A tutorial on Stålmarck's proof procedure for propositional logic

Mary Sheeran and Gunnar Stålmarck

Prover Technology AB and Chalmers University of Technology, Sweden

Abstract. We explain Stålmarck's proof procedure for classical propositional logic. The method is implemented in a commercial tool that has been used successfully in real industrial verification projects. Here, we present the proof system underlying the method, and motivate the various design decisions that have resulted in a system that copes well with the large formulas encountered in industrial-scale verification. We also discuss possible applications in Computer Aided Design of electronic circuits.

To appear in the proceedings of FMCAD'98, published in Springer LNCS.

A tutorial on Stålmarck's proof procedure for propositional logic

Mary Sheeran and Gunnar Stålmarck

Prover Technology AB and Chalmers University of Technology, Sweden

Abstract. We explain Stålmarck's proof procedure for classical propositional logic. The method is implemented in a commercial tool that has been used successfully in real industrial verification projects. Here, we present the proof system underlying the method, and motivate the various design decisions that have resulted in a system that copes well with the large formulas encountered in industrial-scale verification. We also discuss possible applications in Computer Aided Design of electronic circuits.

1 Introduction

In the computer aided design of electronic circuits, a key function is tautology checking, that is testing whether a Boolean expression is true for all truth assignments of its variables. Tautology checking is used not only in hardware verification, but also in synthesis and optimisation. All known methods of tautology checking take time exponential in the size of the input formula, in the worst case. Since the problem is known to be co-NP complete [6], it seems unlikely that we can do any better than this in the worst case. But what about the formulas that actually arise in practice? In many cases, Binary Decision Diagrams (BDDs) and their variants work well, both for tautology checking and for other applications, such as model checking. A glance through the proceedings of the first conference on Formal Methods for CAD confirms that BDDs have become ubiquitous in hardware verification [23].

In this tutorial, we explain Stålmarck's method of tautology checking that may well rival BDDs [5] in some applications, but that is relatively unknown in the hardware verification community [12]. This patented method is implemented in a commercial tool that has been used in many industrial system verification projects [3]. Complex devices, such as engine management units or railway interlocking systems, are modelled in propositional logic, either directly or by translation from industry-standard formats. The required properties of the system are also expressed in propositional logic, and to verify the system is to check that the formula *system* $\rightarrow$ *properties* is a tautology. Often, the verification problem can be expressed as an inductive proof, and the base case and the step checked using Stålmarck's method. Many of these real-world verifications give rise to enormous formulas that could not be handled by current BDD packages. Groote has found Stålmarck's method to be very efficient compared to BDD-based methods and the Otter prover in the verification of the safety guaranteeing system at a particular Dutch railway station [11]. The largest formula encountered so far arose in railway interlocking; it had $350,000$ connectives and the log recording the proof (for later independent checking) was 780 megabytes long.

The exciting thing about Stålmarck's method is that it copes with such formulas with aplomb, provided that they are *easy* according to a proof-theoretic measure that we will discuss later. And what is even more surprising is that real-world problems do indeed give rise to large but easy formulas.

In this tutorial, we first briefly present two standard proof systems for propositional logic: Gentzen's cut-free sequent calculus and semantic tableaux. Next, we show how cut-free proofs are intrinsically redundant, and motivate a rather different proof method, in which we use relations on formulas, rather than just sets of formulas that are known to be true or false. The system KE and the Davis-Putnam procedure can be seen as special cases of this approach. Next, we add a new kind of rule, to give the Dilemma proof system. This is the system that underlies Stålmarck's method. Finally, we outline the Dilemma proof procedure, give some important complexity results, and discuss applications.

2 Necessary Background: Proof Systems

In order to understand why a proof procedure is efficient from a practical point of view, we *must* study the underlying proof system. A proof procedure consists of two parts: an inductive definition of the classical consequence relation (that is rules about what it means to be a tautology) and a related algorithm for generating proofs. The algorithm can really only be understood by showing how it relates to a particular way of defining the consequence relation. And the way in which one defines the consequence relation – the choice of underlying proof system – has a surprisingly large effect on the performance of the resulting algorithm. So the first step in designing an efficient proof procedure is to choose a suitable proof system. Let us review some standard proof systems for classical propositional logic, and their properties.

2.1 Gentzen's Sequent Calculus with Cut

Gentzen introduced the sequent calculus during the thirties in order to prove his *Hauptsatz*. It states that all proofs can be brought into a form without roundabouts, obeying the subformula property, so that a proof uses only concepts (and subformulas) from the formula to be proved. Proofs in this calculus contain expressions of the form $\{A_1, \ldots, A_n\} \vdash \{B_1, \ldots, B_m\}$, so called *sequents*, informally read as "if all of the A_is are true, then one of the B_js is true".

Proofs start from obviously valid sequents of the form $A \vdash A$, the axioms. Complex formulas are then built up in the sequents by applications of operational rules. The calculus also includes a rule for introducing new formulas, the thinning rule, and one rule that removes formulas, the cut rule. Thinning can be used to minimize the number of different sequents in proofs and hence, to reduce proof complexity when proofs are viewed as directed acyclic graphs, rather than trees. When presenting the rules, we write Γ, A for the set $\Gamma \cup \{A\}$.

