Stålmarck's Algorithm as a HOL Derived Rule

John Harrison

Åbo Akademi University, Department of Computer Science
Lemminkäisenkatu 14a, 20520 Turku, Finland

Abstract. Stålmarck's algorithm is a patented technique for tautology-checking which has been used successfully for industrial-scale problems. Here we describe the algorithm and explore its implementation as a HOL derived rule.

1 Introduction

To test whether a Boolean expression is a tautology, i.e. is true for all truth assignments of the variables it involves, there is no shortage of available methods. For example, one can construct its truth-table and make sure each row evaluates to 'true', convert it to conjunctive normal form and check that each conjunct contains some variable disjoined with its negation, or even translate it into an integer programming problem and solve it using a variety of standard techniques [9]. Moreover, many standard proof search procedures for first order logic yield a decision procedure for propositional logic as a special case, e.g. the Davis-Putnam procedure, resolution, model elimination and tableaux, including the more efficient variant of tableaux studied by d'Agostino [5].

Whatever their particular strengths and weaknesses, all known methods take time exponential in the size of the input formula, in the worst case. Since Cook [4] showed the dual problem of testing Boolean satisfiability to be NP complete, tautology checking is co-NP complete, and so it seems quite likely that *any* algorithm will have exponential complexity characteristics. However this still leaves open the possibility that some algorithms are acceptably efficient in a large number of important cases. Sometimes the traditional methods can shine in the right problem domain [17]. But in practical applications, the two most promising techniques seem to be binary decision diagrams [2], and a recent patented algorithm due to Stålmarck [14].

Binary decision diagrams have been most successful in hardware verification, though other applications have been explored — Bryant [3] gives a survey. Stålmarck's algorithm has also been applied to hardware verification, in a large number of real industrial situations. Some early examples [12] include reverse flushing control in a nuclear power station's emergency cooling system, and landing gear control logic for Saab military aircraft. More recently, the algorithm has also been applied to verification of, inter alia, car engine management systems, special service non-interaction in telephone networks, programmable logic controllers (PLCs) in various process industries (wood, steel, food etc.), and railway interlocking systems. This last application generally requires checking of tautologies involving something like 10^5 variables. This is too much for many traditional methods, including BDDs, though by using careful problem partitioning, e.g. a hiding technique proposed by Groote [7], they might still be tractable. Stålmarck's algorithm seems less sensitive to the number of variables than other methods, with runtimes depending more on certain proof-theoretic properties of the formula.

This paper discusses the implementation of Stålmarck's algorithm as a HOL derived rule. Part of our motivation was that it is an interesting experiment in LCF-style reduction of standard decision procedures, potentially yielding another valuable data point. Moreover, the algorithm itself is not yet widely known, and it's interesting to investigate how it performs, with or without integration as a HOL derived rule. But there are also deeper reasons.

The present author [8] has implemented the BDD algorithm as a HOL derived rule. This turned out to be possible with only a constant factor slowdown as compared with a direct ML implementation, but the constant factor was significant (40–50), and the implementation needed quite sophisticated tricks.[1] Stålmarck's algorithm has two features that make it seem a more promising target for HOL implementation. First, it is conceptually closer to a traditional natural deduction system of the kind HOL implements, so steps in the algorithm may be translated quite directly into HOL inferences. Second, the algorithm permits a significant separation of proof search from proof checking. It is well-known that this is advantageous for HOL implementation. The proof search stage need not produce HOL inferences; it need only record the proof eventually found, which may then be 'checked', i.e. translated to primitive inferences. By contrast, the HOL implementation of BDDs needs to track all stages of BDD construction by inference rules. (On the other hand the matter of deciding on a suitable variable ordering, which is of considerable significance for the efficiency of the BDD technique, *can* be separated.)

Another motivation is that Stålmarck's company Logikkonsult NP AB[2] is becoming increasingly interested in proof logging and proof checking for its implementation. Indeed, some of its customers are requesting (or insisting on) some kind of proof that can be independently checked. This phenomenon is likely to become increasingly common in applications of formal methods, as suggested by the U.K. Ministry of Defence [11]:

> 32.3.1 [...] It is [...] possible to remove the reliance on the correctness of the theorem proving assistant from the case for correctness of an application by arranging that a version of the final proof (omitting all history of its construction) is passed from the theorem proving assistant to a proof checker. For reasonable languages, such a proof checker could be a very simple program (perhaps ten pages in a functional programming language) that could be developed to the highest level of assurance.

