Approximate Quantifier Elimination for Propositional Boolean Formulae

Jörg Brauer[1] and Andy King[2]

[1] Embedded Software Laboratory, RWTH Aachen University, Germany
[2] Portcullis Computer Security, Pinner, UK

Abstract. This paper describes an approximate quantifier elimination procedure for propositional Boolean formulae. The method is based on computing prime implicants using SAT and successively refining over-approximations of a given formula. This construction naturally leads to an *anytime* algorithm, that is, it can be interrupted at anytime without compromising soundness. This contrasts with classical monolithic (all or nothing) approaches based on resolution or model enumeration.

1 Introduction

Model checking and abstract interpretation are sub-disciples of formal methods that, for many years, have been diametrically opposed. In model checking a programmer prescribes a so-called model that formally specifies the behaviour of the system or program. All paths through the program are then exhaustively checked against this requirement. Either the requirement is discharged or a counterexample is found that illustrates how the program is faulty. The detailed nature of the requirements entails that the program is simulated in a fine-grained way, sometimes down to the level of individual bits. Enumerating all these combinations is computationally infeasible. Thus, there has been much interest in representing all the states of a program symbolically, which enables states that share commonality to be represented without duplicating their commonality.

In abstract interpretation, the key idea is to abstract away from the detailed nature of states. Then the program checker operates over classes of related states — collections of states that are equivalent in some sense — rather than individual states. If the number of classes is small, then all the paths through the program can be enumerated one-by-one without incurring the problems of state-space explosion. When carefully constructed, the classes of states can preserve sufficient information to prove the correctness requirements.

Despite their philosophical differences, the fields of model checking and abstract interpretation are converging, partly because they draw on similar computational techniques. A case in point is given by Boolean formulae that are typically either represented with BDDs [5] or manipulated using SAT [22]. BDDs have been widely applied, both in symbolic model checking [7], and as an abstract domain for tracking dependences [1]. Although some niche problems remain difficult for SAT [6], clever ideas and careful engineering have advanced

DPLL-based SAT solvers [22] to the point they can rapidly decide the satisfiability of structured problems that involve thousands of variables. Conseqently SAT has been almost universally adopted within symbolic model checking [9].

1.1 Quantifier elimination and abstract interpretation

Yet SAT remains a comparative novelty in abstract interpretation where it is more often than not relegated to solving auxiliary problems such as that of synthesising best transformers [4, 20, 30] rather then being integrated into the heart of the analysis itself [18]. This is not because there is no interest in using Boolean functions as an abstract domain [1, 16, 18] but rather because projection operations, namely existential and universal quantifier elimination, fit less comfortably with SAT than with BDDs. Eliminating a single variable from a BDD, either existentially or universally, is worst-case quadratic in size of the input BDD [5, Sect. 3.3]. By way of contrast, the natural way to existentially quantify using a SAT solver is to systematically enumerate the models of a formula using blocking clauses. Even when the blocking clauses only constrain the variables in the projection space, such methods are inefficient when compared to BDD-based techniques because of the large number of models that may need to be enumerated [6]. This would be less of a problem if projection was an infrequent operation in abstract interpretation; the guiding principle in domain design is that the commonly arising operations should be fast whereas the speed of the infrequent operations is less critical. However, in dependency analysis, elimination is applied whenever a call is encountered. This is because the dependencies at the call site need to be restricted to those variables that occur as the arguments of a call so as to propagate dependency information across the body of the callee. Existential quantification is applied to flow information in the direction of the control-flow [1] whereas universal quantification is needed to propagate requirements against the control-flow [13]. The frequency of call handling and the inefficiency of SAT-based elimination methods have tended to bias abstract interpretation towards BDDs [1], though new algorithms for elimination would break this dependency.

1.2 Quantifier elimination by resolution and striking out literals

For formulae presented in CNF, existential and universal quantifiers can alternatively be eliminated by resolution and striking out literals [22]. To illustrate, let $f = (\wedge_{i=0}^{n_1} x \vee C_i) \wedge (\wedge_{j=0}^{n_2} \neg x \vee D_j) \wedge (\wedge_{k=0}^{n} E_k)$ and consider $\exists x : f$ and $\forall x : f$ where C_i, D_j and E_k are clauses that involve neither x nor $\neg x$. A quantifier-free version of $\exists x : f$ can be obtained by resolving each $x \vee C_i$ with $\neg x \vee D_j$ to give $\exists x : f = (\wedge_{i=0}^{n_1} \wedge_{j=0}^{n_2} C_i \vee D_j) \wedge (\wedge_{k=0}^{n} E_k)$, *increasing* the representation size by as many as $n_1 n_2 - n_1 - n_2$ clauses. By way of contrast, $\forall x : f$ can be found by removing the x and $\neg x$ literals to give $\forall x : f = (\wedge_{i=0}^{n_1} C_i) \wedge (\wedge_{j=0}^{n_2} D_j) \wedge (\wedge_{k=0}^{n} E_k)$, *reducing* the size of the representation.

One might be forgiven for thinking that calculating a quantifier-free version of $\forall \boldsymbol{y} : f$ is straightforward when f is propositional and $\boldsymbol{y}$ is a vector of variables.

For such an f, an equisatisfiable CNF formula g can be found [28] by introducing fresh variables $\boldsymbol{z}$ to give $f = \exists \boldsymbol{z} : g$ [33] . But then $\forall \boldsymbol{y} : f$ amounts to solving $\forall \boldsymbol{y} : \exists \boldsymbol{z} : g$ and the quadratic nature of resolution compromises the tractability of this approach as the size of $\boldsymbol{z}$ increases.

1.3 Contributions to approximate quantifier elimination

In this paper, we show how upper-approximation can be applied to eliminate $\boldsymbol{z}$ from $\exists \boldsymbol{z} : g$ where g is presented in CNF. We show how a SAT solver can be repeatedly called to compute a sequence of CNF formulae $h_0, h_1, \ldots$ that converge onto $\exists \boldsymbol{z} : g$ from above in the sense that $\exists \boldsymbol{z} : g$ entails h_i (each model of $\exists \boldsymbol{z} : g$ is also a model of h_i). Each h_{i+1} strictly entails h_i so the sequence is ultimately stationary. However, each h_i is free from all variables in $\boldsymbol{z}$, hence this approach has the attractive property that generation of the sequence $h_0, h_1, \ldots, h_t$ can be stopped prematurely, at any time t, without compromising soundness since each h_i is an upper-approximation of $\exists \boldsymbol{z} : g$.