Axiom

$$A \vdash A$$

Structural Rules

$$(\textit{Thinning}) \quad \frac{\Gamma \vdash \Delta}{\Gamma, \Theta \vdash \Delta, \Lambda} \qquad (\textit{Cut}) \quad \frac{\Gamma, A \vdash \Delta \quad \Gamma \vdash \Delta, A}{\Gamma \vdash \Delta}$$

Operational Rules

$$(\textit{Or-left}) \quad \frac{\Gamma, A \vdash \Delta \quad \Gamma, B \vdash \Delta}{\Gamma, A \vee B \vdash \Delta} \qquad (\textit{Or-right}) \quad \frac{\Gamma \vdash \Delta, A, B}{\Gamma \vdash \Delta, A \vee B}$$

$$(\textit{And-left}) \quad \frac{\Gamma, A, B \vdash \Delta}{\Gamma, A \wedge B \vdash \Delta} \qquad (\textit{And-right}) \quad \frac{\Gamma \vdash \Delta, A \quad \Gamma \vdash \Delta, B}{\Gamma \vdash \Delta, A \wedge B}$$

$$(\textit{Imp-left}) \quad \frac{\Gamma \vdash \Delta, A \quad \Gamma, B \vdash \Delta}{\Gamma, A \rightarrow B \vdash \Delta} \qquad (\textit{Imp-right}) \quad \frac{\Gamma, A \vdash \Delta, B}{\Gamma \vdash \Delta, A \rightarrow B}$$

$$(\textit{Neg-left}) \quad \frac{\Gamma \vdash \Delta, A}{\Gamma, \neg A \vdash \Delta} \qquad (\textit{Neg-right}) \quad \frac{\Gamma, A \vdash \Delta}{\Gamma \vdash \Delta, \neg A}$$

2.2 Removing Cut, and Gaining the Subformula Principle

The sequent calculus is complete, even if we remove the Cut rule. That is what Gentzen's Haupsatz says. All of the rules of the cut-free sequent calculus are elimination rules, so the system trivially obeys the *subformula principle*. Every proof uses only subformulas of the formula to be proved. No extraneous formulas or definitions are used, so that only concepts that were already there in the formula to be proved are used. The proof is direct, or as Gentzen put it, it is not roundabout [10]. As we shall see later, having the subformula principle allows us to place bounds on proof size, and this is very important in practice.

2.3 Removing Thinning

If, in addition, we take as axioms sequents of the form $\Gamma, A \vdash A, \Delta$, then we can add extra formulas using axioms, and the Thinning Rule becomes redundant and can be removed. The resulting system is essentially the same as Kleene's system G4 [15]. This system has been particularly important in automated deduction because it lends itself to a goal oriented proof search. One can start with the sequent that is to be proved and use the rules backwards, aiming to reach axioms or obviously unprovable sequents. This proof procedure works because all of the rules in G4 are *invertible*: the provability of the sequent below the line implies the provability of the sequents above the line. Note that the thinning rule is not invertible.

2.4 The Semantic Tableau Method

Smullyan's system of 'analytic tableaux' [22] is another classic proof method. Any valuation of a formula (an assignment of $\top$ (true) or $\perp$ (false)) to the propositional variables) must make the formula either true or false. So, for each connective, we examine the possible cases. If $A \wedge B$ is true, then A must be true and B must be true (the And rule). If $A \wedge B$ is false, then A is false or B is false and we explore both possibilities (the Not-And rule). This combination of the law of the excluded middle with the usual semantic interpretation of the connectives gives the following tableau rules for propositional logic:

$$\textit{(And)}\quad \frac{A \wedge B}{\begin{array}{c}A\\ B\end{array}} \qquad\qquad \textit{(Not-Or)}\quad \frac{\neg(A \vee B)}{\begin{array}{c}\neg A\\ \neg B\end{array}}$$

$$\textit{(Or)}\quad \frac{A \vee B}{A|B} \qquad\qquad \textit{(Not-And)}\quad \frac{\neg(A \wedge B)}{\neg A|\neg B}$$

$$\textit{(Impl)}\quad \frac{A \rightarrow B}{\neg A|B} \qquad\qquad \textit{(Not-Impl)}\quad \frac{\neg(A \rightarrow B)}{\begin{array}{c}A\\ \neg B\end{array}}$$

$$\textit{(Not-Not)}\quad \frac{\neg\neg A}{A}$$

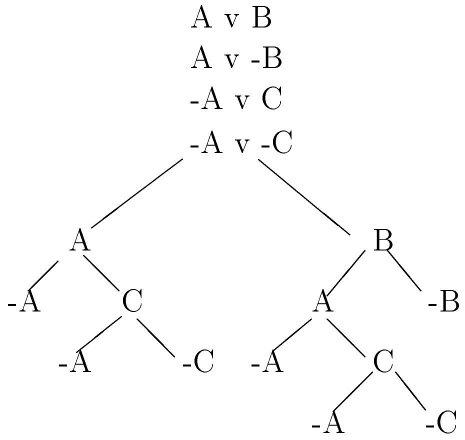
This method analyses a formula by progressively breaking it into its component parts, using these rules. The branching rules (Or, Impl and Not-And) are such that the disjunction of the formulas in the branches is a consequence of the formula above the line. Rules such as Not-Or that have two formulas below the line are just shorthand for a pair of rules, each giving a single formula. We start with the formula to be proved at the root of a tree and repeatedly apply the above rules. Along any path through the tree, we build up information about one possible valuation of the subformulas, by gathering a set that contains all of the formulas along that path. Think of this as being all of the formulas that we know to be true in a particular valuation. If we end up with an explicitly contradictory set, one containing both A and $\neg A$ for some formula A, then the exploration of that branch has failed to find a model (a setting of the propositional variables the makes the formula true). If all of the branches of the tree are contradictory in this way, then we know that there is no model of the formula, so it is contradictory and its negation is valid.