Implementation as a HOL rule with the 2-pass structure that we have outlined *is* a form of proof checking. Indeed, it might be argued that it offers a rather high level of reliability. But even if HOL checkability doesn't coincide with the proof-checking interests of Logikkonsult or its customers, many of the same issues arise with other proof-checking arrangements, so some of our experiments here have a wider significance.

---

[1] In fact, not all of these are necessary given certain assumptions about equality testing and sharing inside terms.

[2] Swedenborgsgatan 2, S-118 48 Stockholm, Sweden.

2 Stålmarck's Algorithm

Stalmarck's method can deal with formulas involving all the usual logical connectives, $\neg$ (negation), $\wedge$ (conjunction), $\vee$ (disjunction), $\Rightarrow$ (implication) and $\Leftrightarrow$ (logical equivalence). However it avoids duplication later if some initial canonicalization is applied. The exact method chosen is not particularly important, though it should avoid the blowup in formula size that results from splitting logical equivalences. An early version of the algorithm [15] systematically pulled negations *up* the formula, leaving a body involving only conjunction, disjunction, implication and logical equivalence applied to unnegated propositional variables. This has the appealing feature that if the resulting formula is negated, it is immediately clear that it cannot be a tautology (set all propositional variables to 'true'); therefore in the main part of the algorithm, negation is avoided completely. However, we follow a more recent version of the algorithm in reducing the formula so that it uses only negation, conjunction and logical equivalence. Specifically, we apply the following transformations in a single bottom-up sweep, applying the first in the list where more than one is possible:

$$
\begin{aligned}
\neg\neg p &\longrightarrow p \\
\neg p \vee \neg q &\longrightarrow \neg(p \wedge q) \\
\neg p \vee q &\longrightarrow \neg(p \wedge \neg q) \\
p \vee \neg q &\longrightarrow \neg(\neg p \wedge q) \\
p \vee q &\longrightarrow \neg(\neg p \wedge \neg q) \\
\neg p \Rightarrow \neg q &\longrightarrow \neg(\neg p \wedge q) \\
\neg p \Rightarrow q &\longrightarrow \neg(\neg p \wedge \neg q) \\
p \Rightarrow \neg q &\longrightarrow \neg(p \wedge q) \\
p \Rightarrow q &\longrightarrow \neg(p \wedge \neg q) \\
\neg \top &\longrightarrow \bot \\
\neg \bot &\longrightarrow \top \\
p \wedge \top &\longrightarrow p \\
\top \wedge p &\longrightarrow p \\
p \wedge \bot &\longrightarrow \bot \\
\bot \wedge p &\longrightarrow \bot \\
p \vee \top &\longrightarrow \top \\
\top \vee p &\longrightarrow \top \\
p \vee \bot &\longrightarrow p \\
\bot \vee p &\longrightarrow p \\
p \Rightarrow \top &\longrightarrow \top \\
\top \Rightarrow p &\longrightarrow p \\
p \Rightarrow \bot &\longrightarrow \neg p \\
\bot \Rightarrow p &\longrightarrow \top
\end{aligned}
$$

$$
\begin{aligned}
p \Leftrightarrow \top &\longrightarrow p \\
\top \Leftrightarrow p &\longrightarrow p \\
p \Leftrightarrow \bot &\longrightarrow \neg p \\
\bot \Leftrightarrow p &\longrightarrow \neg p
\end{aligned}
$$

If the formula is thus reduced to the logical constant $\top$ or $\bot$, then we are finished: in the first case the formula is a tautology; in the second it is not; indeed it is unsatisfiable. Otherwise we may suppose that the formula now involves only negation, conjunction and logical equivalence applied to propositional variables.

Reduction to triplets

Next, we imagine introducing a new propositional variable to represent each subformula that is either a conjunction or an equivalence, and assuming a set of logical equivalences representing their 'definitions'. For example given the formula:

$$\neg((a \Leftrightarrow b \wedge c) \wedge (b \Leftrightarrow \neg c) \wedge a)$$

we introduce new variables $v_1, \ldots, v_5$ defined by:

$$
\begin{aligned}
&v_1 \Leftrightarrow b \wedge c \\
&v_2 \Leftrightarrow a \Leftrightarrow v_1 \\
&v_3 \Leftrightarrow b \Leftrightarrow \neg c \\
&v_4 \Leftrightarrow v_3 \wedge a \\
&v_5 \Leftrightarrow v_2 \wedge v_4
\end{aligned}
$$

Under these assumptions, the whole formula is equivalent to $\neg v_5$. Therefore to prove that formula it suffices to derive a contradiction from the above definitions together with the additional assumption v_5.

