Second, it recognizes two-level nestings of gates representing a mux (if-then-else) structure, and generates a special set of clauses:

$$o = \mathsf{if}\ a\ \mathsf{then}\ b\ \mathsf{else}\ c \quad \longrightarrow \quad \begin{array}{rcl} a \wedge b & \Rightarrow & o \\ \neg a \wedge c & \Rightarrow & o \\ a \wedge \neg b & \Rightarrow & \neg o \\ \neg a \wedge \neg c & \Rightarrow & \neg o \\ b \wedge c & \Rightarrow & o \\ \neg b \wedge \neg c & \Rightarrow & \neg o \end{array}$$

The last two clauses are unnecessary but are included to improve SAT performance [15]; they are omitted in the degenerate case where $b = \neg c$, i.e. an XOR gate.

The CNF generation function has the following signature:

$$\textsc{AddCnf}(id, marks, cnf, aignet) \rightarrow (marks', cnf').$$

Here, *cnf* is the accumulator for clauses, and *marks* is a bit array that tracks the identifiers whose structural constraints (recursively, back to the CIs) are encoded in the CNF.

We prove that AddCnf marks *id* and preserves an invariant, $\textsc{CnfForAignet}(cnf, marks, aignet)$. When this invariant holds, we can map evaluations of *aignet* to satisfying assignments of *cnf* and vice versa, as described above. To state the invariant we need some additional definitions:

- CNFEVAL$(cnf, env)$ evaluates a CNF formula under a bit array *env* mapping IDs to values.
- AIGNETEVAL$(aignet, env)$ updates *env* so that the values assigned to CI IDs are preserved, but for all other IDs, the values under those CI assignments are stored.
- MARKEDVALSCORRECT$(aignet, marks, env)$ is true when *env* binds every marked ID to its evaluation under the CI assignments, i.e., when

$$\forall id : marks[id] \Rightarrow \text{AIGNETEVAL}(aignet, env)[id] = env[id].$$

We can now define our invariant:

$$\begin{aligned}&\text{CNFFORAIGNET}(cnf, marks, aignet) \triangleq \\ &\quad \forall env : (\text{CNFEVAL}(cnf, env) \Rightarrow \text{MARKEDVALSCORRECT}(aignet, marks, env)) \\ &\qquad \wedge \text{CNFEVAL}(cnf, \text{AIGNETEVAL}(aignet, env)).\end{aligned}$$

The two conjuncts are the two directions. The first says, for any satisfying assignment of the CNF, how to obtain a consistent evaluation of the circuit. The second says, given an evaluation of the circuit, how to obtain a satisfying assignment of the CNF.

When *cnf* is empty and no nodes are marked, CNFFORAIGNET is trivially true. This is the starting point for adding nodes to *cnf*. Then, we prove our invariant is preserved by ADDCNF, i.e.,

$$\begin{aligned}&\mathsf{let}\ (marks', cnf') = \text{ADDCNF}(id, marks, cnf, aignet)\ \mathsf{in} \\ &\quad \text{CNFFORAIGNET}(cnf, marks, aignet) \Rightarrow \text{CNFFORAIGNET}(cnf', marks', aignet).\end{aligned}$$

When we know that CNFFORAIGNET holds, then we know that satisfiability proofs about the CNF (with additional constraints) imply satisfiability results about the circuit. For example, we define a function AIGNETCISTOCNFENV$(V_{in}, V_{reg}, aignet)$ and prove

$$\begin{aligned}&\text{CNFFORAIGNET}(cnf, marks, aignet) \\ &\wedge \text{LITEVAL}(lit, V_{in}, V_{reg}, aignet) \\ &\wedge marks[\text{LITID}(lit)] \\ \Rightarrow\ &\text{CNFEVAL}\left(cnf \cup \{lit\}, \text{AIGNETCISTOCNFENV}\left(V_{in}, V_{reg}, aignet\right)\right).\end{aligned}$$