This approach leads to a so-called *anytime* (or *interruptible* [2, Sect. 2.6]) formulation of projection that compares favourably against resolution and model enumeration techniques, which lead to all or nothing, monolithic approaches. Specifically, if $g_0 = g$ and g_{i+1} is obtained from g_i by applying resolution to remove another variable of $\boldsymbol{z}$, then it is only the final formula $g_{|\boldsymbol{z}|}$ that is free from $\boldsymbol{z}$. Moreover, the number of clauses in g_i do not necessarily decrease as i increases, and the size of intermediate g_i can be significantly larger than both g and its projection $g_{|\boldsymbol{z}|}$. By way of contrast, the size of the h_i increases monotonically as the sequence converges. We also show how to construct a sequence $h_0, h_1, \ldots, h_t$ which rapidly converges onto $\exists \boldsymbol{z} : g$ based on the enumeration of prime implicants, that is, small conjunctions of literals which entail $\exists \boldsymbol{z} : g$. As a final contribution, we show how this scheme can be implemented with incremental SAT [35] and sorting networks [14, 21].

Our paper makes a specific contribution to a specific problem, yet that problem appears in various guises in model checking and abstract interpretation. As already stated, projection arises in dependency analysis which is itself finding new applications in, for example, information flow analysis [16]. Projection arises when computing transfer functions [4] and, very recently, in the synthesis of ranking functions from template constraints for low-level code [10]. The existence of a ranking function on a path π with a transition $r_\pi(\boldsymbol{x}, \boldsymbol{x}')$ amounts to solving the formula $\exists \boldsymbol{c} : \forall \boldsymbol{x} : \forall \boldsymbol{x}' : r_\pi(\boldsymbol{x}, \boldsymbol{x}') \rightarrow p(\boldsymbol{c}, \boldsymbol{x}) < p(\boldsymbol{c}, \boldsymbol{x}')$ where $p(\boldsymbol{c}, \boldsymbol{x})$ is a polynomial over the bit-vector $\boldsymbol{x}$ whose coefficients constitute the vector $\boldsymbol{c}$. However, if intermediate variables are needed to express $r_\pi(\boldsymbol{x}, \boldsymbol{x}')$, the polynomials $p(\boldsymbol{c}, \boldsymbol{x})$ and $p(\boldsymbol{c}, \boldsymbol{x}')$ or the size relation $<$ in CNF, then the quantifiers take the form $\exists \boldsymbol{c} : \forall \boldsymbol{x} : \forall \boldsymbol{x}' : \exists \boldsymbol{z}$ where $\boldsymbol{z}$ is the vector of intermediate variables. The authors proceed by instantiating elements of the $\boldsymbol{c}$ vector to values drawn from the set $\{-1, 0, 1\}$, then testing the formula $\neg \exists \boldsymbol{x} : \exists \boldsymbol{x}' : r_\pi(\boldsymbol{x}, \boldsymbol{x}') \wedge \neg(p(\boldsymbol{c}, \boldsymbol{x}) < p(\boldsymbol{c}, \boldsymbol{x}'))$ for *unsatisfiability*. The method advocated in this paper suggests a more direct approach, which avoids enumerating combinations of coefficients, and restricts the coefficients to a small set of allowable values.

2 Existential quantification in five steps

The idea behind our approach is to converge onto the set of solutions of a formula φ by adding constraints formed from the prime implicants of $\neg\varphi$ that are derived using SAT solving. This approach contrasts with existing techniques in that it is based on successive refinement and thereby provides an anytime approach to existential quantifier elimination. We build towards the technique in five steps.

2.1 Under-approximation using implicants

We first show how to under-approximate an existentially quantified formula by deriving an implicant ν of $\exists z : \varphi$, that is, $\nu \models \exists z : \varphi$. To illustrate, let:

$$\varphi = (\neg x \vee z) \wedge (y \vee z) \wedge (\neg x \vee \neg w \vee \neg z) \wedge (w \vee \neg z)$$

Let $X = \{w, x, y, z\}$ denote the set of variables in φ. To project φ onto $Y_1 = \{w, x, y\}$, i.e. remove all information pertaining to the variables $Y_2 = X \setminus Y_1 = \{z\}$, we introduce fresh sets of variables $Y_1^+ = \{v^+ \mid v \in Y_1\}$ and $Y_1^- = \{v^- \mid v \in Y_1\}$. Each occurrence of the literal v in φ is replaced with v^+ if $v \in Y_1$ and each occurrence of $\neg v$ is replaced with v^- if $v \in Y_1$. The transformed formula is augmented with a constraint $\neg v^+ \vee \neg v^-$ for each $v \in Y_1$ so as to prevent v^+ and v^- holding simultaneously. Let t_{Y_1} denote this transformation, hence:

$$t_{Y_1}(\varphi) = \begin{cases} (x^- \vee z) \wedge (y^+ \vee z) \wedge (x^- \vee w^- \vee \neg z) \wedge (w^+ \vee \neg z) \wedge \\ (\neg w^+ \vee \neg w^-) \wedge (\neg x^+ \vee \neg x^-) \wedge (\neg y^+ \vee \neg y^-) \end{cases}$$

Then the formula $t_{Y_1}(\varphi)$ is defined over the set of variables $X' = Y_1^+ \cup Y_1^- \cup Y_2$, and a model of $t_{Y_1}(\varphi)$ is a map $\mathcal{M} : X' \to \mathbb{B}$ such as:

$$\mathcal{M} = \left\{ w^+ \mapsto 1,\, w^- \mapsto 0,\, x^+ \mapsto 0,\, x^- \mapsto 1,\, y^+ \mapsto 0,\, y^- \mapsto 0,\, z \mapsto 1 \right\}$$

The model $\mathcal{M}$ can be equivalently represented by the set $\{v \in X' \mid \mathcal{M}(v) = 1\}$, and henceforth we shall use the map and set representation interchangeably. The variables of $\mathcal{M} \cap (Y_1^+ \cup Y_1^-)$ define a cube (a conjunction of literals) that is given by $\nu = (\bigwedge_{v^+ \in \mathcal{M} \cap Y_1^+} v) \wedge (\bigwedge_{v^- \in \mathcal{M} \cap Y_1^-} \neg v)$. Therefore $\nu = (\neg x \wedge w)$. Observe that $\nu \models \exists Y_2 : \varphi$ hence ν is a so-called implicant of $\exists Y_2 : \varphi$ which constitutes an under-approximation of $\exists Y_2 : \varphi$. This can be seen since ν is free from any variables of Y_2 and the conjunction $\neg\varphi \wedge \nu$ is unsatisfiable. To converge onto $\exists Y_2 : \varphi$ from below, we augment $t_{Y_1}(\varphi)$ with the blocking clause $(\neg x^- \vee \neg w^+)$ which suppresses the previously derived solution. The blocking clause ensures that any cube that is subsequently found does not entail ν. Then $t_{Y_1}(\varphi) \wedge (\neg x^- \vee \neg w^+)$ is checked for satisfiability, yielding a model:

