Formal Methods in System Design, 16, 23–58 (2000)
© 2000 Kluwer Academic Publishers. Manufactured in The Netherlands.

# A Tutorial on Stålmarck's Proof Procedure for Propositional Logic

MARY SHEERAN mary@prover.com
GUNNAR STÅLMARCK gunnar@prover.com
*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.

**Keywords:** propositional logic (Boolean logic), tautology checking (propositional theorem proving), Dilemma proof system, *k*-saturation, proof depth, tautological hardness, proof complexity, industrial applications of formal methods

## 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 [9], 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 two conferences on Formal Methods for CAD confirms that BDDs have become ubiquitous in hardware verification [15, 33].

In this tutorial, we explain Stålmarck's method of tautology checking that may well rival BDDs [8] in some applications, but that is relatively unknown in the hardware verification community [18]. This patented method is implemented in a commercial tool that has been used in many industrial system verification projects [4]. 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 in the verification of the safety guaranteeing system at a particular Dutch railway station [17]. The largest formulas encountered so far in industrial verification using Stålmarck's method have arisen in railway interlocking. It is not unusual to generate formulas with half a million connectives.

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 important classes of real-world problems do indeed give rise to large but easy formulas.

Let us first give a brief and informal description of the method, before beginning a deeper analysis of it.

## 2. Stålmarck's method in brief

Stålmarck's proof method for propositional logic can be understood in various different ways. Here, we first present the algorithm very briefly. Our intention is to give the reader some intuition about the operation of the algorithm, while whetting his appetite for the deeper study that follows. The presentation in this section is much influenced by an early paper by Stålmarck and Säflund [35].

There is a straightforward translation from formulas in propositional logic (say with negation, conjunction, disjunction and implication) to formulas built from only implication and $\bot$ (false). We repeatedly apply the following transformations:

$$\begin{array}{lll} A \vee B & \textit{to} & \neg A \rightarrow B \\ A \wedge B & \textit{to} & \neg(A \rightarrow \neg B) \\ \neg\neg A & \textit{to} & A \\ \neg A & \textit{to} & A \rightarrow \bot \end{array}$$

Let such an implication formula have propositional variables $a_1, \ldots, a_n$, and compound subformulas $B_1, \ldots, B_k$. $B_k$ is $A$ itself, and $B_i$ is $C_i \rightarrow D_i$, where $C_i$ and $D_i$ are subformulas of $A$. We invent a new name $b_i$ (different from each $a_j$) for each compound subformula $B_i$. Thus $b_i$ is the variable representing the formula $B_i$, and we write $rep(B_i) = b_i$. A propositional variable represents the formula that is just that variable: $rep(a_i) = a_i$.

Now, the formula $A$ can be represented by the set of *triplets*

$$\{(b_1, rep(C_1), rep(D_1)), \ldots, (b_k, rep(C_k), rep(D_k))\}$$

where a triplet $(x, y, z)$ is an abbreviation for $x \leftrightarrow (y \rightarrow z)$. We treat $\bot$ as a special case of a propositional variable, and write it as 0 in triplets. We write $\top$ (true) as 1.

*Example 1.* The formula $p \rightarrow (q \rightarrow p)$ becomes

$$\begin{array}{l} (b1, q, p) \\ (b2, p, b1) \end{array}$$

where $b2$ represents the formula itself.

To prove a formula valid, we assume it to be false and try to derive a contradiction using either *simple rules* or a branching rule called the *dilemma rule*.

*Simple rules.* A simple rule takes a *triggering triplet* and derives new information about its variables. For example, we know that if $y \to z$ is false, then $y$ must be true and $z$ false. We write this rule as

$$(r1) \quad \frac{(0, y, z)}{y/1 \quad z/0}$$

A *terminal* triplet is one that is contradictory. For example, the triplet $(1, 1, 0)$ is terminal because $1 \to 0$ cannot be true. The only other terminal triplets are $(0, x, 1)$ and $(0, 0, x)$.

Applying a rule to an element of a set of triplets gives a new set of triplets into which we substitute the newly calculated variable instantiations. The new set need not contain the triggering triplet, as a triplet can only be triggered once, and a triggered triplet cannot be terminal.

*Example 2.* The formula $p \to (q \to p)$ gave triplets

$$(b1, q, p)$$
$$(b2, p, b1)$$

Assume $b2$ (which corresponds to the formula itself) to be false and apply rule $r1$ to the triplet $(0, p, b1)$, dropping that triplet from the set:

$$(b1, q, p)$$
$$(0, p, b1) \Rightarrow$$
$$(b1, q, p)[p/1, b1/0] \Rightarrow$$
$$(0, q, 1)$$

The single triplet that results is terminal. Since the assumption that $p \to (q \to p)$ is false gives a terminal triplet, we conclude that the formula is *valid*.

We now list the remaining 6 simple rules:

$$(r2) \quad \frac{(x, y, 1)}{x/1} \qquad (r3) \quad \frac{(x, 0, z)}{x/1}$$

$$(r4) \quad \frac{(x, 1, z)}{x/z} \qquad (r5) \quad \frac{(x, y, 0)}{x/\neg y}$$

$$(r6) \frac{(x, x, z)}{x/1 \quad z/1} \qquad (r7) \quad \frac{(x, y, y)}{x/1}$$

Note that in rules $r4$ and $r5$ we gain information not about the exact value of a variable, but about the equality or inequality of two variables. The reader might like to check that a triggered triplet is never terminal.

The simple rules alone are not complete. There are formulas that cannot be proved with just these rules. We need some form of branching. (Later, we will delve more deeply into proof systems, and will see more proof systems that consist of a set of simple rules and a branching rule that gives completeness.)

*Dilemma rule*

$$\begin{array}{cc} \multicolumn{2}{c}{T} \\ \hline T[x/1] & T[x/0] \\ D_1 & D_2 \\ U[S_1] & V[S_2] \\ \hline \multicolumn{2}{c}{T[S]} \end{array}$$

$D_1$ and $D_2$ are derivations (or proofs). $D_1$ starts from the set of triplets $T$ and the assumption that $x$ is true. $D_2$ starts from $T$ and the assumption that $x$ is false. If one of these derivations gives a terminal triplet, then the result of applying this rule is the result of the other derivation. If neither $D_1$ nor $D_2$ leads to a contradiction, then the resulting substitution (or variable instantiation) $S$ is the intersection of $S_1$ and $S_2$. Any information gained both from assuming that $x$ is true and from assuming that it is false must hold independent of the value of $x$.

*Example 3.* We show the formula

$$(((p \to p) \to p) \to (p \to q)) \to (((p \to q) \to p) \to q)$$

to be a tautology. The triplets produced are

$(b1, p, q)$
$(b2, b1, p)$
$(b3, b2, q)$
$(b4, p, q)$
$(b5, p, p)$
$(b6, b5, p)$
$(b7, b6, b4)$
$(b8, b7, b3)$

and the following shows how the $b$s correspond to subformulas.

$$(((p \overset{b5}{\to} p) \overset{b6}{\to} p) \overset{b7}{\to} (p \overset{b4}{\to} q)) \overset{b8}{\to} (((p \overset{b1}{\to} q) \overset{b2}{\to} p) \overset{b3}{\to} q)$$

We set $b8$ to be false and apply the simple rules repeatedly. We start by applying rule $r1$ to the last triplet, and applying the resulting substitution, which we list in place of the triggering triplet, since that can be dropped:

$(b1, p, q)$

$(b2, b1, p)$

$(0, b2, q)$

$(b4, p, q)$

$(b5, p, p)$

$(b6, b5, p)$

$(1, b6, b4)$

$[b7/1, b3/0]$

This gives another opportunity to apply rule $r1$ on the triplet $(0, b2, q)$:

$(b1, p, 0)$

$(1, b1, p)$

$[b2/1, q/0]$

$(b4, p, 0)$

$(b5, p, p)$

$(b6, b5, p)$

$(1, b6, b4)$

$[b7/1, b3/0]$

Next, we apply rule $r7$ to the triplet $(b5, p, p)$, followed by $r4$ to the triplet below it.

