Using Stålmarck's algorithm to prove inequalities

Byron Cook and Georges Gonthier

Microsoft Research

Abstract. Stålmarck's 1-saturation algorithm is an incomplete but fast method for computing partial equivalence relations over propositional formulae. Aside from anecdotal evidence, until now little has been known about what it can prove. In this paper we characterize a set of formulae with bitvector-inequalities for which 1-saturation is sufficient to prove unsatisfiability. This result has application to fast predicate abstraction for software with fixed-width bit-vectors.

1 Introduction

Stålmarck's n-saturation algorithm [7, 12] is a method for automatically finding consequences of propositional logic formulae. The complexity of n-saturation is $O(g^{2n+1})$, where g is the number of nodes in the graph representing the formula. In practice, when $n \leq 2$, Stålmarck's algorithm is fast but incomplete. The limited forms of saturation (where $n \leq 2$) can be used in situations when completeness is not required. Alternatively, if completeness is required, it can be used as a method of pruning the search space traversed with complete techniques—as was done in [1]. The advantage to this approach is the fact that Stålmarck's algorithm can infer many consequences simultaneously. The disadvantage, given that the limited forms of saturation are not complete, is that little is known about the category of formulae for which they are sufficient to prove unsatisfiability.

The goal of this paper is to address the question of what 1-saturation can prove with respect to an important class of formulae that often arise during model checking: transitive arguments using arithmetic relations such as $\leq$, $<$ and $=$ over Boolean vectors. Informally stated, we show that Stålmarck's algorithm can prove the unsatisfiability of unsatisfiable formulae containing inequalities on vectors of Boolean variables, such as $\ldots w \leq x \wedge \ldots \wedge x \leq y \wedge \ldots \wedge y \leq z \wedge \ldots \wedge \neg(w \leq z) \wedge \ldots$, or $\ldots w = x \wedge \ldots \wedge x < y \wedge \ldots \wedge y \leq z \wedge \ldots \wedge z \leq w \wedge \ldots$, etc. The aspect that makes proving this difficult is the fact that the outcome of saturation greatly depends on how $\leq$, $<$, and $=$ are represented in the propostional formulae.

1.1 Application

The motivation for this paper is rooted in our search for fast proof methods for propositional reasoning within the SLAM software model checker for C pro-

grams[1]. We would like to find fast approximative methods that are guaranteed to be able to prove at least a certain limited class of properties.

SLAM currently treats program variables as unbounded integers, and treats bitwise operations as uninterpreted functions. In reality C programs are primarily written over fixed-width types, and often use bitwise operators in non-trivial ways. In order to switch SLAM's semantics from arbitrary-length to the more accurate fixed-width types we must adapt our methods used to improve the performance of SLAM's implementation of predicate abstraction [6, 10].

The key step behind predicate abstraction is the computation of *coverings*. A covering C of a formula f is a set of monomials drawn from a set of predicates P such that for all $m \in C$, $m \Rightarrow f$. In principle the covering can be computed by enumerating over candidate monomials and calling an automatic theorem prover (such as COGENT [4] or ZAPATO [3]) with arithmetic support to determine which monomials imply f.

The strongest covering can be computed using enumeration over all $3^{|P|}$ possible monomials. However, as described in [9, 10], a faster approach is to use incomplete *symbolic decision procedures* that quickly compute an approximation of the needed coverings. Later, if the approximation is not strong enough, we use a technique described in [2] to lazily refine the quality of the abstraction.

Stålmarck's algorithm is a candidate method for computing these appoximative coverings when a bit-level semantics is used (see [4] for more information on the encoding of C expressions into propositional logic). But how good will the approximative coverings be if 1-saturation is used? Since the vast majority of the formulae involved in this application involve $\leq$, $<$, and $=$, it is important that our initial incomplete abstraction method at least finds the connections between these relations even if it is incomplete over others. This is the question that we are addressing in this paper.

2 Stålmarck's algorithm

In this section we provide a formal definition for Stålmarck's n-saturation algorithm and a sound but incomplete validity procedure based on 1-saturation. We also prove a number of basic properties about the algorithm.

Finite Boolean vectors are sequences whose elements can be $\top$ (`true`), $\bot$ (`false`), propositional variables, and negated propositional variables. Fig. 1 defines the set of all finite Boolean vectors, $\mathcal{S}$. We use subscripts to indicate indexing into vectors. For example, if x is a Boolean vector, then x_5 represents the element at the 5th position of x. Vectors are addressed starting at 1. We use superscripts to differentiate between vectors. For example, x^1 and x^2 should be considered different vectors. They could, of course, have equivalent values.

A partial equivalence relation (PER) over a set S is a relation that is transitive and symmetric. A PER may not necessarily be defined for all arguments in S, but it is reflexive for those in which it is. A finite PER (or FPER) is defined only for a finite subset of S.

[1] SLAM is the basis of Microsoft's Static Driver Verifier product [11].

$$
\begin{array}{rl}
\mathcal{B} & ::= \top \mid \bot \\
\mathcal{V} & ::= \ldots \mid p \mid q \mid r \mid \ldots \\
\mathcal{E} & ::= \mathcal{B} \mid \mathcal{V} \mid \neg\mathcal{V} \\
\mathcal{S} & ::= \langle \mathcal{E}_1, \ldots, \mathcal{E}_n \rangle
\end{array}
$$

Fig. 1. Grammar defining Finite Boolean vectors

Stålmarck's algorithm is defined in terms of finite PERs over the type $\mathcal{E}$ (from Fig. 1). For simplicity we will fix the set of variables for which PERs that we consider are defined. We define $=$, $<$ and $\leq$ on finite PERs in the standard way using when the PER is treated as a set of pairs. We say that a PER R is unsatisfiable iff $R(\top, \bot)$:

$$\textsc{Unsatisfiable}(R) \triangleq R(\top, \bot)$$

We assume that redundant $\neg$ symbols in arguments to equivalence relations are removed. For example, $\neg\top$ is treated as an alternative notation for $\bot$ and $\neg(\neg x)$ is considered the same as x.