We said 'imagine' doing the above because it is, from a logical point of view, unnecessary. The intention is only to provide convenient short identifiers for subformulas, which is important since the algorithm works by assigning values to, and recording equivalences between, these subformulas. But the final proof discovered can perfectly well be written out with the subformulas used directly as their own names. This is what happens in the HOL version of the algorithm which we describe later. The above reduction may therefore be regarded as completely metalogical, a mere implementation convenience.

We have reduced the original formula to a set of 'triplets' of the form $p \Leftrightarrow q \otimes r$ where $\otimes$ is either conjunction or equivalence, p is a propositional variable (real or imaginary) and q and r are literals, i.e. either variables or their negations. The algorithm works by using these triplets to make logical inferences. All facts used and deduced by the algorithm can be considered as equations between literals,[3] if for the sake of regularity we treat $\top$ as another variable. That is, rather than v and $\neg v$, we use $v = \top$

[3] Or bi-implications, if the reader prefers to think of them that way.

and $v = \neg\top$ for actual truth-assignments. The starting point is the single equation $v^* = \neg\top$ where v^* represents the whole formula. In our example we start with $\neg v_5 = \neg\top$, or equivalently, $v_5 = \top$. The objective is to reach a contradictory equation of the form $v = \neg v$.

Simple rules

The basic means for deriving new equations from old is a set of 'simple rules'. These simply use the triplets together with the existing equations to deduce new equations via some obvious deductions. First there are rules for conjunctive triplets $p \Leftrightarrow q \wedge r$:

- if $p = \neg q$ then $q = \top$ and $r = \neg\top$
- if $p = \neg r$ then $q = \neg\top$ and $r = \top$
- if $q = r$ then $p = r$
- if $q = \neg r$ then $p = \neg\top$
- if $p = \top$ then $q = \top$ and $r = \top$
- if $q = \top$ then $p = r$
- if $q = \neg\top$ then $p = \neg\top$
- if $r = \top$ then $p = q$
- if $r = \neg\top$ then $p = \neg\top$

And there is also a similar set of rules for equivalential triplets $p \Leftrightarrow q \Leftrightarrow r$:

- if $p = q$ then $r = \top$
- if $p = \neg q$ then $r = \neg\top$
- if $p = r$ then $q = \top$
- if $p = \neg r$ then $q = \neg\top$
- if $q = r$ then $p = \top$
- if $q = \neg r$ then $p = \neg\top$
- if $p = \top$ then $q = r$
- if $p = \neg\top$ then $q = \neg r$
- if $q = \top$ then $p = r$
- if $q = \neg\top$ then $p = \neg r$
- if $r = \top$ then $p = q$
- if $r = \neg\top$ then $p = \neg q$

If we forget about the metalogical assignment of local variables and the breakdown into triplets, we can see all these as quite straightforward logical rules. For example the rule for conjunctive triplets 'if $p = \neg q$ then $q = \top$ and $r = \neg\top$' can be seen as:

$$\frac{\neg q \Leftrightarrow (q \wedge r)}{q \quad \neg r}$$

while 'if $p = \top$ then $q = \top$ and $r = \top$' looks even more like a standard natural deduction rule:

$$\frac{q \wedge r}{q \quad r}$$

0-saturation

Given a set of equations, the process of 0-saturation simply means deducing as many other equations as possible using only the simple rules. We also assume the use of symmetry and transitivity of equality together with the involution property of negation, e.g. going from $p = q$ and $\neg p = r$ to $q = \neg r$. From a theoretical point of view we can imagine deriving all possible equations using these extra properties, though of course in the actual implementation we do not derive such a heavily redundant set.