Semantic Tableaux systems were first introduced during the fifties, by Beth, Kanger, Hintikka and Shütte almost simultaneously [1, 14, 13, 18]. Because the system has only elimination rules, it trivially obeys the subformula principle. This has the important effect of placing a limit on the size of proofs in relation to the size of the formula to be proved. We write the size of formula A as $|A|$. It is the number of variable occurrences plus the number of connectives. The number of subformulas of a formula is the same as the size of the formula, and is also the maximum length of any path in a semantic tableau, since each step along the path adds one formula to a set of formulas.

2.5 The Intrinsic Redundancy of Cut-free Proofs

No matter what procedure is used to search for proofs in G4 or the tableau system, the search tree can grow explosively, even for simple commonly occuring examples. Such growth happens even for the smallest possible proof, so the problem is not in the procedure but in the cut-free nature of the proof system. D'Agostino, in his thesis [7], presents a small and enlightening example: the minimal tableau refutation (without Cut or Thinning) of the formula

$$(A \vee B) \wedge (A \vee \neg B) \wedge (\neg A \vee C) \wedge (\neg A \vee \neg C)$$



Each of the seven paths explored results in a contradiction. Note, however, that in the righthand side of the tree, we explore a part of the search space that has already been explored in the left subtree. After building the left subtree, we know that assuming that A is true leads to a contradiction, yet we repeat this search even after assuming that B is true. (Don't be misled by the picture into thinking that using graphs instead of trees might help! Our trees are really decorated at each node with the set of formulas along the path from the root, and each node in the above tree corresponds to a different set.) This kind of redundant pattern can be repeated inside the redundant subtrees, so that a combinatorial explosion results. The semantic tableaux rules given above don't really match the search space that we are trying to explore. We would like our refutation trees to be a better match with the search space. The solution is to put back in a form of cut, while keeping the subformula principle.

To understand this step, we must study the space that we are searching, and what rules and proofs look like.

2.6 Rules

Rules in tableau systems correspond to clauses in the defintion of true in a valuation. So, for example, if $A \vee B$ is true and A is false, then B is true. This and the other two elimination rules for $\vee$ are written

$$\frac{\begin{array}{c}A \vee B \equiv \top \\ A \equiv \bot\end{array}}{B \equiv \top} \qquad \frac{\begin{array}{c}A \vee B \equiv \top \\ B \equiv \bot\end{array}}{A \equiv \top} \qquad \frac{A \vee B \equiv \bot}{\begin{array}{c}A \equiv \bot \\ B \equiv \bot\end{array}}$$

The introduction rules for $\vee$ are

$$\frac{A \equiv \top}{A \vee B \equiv \top} \qquad \frac{B \equiv \top}{A \vee B \equiv \top} \qquad \frac{\begin{array}{c}A \equiv \bot \\ B \equiv \bot\end{array}}{A \vee B \equiv \bot}$$

Here, when analysing a connective $\circ$, we consider not only $A \circ B$ but also its immediate subformulas and their complements. This gives a different set of rules from the classic tableau rules that we have already seen.

We can extend this idea by considering not just truth values of subformulas, but also whether two formulas must have the same truth value. For example, if A and B have the same value, then $A \vee B$ also has that value, and if A and B have different values, then $A \vee B$ must be true.

$$\frac{A \equiv B}{\begin{array}{c}A \equiv A \vee B \\ B \equiv A \vee B\end{array}} \qquad \frac{A \equiv \neg B}{A \vee B \equiv \top}$$

By examining each connective in turn, we can generate a large set of propagation rules. We include only the *proper* rules. In general, the rules for a connective $\circ$ look like

$$\frac{F_1 \equiv G_1, \ldots F_n \equiv G_n}{F \equiv G}$$

where each $F_i, G_i \in \{A, B, A \circ B, \neg A, \neg B, \top, \bot\}$. A rule is *proper* if and only if

$$\begin{array}{r}\{F_1 \equiv G_1, \ldots F_n \equiv G_n\} \models F \equiv G \\ \{F_1 \equiv G_1, \ldots F_n \equiv G_n\} \not\models \bot \\ \{F_1 \equiv G_1, \ldots F_n \equiv G_n\} - \{F_i \equiv G_i\} \not\models F \equiv G\end{array}$$

2.7 The Systems KE/I and KE

The subset of the proper rules for which $G, G_i \in \{\top, \bot\}$ corresponds to the introduction and elimination rules of the propositional fragment of the system KE/I. This proof system was introduced by Mondadori, and has been further studied by D'Agostino [17, 7]. Using just the introduction and elimination rules does not give a system that is complete for propositional logic. However, adding a single branching rule, the principle of bivalence,

$$\textit{(PB)} \quad \frac{}{A \equiv \top | A \equiv \bot}$$

gives a proof system that is complete for propositional logic and that does not suffer from the kind of redundancy that we illustrated earlier. We have put back the cut rule! A is restriced to be a subformula of the formula to be proved, so we again have the subformula principle (with the associated bound on proof size) and because of this the PB rule is known as an *analytic* form of cut. Indeed, just the elimination rules plus the PB rule form a complete system, KE, which is extensively studied in D'Agostino's thesis [7].

The formula $(A \vee B) \wedge (A \vee \neg B) \wedge (\neg A \vee C) \wedge (\neg A \vee \neg C)$, for which we earlier showed the minimal tableau refutation, gives the following KE-refutation.

```
          A v B
          A v -B
          -A v C
          -A v -C
        /         \
    -A              - -A
    B               C
    -B              -C
```

Let us start from KE and extend the language of formulas to include generalised conjunction and disjunction. If we now add two simplification rules (affirmative-negative and subsumption), we get a proof procedure that is equivalent to the well-known Davis-Putnam procedure [9] (in the version of [8]) for formulas in conjunctive normal form (CNF). So, KE can be seen as a generalisation of Davis-Putnam that does not require reduction to CNF.