Let R be a PER. As in [12], we use the notation $R(x \equiv y)$ to represent a *union* operation over the equivalence classes in the PER. This operation constructs a new relation based on R where x and y's equivalence classes have been merged. This can be (naively) implemented as:

$$
\begin{array}{rl}
R(x \equiv y) \triangleq & \begin{cases} \mathcal{E}^2 \textbf{ if } \textsc{Unsatisfiable}(Q) \\ Q \textbf{ if } \neg\textsc{Unsatisfiable}(Q) \end{cases} \\
& \textbf{where } Q \triangleq (R \cup \{(x,y),(y,x),(\neg x,\neg y),(\neg y,\neg x)\})^*
\end{array}
$$

P^* is the transitive closure of the relation P. We assume the existence of a base finite PER called Base that is defined for $\top$ and $\bot$:

$$\textsc{Base} = \{(\top,\top),(\bot,\bot)\}$$

We assume that all finite PERs are constructed from Base and a series of $\equiv$ operations. Quantification over finite PERs will be limited to this set.

A meet operation over finite PERs is defined as: $\textsc{Meet}(Q, R) \triangleq Q \cap R$.

Lemma 1. $\forall A, B, C.\ A \leq B \wedge A \leq C \Rightarrow A \leq \textsc{Meet}(B, C)$

Lemma 2. $\forall R, x, y.\ R \leq R(x \equiv y)$

Lemma 3. $\forall x, y, R.\ R(x \equiv y)(x, y)$

Fig. 2 defines a function called Initial which constructs a finite identity PER. The type `FPER` is used to represent finite PERs. The polymorphic type constructor `FSet` is used to represent finite sets. We assume that a `foreach` statement over a finite set (as found in Fig. 2) always terminates so long as the loop's body terminates. Therefore (given that the $\equiv$ operation terminates) Initial terminates.

```
FPER INITIAL (FSet<Formulae> S)
{
    FPER R := BASE;
    foreach x ∈ S { R := R(x≡x); }
    return R;
}
```

Fig. 2. INITIAL – Constructs a finite identity PER

2.1 Triples

The first step in Stålmarck's algorithm is to break the input propositional logic formula into an equisatisfiable directed acyclic graph represented as a set of *triples*. A triple is defined by the grammar $\mathcal{T} ::= \mathcal{E} \Leftrightarrow (\mathcal{E} \Rightarrow \mathcal{E}) \mid \mathcal{E} \Leftrightarrow (\mathcal{E} \Leftrightarrow \mathcal{E})$. We assume the existence of a function, called TRIPLES, which returns the representative directed acyclic graph together with a variable representing the original input. As an example, let $\leq_n$, $<_n$, and $=_n$ be defined as:

$$x \leq_n y \triangleq \begin{cases} \top & \textbf{if } n = 0 \\ (\neg x_n \vee y_n) \wedge ((\neg x_n \Leftrightarrow y_n) \vee (x \leq_{n-1} y)) & \textbf{if } n > 0 \end{cases}$$

$$x <_n y \triangleq \begin{cases} \bot & \textbf{if } n = 0 \\ (\neg x_n \vee y_n) \wedge ((\neg x_n \Leftrightarrow y_n) \vee (x <_{n-1} y)) & \textbf{if } n > 0 \end{cases}$$

$$x =_n y \triangleq \bigwedge_{i \in \{1 \ldots n\}} x_i \Leftrightarrow y_i$$

Let x and y be 1-length Boolean vectors. In this case we assume that TRIPLES will produce the following output:

$$\begin{aligned} \text{TRIPLES}(x \leq_1 y) = (\beta_1 \\ , \quad & (\rho_1^1 \Leftrightarrow (\neg x_1 \Leftrightarrow y_1)) \wedge \ (\rho_2^1 \Leftrightarrow (x_1 \Rightarrow y_1)) \\ & \wedge (\rho_3^1 \Leftrightarrow (\neg \rho_1^1 \Rightarrow \beta_2)) \wedge \ (\neg \beta_1 \Leftrightarrow (\rho_2^1 \Rightarrow \neg \rho_3^1)) \\ & \wedge (\beta_2 \Leftrightarrow \top) \\ ) \end{aligned}$$

The variable β_1 is equisatisfiable with $x \leq_1 y$ when constrained by these triples. The variables ρ_1^1, ρ_2^1, ρ_3^1 and β_1 are assumed to be previously unused. In the case where the vectors are of length 2, TRIPLES would return:

$$\begin{aligned} \text{TRIPLES}(x \leq_2 y) = (\beta_1 \\ , \quad & (\rho_1^2 \Leftrightarrow (\neg x_2 \Leftrightarrow y_2)) \wedge \ (\rho_2^2 \Leftrightarrow (x_2 \Rightarrow y_2)) \\ & \wedge (\rho_3^2 \Leftrightarrow (\neg \rho_1^2 \Rightarrow \beta_3)) \wedge \ (\neg \beta_2 \Leftrightarrow (\rho_2^2 \Rightarrow \neg \rho_3^1)) \\ & \wedge (\rho_1^1 \Leftrightarrow (\neg x_1 \Leftrightarrow y_1)) \wedge \ (\rho_2^1 \Leftrightarrow (x_1 \Rightarrow y_1)) \\ & \wedge (\rho_3^1 \Leftrightarrow (\neg \rho_1^1 \Rightarrow \beta_2)) \wedge \ (\neg \beta_1 \Leftrightarrow (\rho_2^1 \Rightarrow \neg \rho_3^1)) \\ & \wedge (\beta_3 \Leftrightarrow \top) \\ ) \end{aligned}$$