Let us see how our example can already be proved by 0-saturation. We start with just the equation $\neg v_5 = \neg\top$, i.e. $v_5 = \top$, and proceed as follows:

- By $v_5 = \top$ and triplet 5, $v_2 = \top$ and $v_4 = \top$
- By $v_4 = \top$ and triplet 4, $v_3 = \top$ and $a = \top$
- By $v_3 = \top$ and triplet 3, $b = \neg c$
- By $b = \neg c$ and triplet 1, $v_1 = \neg\top$
- By $v_2 = \top$, $a = \top$ and triplet 2, $v_1 = \top$
- By $v_1 = \top$ and $v_1 = \neg\top$, $v_1 = \neg v_1$, a contradiction.

Indeed, note that in general if the original formula uses local variable assignments in the following style:

$$\bigwedge_i (v_i = E_i) \Rightarrow \phi[v_1, \ldots, v_n]$$

then all the equivalences $v_i = E_i$ will be discovered by 0-saturation.

The Dilemma Rule

In general, 0-saturation alone is not sufficient to prove formulas. If after 0-saturation, no contradictory assignment has been reached, the next step is to use the so-called *dilemma rule*. This involves a case-split over a variable, though one of a rather sophisticated kind. Suppose that 0-saturation has yielded a set of equations Σ, and that we choose v as the variable to split over. Then we 0-saturate the sets $\Sigma \cup \{v = \top\}$ and $\Sigma \cup \{v = \neg\top\}$ to produce new sets of equations $\Delta_\top$ and $\Delta_\bot$ respectively. Even if we have not gained a contradictory assignment in both $\Delta_\top$ and $\Delta_\bot$, the case split may still yield new information. Set $\Delta = \Delta_\top \cap \Delta_\bot$ (if a set of equations contains a contradictory assignment, we think of it as contaning all possible equations between pairs of literals; so for example if $\Delta_\top$ contains a contradiction then $\Delta = \Delta_\bot$). It is clear that we may now assume the set of equations Δ, since they hold whatever the value of v may be. We certainly have $\Sigma \subseteq \Delta$, and if Δ is a *proper* superset of Σ, new information has been obtained by the case split.

The process of 1-saturation means applying the dilemma rule to each variable (real or imaginary) in turn, repeating for all variables as long as new equations are obtained. If 1-saturation does not solve the problem, then 2-saturation is attempted. This is like 1-saturation, but case splits are tried over *pairs* of variables simultaneously. That is, for each pair of variables v and w, one 0-saturates the sets $\Sigma \cup \{v = \top, w = \top\}$, $\Sigma \cup \{v = \top, w = \neg\top\}$, $\Sigma \cup \{v = \neg\top, w = \top\}$ and $\Sigma \cup \{v = \neg\top, w = \neg\top\}$, and takes the intersection of the results, again repeating as often as new information is obtained. Similarly, one can n-saturate for any natural number n, case-splitting over n-tuples of variables simultaneously.

Stålmarck's method differs from naive methods using case splits in two important respects. First, it is possible to case-split not just over the primitive propositional variables, but over the imaginary ones too, i.e. over nontrivial subformulas. This may cause truth-assignments to propagate both up *and down* the formula's 'syntax tree'. For example if the formula contains a subformula $p \wedge (q \wedge r)$, then the assignment $q \wedge r = \neg\top$ will propagate up to yield $p \wedge (q \wedge r) = \neg\top$, while the assignment $q \wedge r = \top$ will propagate down to give $q = \top$ and $r = \top$. Second, rather than case-split over every-increasing numbers of variables until a contradiction is reached directly, the number of simultaneous case splits is kept as low as possible, with all new information arising carefully garnered and fed back into the next iteration. This avoids an immediate exponential blowup.

The algorithm gives rise to a new and interesting classification of tautologies according to their hardness. A tautology is said to be n-easy if it can be proved by n-saturation, and n-hard if it cannot be solved by $(n-1)$-saturation. Obviously, the practicality of the algorithm for a large problem is likely to depend on the problem's being n-easy for reasonably small n. In fact, Stålmarck [13] shows that if a tautology A with size (number of connectives) $|A|$ is n-easy, then there is a proof of it with size $\leq |A|^{n+1}$. Moreover, the running time of his n-saturation algorithm, which is guaranteed to find such a proof, is bounded by $O(|A|^{2n+1})$.[4] For large formulas, even 2-saturation can be impractical, so the homophony of '2-hard' and 'too hard' is quite apt. Fortunately it turns out that many practical formulas arising in verification are in fact 1-easy. So often, in fact, that Logikkonsult's tools perform 1-saturation first, and if that fails, try to find a falsifying assignment, since experience shows that a formula is either 1-easy or not a tautology at all!