$$\mathcal{M}' = \left\{ w^+ \mapsto 0,\, w^- \mapsto 0,\, x^+ \mapsto 0,\, x^- \mapsto 1,\, y^+ \mapsto 1,\, y^- \mapsto 0,\, z \mapsto 0 \right\}$$

which defines another implicant $(\neg x \wedge y)$ of $\exists Y_2 : \varphi$, hence the refined under-approximation $(\neg x \wedge y) \vee (\neg x \wedge w)$. Adding another blocking clause and passing $t_{Y_1}(\varphi) \wedge (\neg x^- \vee \neg w^+) \wedge (\neg x^- \vee \neg y^+)$ to a SAT solver reveals the formula to be unsatisfiable. Convergence onto $\exists Y_2 : \varphi$ has thus been achieved

and $\exists Y_2 : \varphi = (\neg x \wedge y) \vee (\neg x \wedge w)$. This can be checked by applying Schröder-expansion [22, Sect. 9.2.3] to compute $\exists Y_2 : \varphi = \varphi[z \mapsto 0] \vee \varphi[z \mapsto 1] = ((\neg x) \wedge (y)) \vee ((\neg x \vee \neg w) \wedge (w)) = (\neg x \wedge y) \vee (\neg x \wedge w)$.

2.2 Over-approximation using implicants

To derive an over-approximation of $\exists Y_2 : \varphi$, a formula κ is constructed which is equisatisfiable to $\neg\varphi$:

$$\kappa = \begin{cases} (x \vee t_1) & \wedge (\neg z \vee t_1) \wedge \\ (\neg y \vee t_2) & \wedge (\neg z \vee t_2) \wedge \\ (x \vee t_3) & \wedge (w \vee t_3) \quad \wedge (z \vee t_3) \qquad\qquad \wedge \\ (\neg w \vee t_4) & \wedge (z \vee t_4) \quad \wedge (\neg t_1 \vee \neg t_2 \vee \neg t_3 \vee \neg t_4) \end{cases}$$

The formula κ is obtained by a standard CNF translation [28] which introduces fresh variables $T = \{t_1, \ldots, t_4\}$ such that $\neg\varphi \equiv \exists T : \kappa$. The variable t_i indicates whether a truth assignment violates the i^{th} clause of φ. Applying the transformation introduced previously then gives:

$$t_{Y_1}(\kappa) = \begin{cases} (x^+ \vee t_1) & \wedge (\neg z \vee t_1) & \wedge \\ (y^- \vee t_2) & \wedge (\neg z \vee t_2) & \wedge \\ (x^+ \vee t_3) & \wedge (w^+ \vee t_3) & \wedge (z \vee t_3) \qquad\qquad \wedge \\ (w^- \vee t_4) & \wedge (z \vee t_4) & \wedge (\neg t_1 \vee \neg t_2 \vee \neg t_3 \vee \neg t_4) \wedge \\ (\neg w^+ \vee \neg w^-) & \wedge (\neg x^+ \vee \neg x^-) & \wedge (\neg y^+ \vee \neg y^-) \end{cases}$$

To see how $t_{Y_1}(\kappa)$ can be applied to find an over-approximation $\neg\nu$ of $\exists Y_2 : \varphi$ observe that $\nu \models \forall Y_2 : \exists T : \kappa$ iff $\neg\forall Y_2 : \exists T : \kappa \models \neg\nu$ iff $\exists Y_2 : \neg\exists T : \kappa \models \neg\nu$ iff $\exists Y_2 : \varphi \models \neg\nu$. Hence to find an over-approximation of $\exists Y_2 : \varphi$ it suffices to find an implicant of $\forall Y_2 : \exists T : \kappa$. To find such an implicant observe that $\forall Y_2 : \exists T : \kappa \models \exists Y_2 : \exists T : \kappa$ hence every implicant of $\forall Y_2 : \exists T : \kappa$ is also an implicant of $\exists Y_2 : \exists T : \kappa$. This suggests a strategy in which the implicants of $\exists Y_2 : \exists T : \kappa$ are filtered to find the implicants of $\forall Y_2 : \exists T : \kappa$, that is, the implicants $\nu \models \exists Y_2 : \exists T : \kappa$ are filtered by checking $\exists Y_2 : \varphi \models \neg\nu$. Moreover, the check $\exists Y_2 : \varphi \models \neg\nu$ amounts to deciding whether the conjoined formula $\varphi \wedge \nu$ is unsatisfiable. Thus an unsatisfiability check can be used for filtering. To illustrate, suppose that a SAT solver produces the following solution to the formula $t_{Y_1}(\kappa)$:

$$\mathcal{M} = \left\{ \begin{matrix} w^+ \mapsto 0, w^- \mapsto 1, x^+ \mapsto 0, x^- \mapsto 0, y^+ \mapsto 0, y^- \mapsto 0 \\ z \quad \mapsto 1, t_1 \ \mapsto 1, t_2 \ \mapsto 1, t_3 \ \mapsto 1, t_4 \ \mapsto 0 \end{matrix} \right\}$$

The cube $\nu = (\neg w)$ is an implicant of $\exists Y_2 : \exists T : \kappa$ and therefore it remains to check whether $\exists Y_2 : \varphi \models \neg\nu$. Since $\varphi \wedge \nu$ is satisfiable, the cube is discarded. However, before doing so, the formula $t_{Y_1}(\kappa)$ is augmented with $\neg w^- \vee x^+ \vee x^- \vee y^- \vee y^+$ to avoid the cube being found again. This blocking clause can be interpreted as an implication $w^- \rightarrow (x^+ \vee x^- \vee y^- \vee y^+)$ which ensures that any cube subsequently found that entails ν also has more literals than ν. Applying a SAT solver then yields a model:

$$\mathcal{M}' = \left\{ \begin{matrix} w^+ \mapsto 0, w^- \mapsto 0, x^+ \mapsto 1, x^- \mapsto 0, y^+ \mapsto 0, y^- \mapsto 0 \\ z \quad \mapsto 0, t_1 \ \mapsto 0, t_2 \ \mapsto 1, t_3 \ \mapsto 1, t_4 \ \mapsto 0 \end{matrix} \right\}$$