$(b1, p, 0)$

$(1, b1, p)$

$[b2/1, q/0]$

$(b4, p, 0)$

$[b5/1]$

$[b6/p]$

$(1, p, b4)$

$[b7/1, b3/0]$

Finally, we apply rule $r5$ to each of the triplets ending with 0.

$[b1/\neg p]$<br>
$(1, \neg p, p)$<br>
$[b2/1, q/0]$<br>
$[b4/\neg p]$<br>
$[b5/1]$<br>
$[b6/p]$<br>
$(1, p, \neg p)$<br>
$[b7/1, b3/0]$

Now, we have only two triplets left: $(1, \neg p, p)$ and $(1, p, \neg p)$ and none of the simple rules applies. We must apply the dilemma rule. In the left branch, we assume $p$ to be true and get the triplets $(1, 0, 1)$ and $(1, 1, 0)$, the second of which is terminal. In the other branch, we get the same two triplets but in the reverse order, so we reach a contradiction in both branches and the application of the dilemma rule also results in a contradiction. This means that the assumption that the formula is false leads to a contradiction, so we conclude that the formula is a tautology.

### *2.1. From proof system to proof method*

The proof system $M$ consisting of the simple rules $r1$ to $r7$ and the dilemma rule is sound and complete for formulas made from variables and implication. Any boolean formula can be translated to such a formula in linear time by a procedure described by Stålmarck [34]. So the system $M$ is sound and complete for full propositional logic.

We make the proof system into a proof method by making a sequence of increasingly more powerful subsystems of $M$. $M_0$ is $M$ without the dilemma rule. $M_{i+1}$ is $M$ in which the derivations in the two branches of the dilemma rule are restricted to be $M_i$ derivations. So, proofs in $M_1$ have at most one open assumption about the value of a variable, proofs in $M_2$ have at most two simultaneous assumptions, and so on.

Stålmarck's method can be seen as a family of algorithms that efficiently search for short proofs in $M_i$ for a given $i$. One can find a proof in $M_0$ in linear time; this is just the closure of the simple rules. The time required to exhaustively search for a proof in $M_k$ is $O(n^{2k+1})$, where $n$ is the size of the formula.

We say that a valid formula is *i-hard* if it is provable in $M_i$, but not provable in $M_j$ for any $j < i$. This notion of formula hardness is important, and we will return to it. For now, it is sufficient to note that many industrial verification problems give rise to formulas whose hardness degree is 0 or 1. The formulas may be large, but the method is much more sensitive to the hardness degree of a formula than to its size in terms of number of variables or connectives. This means that the method is applicable for industrial verification, even at large scale.

It is tempting to believe that the reader who understands the above description of Stålmarck's method knows all that he needs to know about the method. Our contention is that this is not the case. One can gain a much deeper understanding of the method by placing it in a wider context, and by comparing it to more standard approaches. So please read on!

In the remainder of 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. 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.

## 3. 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.

### *3.1. 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 in which the only formulas appearing in the proof are subformulas of 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 $A_i$s are true, then one of the $B_j$s 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 minimise 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 $\Gamma, A$ 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*

$$
\begin{array}{llll}
(Or_L) & \dfrac{\Gamma, A \vdash \Delta \quad \Gamma, B \vdash \Delta}{\Gamma, A \vee B \vdash \Delta} & (Or_R) & \dfrac{\Gamma \vdash \Delta, A, B}{\Gamma \vdash \Delta, A \vee B} \\
(And_L) & \dfrac{\Gamma, A, B \vdash \Delta}{\Gamma, A \wedge B \vdash \Delta} & (And_R) & \dfrac{\Gamma \vdash \Delta, A \quad \Gamma \vdash \Delta, B}{\Gamma, \vdash \Delta, A \wedge B} \\
(Imp_L) & \dfrac{\Gamma \vdash \Delta, A \quad \Gamma, B \vdash \Delta}{\Gamma, A \rightarrow B \vdash \Delta} & (Imp_R) & \dfrac{\Gamma, A \vdash \Delta, B}{\Gamma \vdash \Delta, A \rightarrow B} \\
(Neg_L) & \dfrac{\Gamma \vdash \Delta, A}{\Gamma, \neg A \vdash \Delta} & (Neg_R) & \dfrac{\Gamma, A \vdash \Delta}{\Gamma, \vdash \Delta, \neg A}
\end{array}
$$

### *3.2. 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. The proof is direct, or as Gentzen put it, it is not roundabout [14]. As we shall see later, having the subformula principle allows us to place bounds on proof size, and this is very important in practice.

### *3.3. 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 [21]. 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.

### *3.4. The semantic tableau method*

Smullyan's system of 'analytic tableaux' [32] is another classic proof method. Any valuation of a formula (an assignment of $\top$ (true) or $\bot$ (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:

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

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

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

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

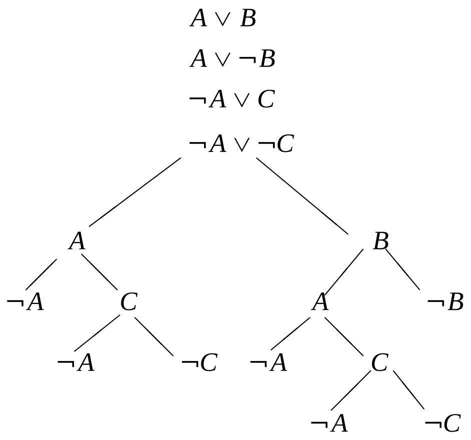
This method analyses a formula by progressively breaking it into its component parts, using these rules. The branching rules (*Or*, *Impl* and *Not-And*) are such that the disjunction of the formulas in the branches is a consequence of the formula above the line. Rules such as *Not-Or* that have two 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 [2, 19, 20, 27]. 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.

### 3.5. *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 occurring 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 [10], 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 right-hand 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.

### 3.6. Rules

Rules in tableau systems correspond to clauses in the definition 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

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

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

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

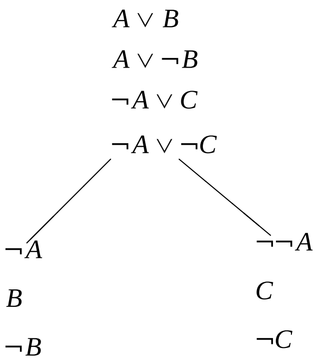
### 3.7. *The systems KE/I and KE*

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 [10, 26]. 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,

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

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 restricted 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 [10].

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.



The well-known Davis-Putnam procedure [12] (in the version of [11]) can be seen as a proof procedure for a variant of KE in which the language of formulas is extended to include generalised conjunction and disjunction, and to which two simplification rules (affirmative-negative and subsumption) are added.

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.

### 3.8. *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 $B$. We call $A \equiv B$ an *association*. If $m$ is the association $A \equiv B$, then $m'$ is the complementary association. $A \equiv B'$. Of course $(R(m))(m')$ is explicitly contradictory. We extend the notation for addition to formula relations from single associations to sets of associations in the obvious way. If $M = \{m_0, m_1, \ldots, m_i\}$ is a set of associations, then $R(M)$, which we write $R(m_0, m_1, \ldots, m_i)$, is $(R(m_0))(m_1, \ldots, m_i)$.