In particular, notice that 1-saturation will always discover common subformulas, modulo the symmetry of conjunction and equivalence, so assuming the formula is at least 1-hard, there is not much advantage in clever structure sharing when forming the triplets. If two identical formulas $p \otimes q$ appear in different places, giving rise to triplets $r_1 = p \otimes q$ and $r_2 = p \otimes q$, then splitting over either p or q will yield the equation $r_1 = r_2$. By the iteration of 1-saturation, this effect propagates up arbitrary common subformulas.

Patent

Note that the above algorithm is patented for commercial use.

[4] The algorithm described there and in other written presentations is more limited than the current version as we describe it here: rather than finding and using arbitrary equations between literals, it only uses true/false assignments. This makes it simpler, and also much closer to a standard natural deduction presentation. However the extension with arbitrary equations between variables is significantly more powerful, and is used in Logikkonsult's industrial practice.

3 Implementation

We shall first describe a direct implementation, since only minor modifications are required to produce a HOL derived rule. The algorithm is fairly straightforward to implement, the main difficulty being to represent efficiently the potentially quadratic number of equations derived.

Storing equations

Each (real or imaginary) variable is allocated a positive integer as an identifier. Logical negation is represented by numerically negating the corresponding integer. This means of course that we can't use 0. Moreover, we allocate 1 to the pseudo-variable $\top$. Now out of all the equations between variables $\pm v = \pm w$, we store only canonical ones in the form $v = \pm w$ where $w < v$ and w is the minimum possible value. So in particular, if a variable is assigned to $\top$ or $\bot$, then that assignment is the one stored. From these, other equations between variables may be created when required by plugging the stored equations together via transitivity and symmetry: if $x = y$ is derivable, then we must have $x = \pm v$ and $y = \pm v$ for the same canonical assignment v.

These canonical variable assignments are stored in assignment arrays. These have one cell per variable, which contains either its canonical assignment, if it has one, or otherwise the list of variables (with their signs) which are assigned to *it* (this list may be empty — initially all lists are empty).

When a new assignment $x = y$ is deduced, first of all the canonical assignments of x and y are taken from the assignment arrays, giving an equation $x' = y'$ between unassigned literals. Moreover, we orient it so that it is of the form $v = \pm w$ where $w < v$. Now this assignment for v is entered in the table, and moreover, each element in the list already assigned to v now becomes assigned to w (with the appropriate sign). The list of variables assigned to w has appended to it the list of those which were previously assigned to v.

Main loop

The main part of the algorithm is now quite simple. We repeatedly make the new assignments indicated then attempt to derive new facts using the simple rules, repeating until no new information can be derived. To avoid too much redundant searching of the simple rules, we initially produce lists of the triplets involving each variable. Then after a variable assignment to $v_1, \ldots, v_n$ (after canonicalization), only the triplets involving $v_1, \ldots, v_n$ need be examined for new information. Variables are tried in order of the number of triplets in which they appear, so that more apparently important variables are favoured.

During n-saturation, it is necessary to form intersections of the assignment arrays arising from the different settings of the variables. This is done by repeated pairwise intersection. Pairwise intersection, in the case where neither array is contradictory (this is indicated by assigning variable 0, which is otherwise unused), proceeds as follows. For each variable v assigned in both arrays (lists of assigned variables are maintained, again to avoid excessive searching), we find the respective canonical assignments w_1

and w_2. The assigned-to lists of these variables are examined, and the least common element, if any, extracted. This is now assigned to v in the result. When at least one array contains a contradiction, then it is merely necessary to copy the other. To avoid copying arrays, we proceed as follows. Three assignment arrays are allocated once and for all. The pointers to them are rotated if necessary by the intersection function, i.e. rather than copy one array to another, the pointers are merely exchanged. No other storage is needed.

After a saturation step resulting in new information, the triplets are rewritten with the assignments and all the assignment arrays reset. However, whatever internal assignments are derived during saturation, the triplets are not changed then.