and hence $\nu' = (x)$. Since $\varphi \wedge \nu'$ is unsatisfiable, we conclude that $\exists Y_2 : \varphi \models \neg\nu'$, hence $\neg\nu'$ constitutes an over-approximation of $\exists Y_2 : \varphi$. The blocking clause $\neg x^+$ is then added to $t_{Y_1}(\kappa)$ to prevent any cube which entails ν' being found. Note too that this blocking clause differs in structure from the one imposed previously, and indeed the number of literals in the clause is merely n where n is the number of literals in the cube. In the previous case, the number of literals in the blocking clause is $2|Y_1| - n$. Reapplying a SAT solver yields a further model:

$$\mathcal{M}'' = \left\{ \begin{array}{llllllll} w^+ \mapsto 0, & w^- \mapsto 1, & x^+ \mapsto 0, & x^- \mapsto 0, & y^+ \mapsto 0, & y^- \mapsto 1 \\ z \quad \mapsto 1, & t_1 \quad \mapsto 1, & t_2 \quad \mapsto 1, & t_3 \quad \mapsto 1, & t_4 \quad \mapsto 0 \end{array} \right\}$$

which defines the cube $\nu'' = \neg w \wedge \neg y$. Since $\varphi \wedge \nu''$ is unsatisfiable, it again follows that $\exists Y_2 : \varphi \models \neg\nu''$, which refines the over-approximation of $\exists Y_2 : \varphi$ to the conjunction $(\neg\nu') \wedge (\neg\nu'')$. The blocking clause $\neg w^- \vee \neg y^-$ is then added to the augmented formula at which point one final application of the solver indicates that the conjoined formula is unsatisfiable. Hence convergence onto $\exists Y_2 : \varphi$ has been obtained from above where $\exists Y_2 : \varphi = \neg\nu' \wedge \neg\nu'' = (\neg x) \wedge (w \vee y)$. Terminating the procedure early, before ν'' is computed, would yield the over-approximation $\neg\nu' = \neg x$ which, though safe, has strictly more models than $(\neg x) \wedge (w \vee y)$. Thus the method is diametrically opposed to resolution: In the resolution based scheme, the projection is found in the last step only when all variables have been eliminated one after the other. In the above SAT based scheme, a clause in the projection space is obtained in the first step, as in a parallel form of elimination, which is subsequently refined by adding further clauses.

2.3 Approximation using prime implicants

Thus far we have seen how upper- and lower-approximation can be reduced to finding an implicant c of a formula f where c is a cube, namely a conjunction of literals. Suppose $c_1 \models f$ and $c_2 \models f$ where the cubes c_1 and c_2 are related by $c_1 \models c_2$. Then $\neg f \models \neg c_2 \models \neg c_1$ where $\neg c_2$ and $\neg c_1$ are clauses. Furthermore, if c_2 is shorter than c_1, that is, if c_2 is constructed from fewer literals than c_1, then $\neg c_2$ constitutes a stronger (more descriptive) approximation than $\neg c_1$. Rather than using any implicant to approximate $\neg f$, it is better to use a shorter one, and better still to use one that is said to be prime. The implicant c_2 of f is prime (or irreducible) if there is no shorter implicant c_3 of f such that $c_2 \models c_3 \models f$. The best approximations are thus constructed from the shortest prime implicants.

To derive shortest prime implicants, we turn to sorting networks [14, 21]. Examples of sorting networks for 3 and 4 bits are given in Fig. 1. The 3-bit sorter has 3 input bits on the left and 3 output bits on the right. It also has 3 comparison operations, indicated with vertical bars, which compare and if necessary swap bits. A comparator assigns its outgoing upper bit to the maximum of its two incoming bits and its outgoing lower bit to the minimum. A comparator with incoming bits i_1 and i_2 with outgoing bits u and ℓ can be encoded propositionally as the formula $(u \leftrightarrow i_1 \vee i_2) \wedge (\ell \leftrightarrow i_1 \wedge i_2)$. The value of a sorting network is that it can be applied to compute the sum of a series of 0/1 values [14] where the

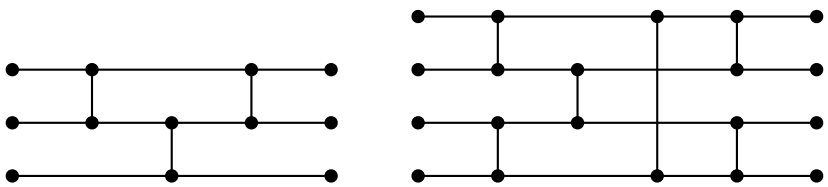

Fig. 1. Sorting networks for 3 and 4 bits

sum is represented in a unary fashion. Moreover, by instantiating the output bits to fixed unary value, a cardinality constraint can be obtained. For example, by constraining the output bits of the 4-bit sorter to 1100, the cardinality constraint is derived which ensures that exactly two of the input bits to the sorter are set. Constraining the output bits to 1110 would ensure that exactly three input bits are set. Such cardinality constraints can be imposed in conjunction with the formula $t_{Y_1}(\kappa)$ to rule out the discovery of implicants that are not prime.

Let us return to the formula $t_{Y_1}(\kappa)$ from Sect. 2.2 where $Y_1 = \{w, x, y\}$. The construction proceeds by introducing variables, denoted $v^{\pm}$ for each $v \in Y_1$, which serve as input to the sorting network. Each $v^{\pm}$ indicates whether v or $\neg v$ appear in the implicant, hence the relationship $v^{\pm} \leftrightarrow (v^{+} \vee v^{-})$. A 3-bit network is then used to constrain the output bits o_1, o_2, o_3 (top-to-bottom) to the unary sum of the inputs $w^{\pm}, x^{\pm}, y^{\pm}$ (again oriented top-to-bottom). Overall, this construction yields the following propositional encoding, where h_1, h_2, h_3 are intermediate variables computed by the comparators:

$$\begin{aligned}\mu = t_{Y_1}(\kappa) \wedge (w^{\pm} \leftrightarrow w^{+} \vee w^{-}) \wedge (x^{\pm} \leftrightarrow x^{+} \vee x^{-}) \wedge (y^{\pm} \leftrightarrow y^{+} \vee y^{-}) \wedge \\ (h_1 \leftrightarrow w^{\pm} \vee x^{\pm}) \wedge (h_2 \leftrightarrow w^{\pm} \wedge x^{\pm}) \wedge (h_3 \leftrightarrow h_2 \vee y^{\pm}) \wedge \\ (o_1 \leftrightarrow h_1 \vee h_3) \wedge (o_2 \leftrightarrow h_1 \wedge h_3) \wedge (o_3 \leftrightarrow h_2 \wedge y^{\pm})\end{aligned}$$

To enforce the cardinality constraint, we set $\mu_{k=1} = \mu \wedge o_1 \wedge \neg o_2 \wedge \neg o_3$. Invoking a SAT solver on $\mu_{k=1}$ yields candidates $\neg w$, x and $\neg y$, but only x is implied by $\exists Y_2 : \varphi$. Then $\mu_{k=1}$ is unsatisfiable, and we derive implicants for $\mu_{k=2} = \mu \wedge o_1 \wedge o_2 \wedge \neg o_3$, which yields the clause $w \vee y$ that is implied by $\exists Y_2 : \varphi$. Enumerating implicants by their size may require more SAT instances, but it ensures that the upper-approximation is always conjoined with a clause that is as short as possible. Short clauses are likely to remove more models from the approximation than long ones, thereby encouraging rapid convergence.