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 4.* 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.

We call the equivalence class containing $\top$ the True-class, that containing $\bot$ the False-class, and the remaining classes the *indeterminate* classes.

### 3.9. *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. (For convenience, we overload the $\bot$ symbol, and use it also to represent all such explicitly contradictory formula relations.) So, this sequence of rule applications constitutes a refutation of $C \wedge \neg C$. From the assumption that $C \wedge \neg C$ is true, we have derived a contradiction.

In the above examples, we have seen that each successful application of a simple rule merges (at least) two pairs of equivalence classes. And indeed, if one class contains a particular set of subformulas, then there is another ‘shadow’ class containing the complements of those subformulas. In real implementations, we halve the number of equivalence classes by making the shadow classes implicit. When two equivalences classes are merged, their shadows are also merged. The result is that each successful application of a proper rule merges at least one pair of (explicit) equivalence classes, and so reduces the number of equivalence classes by at least one.

## 4. 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.

### *4.1. The Dilemma rule*

The Dilemma rule is pictured as

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

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 a formula and its complement 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.

### *4.2. 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 $d(\Pi)$, 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 composition contains only one copy of the intermediate relation $R_2$. The proof depth of the composition is defined to be $max(d(\Pi_1), d(\Pi_2))$.
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

$$\begin{array}{cc} \multicolumn{2}{c}{R} \\ \hline \Pi_1 & \Pi_2 \\ \hline \multicolumn{2}{c}{R_1 \sqcap R_2} \end{array}$$

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

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 5.*

$$\begin{array}{ccc} \multicolumn{3}{c}{R_1} \\ \multicolumn{3}{c}{R_2} \\ \multicolumn{3}{c}{R_3} \\ \hline \multicolumn{2}{c}{R_3(A \equiv B)} & R_3(A \equiv \neg B) \\ \multicolumn{2}{c}{R_4} & \\ \multicolumn{2}{c}{R_5} & R_{10} \\ \cline{1-2} R_6(C \equiv D) & R_6(C \equiv \neg D) & \\ R_7 & & \\ R_8 & R_9 & \\ \cline{1-2} \multicolumn{2}{c}{R_8 \sqcap R_9} & R_{11} \\ \hline \multicolumn{3}{c}{(R_8 \sqcap R_9) \sqcap R_{11}} \end{array}$$

This proof has depth 2. The *length* of a Dilemma proof $\Pi$, denoted $|\Pi|$, is the number of occurrences of formula relations in $\Pi$. The length of the above proof is 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 $X^{+}(X \equiv \bot)$.

### *4.3. Proof hardness*

A formula relation $R$ is *k-easy* if and only if there is a derivation $\Pi : R \Rightarrow \bot$ with $d(\Pi) \leq k$. A relation $R$ is *k-hard* if and only if there is no derivation $\Pi : R \Rightarrow \bot$ with $d(\Pi) < k$. A relation $R$ has *hardness degree k*, $h(R) = k$, if and only if $R$ is $k$-easy and $k$-hard. If a relation has hardness degree $c$, then there must exist a proof $\Pi : R \Rightarrow \bot$ whose depth is $c$. We write *proof*$(R)$ to represent such a proof and this will prove convenient later in proofs about complexity.

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 [18]. 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 subsection 7.3. 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.

## 5. 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.

### *5.1. 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 6.* The formula $(A \wedge (B \vee C)) \rightarrow ((A \wedge B) \vee (A \wedge C))$ is reduced to triplets as follows:

$$\begin{aligned} t &: \top \\ a &: A \\ b &: B \\ c &: C \\ d &: b \vee c \\ e &: a \wedge d \\ f &: a \wedge b \\ g &: a \wedge c \\ h &: f \vee g \\ i &: e \rightarrow h \end{aligned}$$