As a derived rule

To implement the algorithm as a HOL derived rule requires remarkably modest changes to the overall structure of the code. The guiding principle is that each equation $v = \pm w$ is paired with a theorem (in general, with assumptions) justifying the equation between the corresponding subterms. The algorithm can be performed more or less as above, using symmetry and transitivity to justify certain procedures, like allocating the variables previously assigned to v to w when a new equation $v = w$ is derived. The simple rules also produce theorems, which are produced by matching against one of 30 pre-proved tautologies. The process of intersection becomes slightly more complex. After finding the least assignment common to both arrays, the corresponding theorems are combined using `DISJ_CASES`. For example, in 1-saturation, one theorem will have an assumption $v = \top$, the other will have $v = \neg\top$. The result will have an assumption $(v = \top) \vee (v = \neg\top)$. This assumption is itself a simple tautology and can therefore be eliminated quite easily. Where one assignment is contradictory, we first use *ex falso quodlibet* to derive identical conclusions. For example, from $\Gamma, v = \top \vdash x = y$ and $\Gamma, v = \neg\top \vdash \bot$, we first derive $\Gamma, v = \neg\top \vdash x = y$ then proceed as before. It would be much more efficient to retain a separate list of those assignments which gave a contradiction and eliminate them only at the end, but this is more tedious to implement.

Two passes

During the search for case splits which yield new information, there is no need to perform inference. Accordingly, we proceed as follows. The version which does not perform inference is run to discover the sequence of case splits which yields the result. Only these are then run performing inference. With more effort, the amount of work done in the inference phase could be cut back further. For example, once a branch of the case-split has derived all the equations that are known to hold in the final intersection, there is no need to proceed to others, since these will merely be thrown away. But the present organization seems reasonably satisfactory, and the proof log is of a very simple form: just a list of variable tuples to split over.

4 Results

In the long term, we hope to give results from Logikkonsult's own industrial examples. But some practical obstacles need to be cleared first. In any case, it is well not to let our implementation loose on really big examples until it has been improved. For the present, we use tautologies taken from three main sources.

1. An IFIP International Workshop on Applied Formal Methods For Correct VLSI Design 13-16 November 1989, IMEC Leuven, Belgium included a set of examples arising from hardware verification. These all involve verifying that two different combinational logic circuits implement the same function (in a couple of cases, modulo a few 'don't care' states).
2. The Second DIMACS Challenge[5] included a large number of Boolean satisfiability test cases. Those purported to be unsatisfiable were negated and used as test cases for tautology checking. They come from a range of applications; some are encodings of problems like graph-colouring, the Towers of Hanoi or the pigeonhole problem, some arise from circuit fault analysis, others are generated randomly. All are (when negated for satisfiability testing) in conjunctive normal form.
3. The validity of formulas $\forall x_1, \ldots, x_n. \exists y_1, \ldots, y_n. P[x_1, \ldots, x_n, y_1, \ldots, y_n]$, where no function symbols are involved, can be reduced to tautology checking [1]. We took most problems from the TPTP problem library [16] that fell into this subset (i.e. that when negated and reduced to clausal form for refutation, involved no non-nullary function symbols)[6] and generated the corresponding tautologies. Again, they are all in conjunctive normal form for refutation. Jeroslow [10] suggests more efficient translation techniques, e.g. exploiting 'type' information or checking smaller disjunctions first. But since we are interested in tautology checking per se, an inefficient translation is quite welcome! Nevertheless, a few examples, notably 'SAM's Lemma', were excluded because the resulting tautology was rather big (many millions of connectives).

All results are in seconds of user CPU on a Sparc 10. Note that this is running in interpreted CAML Light. It would probably run several times faster in a compiled version of ML. Moreover the inference-free 'oracle' could be rewritten in C and using a more efficient imperative style. We estimate that such an implementation would run at about 50 times the speed, even without improving the basic implementation structure explained above. We will first give results for some problems which are 1-easy. We list the problem name and the source (from the above three categories). Then we give the number of (primitive) propositional variables[7] and the number of connectives (conjunction and equivalence) they contain. The problems are sorted according to this latter figure, more or less the 'size' of the formula. Then runtimes are given to find the

[5] See `http://mat.gsia.cmu.edu/challenge.html`.

[6] Other first order logic problems, e.g. those involving only monadic predicates, could be reduced to this class [6].