The relation LTE is defined in Fig. 3. This relation takes three inputs (i, p, and q) and produces an output o. The t is used as a source of intermediate variables. We can use this relation to characterize the translation of triples over $\leq_n$:

$$\textsc{Triples}(x \leq_n y) = (\beta_1, \wedge_{i \in \{1 \ldots n\}} \{\mathrm{LTE}(\beta_i, \beta_{i+1}, x_i, y_i, \rho^i)\} \wedge \beta_{n+1} \Leftrightarrow \top)$$

where β and ρ are families of fresh variables. Think of LTE as a basic cell (or circuit building-block). The β variables are used to establish communication between other cells. $\textsc{Triples}(x \leq_n y)$ has a similar definition:

$$\textsc{Triples}(x <_n y) = (\beta_1, \wedge_{i \in \{1 \ldots n\}} \{\mathrm{LTE}(\beta_i, \beta_{i+1}, x_i, y_i, \rho^i)\} \wedge \beta_{n+1} \Leftrightarrow \bot)$$

$$\begin{array}{l|l}
\mathrm{LTE}(o,i,p,q,t) \triangleq (t_1 \Leftrightarrow (\neg p \Leftrightarrow q)) & \mathrm{GT}(o,i,p,q,t) \triangleq (t_1 \Leftrightarrow (p \Leftrightarrow q)) \\
\quad \wedge (t_2 \Leftrightarrow (p \Rightarrow q)) & \quad \wedge (\neg t_2 \Leftrightarrow (p \Rightarrow q)) \\
\quad \wedge (t_3 \Leftrightarrow (\neg t_1 \Rightarrow i)) & \quad \wedge (\neg t_3 \Leftrightarrow (t_1 \Rightarrow \neg i)) \\
\quad \wedge (\neg o \Leftrightarrow (t_2 \Rightarrow \neg t_3)) & \quad \wedge (o \Leftrightarrow (\neg t_2 \Rightarrow t_3))
\end{array}$$

$$\mathrm{EQ}(o,i,p,q,t) \triangleq (t_1 \Leftrightarrow (p \Leftrightarrow q)) \wedge (\neg o \Leftrightarrow (t_1 \Rightarrow \neg i))$$

Fig. 3. The relations LTE, GT, and EQ

We can negate LTE to produce GT, also defined in Fig. 3. We can use GT to characterize the translation of triples over the negation of $\leq_n$:

$$\textsc{Triples}(\neg(x \leq_n y)) = (\beta_1, \wedge_{i \in \{1 \ldots n\}} \{\mathrm{GT}(\beta_i, \beta_{i+1}, x_i, y_i, \rho^i)\} \wedge \beta_{n+1} \Leftrightarrow \bot)$$

To characterize the $=_n$ relation we can use EQ (Fig. 3):

$$\textsc{Triples}(x =_n y) = (\beta_1, \wedge_{i \in \{1 \ldots n\}} \{\mathrm{EQ}(\beta_i, \beta_{i+1}, x_i, y_i, \rho^i)\} \wedge \beta_{n+1} \Leftrightarrow \top)$$

2.2 0-saturation

The function in Fig. 4, called ZeroSaturate, implements a version of 0-saturation. Essentially ZeroSaturate iteratively applies the 0-saturation step function ZeroSaturateStep from Fig. 9 until a fixpoint is reached. Note that Fig. 9 appears later in the paper so that the proofs in Section 3.1 are easier to follow.

Lemma 4. ZeroSaturateStep *terminates.*

Proof. By the structure of ZeroSaturateStep's control-flow graph and the termination assumption about `foreach` statements over finite sets. □

```
FPER ZeroSaturate(FPER Q,FSet<Formulae> F)
{
     FPER newQ :=  ZeroSaturateStep (Q,F);
     if (newQ ≠ Q) {
        return  ZeroSaturate(newQ,F);
     } else {
        return Q;
     }
}
```

Fig. 4. ZeroSaturate: 0-saturation algorithm.

Lemma 5. $\forall R, F.\ R \leq \textsc{ZeroSaturateStep}(R, F)$

Proof. All paths through ZeroSaturateStep lead to an update of R with $\equiv$ or R itself. Therefore, by Lemma 2, $R \leq \textsc{ZeroSaturateStep}(R, F)$. □

Lemma 6. ZeroSaturate *computes a fixpoint.*

Proof. The powerset of finite PERs over a fixed set of variables is a finite set. By Lemma 5, ZeroSaturateStep is order-preserving. Therefore, by the Knaster-Tarski fixpoint theorem ([5], p.93), ZeroSaturate computes a fixpoint. □

Lemma 7. $\forall R, F.\ R \leq \textsc{ZeroSaturate}(R, F)$

Proof. By induction on the structure of ZeroSaturate. Due to Lemma 6 we know that the result of ZeroSaturate is a finite composition of applications of ZeroSaturateStep. By Lemma 5 and the transitivity of $\leq$, $R \leq \textsc{ZeroSaturate}(R, F)$. □.

2.3 General *n*-saturation

Stålmarck's n-saturation procedure is defined in Fig 5. Based on this procedure, we also define a validity procedure, called StålmarckValidity, in Fig. 6. The function Vars, when applied to a finite PER, returns the variables for which the PER is defined.