### *5.2. The saturation algorithm*

A relation $R$ is *k-saturated* if and only if for every Dilemma derivation $\Pi : R \Rightarrow S$ with $d(\Pi) \leq k$, 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 affected(R(q)-R)
              R := R(q)
     od
    return R
```

The set *Compound*$(R)$, the initial pool of subformulas to be processed, contains compound subformulas in the domain of $R$. However, it is restricted to contain only one representative from each of the equivalence classes. This method of using representatives of the indeterminate equivalence classes is vital for the complexity of the algorithm, as we shall see later. We choose 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, $R(q) - R$. Call the set of subformulas mentioned in these new equivalences $N$. We might expect to add back to the pool the subformulas in $N$, all the subformulas from the same equivalence classes (and their complements) as those in $N$, and the 'parents' of all these formulas. (The parent of a formula contains that formula as an immediate subformula.) These are the triplets that might be affected by the new information gained in applying the rule. Adding $N$ and formulas from the same equivalence classes allows information to be propagated downwards in the formula. Adding the parents allows information to propagate upwards.

Again, we make some optimisations that are important for the complexity of the algorithm. If a formula contains a variable $v$ that has been added to one of the indeterminate classes, then that formula is only placed in the pool if it contains a second variable from the same equivalence class or its complement class. In addition, we continue to use a single variable to represent each of the indeterminate classes, replacing variables by their representatives as triplets enter the pool. We call the set of triplets added back to the pool by this procedure `affected(R(q)-R)`. 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. One might think that a better strategy would be to try to merge the two largest equivalence classes, but this does not seem to give better performance in practice.

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

    repeat
      L := Sub(R)
      R' := R
      for each l in L
       do
        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. As in the case of 0-saturation, we place into this pool of formulas only one representative of each of the indeterminate equivalence classes. 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 iterative deepening.

The procedure just presented has been implemented in a tool called Prover Plug-In, which in turn is the proof engine for a number of products, including the verification toolkit NP-Tools. A separate proof-logging and proof-checking module is available, and this was introduced because of pressure from customers, particularly in the railway field. In fact, Prover Plug-In implements an extension of Stålmarck's method to finite domain integer arithmetic, and formulas can also declare and use enumerated types, which is very useful for system modelling. We call the resulting extended propositional logic PROP+. The addition of integer arithmetic is done by adding simple rules for the arithmetic operators, resulting in a rather natural extension of Stålmarck's method. The method has also been extended in various other ways, for example to many sorted first order logic and to propositional linear temporal logic. In this paper, we have chosen, for simplicity, to concentrate only on the propositional logic case.

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 [22]. 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).

## 6. Complexity results

### *6.1. A lower bound on proof length in Dilemma*

If a formula has hardness degree $k$, then a lower bound on the length of its proof is $2^{k/2}$.

Before showing the proof, we introduce a useful proof-theoretic trick. In proving the lower bound, we must construct derivations of limited depth. For a relation $R$ with hardness degree less than $c$, we know that $\mathit{proof}(R) : R \Rightarrow \bot$ has depth less than $c$.

What happens, though, if we are dealing with derivations that do not end in $\bot$? Let us assume that we have a proof $\Pi : R \Rightarrow R(M)$ where $M = \{m_0, m_1, \dots, m_j\}$ is a set of associations. Is there anything that we can do to build a proof of known depth? Yes, indeed there is. Our building blocks are variants on $\Pi$ that lead to $\bot$. Let $M_i' = \{m_0, m_1, \dots, m_{i-1}, m_i'\}$.

Then, construct $\Pi_i : R(M_i') \Rightarrow \bot$ from $\Pi$ by extending each relation in the proof with $M_i'$. Because the final relation in the proof then always contains both $m_i$ and $m_i'$, then it must be contradictory.

So, we can safely use $\mathit{proof}(R(M_i'))$ as a building block in the following derivation, which we call $\mathit{unroll}(R, M)$.

$$\begin{array}{c}
R \\ \hline
\mathit{proof}(R(m_0')) \quad R(m_0) \\ \hline
R(m_0) \\ \hline
\mathit{proof}(R(m_0, m_1')) \quad R(m_0, m_1) \\ \hline
R(m_0, m_1) \\
\vdots \\
R(m_0, m_1, \dots, m_{j-1}) \\ \hline
\mathit{proof}(R(m_0, m_1, \dots, m_{k-1}, m_j')) \quad R(m_0, m_1, \dots, m_j) \\ \hline
R(M)
\end{array}$$

If, in addition, we know that $h(R(M_i')) < c$ for each $i$, then this derivation has depth less than $c + 1$. Note that the trick would have worked equally well had the original $\Pi$ been a derivation from $R$ to $R_1(M)$. The resulting derivation is still from $R$ to $R(M)$, however.

Now, we are ready to show that if a formula has hardness degree at least $k$, then the length of its proof in Dilemma must be at least $2^{k/2}$. We do this by turning the implication around (and negating its components). We show that if the length of a proof $\Pi : R \Rightarrow \bot$ is less than $2^n$, then the hardness of the relation $R$ must be less than $2n$.

The proof is by induction on the length of $\Pi$.

*Base Case*: If $|\Pi| = 1$ then $h(\Pi) = 0$. The proof contains a single relation.

*Induction Step*: Assume $1 < |\Pi| < 2^n$. There are two cases to consider. Either the proof starts with a simple rule application or it starts with an application of the dilemma rule.

1. $\Pi = \Pi_1\Pi_2$ with $\Pi_1 : R \Rightarrow R_1$, a simple rule application, and $\Pi_2 : R_1 \Rightarrow R_2$. By the induction hypothesis, $h(R_1) < 2n$, and $R_1$ is derived from $R$ by the application of a simple rule. Hence, $h(R) < 2n$.

2. $\Pi$ has the form

$$\begin{array}{cc} \multicolumn{2}{c}{R} \\ \hline R_1 & R_2 \\ S & T \\ R_3(M) & R_4(M) \\ \hline \multicolumn{2}{c}{R(M)} \\ \multicolumn{2}{c}{U} \\ \multicolumn{2}{c}{\bot} \end{array}$$

where $M$ is the set of associations found by (both branches of) the Dilemma rule. Obviously, $|\Pi| = |S| + |T| + |U| + 1$, so one of the following three cases must apply to $\Pi$:

a) $|S| < 2^{n-1}$ and $|T| < 2^{n-1}$
b) $|S| < 2^{n-1}$ and $|U| < 2^{n-1}$
c) $|T| < 2^{n-1}$ and $|U| < 2^{n-1}$

a) Since both $S$ and $T$ are short, we apply our trick to both of them. Take the case of $S : R_1 \Rightarrow R_3(M)$. When we make variants of $S$ that lead to $\bot$ as explained above, then we know that those derivations also have length less than $2^{n-1}$, so by the induction hypothesis, each of the $R_1(M'_i)$ must have hardness less than $2n - 2$. So $\mathit{unroll}(R_1, M) : R_1 \Rightarrow R_1(M)$ has depth less than $2n - 1$. A similar argument applies to $T$. For $U$, we know by the induction hypothesis, that $h(R(M)) < 2n$, so $\mathit{proof}(R(M))$ has depth less than $2n$. Plugging everything together, the following derivation

$$\begin{array}{cc} \multicolumn{2}{c}{R} \\ \hline R_1 & R_2 \\ \mathit{unroll}(R_1, M) & \mathit{unroll}(R_2, M) \\ R_1(M) & R_2(M) \\ \hline \multicolumn{2}{c}{R(M)} \\ \multicolumn{2}{c}{\mathit{proof}(R(M))} \\ \multicolumn{2}{c}{\bot} \end{array}$$

has depth less than $2n$, as required.

b) The second and third cases are similar. We show only the second. This time, we concentrate on making a derivation in which the $T$ branch (which is the longer one) moves below the dilemma rule. To do this, we must construct a left branch that ends in $\bot$. We can certainly make a shallow proof from $R_1$ to $R_1(M)$ using *unroll*, just as we did in the previous case. Can we make a shallow proof from $R_1(M)$ to $\bot$? First, we take the derivation $U : R(M) \Rightarrow \bot$, and rewrite it into a derivation from $R_1(M) \Rightarrow \bot$ by extending each relation in the proof by $R_1 - R$. The resulting derivation has length less than $2^{n-1}$, and so by the inductive hypothesis, $h(R_1(M)) < 2n - 2$ and $\mathit{proof}(R_1(M))$ has depth less than $2n - 2$.

Similarly, we can make version of $U$ that is a derivation from $R_4(M) \Rightarrow \bot$ by extending each relation in the proof by $R_4 - R$. The derivation that is $T : R_2 \Rightarrow R_4(M)$

followed by this new version of $U$ has length less than $|\Pi|$ and so by the inductive hypothesis, $h(R_2) < 2n$ and $\mathit{proof}(R_2)$ has depth less than $2n$. The following derivation

$$\begin{array}{cc}
\multicolumn{2}{c}{R} \\ \hline
R_1 & R_2 \\
\mathit{unroll}(R_1, M) & \\
R_1(M) & \\
\mathit{proof}(R_1(M)) & \\
\bot & \\ \hline
\multicolumn{2}{c}{R_2} \\
\multicolumn{2}{c}{\mathit{proof}(R_2)} \\
\multicolumn{2}{c}{\bot}
\end{array}$$

has depth less than $2n$, as required.  Q.E.D.

For analytic KE/I, the corresponding lower bound result is that a formula of hardness degree $k$ has a lower bound on proof length of $2^k$ (and not $2^{k/2}$ as in Dilemma). The difference comes not from the move to relations on subformulas but from the series-parallel shape of derivations.

### 6.2. *An upper bound on the length of derivations in Dilemma*

A proper $k$-derivation is a derivation of depth $k$ without redundant rule applications. (A redundant rule application is one that does not draw any new conclusions, so that the concluding relation is the same as the starting one.) For a given $k$, there is a polynomial upper bound on the length of proper $k$-derivations in Dilemma.

We define a function $f$ such that $f(k, m) \leq m^{k+1}$ for all natural numbers $k$ and $m$.

$$f(k, m) = m, \quad k = 0 \vee m \leq 3$$

$$f(k+1, m) = \left(2 \cdot \sum_{j=1}^{m-1} f(k, j)\right) + m, \quad m > 3$$

We show that if $\Pi : R \Rightarrow R'$ is a proper $k$-derivation, then $|\Pi| \leq f(k, m)$, where $m$ is the number of equivalence classes in $R$. The proof is by induction on $k$.

*Base Case*: $\Pi$ is a 0-derivation, in which only simple rules are applied, and so is just a sequence of relations. How long can the sequence be? Well, there are only $m$ equivalence classes in $R$, and $m$ is reduced by at least one each time a simple rule is applied, so the maximum length of the sequence of relations is $m$. Hence $|\Pi| \leq f(0, m) = m$.

*Induction Step*: $\Pi : R \Rightarrow R'$ is a $(k+1)$-derivation. We treat the worst case, in which the dilemma rule is applied as much as possible. Then, the proof must consist of $m - 1$ applications of the dilemma rule, each of which has branches of depth less than or equal to $k$. (In a proof with fewer uses of dilemma, some of the $S_i$, $T_i$ pairs would simply be removed, corresponding to the application of a simple rule instead, or would themselves contain fewer uses of dilemma.)

$$\begin{array}{c} R \\ \hline S_1 \quad T_1 \\ \hline R_1 \\ \hline S_2 \quad T_2 \\ \hline R_2 \\ \vdots \\ R_{m-2} \\ \hline S_{m-1} \quad T_{m-1} \\ \hline R' \end{array}$$

Each $S_i$ and $T_i$ has a "top relation" that contains at most $m - i$ equivalence classes. So, $|S_i| = |T_i| \leq f(k, i)$, by the induction hypothesis. Thus, we can calculate that

$$|\Pi| \leq \left(2 \cdot \sum_{j=1}^{m-1} f(k, j)\right) + m \;=\; f(k+1, m)$$

Because of the polynomial bound on $f$, we can conclude that an upper bound on the length of a proper $k$-derivation $\Pi : R \Rightarrow R'$ is $m^{k+1}$, where $m$ is the number of equivalence classes in $R$. Note that the upper bound on proof length is subexponentional with respect to the length of the shortest proof.

### *6.3. The hierarchy of hardness*

An interesting question is "Do the hardness degrees actually form a hierarchy, or does everything collapse at some particular degree $k$?". The answer is that the (infinite) hierarchy does indeed exist. Ajtai has shown that there cannot exist polynomial size proofs of the so-called pigeon hole formulas (a propositional encoding of the pigeon hole principle) in bounded depth Frege systems [1]. Furthermore, it is a routine task to polynomially embed $k$-proofs in Dilemma of the pigeon hole formulas into a bounded depth Frege system. So, from our upper bound result about the lengths of derivations in Dilemma, it follows that the hardness of these formulas must be increasing. Perhaps it is not so surprising that the hierarchy exists. If it collapsed, then all tautologies would be $k$-easy and so have polynomial size Dilemma proofs. This would mean that NP $=$ co-NP, contrary to the intuition of most complexity theorists.

From a complexity theoretic point of view, it is interesting to note that our proof theory inspired notion of hardness actually captures the traditional complexity inspired notion of hardness in terms of proof size. Easy formulas have short Dilemma proofs while hard formulas (only) have long Dilemma proofs. This can be generalised to formula sequences. A formula sequence $(F_1, F_2, \ldots, F_i)$ with linearly increasing hardness requires exponentially increasing proof sizes. If hardness grows logarithmically, the proof sizes increase (at most) superpolynomially but not exponentially, and for constant hardness we get polynomial size proofs.

### *6.4. Dilemma versus analytic KE/I*

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 degrees 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. The relational version seems to open opportunities to remove repetitions by the invertible thinning used in the Dilemma rule.

### *6.5. Time complexity of the saturation procedure*

We show that the time required to search exhaustively for a proof of depth $k$ of a formula $A$ is bounded by $O(m^{2k+1})$, where $m$ is the size of $A$.

In order to calculate the complexity of the saturation algorithm, we count the maximum number of triplet evaluations during $k$-saturation of a relation $R$ with $m$ different equivalence classes. (For a relation derived directly from a formula, the number of equivalence classes is the same as the number of triplets or the size of the formula.)

We first note that a triplet with only one unassigned variable either propagates a value to that variable or is useless and can be removed. We will assume that we do not add such useless triplets to the pool of triplets more than once. We also assume that there is a unique variable in each equivalence class that is used as a representative for the class. The pool of triplets contains only such representatives, since other variables are replaced by their representatives before entering the pool.

Consider the 0-saturation procedure. First note that triplets in the pool contain at most two different variables. Why is this? There are two kinds of triplets in the pool: those that contain a variable that has been added to the True- or False-class (leaving only two remaining variables), and those that contain a variable that has been added to another class $C$. In the latter case, we only add the triplet to the pool if it contains a second variable that also belongs to $C$ or its complement class, and both of those variables end up being replaced by the class representative.

If such a triplet with at most two distinct variables propagates when it is evaluated, it will be useless for further evaluation, and so never put back into the pool. Either all of its variables are instantiated or the value of the only variable left is redundant. If on the other hand, the triplet is lifted out of the pool without propagating, then it will, by the same argument as above, only contain one variable the next time it enters the pool.

So, in 0-saturation, each triplet is evaluated at most twice, and the maximum number of evaluations of triplets can be bounded by $2m$.

To calculate the complexity of $(k+1)$-saturation, it is helpful to consider again the pseudo code that was shown earlier.

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

repeat
  L := Sub(R)
  R' := R
  for each l in L
   do
    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
```