2.4 Solution-space reduction using instantiation

In the example in Sect. 2.2, a SAT solver generates several false candidates ν for implicants, which are then refuted by checking $\varphi \models \neg\nu$. This scheme is based on the observation that every implicant of $\forall Y_2 : \exists T : \kappa$ is also an implicant of $\exists Y_2 : \exists T : \kappa$, where in the case of the example $Y_2 = \{z\}$. However, observe that $\forall Y_2 : \exists Y : \kappa \models \exists Y : \kappa_{z \leftarrow 0}$ where $\kappa_{z \leftarrow 0}$ denotes the formula obtained by replacing

each occurrence of z in κ with the truth value 0 (instantiation). Therefore every implicant of $\forall Y_2 : \exists T : \kappa$ is also an implicant of $\exists T : \kappa_{z\leftarrow 0}$. The formula $\exists T : \kappa_{z\leftarrow 0}$ is not only a simplification of $\exists T : \kappa$ but $\exists T : \kappa_{z\leftarrow 0}$ will possess fewer models and hence fewer implicants than $\exists Y_2 : \exists T : \kappa$ provided $\kappa \not\models \neg z$.

Consider again the formula $t_{Y_1}(\kappa)$ given in Sect. 2.2 and consider $t_{Y_1}(\kappa_{z\leftarrow 0}) = t_{Y_1}(\kappa)_{z\leftarrow 0}$. Recall that originally the candidate implicant $\nu = (\neg w)$ was derived which was then refuted because $\varphi \not\models \neg\nu$. This candidate is suppressed by the instantiation and is not a solution of $t_{Y_1}(\kappa)_{z\leftarrow 0}$. It turns out that 13 SAT instances are required to converge onto $\exists Y_2 : \varphi$ whereas operating on $t_{Y_1}(\kappa)_{z\leftarrow 0}$ and $t_{Y_1}(\kappa)_{z\leftarrow 1}$ only requires 9 and 10 SAT instances, respectively. Interestingly, the formulae derived for these cases are equivalent but different. For $t_{Y_1}(\kappa)_{z\leftarrow 0}$ we obtain the limit $(\neg x) \wedge (w \vee y)$ as expected, but operating on $t_{Y_1}(\kappa)_{z\leftarrow 1}$ yields $(w \vee y) \wedge (w \vee \neg x) \wedge (\neg w \vee \neg x)$ which is equivalent to $(\neg x) \wedge (w \vee y)$.

2.5 Solution-space reduction using multiple instantiations

Instantiating the variables of Y_2 with truth values can decrease the number of spurious implications that are generated. This suggests instantiating κ in several different ways and then combining the instantiations so as to limit the search space a priori. Thus the basic idea is to derive multiple instantiations, say, $t_{Y_1}(\kappa)_{z\leftarrow 0}$ and $t_{Y_1}(\kappa)_{z\leftarrow 1}$ and solve the conjunction $\mu = t_{Y_1}(\kappa)_{z\leftarrow 0} \wedge t_{Y_1}(\kappa)_{z\leftarrow 1}$. In actuality, care is needed to avoid accident coupling between the T variables in the different instantiations. This can be avoided by introducing fresh, disjoint sets of variables $T_1 = \{t_{i,1} \mid t_i \in T\}$ and $T_2 = \{t_{i,2} \mid t_i \in T\}$ by applying renamings $\rho_1(t_i) = t_{1,i}$ and $\rho_2(t_i) = t_{2,i}$ to $\kappa_{z\leftarrow 0}$ and $\kappa_{z\leftarrow 1}$, respectively. By applying these renamings, combining and then applying simplification we obtain:

$$\mu = \begin{cases} (x^+ \vee t_{1,1}) & \wedge (y^- \vee t_{1,2}) & \wedge & (\neg t_{1,1} \vee \neg t_{1,2}) \wedge \\ (x^+ \vee t_{2,3}) & \wedge (w^+ \vee t_{2,3}) & \wedge (w^- \vee t_{2,4}) & \wedge (\neg t_{2,3} \vee \neg t_{2,4}) \wedge \\ (\neg w^+ \vee \neg w^-) & \wedge (\neg x^+ \vee \neg x^-) & \wedge (\neg y^+ \vee \neg y^-) & \end{cases}$$

When solving for μ, the sequence of upper-approximations converges onto the limit $(w \vee y) \wedge (w \vee \neg x) \wedge (\neg w \vee \neg x)$ without encountering any spurious implicants. Observe too that μ consists of 10 clauses whereas the $t_{Y_1}(\kappa)$ formula given in Sect. 2.2 has 13 clauses. This is because instantiating the variables of Y_2 often confers significant opportunities for simplification, offering scope for applying multiple instantiation without generating a formula that is unwieldy.

3 Correctness of the Transformation

The techniques presented thus far for computing under- and over-approximations of existentially quantified formula all rest on finding an implicant of a formula of the form $\exists Y_2 : \varphi$ (Sect. 2.1) or $\exists Y_2 : \exists T : \kappa$ (Sect. 2.2 onwards). The transformation t_{Y_1} reduces this problem SAT. This section is concerned with correctness of this transformation. The style of presentation is necessarily formal and a reader who is concerned with the application of the technique (rather than establishing its correctness) can proceed onto the following section.

3.1 Transforming clauses

Let $Bool_X$ denotes the class of propositional formulae over the set of variables X and suppose X is partitioned into two disjoint subsets Y_1 and Y_2. We shall consider the problem of computing an implicant of $\exists Y_2 : f$ where the formula $f \in Bool_X$ is presented in CNF. The transformation is formalised as a map t_{Y_1} on the set of literals $Lit_X = \{x, \neg x \mid x \in X\}$. This map is, in turn, defined in terms of sets of propositional variables $Y_1^+ = \{x^+ \mid x \in Y_1\}$ and $Y_1^- = \{x^- \mid x \in Y_1\}$ for which we assume that $Y_1^+ \cap Y_1^- = \emptyset$ and $(Y_1^+ \cup Y_1^-) \cap X = \emptyset$.