The proof system underlying Stålmarck's method uses the larger set of propagation rules in which the G and G_i are no longer constrained to be in $\{\top, \bot\}$. It is no longer sufficient to maintain sets of formulas that are known to be true or false; we must also maintain information about sets of formulas that are known to have the same value. To do this, we introduce formula relations.

2.8 Formula Relations

The complement of a formula A, written A', is B if $A = \neg B$ and is $\neg A$ otherwise. Let $S(X)$ be the set containing all the subformulas of X (including $\top$) and their complements.

A *formula relation* $\sim$ on X is an equivalence relation with domain $S(X)$, with the constraint that if $A \sim B$ then $A' \sim B'$. If $A \sim B$, that means that A and B are in the same equivalence class and must have the same truth value. Working with $S(X)$, which includes the complements of subformulas of X, allows us to encode both equalities and inequalities between subformulas. $A \not\sim B$ is encoded as $A' \sim B$.

We write $R(A \equiv B)$ for the least formula relation containing R and relating A and B. The smallest formula relation is the identity relation on $S(X)$, written X^+. It simply places each element of $S(X)$ in its own equivalence class. Rather more interesting is $X^+(X \equiv \top)$, which we abbreviate to $X^\top$. This *partial valuation* will later be the starting point when we attempt to refute X.

Example Let C and D be propositional variables, and $X = C \wedge D$. Then,

$$X^\top = \{[C \wedge D, \top], [C], [D], [\neg(C \wedge D), \bot], [\neg C], [\neg D]\}$$

where equivalence classes are shown using square brackets.

2.9 Applying Rules to Formula Relations

Each schematic rule

$$\frac{F_1 \equiv G_1, \ldots F_n \equiv G_n}{F \equiv G}$$

corresponds to a partial function on formula relations. It takes a formula relation R in which each $F_i \equiv G_i$ for i in $\{1..n\}$, and returns the larger $R(F \equiv G)$.

Continuing the previous example: applying the two $\wedge$-elimination rules

$$\frac{A \wedge B \equiv \top}{A \equiv \top} \qquad \frac{A \wedge B \equiv \top}{B \equiv \top}$$

in sequence to $X^\top = \{[C \wedge D, \top], [C], [D], [\neg(C \wedge D), \bot], [\neg C], [\neg D]\}$ gives R_1 and then R_2.

$$R_1 = \{[C \wedge D, C, \top], [D], [\neg(C \wedge D), \neg C, \bot], [\neg D]\}$$
$$R_2 = \{[C \wedge D, C, D, \top], [\neg(C \wedge D), \neg C, \neg D, \bot]\}$$

R_2 contains exactly two equivalence classes, and no further simple rules are applicable to it. It gives a *model* of $C \wedge D$, which therefore cannot be refuted.

If, instead, we take $Y = C \wedge \neg C$ and apply the two $\wedge$-elimination rules to $Y^\top$, we get R_3 and R_4.

$$R_3 = \{[C \wedge \neg C, C, \top], [\neg(C \wedge \neg C), \neg C, \bot]\}$$
$$R_4 = \{[C \wedge \neg C, C, \top, \neg(C \wedge \neg C), \neg C, \bot]\}$$

R_4 groups all of the formulas in its domain into a single equivalence class, and so is the largest formula relation on Y. It is explicitly contradictory since it (several times) places a formula and its complement in the same equivalence class. So, this sequence of rule applications constitues a refutation of $C \wedge \neg C$. From the assumption that $C \wedge \neg C$ is true, we have derived a contradiction.

3 The Dilemma Proof System

We are now ready to present the Dilemma proof system that underlies Stålmarck's method. The large set of *proper* rules that we have introduced allows us to reach conclusions that would require branching in systems with a smaller set of simple rules. Since the lengths of paths in refutation graph proofs are bounded by the size of the formula to be proved, it is branching that is the critical factor for complexity speed up.

Of course branching cannot be avoided altogether, since the simple rules are not complete for propositional logic. We introduce a special 'branch and merge' rule called the Dilemma rule. The Dilemma proof system is just this rule plus the simple rules.

3.1 The Dilemma Rule

The Dilemma rule is pictured as

$$
\begin{array}{cc}
\multicolumn{2}{c}{R} \\
\hline
R(A \equiv B) & R(A \equiv \neg B) \\
(derivation) & (derivation) \\
R_1 & R_2 \\
\hline
\multicolumn{2}{c}{R_1 \sqcap R_2}
\end{array}
$$

Given a formula relation R, we apply the Dilemma rule by choosing A and B from different (and non-complementary) equivalence classes in R. We make two new Dilemma derivations starting from $R(A \equiv B)$ and $R(A \equiv \neg B)$, to give R_1 and R_2 respectively. Finally, we intersect R_1 and R_2, to extract the conclusions that are common to both branches. If an explicitly contradictory formula relation is always extended to the relation with a single equivalence class, then the intersection operation is simply set intersection ($\cap$) of the relations viewed as sets of pairs. In practice, we stop a derivation as soon as two formulas A and $\neg A$ are placed in the same equivalence class. Then $R_1 \sqcap R_2$ is defined to be R_2 if R_1 is explicitly contradictory, R_1 if R_2 is explicitly contradictory, and $R_1 \cap R_2$ otherwise.