Each iteration of the body of the `repeat` loop reduces the number of equivalence classes in the relation $R$ by at least one, apart from the last iteration, which leaves $R$ unchanged. So, the body of the `repeat` loop is executed at most $m$ times, where $m$ is the number of equivalence classes in the top relation. Initially, the list $L$ has length $m$, and its length is reduced by at least one at each iteration. This means that the body of the `for` loop, which contains two $k$-saturations, is executed $m$ times in the first iteration, $m-1$ times in the next, and so on. So, we can define a recursive function that captures this counting argument. The function $g(k, m)$ characterises the maximum number of evaluated triplets, where $k$ is the saturation degree and $m$ is the number of equivalence classes. The base case comes from the argument about the complexity of 0-saturation. The step reflects the above argument about iterations of the `repeat` and `for` loops.

We define

$$
\begin{aligned}
g(0, m) &= 2 \cdot m \\
g(k+1, m) &= \sum_{i=1}^{m} 2 \cdot i \cdot g(k, m)
\end{aligned}
$$

The rate of growth of $g(k, m)$ is bounded by $O(m^{2k+1})$.

Our conclusion is that using the saturation procedure, the time required to search exhaustively for a proof of depth $k$ of a formula $A$ is bounded by $O(m^{2k+1})$, where $m$ 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 subexponential related to the shortest possible proof. The saturation algorithm performs within $O(|A| \cdot s^{4 \cdot \log |A|})$ time, where $s$ is the size of a shortest possible Dilemma proof of $A$. The argument goes as follows. If $A$ has hardness degree $k$, then we can find a proof within $O(|A|^{2k+1})$ steps. Furthermore, the shortest possible proof has length greater than or equal to $2^{k/2}$. Now $|A| \cdot (2^{k/2})^{4 \cdot \log |A|} = |A|^{2k+1}$.

## 7. Industrial applications

We give brief descriptions of three industrial applications of Stålmarck's method.

### *7.1. Minimal models as a method of documentation*

To illustrate the surprising variety of ways in which propositional logic can be used in the analysis of existing systems, we briefly describe a recent study in the area of power station control.

At one of Sweden's nuclear power plants, the function of safety critical control systems is documented in so-called logic schemas. A graphical notation shows how three kinds of components, combinational, memory and real-time elements, are connected to realise a particular function. A typical such function might control the opening or closing of a single valve.

The logic schemas have the advantage of giving an implementation-independent description of the required function. They are used both in communication with regulating authorities and suppliers, and by control-room staff when searching for errors. However, the documents are hand-drawn, and difficult to maintain and analyse.

Prover Technology AB was recently asked to propose ways to improve, and also computerise, this system of documentation. They proposed that each logic schema be hand-translated to the NP-Tools format—a graphical notation for propositional logic with arithmetic used in Prover Technology's verification toolkit. A system containing memory elements is represented by a state transition function in which previous and current states are inputs and outputs.