Lemma 8. $\forall R, F, n.\ R \leq \textsc{Saturate}(n, R, F)$

Proof. By induction on n with Lemma 7 in the base case and an argument based on Lemma 2 and Lemma 1 in the inductive case. □

Lemma 9. Saturate *computes a fixpoint.*

Proof. The powerset of finite PERs over a fixed set of variables is a finite set. By Lemma 8, Saturate is order-preserving. Therefore, by the Knaster-Tarski fixpoint theorem, Saturate computes a fixpoint. □

```
 1  FPER SATURATE(int n,FPER R,FSet<Formulae> F)
 2  {
 3       if (n<1) { return ZEROSATURATE(R,F); }
 4       FPER prevR;
 5       do {
 6          prevR := R;
 7          foreach v ∈ VARS(F) {
 8             R :=  SATURATESTEP(n,v,R,F);
 9          }
10       } while (R ≠ prevR);
11       return R;
12  }
13
14  FPER SATURATESTEP(int n,Formulae v,FPER R,FSet<Formulae> F)
15  {
16       assume(n>0);
17       FPER R1 := SATURATE(n-1,R(v≡⊥),F)
18       FPER R2 := SATURATE(n-1,R(v≡⊤),F)
19       return  MEET(R1,R2);
20  }
```

Fig. 5. Stålmarck's SATURATE algorithm.

```
bool STÅLMARCKVALIDITY(Formulae f)
{
     Formulae root;
     FSet<Formulae> F;
     (root,F) := TRIPLES(¬f);
     FPER S := INITIAL(VARS(F));
     FPER Q := SATURATE(1,S(root≡⊤),F);
     return UNSATISFIABLE(Q);
}
```

Fig. 6. STÅLMARCKVALIDITY: a validity procedure based on 1-saturation

Lemma 10. $\forall R, F, v, n \geq 0.\ \textsc{ZeroSaturate}(R, F) \leq \textsc{SaturateStep}(n, v, R, F)$

Proof. By induction on n, the definition of SATURATE, and Lemma 1. □

We use the following two properties to structure the proof in Section 3.

3 What can 1-saturation prove?

Now that we have defined Stålmarck's algorithm we are prepared to reason about what it can prove. Due to Lemma 9 we know that SATURATE does terminate and that there is a finite PER that is a fixpoint of SATURATE. We use the symbol $\Leftrightarrows$ to denote this relation. The reasoning in this section is about the equivalences contained in $\Leftrightarrows$.

In this paper we consider formulae of the following form: $\bigwedge E \Rightarrow \bigvee F$, where $\{\mathcal{R}^1(x^1, x^2), \ldots, \mathcal{R}^{k-1}(x^{k-1}, x^k)\} \subseteq E$ and $\mathcal{R}^k(x^1, x^k) \in F$. The idea is that the $\mathcal{R}^i$s are instances of $\leq$, $<$, etc. We assume that $x^1 \ldots x^k$ are Boolean vectors of length n. The proof could be easily generalized to cases where the vectors are of different size—we limit ourselves to n-length vectors to simplify the notation. We assume that there exists a family of relations (or cells), $\mathcal{C}$, such that for all $j \in \{1, \ldots, k-1\}$,

$$\textsc{Triples}(\mathcal{R}^j(x, y)) = (\beta_1^j, e \wedge_{i \in \{1 \ldots n\}} \{\mathcal{C}^{j,i}(\beta_i^j, \beta_{i+1}^j, x_i, y_i, \rho^{j,i})\} \wedge \beta_{n+1}^j \Leftrightarrow \top)$$

We assume that $\textsc{Triples}(\neg(\bigwedge E \Rightarrow \bigvee F))$ is equivalent to $\textsc{Triples}(\bigwedge(E \cup \neg F))$. If we push the $\neg$ through F (which is in disjunctive form) the result is another conjuction. Therefore, for our purposes, it is sufficient to consider the following conjunction

$$\bigwedge \{\mathcal{R}^1(x^1, x^2), \ldots, \mathcal{R}^{k-1}(x^{k-1}, x^k), \neg\mathcal{R}^k(x^1, x^k)\}$$

and then assume that

$$\textsc{Triples}(\neg\mathcal{R}^k(x, y)) = (\beta_1^k, \wedge_{i \in \{1 \ldots n\}} \{\mathcal{C}^{k,i}(\beta_i^k, \beta_{i+1}^k, x_i, y_i, \rho^{k,i})\} \wedge \beta_{n+1}^k \Leftrightarrow \bot)$$

Fig. 7 displays an instance of this configuration where $n = 3$ and $k = 4$.

We will prove that, when several properties hold of $\{\mathcal{R}^1(x^1, x^2), \ldots, \mathcal{R}^{k-1}(x^{k-1}, x^k), \neg\mathcal{R}^k(x^1, x^k)\}$ that $\textsc{StålmarckValidity}(\bigwedge E \Rightarrow \bigvee F) = \top$. We assume (based on Lemma 10) that the first iteration of saturation in Fig. 5 will find $\beta_1^i \Leftrightarrow \top$ for each $i \in \{1, \ldots, k\}$.

Assumption 1. $\forall i \in \{1, \ldots, k\}.\ \beta_1^i \Leftrightarrow \top$.

This is a key assumption: it allows us to ignore the exact structure of the original formula and relations that are passed to SATURATE. In the case of $\bigwedge E \Rightarrow \bigvee F$, this will be found by 0-saturation. There may be many additional triples and equivalences, and they may have incrementally been added or discovered—but all we need to know is that each component in the transitive argument has been asserted to $\top$.

We structure the proof of $\textsc{StålmarckValidity}(\bigwedge E \Rightarrow \bigvee F) = \top$ as follows:

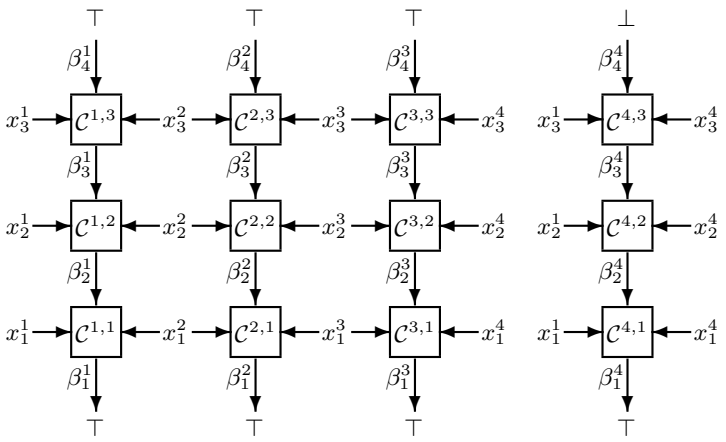

Fig. 7. An example configuration of triples where $n = 3$ and $k = 4$. $\mathcal{C}^{1,1}$ to $\mathcal{C}^{4,3}$ represent cells (see Fig. 3 for examples of cells).