Note that R is a subset of both R_1 and R_2, since a derivation only adds to the relation. This means that R is a subset of $R_1 \sqcap R_2$, and the rule is sound. The rule can be seen as the combination of a cut (the branch) with two (backwards) applications of thinning, from $R_1 \sqcap R_2$ to R_1, and from $R_1 \sqcap R_2$ to R_2. In the context in which these thinning applications appear, they are invertible. If we can refute $R_1 \sqcap R_2$, then we have refuted R, so it must be possible to refute both R_1 and R_2. If we omit the use of thinning, and simply refute R_1 and R_2 separately, then the two proofs are likely to have much in common (since R_1 and R_2 have R in common). Using thinning avoids this repetition, and so has an effect akin to that of having lemmas.

3.2 Dilemma Derivations

The following three clauses define what it means to be a Dilemma derivation, and also the related notion of proof depth.

1. *Simple Rules.* If the application of one of the simple rules to R_1 gives R_2, then $\Pi = R_1 R_2$ (R_1 followed by R_2) is a Dilemma derivation of R_2 from R_1. We write that assertion as $\Pi : R_1 \Rightarrow R_2$. If none of the simple rules applies to R, we say that R itself is a derivation of R from R. In both cases, the proof depth, written $depth(\Pi)$, is zero.
2. *Composition.* If $\Pi_1 : R_1 \Rightarrow R_2$ and $\Pi_2 : R_2 \Rightarrow R_3$, then we can compose the proofs: $\Pi_1 \Pi_2 : R_1 \Rightarrow R_3$. The proof depth of the composition is defined to be $max(depth(\Pi_1), depth(\Pi_2))$.
3. *Dilemma Rule.* If $\Pi_1 : R(A \equiv B) \Rightarrow R_1$ and $\Pi_2 : R(A \equiv \neg B) \Rightarrow R_2$, then

$$
\begin{array}{c}
R \\ \hline
\Pi_1 \qquad \Pi_2 \\ \hline
R_1 \sqcap R_2
\end{array}
$$

   is a derivation from R to $R_1 \sqcap R_2$ and its depth is $max(depth(\Pi_1), depth(\Pi_2)) + 1$

Proofs built using these rules have a series-parallel shape. The depth of a proof is the same as the maximum number of simultaneously open branches.

Example

$$
\begin{array}{c}
R_1 \\ R_2 \\ R_3 \\ \hline
\begin{array}{cc}
\begin{array}{c}
R_3(A \equiv B) \\ R_4 \\ R_5 \\ \hline
\begin{array}{cc}
R_6(C \equiv D) & R_6(C \equiv \neg D) \\
R_7 & \\
R_8 & R_9
\end{array} \\ \hline
R_8 \sqcap R_9
\end{array}
&
\begin{array}{c}
R_3(A \equiv \neg B) \\ \\ R_{10} \\ \\ \\ \\ R_{11}
\end{array}
\end{array} \\ \hline
(R_8 \sqcap R_9) \sqcap R_{11}
\end{array}
$$

This proof has depth 2. The *size* of a Dilemma proof Π, denoted $|\Pi|$, is the number of occurrences of formula relations in Π. This proof has size 16.

To make the link back from derivations involving formula relations to proofs about formulas, note that the derivation of a contradictory formula relation from $X^{\top}$ constitutes a refutation of the formula X. Similarly, we can check whether or not X is a tautology by attempting to refute $X^{+}(X \equiv \bot)$.

3.3 Proof hardness

A formula relation R is k-easy if and only if there is a derivation $\Pi : R \Rightarrow S$ such that S is explicitly contradictory and $depth(\Pi) \leq k$. A relation R is k-hard if and only if there is no derivation $\Pi : R \Rightarrow S$ such that S is explicitly contradictory and $depth(\Pi) < k$. A relation R has *hardness degree* k, $h(R) = k$, if and only if R is k-easy and k-hard.

The formula $(A \wedge (B \vee C)) \rightarrow ((A \wedge B) \vee (A \wedge C))$ has hardness degree 0, while reversing the direction of the implication gives a formula of hardness degree 1. Harrison's paper on Stålmarck's algorithm as a HOL derived rule lists many 1- and 2-hard problems with performance statistics for Harrison's implementation of the method [12]. Many of the examples are circuits taken from the set of examples presented at the IFIP International Workshop on Applied Formal Methods for Correct VLSI Design, held at IMEC in 1989.

In practice, it turns out that real industrial verification problems often produce formulas with hardness degree 0 or 1. Why is this? One reason must be that the Dilemma system has a large set of propagation rules, reducing the need for branching. The only other explanation that we can offer is the observation that systems that have been designed by one person, or in which one person understands and can argue informally for the correctness of the system tend to result in easy formulas. Systems in which components are coupled together in a relatively uncontrolled way, so that behaviour is hard to predict by informal analysis, are also hard to analyse formally. These empirical observations indicate that it is a good idea to introduce design rules that improve verifiability. Indeed, the main supplier of railway interlocking software in Sweden has made an important move in this direction, by introducing software coding rules that guarantee the verifiability of the resulting interlocking control software, see section 6.1. Figuring out how to place constraints on the designer so that it is easy to reason about the correctness of his products seems likely to be an important area of research in formal methods.

4 From proof system to proof procedure

We have now studied the Dilemma proof system. How do we turn it into an efficient proof procedure? The first step is to find a good data structure for representing formulas. The second is to provide an efficient algorithm that searches exhaustively for shallow Dilemma proofs.

4.1 Triplets

For ease of manipulation, compound subformulas are represented by triplets, having the form $x : y \circ z$. The triplet variable x represents the compound formula obtained by applying a binary operator $\circ$ to the triplet operands y and z. x represents a subformula, and y and z are literals, that is either variables (real or triplet variables) or negated variables.