In addition, it was proposed that each schema be translated to a *triggering table* that shows in a concise tabular form the circumstances (values of variables) in which the output of the function goes high or low. These tables are constructed by finding *minimal* partial models of the formula corresponding to the schema [37]. A partial model of a formula $F$ is an assignment of values to some variables such that all possible assignments of the remaining variables give models (in the ordinary sense) of $F$. A partial model is minimal if there is no other partial model that assigns values to fewer variables. In many cases, it is necessary to simplify the schema in order to give a reasonable number of minimal models; however, these simplifications can themselves be carefully documented. In a typical example, a 17-input **and** gate is modelled by a new variable representing its output. This greatly reduces the number of models, but does not remove any of the important ones. Using this approach to simplifying schemas, a transition function containing 29 inputs, one of which is a state variable, and 19 gates is first reduced to a schema with 8 inputs and 8 gates. It then produces the following table for when the `Open` output goes from low to high.

*Table 1.* A typical triggering table.

| I_Y_20s | K40_L1 | L2_or_Sim_L2 | M_C | R_O | o354V51_V58 | V_O |
|---|---|---|---|---|---|---|
| — | — | 0 | — | 1 | — | — |
| 0 | 0 | — | — | 1 | — | — |

There are two rows in the table, so we have two different settings of small subsets of the input variables that trigger a switch from low to high on the `Open` output. And we know that these are the only such settings. (The signal names from the real example have been changed to reduce the width of the table.)

These triggering tables capture system behaviour at just the right level of abstraction and are a good complement to the schematic diagrams. They also facilitate comparison between different versions of a given function—something that has proved rather difficult with the old method of documentation. Indeed, discrepancies that had escaped earlier inspections were found. This is an example where a technically simple technique, related to the well-known theory of prime implicants, could answer a real need in industry.

It is interesting to note that one can use saturation to 'greedily' generate small but not necessarily minimal partial models. In many applications, one would like to generate minimal models, but the tautology checking needed to generate them simply takes too long. Using 0-saturation can be a good way to trade off time against the size of the partial models generated.

### 7.2. *Verification experiments at Saab*

During the 1990s, Saab Military Aircraft, Linköping, Sweden, has put considerable effort into developing a unified system development methodology. Experiments in the practical application of formal methods have formed part of this work, and it is expected that formal methods will be one of the components in the future system developer's toolbox [38].

Saab have applied NP-Tools in case studies on safety and reliability of integrated subsystems in the JAS 39 Gripen and Saab 2000 aircraft. Here, we briefly present one project, in which the landing gear system in the JAS 39 Gripen was modelled and analysed using propositional logic. For further details of projects at Saab, the reader is referred to reference [38].

In the JAS Gripen project, the system to be controlled consists of two large retractable wheels (called gears) and a smaller nose gear. Each gear has a door that can be closed both when the gear is retracted and when it is extended. The hydraulic system moves both gears and doors, but the movement of gears and doors is not physically coupled. The control system contains both a software and a hardware control unit. The software unit collects information about the status of the landing gear system and other related systems to control movement of gears and doors. The hardware unit provides redundancy and can take over in case of computer breakdown.

Both the hardware and software control units were modelled directly at the level of propositional logic. The inputs in the model are status signals sent by sensors, as well as commands from the pilot and information about the engine, electrical power supply, speed and so on. The outputs are control signals to manoeuvre the hydraulic system. Thus, the system modelled was the control part of a standard control loop, in which continuous signals were discretised where necessary. The model was built in bottom-up fashion, from existing documentation, which was in a mixture of tabular and schematic forms. The macro facility of NP-Tools, a simple form of encapsulation, was used to give a hierarchical system description. The complete system—again state transition function with previous and current states explicitly represented—had approximately 100 inputs and 100 outputs.

A typical safety critical requirement that was checked of this system is “Gears and doors must never collide”. (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.) Another typical requirement is “Emergency extension of landing gear must always be possible”.

Once the control system had been shown to fulfil its functional and safety requirements, its fault-tolerant behaviour was analysed using methods related to FTA (fault tree analysis) and FMEA (failure mode and effect analysis). Errors in sensors and in hardware components were explicitly modelled; one could say that the model was made more realistic. In this way, the behaviour of the system under single or multiple faults could be analysed, in a way that is familiar to hardware designers, but perhaps not to software developers.

The modelling took approximately one man-month, and was carried out in part by the Swedish consultancy company LUTAB. 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. Experience showed that simple safety properties of the form “it is always the case that a certain property holds” were easily verified. More complicated reasoning, such as showing that a requirement is fulfilled within a certain number of time steps proved to be more problematic. It seems likely that some of these problems stemmed from the need to do too much manual modelling. The use of high level design and requirements languages, with translators to propositional logic, gives a more user-friendly verification methodology. When using the synchronous data flow language Lustre, one can go one step further, since the related notion of synchronous observer allows modelling of safety properties in the programming language itself [23, 24]. The LUCIFER tool provides a translator from Lustre with synchronous observers to the NP-Tools format [25]. It frees the user from having to think about what kinds of inductive proofs he is performing, by automatically generating the required number of “unrollings” of the transition function and also the formulas for the requirements. LUCIFER also provides an effective model checker. For this kind of system verification, it seems that it is vital to make the proofs invisible to the user if one is aiming for acceptance in real development projects.

One of the conclusions of the project at Saab was that it is necessary to develop user-friendly methods and tools for expressing and validating requirements. This area is the subject of ongoing research and development at Prover, as part of the EU project FAST [13]. Providing help with the management and validation of requirements is likely to be an important niche for formal methods in industry.

### *7.3. Software development for railway interlocking systems*

In Sweden, Adtranz is the main supplier of 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.

Here, we briefly outline how verification is tackled in this application. The overall picture is an interesting one. There is a set of *generic* safety requirements that apply to *every* railyard. These requirements have been formalised in a timed first-order logic. There is also a *generic program*, written in Sternol, that contains code for each kind of generic object found in railyards. Typical examples are signals, points and track sections. An object has parameters, which allow a specific instance to be created—for example a signal with three lamps. Objects also have In, Out and Own integer variables, with finite domains. For each Out and Own variable, $x$, an equation group of the form

$$\begin{aligned} x &:= e_0 \\ x &:= e_1 \quad \texttt{if } c_1 \\ &\vdots \\ x_n &:= e_n \quad \texttt{if } c_n \end{aligned}$$

specifies how the variable should be set on each cycle. Each $e_i$ is an integer expression, and each $c_i$ is a condition, made from integer expressions, relations (like $>$) and logical connectives. If none of the $c_i$s is true, then $x$ is set to $e_0$, the default value shown in the first assignment.

To describe a particular yard, the programmer connects together instances of the generic objects in a way that closely matches the yard's physical layout. A cycle of an object instance is then as follows:

1. *Initialisation*: Fix parameter values and initialise Own variables.
2. *Fixpoint computation*: Repeatedly read In values and evaluate equation groups, until a fixpoint is reached. Write Out values.

Given this computational model, each object can be translated to a formula that is repeated (in what might be called microcycles) to make a single cycle. Since object variables have finite integer domains, the formulas produced are in propositional logic extended with finite domain integer arithmetic. The translation allows us to perform induction proofs about invariant properties of individual objects. These invariants can then be added to the object models, facilitating proofs at the yard level. All such invariants are automatically generated.

Other checks can be performed on objects; for example one can check (that is prove) for each equation group, that the conditions are mutually exclusive. This test for unintentional nondeterminism avoids potential runtime errors in the code. It is entirely automatic, and has been incorporated into Adtranz' development environment since 1990. A yard that encounters such an unwanted *double value* at runtime is automatically set into a safe state (in which nothing moves). The yard must then be manually reset into a known initial state, before being restarted. Such shutdowns are costly. Having an automatic way to avoid them has allowed Adtranz to greatly reduce time spent on testing.