- We define four predicates (PosRgtRipple, PosLftRipple, NegRgtRipple, NegLftRipple) over the signature of the cells ($\mathcal{C}^{i,j}$) used in each $\mathcal{R}^i$ and then constrain the cells using three of these predicates—the fourth predicate is not strictly required due to an uninteresting technicality.
- We inductively find a set of equivalences in $\Leftrightarrow$.
- We then use the equivalences in $\Leftrightarrow$ to demonstrate Unsatisfiable($\Leftrightarrow$).

The predicates are defined in Fig. 8. In a later section we prove that they hold for LTE, GT, EQ, etc.

These predicates are used to represent relationships between the β variables and x variables. For example, PosRgtRipple$(0, C)$ holds when (by using 0-saturation over C) we can prove that if both the output β-variable and left-hand input-variable are true, then the input β-variable and right-hand input-variable must be true.

Assumption 2. $\forall i \in \{1 \dots k-1\}, j \in \{1 \dots n\}$. PosRgtRipple$(0, \mathcal{C}^{i,j})$

In Fig. 7 this corresponds to asserting that the cells in the first three columns are constrained by PosRgtRipple. The next assumption corresponds to asserting that the cells in the fourth column are constrained by NegRgtRipple.

Assumption 3. $\forall j \in \{1 \dots n\}$. NegRgtRipple$(0, \mathcal{C}^{k,j})$

Assumption 4. $\forall i \in \{1 \dots k-1\}, j \in \{1 \dots n\}$. NegLftRipple$(0, \mathcal{C}^{i,j})$

Lemma 11.

$\forall i \in \{1 \dots k\}, m \in \{0 \dots n\}$. $(\beta_{m+1}^i \Leftrightarrow \top) \wedge (\forall j \in \{1 \dots k\}.\ m > 0 \Rightarrow x_m^i \Leftrightarrow x_m^j)$

$$\textsc{Ripple}(n,T,a,b,c,d,v) \triangleq \forall Q.\ [Q(a,\top)\wedge Q(c,v)] \Rightarrow [R(b,\top)\wedge R(d,v)] \quad \textbf{where } R = \textsc{Saturate}(n,Q,T)$$

$$\textsc{PosRgtRipple}(n,C) \triangleq \forall T,a,b,x,y,w.\ C(a,b,x,y,w)\subseteq T \Rightarrow \textsc{Ripple}(n,T,a,b,x,y,\top)$$

$$\textsc{PosLftRipple}(n,C) \triangleq \forall T,a,b,x,y,w.\ C(a,b,x,y,w)\subseteq T \Rightarrow \textsc{Ripple}(n,T,a,b,y,x,\top)$$

$$\textsc{NegRgtRipple}(n,C) \triangleq \forall T,a,b,x,y,w.\ C(a,b,x,y,w)\subseteq T \Rightarrow \textsc{Ripple}(n,T,a,b,x,y,\bot)$$

$$\textsc{NegLftRipple}(n,C) \triangleq \forall T,a,b,x,y,w.\ C(a,b,x,y,w)\subseteq T \Rightarrow \textsc{Ripple}(n,T,a,b,y,x,\bot)$$

Fig. 8. The relation Ripple, and several specialized versions of it

Proof. By induction on m.

Base case ($m = 0$): $(\forall j \in \{1\dots k\}.\ m > 0 \Rightarrow x^i_m \Leftrightarrow x^j_m)$ is trivially true. By Assumption 1, $\beta^i_{m+1} \Leftrightarrow \top$. ✓

Inductive case ($m > 0$): Assume that we are splitting on the variable x^1_m (in Fig. 5). We prove this lemma by cases.

Case $x^1_m \equiv \bot$ (line 17 of Fig. 5): By Lemma 7 and Lemma 3, $\texttt{R1}(x^1_m,\bot)$. By the inductive hypothesis, $\forall i \in \{1\dots k\}.\ \texttt{R1}(\beta^i_m,\top)$. By Assumption 3, $\textsc{NegRgtRipple}(0,\mathcal{C}^{k,m})$. Therefore, because $\mathcal{C}^{k,m}(\beta^k_m,\beta^k_{m+1},x^1_m,x^k_m,\rho^k)$, we know that $\texttt{R1}(x^k_m,\bot)$. By Assumption 4, $\textsc{NegLftRipple}(0,\mathcal{C}^{i,m})$ for $i \in \{1\dots k-1\}$. By induction on i and Assumption 4, $\forall i \in \{1,\dots,k\}.\ \texttt{R1}(x^i_m,\bot)$ and $\forall i \in \{1\dots k\}.\ \texttt{R1}(\beta^i_{m+1},\top)$. By the transitivity of PERs, we know that $\forall i,j.\ \texttt{R1}(x^i_1,x^j_1)$. ✓

Case $x^1_m \equiv \top$ (line 18 of Fig. 5): By Lemma 7 and Lemma 3, $\texttt{R2}(x^1_m,\top)$. By the inductive hypothesis, $\forall i \in \{1\dots k\}.\ \texttt{R2}(\beta^i_m,\top)$. By Assumption 2, $\textsc{PosRgtRipple}(0,\mathcal{C}^{i,m})$ for $i \in \{1\dots k-1\}$. Therefore, by induction on i and Assumption 2, $\forall i \in \{1,\dots,k\}.\ \texttt{R2}(x^i_m,\top)$ and $\forall i \in \{1\dots k\}.\ \texttt{R2}(\beta^i_{m+1},\top)$.. By the transitivity of PERs, $\forall i,j.\ \texttt{R2}(x^i_m,x^j_m)$. ✓