Example The formula $(A \wedge (B \vee C)) \rightarrow ((A \wedge B) \vee (A \wedge C))$ is reduced to triplets as follows:

$$t : \top$$
$$a : A$$
$$b : B$$

$$c : C$$
$$d : b \vee c$$
$$e : a \wedge d$$
$$f : a \wedge b$$
$$g : a \wedge c$$
$$h : f \vee g$$
$$i : e \to h$$

The saturation algorithm

A relation R is k-saturated if and only if for every Dilemma derivation $\Pi : R \Rightarrow S$ with $depth(\Pi) \leq k$, it holds that $R = S$. In other words, proofs of depth k or less add no new equivalences between subformulas.

The k-saturation procedure exhaustively searches for a proof of depth k. If a relation R has hardness degree k, then $saturate(R, k)$ must be explicitly contradictory, and k-saturation finds a disproof of R. The procedure is defined recursively. 0-saturation applies the propagation rules to a relation until no more rules are applicable. It chooses a compound subformula, applies a related simple rule and then continues to apply simple rules on those triplets whose variables were affected by the result of the first rule. The process continues until no further simple rules can be applied.

The following pseudo-code fragment presents 0-saturation.

```
saturate(R,0) =

Q := Compound(R)
while non-empty(Q)
 do
  remove some q from the set Q
  if contradictory(R(q))
     then return R(q)
     else Q := Q union parents(R(q)-R)
          R := R(q)
 od
return R
```

The set $Compound(R)$ is the set of all the compound subformulas in the domain of R, and this is the initial value of the pool of subformulas to be processed. We chose one element q from the pool and apply to R a simple rule related to q. $R(q)$ is $R(F \equiv G)$ if there is a simple rule whose premises all involve formulas from the set containing q and its immediate subformulas and their complements, and whose conclusion is $F \equiv G$. If no such simple rule applies, then $R(q)$ is just R. The application of a rule related to q in this way leads to the discovery of a set of new equivalences, $R(q) - R$. We add back to the pool those subformulas that contain as an immediate subformula one (or more) of the formulas involved in these equivalences - the 'parents' of the subformulas about which we have just discovered something new. These are the triplets that might be affected by the new information gained in applying the rule. Having thus (possibly) augmented the pool, we update R with the new equivalences and repeat, until the pool is empty. At that point, no more simple rules apply to R and 0-saturation is complete.

$(k+1)$-saturation is defined in terms of branching and k-saturation. The version presented here branches on the truth or falsity of a subformula, rather than on equivalence between two arbitrary subformulas. This is the version that is currently implemented. A better strategy might perhaps be to try to merge the two largest equivalence classes.

```
saturate(R,k+1) =

repeat
  L : = Sub(R)
  R' := R
  while non-empty(L)
   do
    remove some l from L
    R1 := saturate(R'(l equiv FALSE),k)
    R2 := saturate(R'(l equiv TRUE) ,k)
    if contradictory(R1) and contradictory(R2)
       then return R1 union R2
       else if contradictory(R1)
               then R' := R2
               else if contradictory(R2)
                       then R' := R1
                       else R' := R1 intersect R2
   od
until R' = R
return R
```

The set $Sub(R)$ contains the subformulas of R, both variables and compound subformulas. We repeatedly branch on each subformula in turn until there are no further consequences in the form of new equivalences. Of course, one does not use $(k+1)$-saturation until after k-saturation has been performed, since branching should be minimised. Information gained during k-saturation is available during the subsequent $(k+1)$ saturation. It is the continuous gathering of information (in the form of equivalences) that distinguishes the algorithm from both breadth first search and interative deepening.

Note how the saturation algorithm propagates information both upwards and downwards in the syntax tree of the formula. In this, it is rather relational.

After the development of the algorithm described here, Kunz et al independently developed *recursive learning*, a method of solving the Boolean satisfiability problem, with applications in digital test generation [16]. The method has much in common with that presented here, and also discovers logical relationships between subformulas of a formula (or nodes in a circuit).

5 Complexity Results

An upper bound on proof lengths for Dilemma (and obviously for KE/I that is a subsystem of Dilemma) in relation to hardness degree is $n^{k+1} + n^k$, where n is the size of the formula of hardness degree k to be proved. This bound assumes that the Dilemma rule branches on the truth or falsity of a subformula

A lower bound is 2^k for analytic KE/I and $2^{k/2}$ for Dilemma. The difference is not due to the extension to formula relations but to the extension to series parallel graphs. It should be noted that the upper bound is "under exponential" over the lower bound.

Is Dilemma superior to analytic KE/I? There are infinite formula sequences for which the hardness of each F_i is less than a constant k in Dilemma, but grows logarithmically in analytic KE/I. It is interesting to note that the hardness degree for analytic KE/I and Dilemma are instead related linearly if either the formula relations are replaced by sets or if the Dilemma rule is replaced by analytic cut, but in the relational version. The relational version seems to open opportunities to remove repetitions by the invertible thinning used in the Dilemma rule.

Using the saturation procedure, the time required to search exhaustively for a proof of depth k of a formula A is bounded by $O(n^{2k+1})$, where n is the size of A. Note that the upper bound on proof search is not more than the square of the upper bound on proof length, and so it is still "under exponential" related to the shortest possible proof.