Definition 1. The literal transformation map $t_{Y_1} : Lit_X \to Lit_{Y_1^+ \cup Y_1^- \cup Y_2}$ (and its inverse $t_{Y_1}^{-1}$) are defined as follows:

$$t_{Y_1}(l) = \begin{cases} x^+ \text{ if } l = x \wedge x \in Y_1 \\ x^- \text{ if } l = \neg x \wedge x \in Y_1 \\ l \text{ otherwise} \end{cases} \qquad t_{Y_1}^{-1}(l) = \begin{cases} x \text{ if } l = x^+ \wedge x \in Y_1 \\ \neg x \text{ if } l = x^- \wedge x \in Y_1 \\ l \text{ otherwise} \end{cases}$$

A clause is considered to be a set of literals to simplify the lifting of the literal transformation map from single literals to clauses. Thus if a clause is merely a set $C \subseteq Lit_X$ then $t_{Y_1}(C) = \{t_{Y_1}(l) \mid l \in C\}$.

3.2 Transforming cubes

The literal transformation map is lifted to cubes and implicants (an implicant is a merely a particular type of cube) by likewise considering these to be sets of (implicitly conjoined) literals. The transformation relates cubes with literals drawn from Lit_X to cubes with literals drawn from $Y_1^+ \cup Y_1^- \cup Lit_{Y_2}$. Our interest is in cubes that are non-trivial, that is, they do not contain opposing literals. These classes of non-trivial cubes are defined below:

Definition 2.

$$Cube_X = \left\{ C \subseteq Lit_X \,\middle|\, \forall x \in X : \{x, \neg x\} \not\subseteq C \;\right\}$$

$$Cube_{Y_1,Y_2} = \left\{ C \cup C' \,\middle|\, \begin{array}{l} C \in Cube_{Y_2} \qquad\qquad\qquad\qquad \wedge\, C' \subseteq Y_1^+ \cup Y_1^- \wedge \\ \forall x \in Y_1 : \{x^+, x^-\} \cap C' \neq \emptyset \wedge \{x^+, x^-\} \not\subseteq C' \end{array} \right\}$$

We transform between these two types of cubes with the following map:

Definition 3. The mapping $c_{Y_1} : Cube_X \to Cube_{Y_1,Y_2}$ is defined:

$$c_{Y_1}(C) = t_{Y_1}(C) \cup \{\neg x^+, \neg x^- \mid x \in Y_1 \wedge \{x, \neg x\} \cap C = \emptyset\}$$

Observe that c_{Y_1} is both injective and surjective, hence it possesses an inverse $c_{Y_1}^{-1} : Cube_{Y_1,Y_2} \to Cube_X$.

3.3 Equivalence

With the t_{Y_1} and c_{Y_1} maps defined on clauses and cubes, we can now state an equivalence result which details how implicants are preserved by transformation. Note that a formula f represented in CNF can be considered to be a set of implicitly conjoined clauses F.

Proposition 1 (equivalence). Let $f = \bigwedge\{\bigvee C \mid C \in F\}$ where $F \subseteq \wp(Lit_X)$ and put $f' = \bigwedge\{\bigvee t_{Y_1}(C) \mid C \in F\}$. Then

- If $D \in Cube_X$ and $(\bigwedge D) \models f$ then $(\bigwedge c_{Y_1}(D)) \models f'$
- If $D' \in Cube_{Y_1,Y_2}$ and $(\bigwedge D') \models f'$ then $(\bigwedge c_{Y_1}^{-1}(D')) \models f$

Proof.

- Let $C \in F$. Since $(\bigwedge D) \models f$ it follows $(\bigwedge D) \models (\bigvee C)$.
  - Suppose $x \in D \cap C$ and $x \in Y_1$. Then $x^+ \in t_{Y_1}(C) \cap c_{Y_1}(D)$.
  - Suppose $\neg x \in D \cap C$ and $x \in Y_1$. Then $x^- \in t_{Y_1}(C) \cap c_{Y_1}(D)$.
  - Suppose $x \in D \cap C$ and $x \in Y_2$. Then $x \in t_{Y_1}(C) \cap c_{Y_1}(D)$.
  - Suppose $\neg x \in D \cap C$ and $x \in Y_2$. Then $\neg x \in t_{Y_1}(C) \cap c_{Y_1}(D)$.

  Hence $(\bigwedge c_{Y_1}(D)) \models (\bigvee t_{Y_1}(C))$ whence $(\bigwedge c_{Y_1}(D)) \models f'$ as required.
- Let $C \in F$. Since $(\bigwedge D') \models f'$ it follows $(\bigwedge D') \models (\bigvee t_{Y_1}(C))$.
  - Suppose $x^+ \in D' \cap t_{Y_1}(C)$ and $x \in Y_1$. Then $x \in C \cap c_{Y_1}^{-1}(D')$.
  - Suppose $x^- \in D' \cap t_{Y_1}(C)$ and $x \in Y_1$. Then $\neg x \in C \cap c_{Y_1}^{-1}(D')$.
  - Suppose $x \in D' \cap t_{Y_1}(C)$ and $x \in Y_2$. Then $x \in C \cap c_{Y_1}^{-1}(D')$.
  - Suppose $\neg x \in D' \cap t_{Y_1}(C)$ and $x \in Y_2$. Then $\neg x \in C \cap c_{Y_1}^{-1}(D')$.

  Hence $(\bigwedge c_{Y_1}^{-1}(D')) \models (\bigvee C)$ whence $(\bigwedge c_{Y_1}^{-1}(D')) \models f$ as required.