Because $\texttt{R1}(x^i_m,x^j_m)$ and $\texttt{R2}(x^i_m,x^j_m)$, by Lemma 1, $x^i_m \Leftrightarrow x^j_m$. Similarily, because $\texttt{R1}(\beta^i_{m+1},\top)$ and $\texttt{R2}(\beta^i_{m+1},\top)$, $\beta^i_{m+1} \Leftrightarrow \top$. ✓

□

Theorem 1. $\textsc{Unsatisfiable}(\Leftrightarrow)$

Proof. As a consequence of Lemma 11, $\beta^k_{n+1} \Leftrightarrow \top$. However, by definition, $\textsc{Triples}(\neg\mathcal{R}^k(x^1,x^k))$ is

$$\wedge_{i\in\{1\dots n\}}\{\mathcal{C}^{k,i}(\beta^k_i,\beta^k_{i+1},x^1_i,x^k_i,\rho^{k,i})\} \wedge \beta^k_{n+1} \Leftrightarrow \bot)$$

Therefore, $\beta^k_{n+1} \Leftrightarrow \bot$. By the transitivity of $\Leftrightarrow$, $\top \Leftrightarrow \bot$. And therefore, UNSATISFIABLE($\Leftrightarrow$). □

3.1 Proving conditions about LTE, GT and EQ

```
FPER ZeroSaturateStep (FPER R,FSet<Formulae> T) {
 foreach t ∈ T {
  switch(t) {
   pattern (X ⇔ (Y ⇒ Z)):
  /*Z01*/ if (R(X,⊥) && ¬ R(Y,⊤)) { return R(Y ≡ ⊤); }
  /*Z02*/ if (R(X,⊥) && ¬ R(Z,⊥)) { return R(Z ≡ ⊥); }
  /*Z03*/ if (R(Z,⊤) && ¬ R(X,⊤)) { return R(X ≡ ⊤); }
  /*Z04*/ if (R(Y,⊥) && ¬ R(X,⊤)) { return R(X ≡ ⊤); }
  /*Z05*/ if (R(X,Y) && ¬ R(X,⊤)) { return R(X ≡ ⊤); }
  /*Z06*/ if (R(X,¬Z) && ¬ R(X,⊤)) { return R(X ≡ ⊤); }
  /*Z07*/ if (R(Y,Z) && ¬ R(X,⊤)) { return R(X ≡ ⊤); }
  /*Z08*/ if (R(Y,¬Z) && ¬ R(X,Z)) { return R(X ≡ Z); }
  /*Z09*/ if (R(Y,⊤) && ¬ R(X,Z)) { return R(X ≡ Z); }
  /*Z10*/ if (R(Z,⊥) && ¬ R(X,¬Y)) { return R(X ≡ ¬Y); }
     break;
   pattern (X ⇔ (Y ⇔ Z)):
  /*Z11*/ if (R(X,Y) && ¬ R(Z,⊤)) { return R(Z ≡ ⊤); }
  /*Z12*/ if (R(Y,Z) && ¬ R(X,⊤)) { return R(X ≡ ⊤); }
  /*Z13*/ if (R(X,Z) && ¬ R(Y,⊤)) { return R(Y ≡ ⊤); }
  /*Z14*/ if (R(X,¬Y) && ¬ R(Z,⊥)) { return R(Z ≡ ⊥); }
  /*Z15*/ if (R(Y,¬Z) && ¬ R(X,⊥)) { return R(X ≡ ⊥); }
  /*Z16*/ if (R(X,¬Z) && ¬ R(Y,⊥)) { return R(Y ≡ ⊥); }
  /*Z17*/ if (R(X,⊤) && ¬ R(Y,Z)) { return R(Y ≡ Z); }
  /*Z18*/ if (R(X,⊥) && ¬ R(Y,¬Z)) { return R(Y ≡ ¬Z); }
  /*Z19*/ if (R(Y,⊤) && ¬ R(X,Z)) { return R(X ≡ Z); }
  /*Z20*/ if (R(Y,⊥) && ¬ R(X,¬Z)) { return R(X ≡ ¬Z); }
  /*Z21*/ if (R(Z,⊤) && ¬ R(X,Y)) { return R(X ≡ Y); }
  /*Z22*/ if (R(Z,⊥) && ¬ R(X,¬Y)) { return R(X ≡ ¬Y); }
     break;
  }
 }
 return R;
}
```

Fig. 9. ZeroSaturateStep – Function used in ZeroSaturate

We now use Theorem 1 to show that StålmarckValidity can reliably prove transitive arguments using $=_n$, $<_n$, and $\leq_n$ from Section 2.1. By Theorem 1, if we prove the relations PosRgtRipple, PosLftRipple, etc for LTE, GT, and

EQ then we know that StålmarckValidity can prove transitive arguments of the form $\bigwedge E \Rightarrow \bigvee F$, where E and F contain terms with $\leq_n$, $<_n$, and $=_n$ applied to vectors of Boolean variables.

During the following proofs we are implicitly using Lemmas 6 and 7. That is, we assume throughout these proofs that $Q \leq R$.

Lemma 12. PosRgtRipple$(0, \mathrm{LTE})$ *and* NegLftRipple$(0, \mathrm{LTE})$

Proof. Recall the definition of LTE from Fig. 3:

$$\begin{aligned}\mathrm{LTE}(o, i, p, q, t) \triangleq\ & (t_1 \Leftrightarrow (\neg p \Leftrightarrow q)) \\ & \wedge (t_2 \Leftrightarrow (p \Rightarrow q)) \\ & \wedge (t_3 \Leftrightarrow (\neg t_1 \Rightarrow i)) \\ & \wedge (\neg o \Leftrightarrow (t_2 \Rightarrow \neg t_3))\end{aligned}$$

We can assume $Q(o, \top) \wedge R = \textsc{ZeroSaturate}(Q, T) \wedge \mathrm{LTE}(o, i, p, q, t) \subseteq T$ and we must prove the following three conditions $Q(p, \top) \Rightarrow R(q, \top)$, $Q(q, \bot) \Rightarrow R(p, \bot)$, and $R(p, q) \Rightarrow R(i, \top)$. By $R(o, \top)$, $R(\neg o, \bot)$. By Case `Z01` in Fig. 9, $R(t_2, \top)$. By Case `Z02` in Fig. 9, $R(\neg t_3, \bot)$, hence $R(t_3, \top)$.