Therefore if $cnf \cup \{lit\}$ is unsatisfiable, then there is no $V_{in}, V_{reg}$ for which LITEVAL$(lit, V_{in}, V_{reg}, aignet)$. Conversely, we have CNFENVTOAIGNETCIS$(env, aignet)$ for which we prove:

$$\begin{aligned}&\text{CNFFORAIGNET}(cnf, marks, aignet) \\ &\wedge \text{CNFEVAL}\left(cnf \cup \{lit\}, env\right) \\ &\wedge marks[\text{LITID}(lit)] \\ \Rightarrow\ &\mathsf{let}\ (V_{in}, V_{reg}) = \text{CNFENVTOAIGNETCIS}(env, aignet)\ \mathsf{in} \\ &\quad \text{LITEVAL}(lit, V_{in}, V_{reg}, aignet).\end{aligned}$$

That is, a satisfying assignment for $cnf \cup \{lit\}$ can be converted to a CI assignment under which *lit* is satisfied.

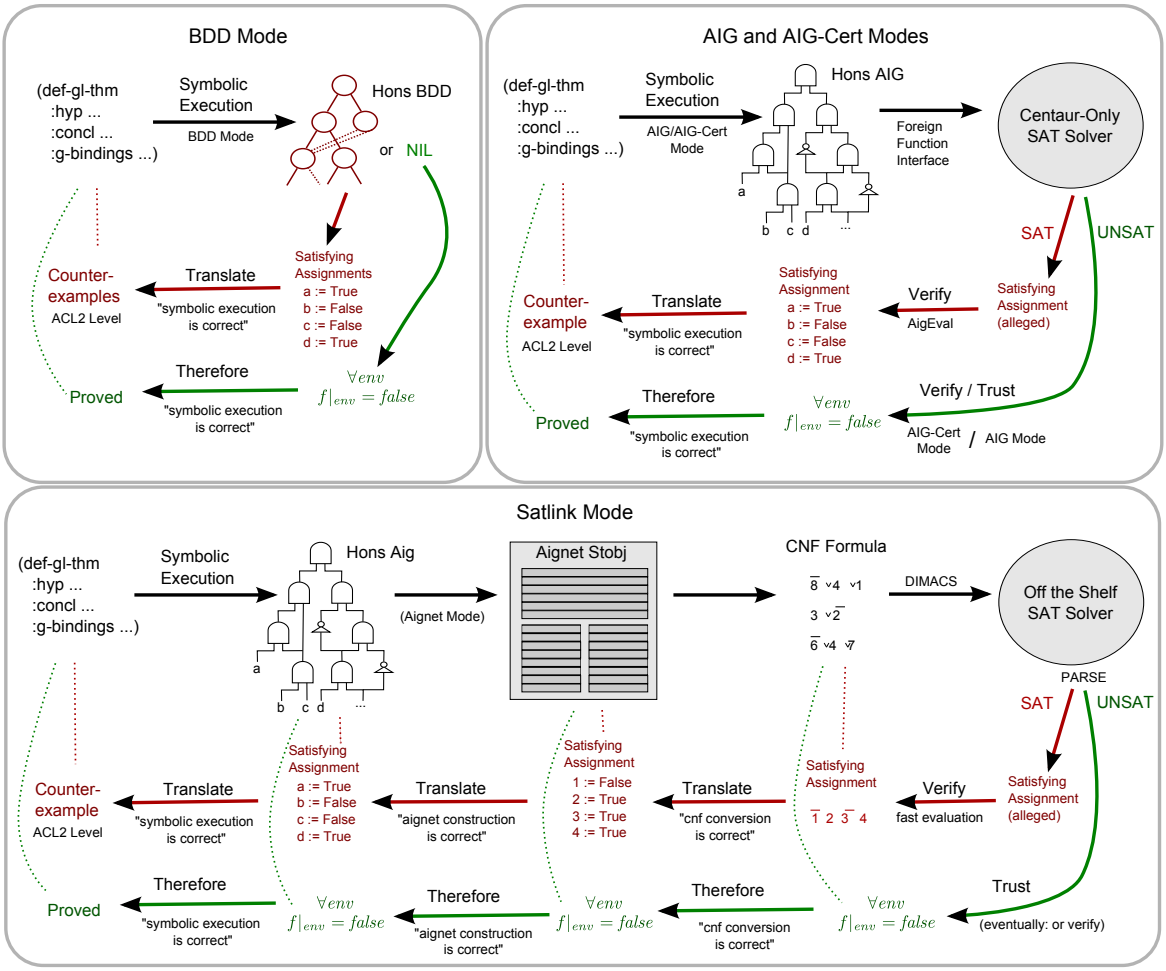
# 4 GL Integration

In previous work [28] we introduced *GL*, a framework for automatically proving "finite" ACL2 theorems. GL works by symbolically executing ACL2 formulas to *bit-blast* (translate) them into Boolean formulas, which can then be solved using a tool like a BDD package or a SAT solver.

We use GL to verify x86 execution units [21]. Its ability to solve these problems depends on the performance of its symbolic execution and the capacity of these solvers. Without help, GL can verify many instructions within a few seconds, and some others in a few minutes. For harder instructions, we have to manually intervene, e.g., we might decompose the proof in some way.

We want to scale up GL to avoid this human effort. Toward this end, we have developed a new GL *mode* that uses Aignet and its CNF conversion algorithm to connect GL to any off-the-shelf SAT solver that implements the DIMACS format. As we will explain shortly, this already has advantages over GL's previous approach to using SAT, and it opens up a promising route for future improvements, viz. simplifying the AIG before calling SAT.