The proof for this and other results is given in an appendix (which will be made available on-line). The following corollary of the above relates implicants with literals drawn from Lit_{Y_1} to the satisfiability of the transformed clause set:

Corollary 1. Suppose f and f' are defined as above. Then

- If $D \in Cube_{Y_1}$ and $\wedge D \models f$ then $(\bigwedge c_{Y_1}(D)) \wedge f'$ is satisfiable
- If $D' \in Cube_{Y_1,\emptyset}$ and $(\bigwedge D') \wedge f'$ is satisfiable then $(\bigwedge c_{Y_1}^{-1}(D')) \models f$

To present the final result, let $[\![f]\!] \subseteq \wp(X)$ denote the set of models of the Boolean function f. (Recall the set-based representation of a model given in Sect 2.1, for example, if $X = \{x, y\}$ then $[\![x \vee y]\!] = \{\{x\}, \{y\}, \{x, y\}\}$.) We can now that state how a prime implicant of the existentially quantifier formula (whose literals are drawn from Lit_{Y_1}) fulfills two satisfiability conditions:

Corollary 2. Suppose f, f' and $F \subseteq \wp(Lit_X)$ are defined as above. Put $g' = f' \wedge \{\neg x^+ \vee \neg x^- \mid x \in Y_1\}$. Then $D \in Cube_{Y_1}$ is a prime implicant of $\exists Y_2 : f$ iff $D = c_{Y_1}^{-1}(M^* \cap (Y_1^+ \cup Y_1^-))$ where

- $M^* \in [\![g']\!]$
- $|M^* \cap (Y_1^+ \cup Y_1^-)| \leq |M \cap (Y_1^+ \cup Y_1^-)|$ for all $M \in [\![g']\!]$

Note that g' does not include any cardinality constraint on the set $M^* \cap (Y_1^+ \cup Y_1^-)$, hence the need to define a prime implicant in terms of an implicant no longer than any other. The above result can straightforwardly adapted to specify how an implicant of a given size can be defined as a SAT instance.

4 Experimental Results

We have implemented the techniques described in this paper in JAVA using the SAT4J solver [23] so as to integrate with our analysis framework for machine code, [MC]SQUARE [32], which is also coded in JAVA. To encode sorting propositionally, we implemented optimal networks for 9 or fewer variables and resorted to bitonic sorting for larger networks [21]. All experiments were performed on a MacBook Pro equipped with a 2.6 GHz dual-core processor and 4 GB of RAM, but only a single core was used in our experiments. The results obtained for deriving upper-approximations using the combination of methods described in Sect. 2.2 and Sect. 2.3 (without applying instantiation) are summarised in Tab. 1.

The formulae originated from the ISCAS benchmark set [17]. For some of these benchmarks, quantifier elimination by model enumeration is intractable due to the large numbers of models presented in column *#models*, and so is resolution. This is highlighted by the benchmark 74L85b, which describes a 4-bit magnitude comparator. Whereas model enumeration required more than 6 minutes for 74182b and 74283b, it ran out of memory for 74L85b after approximately 10 minutes. Column *#vars/clauses* shows the number of propositional variables and clauses in the original formula, whereas column *trans* gives these numbers after applying the transformation t_{Y_1}. The column *length* first contains the maximum length of prime implicants that were enumerated, followed by the size of Y_1. Thus in the 8/8 case the algorithm was run to completion, whereas the 2/8 case was terminated prematurely. Then *#primes* gives the number of implicants found and *#SAT* the total number of calls to a SAT solver. The overall runtime is given in the last column.

It is important to appreciate that the projection of the 74185b formula does not contain any implicants with size between 7 and 10. Likewise 74283b does not contain any implicants of size 7 and 8. This size distribution has been observed elsewhere [19], though not in the context of projection, which suggests that enumerating implicants up to a size threshold can achieve a good approximation of the projection. The ratio of number of calls to the solver to the number of primes

Table 1. Experimental results without instantiation

Formula	models	#vars/clauses	trans.	length	#primes	#SAT	runtime
74182b	262,144	227/526	780/1281	2/5	4	52	0.81
				5/5	4	170	1.80s
74283b	262,144	266/646	966/1,633	4/8	13	1590	5.63s
				6/8	20	4053	14.49s
				8/8	20	4881	16.71s
74L85b	>390,752	412/1084	1582/2747	4/10	6	4496	18.91s
				5/10	14	12349	57.22s
				6/10	30	24960	125.99s
				8/10	30	47536	292.59s
				10/10	30	51522	352.95s

Table 2. Experimental results with a single instantiation

Formula	length	runtime	speedup
74182b	2/5	0.50s	38%
	5/5	0.85s	52%
74283b	4/8	4.26s	24%
	6/8	10.54s	27%
	8/8	12.34s	26%

Formula	length	runtime	speedup
74L85b	4/10	12.61s	23%
	5/10	38.85s	32%
	6/10	84.68s	33%
	8/10	203.45s	30%
	10/10	84.68s	33%

is largely due to spurious candidates (in our experiments, it roughly doubled by increasing the prime length by one or two), which motivates investigating the impact of instantiating variables. Circuits can be simplified after applying instantiation, which involves removing false literals from clauses and removing all clauses that were already satisfied. The effects of single instantiation based on a model of the original formula are highlighted in Tab. 2. The results shown in column *speedup* suggest that instantiation can significantly increase performance.

Finally, we study applying multiple instantiation, accompanied with simplification, for different instances of the 74L85b circuit. Note that simplification reduces the size of the SAT instance which compensates somewhat for multiple instantiation. The instantiations themselves were generated from various models of the formula that were themselves found by applying blocking clauses. By choosing 6 instantiations that constrain the solution space in the 6/10 case a priori, the number of SAT instances reduced from 24960 to 16954, and the runtime decreased to 61.59s. This is a reduction of 32% in terms of the number of calls to a SAT solver and an overall speedup of 51%. Using 10 instantiations, reduced the number of calls to the solver was still further to 14273 and took the runtime down to 52.45s yielding a speedup of 58%. The key point is that a reduction occurs in the ratio of the number of calls to the SAT solver and the number of primes. This is a measure of the effectiveness of the technique, that is, how much effort is needed, on average, to find another implicant and thereby refine the approximation. However, we conjecture, that it is not prudent to apply too many instantiations simultaneously, because at some point the size of the combined SAT instance will become unmanageable (this would correspond to a flattening of quantified bit-vector logic, which can be prohibitively expensive).

5 Related Work

The consensus method has been independently proposed by a number of researchers [3, 29, 31] as a way of enumerating all the prime implicants of a propositional function in disjunctive normal form (DNF). If f is in CNF, then it is straightforward to derive a DNF representation of $\neg f$, to which the consensus procedure can be applied to find its prime implicants. Then $\exists Y : f$ can be found by conjoining all clauses $\neg c$ where c is a prime implicant of $\neg f$ which has no variables in common with Y. One might think that this provides a way to

compute projection, but the key step of the consensus method combines two elementary conjunctions of $\neg f$, say, $x \wedge C$ and $(\neg x) \wedge D$, to form the conjunction $C \wedge D$, which is isomorphic to resolution. Hence the consensus method shares the inefficiency problems associated with applying resolution to a formula in CNF. The complexity of the shortest implicant problem for DNF formulae has been studied by Umans [34] who showed that it is $GC(\log^2(n), coNP)$-complete. Even though this result is not directly transferrable to CNF, it substantiates our application of SAT solvers to the derivation of shortest implicants. Integer linear programming techniques have also be used to find shortest implications, as have SAT engines which have been modified to support inequalities [24]. In this work a transformation is described which is similar to t_{Y_1}. However, the work is not concerned with quantifier elimination, hence pairs of 0-1 variables are introduced for each variable in the formula rather than merely those in Y_1.