The verifications just discussed are done for each *object*. What about the yard? The entire yard also performs a fixpoint computation. It repeatedly reads In values from the yard and the environment, and evaluates objects (as described above) until a fixpoint is reached. It then writes Out values to the yard. We can build a formula in propositional logic with finite domain integer arithmetic corresponding to the transition function of the entire yard. It consists of fixpoint models of object instances, and equalities expressing In/Out connections between objects. The requirements for the yard are generated by instantiating the generic requirements according to the yard layout. Quantifiers are expanded over sets of objects, and the requirements are simplified by partial evaluation. Then, at the push of a button, the instantiated yard model can then be checked against the instantiated requirements. Again the proofs are by induction. The proofs in the base case and step, which are performed using Stålmarck's method, are logged and checked by an independent proof checker.

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.

The verification methods described here are incorporated into a development environment called SVT, which is produced and marketed by Adtranz. The effect of introducing these automatic (and therefore user friendly) formal verification techniques has been positive. Several programming errors were detected just after the introduction of the method. However, safety requirements are now routinely proved for all installations. Adtranz report about 90% time and cost reduction for each installed interlocking system. In a recent project to develop new software to control a yard, it was found that with SVT in place verification took a total of two man days, as compared to 10 man years before the introduction of formal verification. Adtranz' approach to the development of interlocking software builds on the reuse of high level specifications. In this case, the customisation needed to adapt the specification to the yard was small, so this particular development was unusually smooth. Since formal verification was introduced, no program errors have been found in running systems. We would contend that this large scale well-engineered methodology, with its supporting tool, is one of the most serious and convincing applications of formal methods anywhere.

The verification method used at Adtranz relies on the existence of generic requirements, and of a generic system description. Should they exist, the yard layout allows us to generate formulas both for the requirements and for the Sternol program controlling the yard. The final step is to do the actual proofs about the yard in question, using the formulas. We feel that this pattern, with generic requirements and generic system descriptions, is repeated in several different application areas—an example being Programmable Logic Controller programming for factory automation.

Using an early version of the SVT tool, Borälv reports on validation of the interlocking software for the Madrid Subway Station Lago [5]. 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.

## 8. 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, as in the examples described in the previous section. There seems to be no reason why circuits should not be verified in the same style. Some promising experiments in this area have been carried out by Lillieroth and Singh [31].

### *8.1. Verification of FPGA cores using Stålmarck's method*

Field Programmable Gate Arrays (FPGAs) now contain as many as a million gates, and designers rely increasingly on predesigned and verified blocks known as cores or virtual components. For example, the COREGen tool from Xilinx (a leading FPGA manufacturer) generates handcrafted and highly optimised designs in several domains, including arithmetic and digital signal processing. Cores often have a long development time. They are made complicated by the need for portability, and in addition customers demand components that have undergone systematic verification. For this reason, cores are a prime target for formal verification techniques designed to reduce development time, and give greater confidence that the system meets its specification. Lillieroth and Singh propose the following core verification flow, which relies on the availability of a high level specification of the core's desired functionality. (Note that fixed size *instances* of cores are verified, as the aim is to perform fast automatic proofs in propositional logic. The intention is to use this kind of verification *at run time* to verify reconfigurable cores.)

*Core decomposition.* Rather than trying to verify a complete core all at once, the subcomponents are first verified, bottom up. This gives smaller, more tractable proofs, and eases the location of errors.

*Core specification.* Each entity to be verified must be provided with a behavioural or register transfer level description in a hardware description language. This specification must be synthesisable; it must be possible to produce a circuit (a netlist of simple components) from it using standard commercial synthesis tools. Many cores come with such a behavioural specification.

*Core netlist generation.* The core specification is put through a commercial synthesis tool (such as Synopsys Design Compiler) to produce a netlist for a fixed size instance of the core, in the standard interchange format EDIF.

*Implementation netlist generation.* The COREGen tool is used to produce an EDIF netlist for the (real) implementation of the core.

*EDIF to NP-Tools translation.* We need to compare the two EDIF netlists produced from the specification and the implementation. To make this possible, Lillieroth and Singh have developed a translator from EDIF to the NP-Tools format, in which (yet again) the result is either a combinational circuit or a state transition function.

*Proof of equivalence of the two netlists.* Proof is by ordinary combinational equivalence checking (which is just tautology checking) or by induction.

*Failed proofs give countermodels.* Countermodels can be examined either in the NP-Tools system or (via an automatically generated simulation script) in the Model Technology VHDL/Verilog simulator.

This rather pragmatic core verification method has the advantage that it fits very well with the existing design flow. It has been shown to work well on a number of cores, including adders, counters and a constant coefficient multiplier. Work on verifying more complicated cores is in progress. None of the examples tackled so far has required more than two unrollings of the transition function in the inductive proof. However, we expect larger scale systems to require our recently developed model checking technique, which is based on induction, but with additional constraints on the sequences of states considered [29]. Indeed, this application is a useful testbed for newly developed methods of verifying synchronous systems.

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 $\mu$FP and Ruby [3]. 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.

## 9. Relating Stålmarck's method to BDDs and model checking

Many of our colleagues (and referees) have asked us to provide some comparison between Stålmarck's method and BDDs. We do not yet have much data to present, but can only hope that some of the readers of this paper will be sufficiently intrigued that they feel inclined to perform some comparative experiments.

Jan Friso Groote and his coworkers have presented some early work comparing BDDs and Stålmark's method in the verification of railyard control systems [17]. Three static properties of a particular control system were checked using a BDD package enhanced with an optimisation technique developed by Groote [16] and using Stålmarck's method. The propositional formulas produced could be reduced using a form of program slicing. The unenhanced BDD package got nowhere, whereas the enhanced version could prove or disprove sliced formulas in times up to a minute. Unsliced formulas took approximately an hour to prove or disprove. The commercial implementation of Stålmarck's method could handle unsliced formulas in a few seconds. This was of course very encouraging, from our point of view. However, this was a small-scale experiment, and we agree with Groote's assertion that much needs to be done (both theoretical investigations and large scale experiments) if we are to fully understand when to use BDDs and their variants, and when to use various kinds of proof search. Like us, Groote has found that a prover based on Stålmarck's method works well on real system verification examples, but not so well on the small but hard problems that are typically used for benchmarking theorem provers.

In the area of model checking, we have recently developed a complete model checker for safety properties based on induction and Stålmarck's method [29]. The algorithm is guaranteed to terminate, and it finds a countermodel of minimal length, should the required property not hold. We have found a way (in pure propositional logic) to constrain the sequences of states considered in unrollings of the transition relation so that we restrict our attention to shortest paths between pairs of states in the graph. This means that the algorithm never iterates beyond the diameter of the state transition graph. First experiments are promising and we consider this to be an approach that is well worth further investigation.

In another interesting development, Biere et al. have developed and implemented Bounded Model Checking [7], a method of doing a form of symbolic model checking without BDDs, but using SAT-procedures, including Stålmarck's method. The method searches for counterexamples of a certain length $k$, and a propositional formula that is satisfiable if and only if such a counterexample exists is generated. The technique avoids the space blow-up of BDDs. In some examples, it produces counterexamples very fast, while the BDD for the initial state could not be built.

It would be very interesting to compare these two model checkers with more standard approaches, on large scale system verification examples.