GL can operate in different modes that govern how Boolean formulas will be represented during the symbolic execution, and how these formulas will be solved. The mode to use can be chosen at run-time on a per-proof basis using attachments [22], and this choice can have a significant impact on GL's performance. Previously, GL has supported three modes: *BDD mode*, *AIG mode*, and *AIG-Cert mode*. We describe these and our new *Satlink mode* modes below. But first, here is a sketch:



In BDD mode (the default), GL uses a simple Hons-based BDD representation [5] of Boolean functions during symbolic execution, and no back-end solver is needed because the truth of a BDD is immediately evident. This works well when the user knows how to choose a good BDD ordering, but is not well-suited for larger, less-structured problems.

In AIG and AIG-Cert modes, Hons-AIGs are used as the Boolean function representation during symbolic execution, and a SAT solver is used as the back-end. These modes are convenient since a good BDD ordering is not needed, and they tend to outperform BDDs on less-structured problems. In AIG mode, the SAT solver's claims of UNSAT are trusted without verification; in AIG-Cert mode they are checked using a verified checker developed by Matt Kaufmann. Both AIG modes are integrated with a SAT solver that can directly accept AIGs as inputs, but which unfortunately we cannot release.

Our new mode, Satlink mode, still uses Hons-AIGs as the Boolean function representation during symbolic execution. We then convert the goal Hons-AIG into an Aignet and use our verified CNF conversion algorithm to generate clauses that can be sent to any SAT solver that implements the usual DIMACS format. Like AIG mode, the SAT solver is simply trusted to be correct. We have not yet implemented a proof-checking scheme for this Satlink approach, but we are optimistic that recent work by Wetzler, et al. [32] might lead to a high-quality way to fill in this gap.