- Assume $Q(p, \top)$. By Case `Z09` in Fig. 9, $R(t_2, q)$. Because $R(t_2, \top)$, by transitivity, $R(q, \top)$. ✓
- Assume $Q(q, \bot)$. By Case `Z10` in Fig. 9, $R(p, \neg t_2)$. Therefore, $R(\neg p, t_2)$. Because $R(t_2, \top)$, $R(\neg p, \top)$. Hence, $R(p, \bot)$. ✓
- Assume $R(p, q)$. by Case `Z15` in Fig. 9, $R(t_1, \bot)$. Therefore, by Case `Z09` in Fig. 9, $R(t_3, i)$. Because $R(t_3, \top)$, by transitivity of PERs, $R(i, \top)$. ✓

□

Lemma 13. PosLftRipple$(0, \mathrm{GT})$ *and* NegRgtRipple$(0, \mathrm{GT})$

Proof. Recall the definition of GT from Fig. 3:

$$\begin{aligned}\mathrm{GT}(o, i, p, q, t) \triangleq\ & (t_1 \Leftrightarrow (p \Leftrightarrow q)) \\ & \wedge (\neg t_2 \Leftrightarrow (p \Rightarrow q)) \\ & \wedge (\neg t_3 \Leftrightarrow (t_1 \Rightarrow \neg i)) \\ & \wedge (o \Leftrightarrow (\neg t_2 \Rightarrow t_3))\end{aligned}$$

We can assume $Q(o, \top) \wedge R = \textsc{ZeroSaturate}(Q, T) \wedge \mathrm{GT}(o, i, p, q, t) \subseteq T$ and we must prove that $Q(q, \top) \Rightarrow R(p, \top)$, $Q(p, \bot) \Rightarrow R(q, \bot)$, and $R(p, q) \Rightarrow R(i, \top)$.

- Assume $Q(q, \top)$. By Case `Z03` in Fig. 9, $R(\neg t_2, \top)$. Therefore, by Case `Z09` in Fig. 9 $R(o, t_3)$. Because $Q(o, \top)$, by transitivity of PERs, $R(t_3, \top)$. Hence, $R(\neg t_3, \bot)$. By Case `Z01` in Fig. 9, $R(t_1, \top)$. By Case `Z17` in Fig. 9, $R(p, q)$. Because $R(q, \top)$, $Q(p, \top)$. ✓
- Assume $Q(p, \bot)$. By Case `Z04` in Fig. 9, $R(\neg t_2, \top)$. Using the same argument as above, $R(p, q)$. Because $R(p, \bot)$, by transitivity, $R(q, \bot)$,

– Assume $R(p,q)$. By Case `Z07` in Fig. 9, $R(\neg t_2, \top)$. Therefore, by Case `Z09` in Fig. 9, $R(o, t_3)$. By transitivity, $R(t_3, \top)$. That is, $R(\neg t_3, \bot)$. By Case `Z02` in Fig. 9, $R(\neg i, \bot)$. Therefore, $R(i, \top)$. ✓

□

Lemma 14. PosRgtRipple$(0, \mathrm{EQ})$, PosLftRipple$(0, \mathrm{EQ})$, NegRgtRipple$(0, \mathrm{EQ})$, *and* NegLftRipple$(0, \mathrm{EQ})$.

Proof. Recall the definition of EQ from Fig. 3.

$$\mathrm{EQ}(o, i, p, q, t) \triangleq (t_1 \Leftrightarrow (p \Leftrightarrow q)) \wedge (\neg o \Leftrightarrow (t_1 \Rightarrow \neg i))$$

We can assume $Q(o, \top) \wedge R = \textsc{ZeroSaturate}(Q, T) \wedge \mathrm{EQ}(o, i, p, q, t) \subseteq T$ and we must prove that $Q(p, \top) \Rightarrow R(q, \top)$, $Q(q, \bot) \Rightarrow R(p, \bot)$, $Q(p, \bot) \Rightarrow R(q, \bot)$, $Q(q, \top) \Rightarrow R(p, \top)$, and $R(p, q) \Rightarrow R(i, \top)$. By $R(o, \top)$, $R(\neg o, \bot)$. By Case `Z01` in Fig. 9, $R(t_1, \top)$. By Case `Z17` in Fig. 9, $R(p, q)$. The first 4 cases are true by transitivity of PERs. The 5th case is true by Case `Z02` in Fig. 9. □

Alternative implementations There are many other ways to implement $\leq$, $<$, and $=$. For example, we could define $\leq$ in terms of $<$ and $=$:

$$x \leq_n y \triangleq x <_n y \vee x =_n y$$

In this case, we can find a LTE′—displayed in Fig. 10—such that

$$(\beta_3^1, \wedge_{i \in \{1 \ldots n\}} \{\mathrm{LTE}'(\beta^i, \beta^{i+1}, x_i, y_i, \rho^i)\} \wedge \beta_{n+1} \Leftrightarrow \langle \bot, \top \rangle \wedge \beta_3^1 \Leftrightarrow (\beta_1^1 \vee \beta_2^1))$$

We have proven the same result as Theorem 1 for this implementation. Unfortunately we had to modify the proof and Ripple predicate sufficiently that we are not able to include this proof here.

$$\mathrm{LTE}'(o, i, p, q, t) \triangleq \mathrm{LTE}(o_1, i_i, p, q, t') \wedge \mathrm{EQ}(o_2, i_2, p, q, t'')$$

Fig. 10. LTE′: an alternative to LTE. Assume that t' and t'' are vectors with fresh variables. Note that the $\wedge$ is only asserting the triples in LT and EQ and not asserting that the answer is $<$ and $=$. LTE′'s first and second parameters are Boolean vectors of size 2.