6 Industrial applications

In Sweden, Adtranz is the main supplier of railway interlocking software. Tools based on Stålmarck's method have been used to find potential runtime errors, so called double values, in railway interlocking systems since 1990. The checking is done entirely automatically, and the tools are incorporated into Adtranz' development environment. Adtranz report a 90% reduction in time spent on testing, and a reduction in overall development cost of about 15%. Here, we describe in a little more detail a different kind of verification project, also from the railway field [4].

6.1 Safety Analysis in Railway Interlocking Software

The railway interlocking software developed at Adtranz is written in a domain-specific synchronous declarative language called Sternol. That this is (rather surprisingly) the case has undoubtedly aided verification efforts in the area! It is also one of the reasons why we think that it is fair to say that many of the systems verified using tools based on Stålmarck's method have been *hardware-like*. From our point of view, there is very little difference between a Sternol program and a circuit.

A railyard implementation consists of a set of specialised instances of generic railyard objects such as points and signals, one for each physical object in the yard. Specialisation involves setting individual variables, for example the number of lights on a signal object. A control program containing code for each type of generic object then 'runs' the railyard implementation. On each cycle, every object receives some inputs and sets some of its variables depending on the values of boolean guards. Some of the variables are outputs of the object. Inputs and outputs are either booleans or integers. (The double value checking referred to earlier checks that this process is actually deterministic and does not try to set a variable to two different values.) The *same* control program runs *every* Sternol-based yard in Sweden.

A simple way to check safety properties for a railyard is to analyse for each object with safety critical outputs all the objects that follow it in the direction of travel. When a particular safety critical output is emitted by the first object, a certain number of the following objects must be unoccupied, and if they are points, they must point in the direction of travel. Such requirements are easily translated into propositional logic with arithmetic (PA). (The commercial implementations of Stålmarck's method incorporate a weak form of integer arithmetic. Simple rules for the arithmetic operators are added to the propositional rules described here.) The Sternol program for the yard itself encodes a synchronous finite state machine with boolean and arithmetic variables. It can be automatically translated to a PA formula expressing initialisation and the state transition relation. (This kind of translation is routinely used in hardware verification.) Proving (or disproving) the requirements is then straightforward.

This simple but effective verification method has been used in earnest in several projects. In the validation of the interlocking software for the Madrid Subway Station Lago, it was found that some points could be in a neutral position, rather than being locked in the correct position. Such an error might cause minor damage to train and track and is not considered very safety critical. However, the error was corrected. A similar verification was carried out in 1997 for a larger railyard in Finland. There, a serious safety critical error was found, which might have led to a derailment. The error is currently being corrected.

The small Madrid verification produced formulas containing around $36,000$ triplets and took 40 man-hours of work in total. The largest yard analysed so far resulted in a formula containing about half a million triplets.

Based on their experience in performing this kind of formal verification, Adtranz have introduced coding rules for Sternol that guarantee the verifiability of the resulting interlocking control software. These rules constrain the programmer, forcing him to construct hierarchical state machines in a particular style. For example, if resources are claimed in a particular order, then they must be released in reverse order. The result of following these guidelines has been that all formulas generated from Sternol programs since their introduction have been of hardness degree 1 or less, and so quickly proved.

Using Stålmarck's method in CAD

Stålmarck's method provides an efficient way to do tautology checking on large formulas. We expect it to be applicable in CAD. Some of the industrial verification problems that have been tackled using Stålmarck's method are very similar to post hoc hardware verification of systems built using boolean operations and arithmetic. Typically, systems are modelled as synchronous boolean automata. For example, the reverse flushing control in a nuclear power station's cooling system has been verified [24]. There seems to be no reason why circuits should not be verified in the same style, and indeed the Swedish consulting company LUTAB has had some projects in this area.

In an assignment from the Swedish Defence Material Administration, LUTAB verified the hardware and software controlling the landing gear of a Saab military aircraft. Both the hardware and software were modelled manually in propositional logic. Only the electronic part of the hardware (operating in parallel with the software) was modelled, so valves and relays were not covered. The hardware model was based on the functional schematics of the actual implementation. The combined system controlling the opening and closing of the doors to the compartment in which the landing gear is stowed, as well as the lowering of the landing gear, was modelled as a hierarchy of modules. Using the manually constructed system model, it was possible to ask questions like "can the landing gear be lowered at the same time as the doors are closed?". (It is important to check such properties; there have been cases where functional test on a physical test rig has revealed this kind of error and the damage that it can cause.) The analysis checked a set of safety requirements on the system. It also covered failure modes, and so could analyse the behaviour of the system under a single fault, two simultaneous faults, and so on. In those cases, countermodels to the safety requirements were generated and analysed.

The modelling took LUTAB one man-month to do. A lot of that time was spent modelling integer values (which at that time were not supported in the implementation of Stålmarck's method used) by means of boolean variables. Similar projects have since been undertaken for civil aircraft. More recent work in the area is described in [25].

Just as with BDDs, some classes of circuits have proved difficult to verify using Stålmarck's method. Multipliers, in particular, give rise to hard formulas, and the hardness grows with circuit size. We have been experimenting with ways to systematically reduce proof hardness for regular circuits by adding definitions of fresh variables to the system formula – something akin to lemmas. The trick is to know exactly what new definitions to add. Here, we benefit from the fact that we describe circuits not directly in propositional logic but in the functional programming language Haskell, in a combinator-based style that has its roots in μFP and Ruby [2]. We generate propositional logic formulas for instances of generic circuits by symbolic evaluation. The combinator style makes it easier to discover where new definitions should be added. We are guided by the fact that the proof of a circuit instance is closely related to the shape of the circuit. Using this approach, large combinational multipliers have been verified [20]. We are developing novel methods to verify sequential circuits.