Already, Satlink mode appears to be quite promising. For instance, we applied it to a somewhat difficult proof: the correctness of our media unit's 128-bit x86 PSADBW instruction. We have never successfully proved this theorem with BDDs. Even for AIG mode, performance was so bad that we spent time manually *parameterizing* [28] the theorem. State-of-the-art tools like Glucose [2] and Lingeling [3] can solve the full problem quickly, making this work unnecessary!

| 128-bit PSADBW | Parameterized | Full |
|---|---|---|
| AIG Mode | 84.38 sec | ? (>24 hr) |
| Satlink/Lingeling | 62.02 sec | 50.08 sec |
| Satlink/Glucose | 6.11 sec | 14.88 sec |

A promising direction for future work is to simplify the Aignet before converting it to CNF. To explore this, we developed an unverified, experimental way to use ABC's rewriting and fraiging routines to do some light-weight simplification of the AIG that arises from symbolic simulation; the simplified problem is then solved with ABC's built-in SAT solver or by giving it to a back-end tool. It appears that these AIG algorithms may allow GL to scale far beyond its current capability.

To illustrate, we consider a proof of correctness for a hardware module that computes four independent single-precision floating point additions in parallel. The correctness theorem is stated as a single GL theorem, not broken into any cases. This problem is beyond the capacity of our existing BDD or AIG modes. Without ABC, only Satlink/Glucose can solve it in a reasonable amount of time. When we use ABC to perform some light-weight fraiging and rewriting first, Glucose can solve the problem almost 5x faster, and the problem also becomes tractable for other solvers.

| Independent FADDs | Total Time |
|---|---|
| AIG Mode | ? (>24 hr) |
| Satlink/Lingeling | 18,581 sec |
| Satlink/Glucose | 572 sec |
| ABC/integrated | 1,446 sec |
| ABC/Satlink/Lingeling | 442 sec |
| ABC/Satlink/Glucose | 121 sec |

## 5 Conclusions

Hons-AIGs and Aignet are two alternatives for representing AIGs that have complementary strengths and weaknesses. By keeping the DAG representation implicit, Hons-AIGs provide an especially simple logical story: we can deal in self-contained Boolean functions rather than a monolithic graph, and we can reason about AIG operations at the semantic level. In contrast, by making the DAG representation

explicit, Aignet allows us to write more efficient algorithms. Reasoning about these algorithms is more difficult and involves the kinds of invariants associated with any imperative code, but relationships like AIGNETEXTENSION go a long way toward making this tractable and we have been able to implement and verify useful Aignet algorithms, e.g., CNF conversion.

Our work to integrate Aignet into GL allows ACL2 users to make use of state-of-the-art SAT solvers to handle finite ACL2 goals. At present these SAT solvers are simply trusted. In the future, a verified SAT proof checker might be integrated into the flow to ensure correct reasoning.

This work is a starting point toward verified implementations of AIG simplification algorithms like fraiging and rewriting. We have seen that (unverified) implementations of these algorithms can significantly reduce the difficulty of problems that GL gives to the SAT solver, so this is a promising direction for increasing the scale of theorems that GL can automatically prove.

The source code for our AIG representations, algorithms, and GL connection are included in the ACL2 Community Books for ACL2 6.1; see `http://acl2-books.googlecode.com/`. Except for the Aignet library we rely on features specific to ACL2(h), so please see `books/centaur/README.html` for important setup information. Our Hons-AIG representation and algorithms are found in `centaur/aig` and the Aignet library is in `centaur/aignet`. The *Centaur Hardware Verification Tutorial*, available in `centaur/tutorial`, shows how to use the new Satlink mode; see especially `sat.lsp`.

We thank Bob Boyer, Warren Hunt, and Matt Kaufmann for contributing to the development of ACL2(h). We thank Matt Kaufmann for improvements to ACL2 in support of this work, especially related to abstract stobjs and non-executability. We thank Matt Kaufmann, David Rager, Anna Slobadová, and Nathan Wetzler for their corrections and helpful feedback on drafts of this paper.

## References

[1] Michaël Armand, Germain Faure, Benjamin Grégoire, Chantal Keller, Laurent Thery & Benjamin Werner (2011): *A Modular Integration of SAT/SMT Solvers to Coq through Proof Witnesses*. In: *CPP '11*, *LNCS* 7086, Springer, pp. 135–150, doi:10.1007/978-3-642-25379-9_12.