Another possibility is displayed in Fig. 11. This is based on an implementation where the unneccesary $\Leftrightarrow$ has been removed:

$$x \leq_n y \triangleq \begin{cases} \top & \textbf{if } n = 0 \\ (\neg x_n \vee y_n) \wedge ((\neg x_n \wedge y_n) \vee (x \leq_{n-1} y)) & \textbf{if } n > 0 \end{cases}$$

$$\begin{aligned}\mathrm{LTE}''(o,i,p,q,t) \triangleq\ & (t_1 \Leftrightarrow (q \Rightarrow p)) \\ \wedge\ & (t_2 \Leftrightarrow (p \Rightarrow q)) \\ \wedge\ & (t_3 \Leftrightarrow (t_1 \Rightarrow i)) \\ \wedge\ & (\neg o \Leftrightarrow (t_2 \Rightarrow \neg t_3))\end{aligned}$$

Fig. 11. LTE″: an alternative to LTE Assume that t' and t'' are vectors with fresh variables.

Lemma 15. PosRgtRipple$(0, \mathrm{LTE}'')$ *and* NegLftRipple$(0, \mathrm{LTE}'')$

Proof. We can assume $Q(o,\top) \wedge R = \textsc{ZeroSaturate}(Q,T) \wedge \mathrm{LTE}''(o,i,p,q,t) \subseteq T$ and we must prove the following three conditions $Q(p,\top) \Rightarrow R(q,\top)$, $Q(q,\bot) \Rightarrow R(p,\bot)$, and $R(p,q) \Rightarrow R(i,\top)$ (assuming that either $R(p,\top)$ or $R(q,\bot)$). By $Q(o,\top)$, $Q(\neg o,\bot)$. By Case `Z01` in Fig. 9, $R(t_2,\top)$. By Case `Z02` in Fig. 9, $R(t_3,\top)$. The first two conditions are proved by the same argument used in Lemma 12. As for the final condition: if $R(p,\top)$ then, by Case `Z03` in Fig. 9, $R(t_1,\top)$. If $R(q,\bot)$ then, by Case `Z04` in Fig. 9, $R(t_1,\top)$. Since (in either case) $R(t_1,\top)$, by Case `Z09` in Fig. 9, $R(t_3,i)$. Because $R(t_3,\top)$, by transitivity of PERs, $R(i,\top)$. □

4 Conclusion

Stålmarck's 1-saturation is a fast but incomplete method of computing finite partial equivalence relations over propositional logic formulae. It can be used in situations when completeness is not required or as a method of pruning the search space traversed by more complete techniques such as backtracking.

We have proved that, under several implementations of inequalities for finite vectors, 1-saturation can be used to compute transitive arguments. This provides some intuition as to what Stålmarck's algorithm can prove. Notably, we now know that a limited form of 2-saturation can be used to compute a useful approximation of transitive closure over relations such as $\leq$ that is representable by equivalences in the triples. This is precisely what Slam needs.

This paper could be a starting point for future efforts of the same kind. There are other incomplete SAT-based techniques—such as *recursive learning* [8]—that play a role that is similar to Stålmarck's algorithm. We would also like to prove more results about Stålmarck's algorithm (or a similar procedure), such that we could get a complete characterization of its relative completeness over propositional logic extended with linear arithmetic and uninterpreted functions. As these proofs are quite tedious (especially the proofs about the relations PosRgtRipple, etc)—we would like to automate them in a mechanical theorem prover.

Acknowledgements Koen Classen, John Harrison, and Mary Sheeran have made helpful comments regarding this work.

References

1. G. Andersson, P. Bjesse, B. Cook, and Z. Hanna. A proof engine approach to solving combinational design automation problems. In *2002 Design Automation Conference*, 2002.
2. T. Ball, B. Cook, S. Das, and S. K. Rajamani. Refining approximations in software predicate abstraction. In *TACAS 04: Tools and Algorithms for Construction and Analysis of Systems*. Springer-Verlag, 2004.
3. T. Ball, B. Cook, S. K. Lahiri, and L. Zhang. Zapato: Automatic theorem proving for predicate abstraction refinement. In *CAV 04: International Conference on Computer-Aided Verification*, 2004.
4. B. Cook, D. Kroening, and N. Sharygina. Cogent: Accurate theorem proving for program verification. In *To appear at CAV 05: Conference on Computer Aided Verification*, 2005.
5. B. A. Davey and H. Priestley. *Introduction to Lattices and Order*. Cambridge University Press, Cambridge, 1990.
6. S. Graf and H. Saïdi. Construction of abstract state graphs with PVS. In *CAV 97: Conference on Computer Aided Verification*, 1997.
7. J. Harrison. Stålmarck's method as a HOL derived rule. In *TPHOLs 96: International Conference on Theorem Proving in Higher Order Logics*, 1996.
8. W. Kunz and P. K. K. Recursive learning: An attractive alternative to the decision tree for test generation in digital circuits. In *ITC'92: International Test Conference*, 1992.
9. S. K. Lahiri, T. Ball, and B. Cook. Predicate abstraction via symbolic decision procedures. In *To appear at CAV 05: Conference on Computer Aided Verification*, 2005.
10. S. K. Lahiri, R. E. Bryant, and B. Cook. A symbolic approach to predicate abstraction. In *CAV 03: International Conference on Computer-Aided Verification*, pages 141–153, 2003.
11. Microsoft Corporation. Static Driver Verifier. Available at www.microsoft.com/whdc/devtools/tools/SDV.mspx.
12. M. Sheeran and G. Stålmarck. A tutorial on Stålmarck's proof procedure for propositional logic. *Formal Methods in System Design*, 16(1), January 2000.