[7] The IMEC examples use local variable assignments, so this is not an accurate indicator. Perhaps the number of variables remaining after 0-saturation would be a better one.

variable assignment sequence without performing inference, and to translate this into HOL inferences.

Problem	Source	variables	connectives	search time	proof time
syn323_1	TPTP	2	7	0.00	0.10
syn029_1	TPTP	3	9	0.00	0.11
syn052_1	TPTP	2	9	0.00	0.13
syn051_1	TPTP	3	11	0.00	0.13
syn044_1	TPTP	3	12	0.01	0.25
syn011_1	TPTP	7	16	0.01	0.18
syn032_1	TPTP	6	16	0.00	0.36
ex2_be	Imec	7	18	0.00	0.40
syn030_1	TPTP	5	21	0.03	0.31
transp_be	Imec	8	21	0.01	0.65
syn054_1	TPTP	8	23	0.01	0.23
gra001_1	TPTP	5	31	0.08	2.30
syn321_1	TPTP	10	31	0.00	0.20
rip02_be	Imec	9	41	0.20	2.96
puz014_1	TPTP	13	48	0.03	0.31
mul03_be	Imec	54	179	1.83	22.75
puz030_2	TPTP	10	213	4.55	67.23
puz030_1	TPTP	25	221	0.73	11.98
dk27_be	Imec	56	227	1.75	25.58
syn071_1	TPTP	16	233	0.61	1.30
aim_50_1_6_no_3	Dimacs	50	239	3.18	12.15
aim_50_1_6_no_4	Dimacs	50	239	1.98	3.88
hostint1_be	Imec	10	247	0.86	17.06
aim_50_2_0_no_4	Dimacs	50	298	6.91	22.21
aim_50_2_0_no_1	Dimacs	50	299	3.00	17.50
aim_50_2_0_no_2	Dimacs	50	299	6.93	21.81
aim_50_2_0_no_3	Dimacs	50	299	2.91	10.95
mul_be	Imec	14	324	2.73	31.20
dk17_be	Imec	63	327	18.23	92.11
risc_be	Imec	81	337	3.13	48.11
msc006_1	TPTP	32	449	1.98	2.91
syn072_1	TPTP	30	518	2.65	5.60
aim_100_2_0_no_1	Dimacs	100	599	7.71	5.23
aim_100_2_0_no_2	Dimacs	100	599	8.01	6.58
prv001_1	TPTP	115	600	3.00	2.18
ssa0432_003	Dimacs	435	2363	261.40	228.65
jnh211	Dimacs	100	3887	310.78	1451.21

The results are quite variable, but it is clear that to some extent the slowness of performing inference is compensated for by the fact that proof search does not need to be performed. In most cases the proof time is greater, but seldom by a dramatic margin. In some cases it is even shorter. Now let us look at some figures for 2-hard problems.

Problem	Source	variables	connectives	search time	proof time
rip04_be	Imec	19	97	4.02	6.62
ztwaalf2_be	Imec	13	107	9.11	36.75
ztwaalf1_be	Imec	15	111	57.17	32.01
z4_be	Imec	31	145	3.70	16.13
rip06_be	Imec	29	153	20.38	14.52
add1_be	Imec	59	173	11.43	9.95
rip08_be	Imec	39	209	70.08	20.43
aim_50_1_6_no_1	Dimacs	50	238	4.91	3.28
aim_50_1_6_no_2	Dimacs	50	239	8.88	5.25
vg2_be	Imec	77	277	30.10	26.38
misg_be	Imec	108	279	68.36	10.58
x1dn_be	Imec	79	279	261.70	61.36
counter_be	Imec	18	290	182.75	97.22
sqn_be	Imec	66	317	52.43	183.28
add2_be	Imec	144	407	38.11	27.05
dc2_be	Imec	87	409	90.43	231.28
f51m_be	Imec	101	449	371.25	207.10
aim_100_1_6_no_3	Dimacs	100	479	117.10	18.18
dubois20	Dimacs	60	479	335.65	41.31
add3_be	Imec	260	693	270.50	31.03
add4_be	Imec	376	1110	1267.03	85.18

Clearly the balance has shifted: here the proof search stage usually takes longer, significantly so in some cases. This is as expected, since the number of possible 2-saturations increases quadratically with the size of the formula, leading to much more expensive search. It is an interesting question whether very large 1-easy problems exhibit similar characteristics.