## 10. 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.

## Acknowledgments

Thanks to Arne Borälv and Koen Claessen for careful reading of several drafts and to our colleagues in the Formal Methods group at Chalmers, with whom we have had many enjoyable discussions. Thanks also to the anonymous referees who helped us to improve the paper, to Dag Wedelin, who kindly checked the complexity calculations, and to Poul Williams, who pointed out an error in the pseudo-code for $(k + 1)$-saturation in an earlier version of this paper. Thanks to Ganesh Gopalakrishnan for his encouragement and his infinite patience. Sheeran's work at Prover Technology is partly funded by the EU project SYRF (Synchronous Reactive Formalisms). Stålmarck's work at Chalmers is funded by Chalmers Stiftelse.

## References

1. M. Ajtai, "The complexity of the pigeonhole principle," in *Proc. 29th Annual Symposium on Foundations of Computer Science*, IEEE Press, pp. 346–355, 1988.
2. E.W. Beth, "Semantic entailment and formal derivability," Mededelingen der Kon. Nederlandse Akademie van Wetenschappen. Afd. letterkunde, n.s., 18, Amsterdam, pp. 309–342, 1955.
3. P. Bjesse, K. Claessen, M. Sheeran, and S. Singh, "Lava: Hardware design in Haskell," in *Proc. Int. Conf. on Functional Programming, ICFP '98*, ACM Press, 1998.
4. A. Borälv, "The industrial success of verification tools based on Stålmarck's method," in *Proc. 9th Int. Conf. on Computer Aided Verification*, Springer-Verlag, LNCS Vol. 1254, pp. 7–10, 1997.
5. A. Borälv, "Case study: Formal verification of a computerized railway interlocking," *Formal Aspects of Computing*, Vol. 10, No. 4, pp. 338–360, April 1999.
6. A. Borälv and G. Stålmarck, "Automated verification in railways," in M.G. Hinchey and J.P. Bowen (Eds.), *Industrial-Strength Formal Methods in Practice*, Springer-Verlag, London, 1999.
7. A. Biere, A. Cimatti, E. Clarke, and Y. Zhu, "Symbolic model checking without BDDs," in *Proc. 5th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems, TACAS '99*, 1999.
8. R. Bryant, "Graph-based algorithms for boolean function manipulation," *IEEE Trans. Comp.*, Vol. c-35, No. 8, pp. 677–691, August 1986.
9. S.A. Cook, "The complexity of theorem-proving procedures," in *Proc. 3rd ACM Symp. on the Theory of Computing*, 1971.

10. M. D'Agostino, "Investigation into the complexity of some propositional calculi," D. Phil. Dissertation, Programming Research Group, Oxford University, 1990.
11. M. Davis, G. Logemann, and D. Loveland, "A machine program for theorem proving," *Communications of the ACM*, Vol. 5, pp. 394–397, 1962. Reprinted in [30].
12. M. Davis and H. Putnam, "A computing procedure for quantification theory," *Journal of the ACM*, Vol. 7, pp. 201–215, 1960. Reprinted in [30].
13. ESPRIT Project No. 25581-FAST. Integrating Formal Approaches to Specification, Test Case Generation and Automatic Design Verification. http://www.prover.com/fast/
14. G. Gentzen, "Untersuchungen über das logische Schliessen," *Mathematische Zeitschrift*, Vol. 39, pp. 176–210, 1935. English translation in *The Collected Papers of Gerhard Gentzen*, M.E. Szabo (Ed.), North-Holland, Amsterdam, 1969.
15. G. Gopalakrishnan and P. Windley (Eds.), in *Proc. Int. Conf. on Formal Methods in Computer-Aided Design*, LNCS Vol. 1522, Springer-Verlag, 1998.
16. J.F. Groote, "Hiding propositional constants in BDDs," *Formal Methods in System Design*, Vol. 8, pp. 91–96, 1996.
17. J.F. Groote, J.W.C. Koorn, and S.F.M. van Vlijmen, "The safety guaranteeing system at station Hoorn-Kersenboogerd (extended abstract)," in *Proc. 10th Annual Conference on Computer Assurance (COMPASS'95)*, IEEE Press, pp. 57–68, 1995. A longer version appears as Technical Report 121, Logic Group Preprint Series, Department of Philosophy, Utrecht University, 1994.
18. J. Harrison, "The Stålmarck method as a HOL derived rule," *Theorem Proving in Higher Order Logics*, LNCS Vol. 1125, Springer-Verlag, 1996.
19. J.K.J. Hintikka, "Form and content in quantification theory," *Acta Philosophica Fennica*, VII, 1955.
20. S. Kanger, "Provability in logic," *Stockholm Studies in Philosophy*, 1, Acta Universitatis Stockholmiensis, 1957.
21. S.C. Kleene, *Mathematical Logic*, John Wiley and Sons Inc., New York, 1967.
22. 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.
23. N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud, "The synchronous dataflow programming language LUSTRE," *Proc. IEEE*, Vol. 79, No. 9, 1991.
24. N. Halbwachs, F. Lagnier, and C. Ratel, "Programming and verifying real-time systems by means of the synchronous data-flow programming language Lustre," in *IEEE Transactions on Software Engineering*, September 1992.
25. M. Ljung, "Formal modelling and automatic verification of Lustre programs using NP-tools," Master's Project Thesis, Prover Technology AB and Department of Teleinformatics, KTH, Stockholm, 1999.
26. M. Mondadori, "An improvement of Jeffrey's deductive trees," Annali dell'Universita di Ferrara, Sez III, Discussion paper 7, Universita di Ferrara, 1989.
27. K. Schütte. *Proof Theory*, Springer-Verlag, Berlin, 1977.
28. 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).
29. M. Sheeran and G. Stålmarck, "Model checking using induction and boolean satisfiability," Technical Report U-99003, Prover Technology, June 1999.
30. J. Siekman and G. Wrightson (Eds.), *Automation of Reasoning*, Springer-Verlag, New York, 1983.
31. S. Singh and C.J. Lillieroth, "Formal verification of reconfigurable cores," in *Proc. Int. Conf. on Field-Programmable Custom Computing Machines, FCCM'99*, IEEE Press, 1999.
32. R.M. Smullyan, *First Order Logic*, Springer-Verlag, Berlin, 1969.
33. M. Srivas and A. Camilleri (Eds.), in *Proc. Int. Conf. on Formal Methods in Computer-Aided Design*, LNCS Vol. 1146, Springer-Verlag, 1996.
34. G. Stålmarck, "A note on the computational complexity of the pure classical implication calculus," *Information Processing Letters*, Vol. 31, pp. 277–278, June 1989.
35. G. Stålmarck and M. Säflund, "Modeling and verifying systems and software in propositional logic," in *Proc. Int. Conf. on Safety of Computer Control Systems, IFAC SafeComp'90*, Pergamon Press, Oxford, pp. 31–36, 1990.

36. M. Säflund, “Modelling and formally verifying systems and software in industrial applications,” in *Proc. Second Int. Conf. on Reliability, Maintainability and Safety (ICRMS ’94)*, Xu Ferong (Ed.), International Academic Publishers, pp. 169–174, June 1994.
37. J. Åhrman, “Evaluation of an algorithm for generating partial models in propositional logic using Stålmarck’s method,” Master’s Thesis, Royal Institute of Technology, Department of Numerical Analysis and Computing Science, 1998.
38. O. Åkerlund, G. Stålmarck, and M. Helander, “Formal safety and reliability analysis of embedded aerospace systems at Saab,” in *Proc. 7th IEEE Int. Symp. on Software Reliability Engineering (Industrial Track)*, IEEE Computer Society Press, 1996.