7 Summary

We have presented Stålmarck's patented proof procedure for propositional logic. The underlying Dilemma proof system is efficient for two reasons.

1. *Efficient propagation.* It provides efficient propagation of information about subformulas of the formula to be refuted. This efficiency of propagation comes both from the large set of introduction and elimination rules and from the fact that relations are used to maintain equivalences between subformulas.
2. *Series-parallel graphs instead of trees.* The Dilemma rule recombines the results of the two sides of a branch. It is a combination of cut and an invertible form of thinning. This avoids repetition in proofs.

The proof procedure itself is efficient because of the careful design of the k-saturation algorithm, which guarantees to find short proofs if they exist.

The method has been used to perform industrial-strength formal verification of many hardware-like systems. The question of whether or not it can be applied in practice in CAD remains to be answered. We hope that this paper will stimulate others to join the quest for an answer.

Acknowledgements

. Thanks to Koen Claessen for his careful reading of earlier drafts and to the members of the Formal Methods group at Chalmers, with whom we have had many enjoyable discussions. Sheeran's work at Prover Technology is partly funded by the EU project SYRF (Synchronous Reactive Formalisms).

References

1. E.W. Beth: Semantic entailment and formal derivability. Mededelingen der Kon. Nederlandes Akademie van Wetenschappen. Afd. letterkunde, n.s., 18, 309-342, Amsterdam, 1955.
2. P. Bjesse, K. Claessen, M. Sheeran and S. Singh: Lava: Hardware Design in Haskell. Proc. Int. Conf. on Functional Programming, ACM Press, 1998.
3. A. Borälv: The industrial success of verification tools based on Stålmarck's method. Proc. 9^{th} Int. Conf. on Computer Aided Verification, Springer-Verlag LNCS vol. 1254, 1997.
4. A. Borälv and G. Stålmarck. Prover Technology in Railways, In Industrial-Strength Formal Methods, Academic Press, 1998.

5. R. Bryant: Graph-Based Algorithms for Boolean Function Manipulation. IEEE Trans. Comp., vol. c-35, no.8, 1986.
6. S.A. Cook: The complexity of theorem-proving procedures. In Proc. 3rd ACM Symp. on the Theory of Computing, 1971.
7. M. D'Agostino: Investigation into the complexity of some propositional calculi. D. Phil. Dissertation, Programming Research Group, Oxford University, 1990.
8. M. Davis, G. Logemann and D. Loveland: A machine program for theorem proving. Communications of the ACM, 5:394-397, 1962. Reprinted in [21].
9. M. Davis and H. Putnam: A computing procedure for quantification theory. Journal of the ACM, 7:201-215, 1960. Reprinted in [21].
10. G. Gentzen: Untersuchungen über das logische Schliessen. Mathematische Zeitschrift, 39, 176-210, 1935. English translation in The Collected Papers of Gerhard Gentzen, Szabo (ed.), North-Holland, Amsterdam, 1969.
11. J.F. Groote, J.W.C. Koorn and S.F.M. van Vlijmen: The Safety Guaranteeing System at Station Hoorn-Kersenboogerd. Technical Report 121, Logic Group Preprint Series, Utrecht Univ., 1994.
12. J. Harrison: The Stålmarck Method as a HOL Derived Rule. Theorem Proving in Higher Order Logics, Springer-Verlag LNCS vol. 1125, 1996.
13. J.K.J. Hintikka: Form and content in quantification theory. Acta Philosophica Fennica, VII, 1955.
14. S. Kanger: Provability in Logic. Acta Universitatis Stockholmiensis, Stockholm Studies in Philosopy, 1, 1957.
15. S. C. Kleene: *Mathematical Logic*. John Wiley and Sons Inc., New York, 1967.
16. W. Kunz and D.K. Pradhan: Recursive Learning: A New Implication Technique for Efficient Solutions to CAD-problems: Test, Verification and Optimization. IEEE Trans. CAD, vol. 13, no. 9, 1994.
17. M. Mondadori: An improvement of Jeffrey's deductive trees. Annali dell'Universita di Ferrara; Sez III; Discussion paper 7, Universita di Ferrara, 1989.
18. K. Schütte: *Proof Theory*, Springer-Verlag, Berlin, 1977.
19. G. Stålmarck: A system for determining propositional logic theorems by applying values and rules to triplets that are generated from a formula, 1989. Swedish Patent No. 467 076 (approved 1992), U.S. Patent No. 5 276 897 (approved 1994), European Patent No. 0403 454 (approved 1995).
20. M.Sheeran and A. Borälv: How to prove properties of recursively defined circuits using Stålmarck's method. Proc. Workshop on Formal Methods for Hardware and Hardware-like systems, Marstrand, June 1998.
21. J. Siekman and G. Wrightson (editors): *Automation of Reasoning*. Springer-Verlag, New York, 1983.
22. R.M. Smullyan: *First Order Logic*. Springer, Berlin, 1969.
23. M. Srivas and A. Camilleri (editors): Proc. Int. Conf. on Formal Methods in Computer-Aided Design. Springer-Verlag LNCS vol. 1146, 1996.
24. M. Säflund: Modelling and formally verifying systems and software in industrial applications. Proc. second Int. Conf. on Reliability, Maintainability and Safety (ICRMS '94), Xu Ferong (ed.), 1994.
25. O. Åkerlund, G. Stålmarck and M. Helander: Formal Safety and Reliability Analysis of Embedded Aerospace Systems at Saab. Proc. 7^{th} IEEE Int. Symp. on Software Reliability Engineering (Industrial Track), IEEE Computer Society Press, 1996.