5 Conclusions

We have seen that Stålmarck's algorithm can be implemented reasonably straightforwardly as a HOL rule. This results in a substantial slowdown. However this slowdown (which in any case could no doubt be reduced by more efficient inference patterns) need not affect the proof-search phase. For problems where this dominates, particularly problems which are 2-hard, performance as a derived rule is quite good.

At present, neither the inference-free version nor the derived rule has been implemented especially efficiently. Future work will be to remedy this, and see how the balance of the above tables is modified. Rather than expend energy on a highly efficient C implementation of the algorithm, we hope to use Logikkonsult's own implementation to provide a proof log. It will be interesting to see how the HOL version can cope with truly large problems.

Acknowledgements

Thanks are due to Graeme Parkin and Tony Mansfield of the National Physical Laboratory for helping me to understand Stalmarck's algorithm, and most of all to Gunnar Stålmarck himself for explaining the algorithm in detail and providing stimulating discussions about this and many other topics. My work was conducted while I was a member of the Programming Methodology Group of Åbo Akademi University, supported by the European Commission under the Human Capital and Mobility scheme.

References

1. P. Bernays and M. Schönfinkel. Zum Entscheidungsproblem der mathematischen Logik. *Mathematische Annalen*, 99:401–419, 1928.
2. R. E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, C-35:677–691, 1986.
3. R. E. Bryant. Symbolic Boolean manipulation with ordered binary-decision diagrams. *ACM Computing Surveys*, 24:293–318, 1992.
4. S. A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the 3rd ACM Symposium on the Theory of Computing*, pages 151–158, 1971.
5. M. D'Agostino. Investigations into the complexity of some propositional calculi. Technical Monograph PRG-88, Oxford University Computing Laboratory, Programming Research Group, 11 Keble Road, Oxford, OX1 3QD, 1990. Author's PhD thesis.
6. M. Di Manzo, E. Giunchiglia, A. Armando, and P. Pecchiari. Proving formulas through reduction to decidable classes. In P. Torasso, editor, *Proceedings of the 3rd Congress of the Italian Association for Artificial Intelligence, AI*IA '93*, volume 728 of *Lecture Notes in Computer Science*, pages 1–10. Springer-Verlag, 1993.
7. J. F. Groote. Hiding propositional constants in BDDs. *Formal Methods in System Design*, 8:91–96, 1996.
8. J. Harrison. Binary decision diagrams as a HOL derived rule. *The Computer Journal*, 38:162–170, 1995.
9. J. N. Hooker. A quantitative approach to logical inference. *Decision Support Systems*, 4:45–69, 1988.
10. R. G. Jereslow. Computation-oriented reductions of predicate to propositional logic. *Decision Support Systems*, 4:183–197, 1988.
11. U. K. Ministry of Defence. The procurement of safety critical software in defence equipment. Interim Defence Standard 00-55, MOD Directorate of Standardization, Kentigern House, 65 Brown Street, Glasgow G2 8EX, UK, 1991.
12. M. Säflund. Modelling and formally verifying systems and software in industrial applications. Unpublished; available from the National Physical Laboratory, Teddington, Middlesex, TW11 0LW, UK, 1994.
13. G. Stålmarck. A proof theoretic concept of tautological hardness. Unpublished manuscript, 1994.
14. G. Stålmarck. System for determining propositional logic theorems by applying values and rules to triplets that are generated from Boolean formula. United States Patent number 5,276,897; see also Swedish Patent 467 076, 1994.
15. G. Stålmarck and M. Säflund. Modeling and verifying systems and software in propositional logic. In B. K. Daniels, editor, *Safety of Computer Control Systems, 1990 (SAFECOMP '90)*, pages 31–36, Gatwick, UK, 1990. Pergamon Press.

16. C. B. Suttner and G. Sutcliffe. The TPTP problem library. Technical Report AR-95-03, Institut für Infomatik, TU München, Germany, 1995. Also available as TR 95/6 from Dept. Computer Science, James Cook University, Australia, and on the Web.
17. T. Uribe and M. E. Stickel. Ordered Binary Decision Diagrams and the Davis-Putnam procedure. In J.-P. Jouannaud, editor, 1^{st} *International Conference on Constraints in Computational Logics*, volume 845 of *Lecture Notes in Computer Science*, pages 34–49, Munich, 1994. Springer-Verlag.