[2] Gilles Audemard & Laurent Simon (2009): *Predicting Learnt Clauses Quality in Modern SAT Solvers*. In: *IJCAI '09*, IJCAI Organization, pp. 399–404.

[3] Armin Biere (2012): *Lingeling and Friends Entering the SAT Challenge 2012*. In: *Proc. SAT Challenge 2012*, *Department of Computer Science Series of Publications B* B-2012-2, University of Helsinki, pp. 33–34.

[4] Sascha Böhme, Anthony Fox, Thomas Sewell & Tjark Weber (2011): *Reconstruction of Z3's Bit-Vector Proofs in HOL4 and Isabelle/HOL*. In: *CPP '11*, *LNCS* 7086, Springer, pp. 183–198, doi:10.1007/978-3-642-25379-9_15.

[5] Robert S. Boyer & Warren A. Hunt, Jr. (2006): *Function Memoization and Unique Object Representation for ACL2 Functions*. In: *ACL2 '06*, ACM, pp. 81–89, doi:10.1145/1217975.1217992.

[6] Robert S. Boyer & J Strother Moore (2002): *Single-Threaded Objects in ACL2*. In: *PADL '02*, *LNCS* 2257, Springer, pp. 9–27, doi:10.1007/3-540-45587-6_3.

[7] Aaron R. Bradley (2012): *Understanding IC3*. In: *SAT '12*, *LNCS* 7317, Springer, pp. 1–14, doi:10.1007/978-3-642-31612-8_1.

[8] Robert Brayton & Alan Mishchenko (2010): *ABC: An Academic Industrial-Strength Verification Tool*. In: *CAV '10*, *LNCS* 6174, Springer, pp. 24–40, doi:10.1007/978-3-642-14295-6_5.

[9] Robert Brummayer & Armin Biere (2006): *Local Two-Level And-Inverter Graph Minimization without Blowup*. In: *MEMICS '06*.

[10] Robert Brummayer & Armin Biere (2009): *Fuzzing and Delta-Debugging SMT Solvers*. In: *SMT '09*, ACM, pp. 1–5, doi:10.1145/1670412.1670413.

[11] Satrajit Chatterjee, Alan Mishchenko, Robert Brayton & Andreas Kuehlmann (2007): *On resolution proofs for combinational equivalence*. In: *DAC '07*, ACM, pp. 600–605, doi:10.1145/1278480.1278631.

[12] Ashish Darbari, Bernd Fischer & João Marques-Silva (2010): *Industrial-Strength Certified SAT Solving through Verified SAT Proof Checking*. In: *ICTAC '10*, *LNCS* 6255, Springer, pp. 260–274, doi:10.1007/978-3-642-14808-8_18.

[13] Jared Davis (2004): *Finite Set Theory based on Fully Ordered Lists*. In: *ACL2 '04*.

[14] Niklas Eén, Alan Mishchenko & Robert K. Brayton (2011): *Efficient implementation of property directed reachability*. In: *FMCAD '11*, FMCAD, Inc., pp. 125–134.

[15] Niklas Eén, Alan Mishchenko & Niklas Sörensson (2007): *Applying Logic Synthesis for Speeding Up SAT*. In: *SAT '07*, *LNCS* 4501, Springer, pp. 272–286, doi:10.1007/978-3-540-72788-0_26.

[16] Niklas Eén & Niklas Sörensson (2003): *An Extensible SAT-solver*. In: *SAT '03*, *LNCS* 2919, Springer, pp. 502–518, doi:10.1007/978-3-540-24605-3_37.

[17] Anthony Fox (2011): *LCF-Style Bit-Blasting in HOL4*. In: *ITP '11*, LNCS, Springer, pp. 357–362, doi:10.1007/978-3-642-22863-6_26.

[18] Michael J. Gordon, Arthur J. Milner & Christopher P. Wadsworth (1979): *Edinburgh LCF: A Mechanized Logic of Computation*. *LNCS* 78, Springer, doi:10.1007/3-540-09724-4.

[19] David Greve (2009): *Automated Reasoning with Quantified Formulae*. In: *ACL2 '09*, ACM, pp. 110–113, doi:10.1145/1637837.1637855.

[20] Warren A. Hunt, Jr., Matt Kaufmann, Robert Bellarmine Krug, J Moore & Eric Whitman Smith (2005): *Meta Reasoning in ACL2*. In: *TPHOLs '05*, *LNCS* 3603, Springer, pp. 163–178, doi:10.1007/11541868_11.

[21] Warren A. Hunt, Jr., Sol Swords, Jared Davis & Anna Slobodová (2010): *Use of Formal Verification at Centaur Technology*. In David Hardin, editor: *Design and Verification of Microprocessor Systems for High-Assurance Applications*, Springer, pp. 65–88, doi:10.1007/978-1-4419-1539-9_3.

[22] Matt Kaufmann & J Strother Moore (2011): *How Can I Do That with ACL2? Recent Enhancements to ACL2*. In: *ACL2 '11*, pp. 46–60, doi:10.4204/EPTCS.70.4.

[23] Filip Marić (2009): *Formalization and Implementation of Modern SAT Solvers*. *Journal of Automated Reasoning* 43(1), pp. 81–119, doi:10.1007/s10817-009-9127-8.

[24] Alan Mishchenko, Michael Case, Robert Brayton & Stephen Jang (2008): *Scalable and scalably-verifiable sequential synthesis*. In: *ICCAD '08*, IEEE, pp. 234–241, doi:10.1109/ICCAD.2008.4681580.

[25] Alan Mishchenko, Satrajit Chatterjee & Robert Brayton (2006): *DAG-Aware AIG Rewriting: A Fresh Look at Combinational Logic Synthesis*. In: *DAC '06*, ACM, pp. 532–535, doi:10.1145/1146909.1147048.

[26] Alan Mishchenko, Satrajit Chatterjee, Roland Jiang & Robert Brayton (2005): *FRAIGs: A Unifying Representation for Logic Synthesis and Verification*. Technical Report, EECS Dept., UC Berkeley.

[27] Duckki Oe, Aaron Stump, Corey Oliver & Kevin Clancy (2012): *versat: A Verified Modern SAT Solver*. In: *VMCAI '12*, *LNCS* 7148, Springer, pp. 363–378, doi:10.1007/978-3-642-27940-9_24.

[28] Sol Swords & Jared Davis (2011): *Bit-Blasting ACL2 Theorems*. In: *ACL2 '11*, *Electronic Proceedings in Theoretical Computer Science* 70, pp. 84–102, doi:10.4204/EPTCS.70.7.

[29] Sol Swords & Warren A. Hunt, Jr. (2010): *A Mechanically Verified AIG to BDD Conversion Algorithm*. In: *ITP '10*, *LNCS* 6172, Springer, pp. 435–449, doi:10.1007/978-3-642-14052-5_30.

[30] G. S. Tseitin (1968): *On the Complexity of Derivation in Propositional Calculus*. *Zapiski nauchnykh seminarov LOMI* 8, pp. 234–259, doi:10.1007/978-3-642-81955-1_28. English translation of this volume: Consultants Bureau, N.Y., 1970, pp. 115–125.

[31] Tjark Weber & Hasan Amjad (2009): *Efficiently Checking Propositional Refutations in HOL Theorem Provers*. *Journal of Applied Logic* 7(1), pp. 26–40, doi:10.1016/j.jal.2007.07.003.

[32] Nathan Wetzler, Marijn J. H. Heule & Warren A. Hunt, Jr. (2013): *Mechanical Verification of SAT Refutations with Extended Resolution*. In: *To appear in ITP 2013*.