Operating on negated formulae has applications in bounded model checking [8], in particular when using Craig interpolants [25]. Given two inconsistent formulae φ and ψ, that is, $\varphi \wedge \psi$ is unsatisfiable, a smaller upper-approximation ξ of φ can be derived from the proof of unsatisfiability of $\varphi \wedge \psi$ in linear time. This approach is sound in the sense that ξ over-approximates φ, and at the same time serves tractability, and thus can be regarded as a form of widening. Prime implicants have been directly applied to widening Boolean functions represented as ROBDDs [19]. By appling a recursive meta-product construction [12] collections of short primes can be used to derive an ROBDD that is an upper-approximation of the input. Our work on applying SAT to projection was motivated by the emperical finding that collections of short primes, for instance those up to length 5, often yield good approximations of Boolean formulae [19]. Note that SAT-based enlargement of cubes also appears in the work of McMillan [26], who uses SAT-based enumeration for existential quantification. The idea of instantiating (multiple) instances of Boolean formulae with models can be seen as a form of circuit co-factoring as described by Ganai et al. [15]. A recent contribution to reasoning about quantified bit-vector formulae was made by Wintersteiger et al. [36], who most notably used word-level simplifications and template instantiations.

Another approach to quantifier elimination (of linear systems) was recently proposed by Monniaux [27]. In his approach, satisfiability tests of quantified formulae are used to derive witnesses (models). Rather than computing quantifier-free formulae directly, his algorithm uses substitution of witnesses to extend the original system towards a quantifier-free formula. Comparing this technique to our method, a similarity is in the use of witnesses to guide the elimination process. His method, however, is not anytime, and thus, cannot be stopped prematurely.

6 Conclusions

Synopsis This paper advocates using SAT to derive upper-approximations of existentially quantified propositional formulae The approach is designed to be anytime so that it can be stopped early without compromising correctness. This can be considered to be a pragmatic response to the complexity of projec-

tion [11]. Further, the technique avoids the blow-up in the number of clauses in an intermediate representation that is associated with eliminating variables with resolution.

Future Work This work calls for further investigations of ways to reduce the number of spurious candidates that appear when implicants of negations are enumerated, possibly based on the recent work described in [36].

Acknowledgment We thank Olivier Coudert for discussions on the complexity of finding the smallest prime implicant. This work was funded, in part, by a Royal Society travel grant, reference TG092357, and a Royal Society Industrial Fellowship, reference IF081178. Furthermore, we thank Professor Stefan Kowalewski for his generous financial support that was necessary to initiate our collaboration.

References

1. T. Armstrong, K. Marriott, P. Schachte, and H. Søndergaard. Two Classes of Boolean Functions for Dependency Analysis. *Science of Computer Programming*, 31(1):3–45, 1998.
2. E. A. Bender. *Mathematical Methods in Artificial Intelligence*. IEEE Computer Society Press, 1996.
3. A. Blake. *Canonical expressions in Boolean algebra*. University of Chicago, 1938.
4. J. Brauer and A. King. Automatic Abstraction for Intervals using Boolean Formulae. In *SAS*, volume 6337 of *LNCS*, pages 167–183. Springer, 2010.
5. R. E. Bryant. Symbolic Boolean Manipulation with Ordered Binary-Decision Diagrams. *ACM Computing Surveys*, 24(3):293–318, 1992.
6. R. E. Bryant. A View from the Engine Room: Computational Support for Symbolic Model Checking. In *25 Years of Model Checking*, volume 5000 of *LNCS*, pages 145–149. Springer, 2008.
7. J. R. Burch, E. M. Clarke, and K. L. McMillan. Symbolic model checking: 10^{20} states and beyond. *Information and Computation*, 98:142–170, 1992.
8. E. Clarke, D. Kroening, and F. Lerda. A tool for checking ANSI-C programs. In *TACAS*, volume 2988 of *LNCS*, pages 168–176. Springer, 2004.
9. E. M. Clarke, A. Biere, R. Raimi, and Y. Zhu. Bounded model checking using satisfiability solving. *Formal Methods in System Design*, 19(1):7–34, 2001.
10. B. Cook, D. Kroening, P. Rümmer, and C. Wintersteiger. Ranking Function Synthesis for Bit-Vector Relations. In *TACAS*, volume 6015 of *LNCS*, pages 236–250. Springer, 2010.
11. S. Coste-Marquis, D. Le Berre, F. Letombe, and P. Marquis. Complexity Results for Quantified Boolean Formulae Based on Complete Propositional Languages. *JSAT*, (1):61–88, 2006.
12. O. Coudert and J. C. Madre. Implicit and Incremental Computation of Primes and Essential Primes of Boolean Functions. In *DAC*, pages 36–39. IEEE, 1992.
13. E. Duesterwald, R. Gupta, and M. L. Soffa. A Practical Framework for Demand-Driven Interprocedural Data Flow Analysis. *ACM TOPLAS*, 19(6):992–1030, 1997.
14. N. Eén and N. Sörensson. Translating Pseudo-Boolean Constraints into SAT. *JSAT*, 2(1-4):1–26, 2006.

15. M. K. Ganai, A. Gupta, and P. Ashar. Efficient SAT-based unbounded symbolic model checking using circuit cofactoring. In *ICCAD*, pages 510–517. IEEE, 2004.
16. S. Genaim, R. Giacobazzi, and I. Mastroeni. Modeling Secure Information Flow with Boolean Functions. In *IFIP WG 1.7, ACM Workshop on Issues in the Theory of Security*, pages 55–66, Barcelona, Spain, 2004.
17. M. C. Hansen, H. Yalcin, and J. P. Hayes. Unveiling the iscas-85 benchmarks: A case study in reverse engineering. *IEEE Design & Test of Computers*, 16(3):72–80, 1999.
18. J. M. Howe and A. King. Positive Boolean Functions as Multiheaded Clauses. In *ICLP*, volume 2237 of *LNCS*, pages 120–134. Springer, 2001.
19. N. Kettle, A. King, and T. Strzemecki. Widening ROBBDs with Prime Implicants. In *TACAS*, volume 3920 of *LNCS*, pages 105–119. Springer, 2006.
20. A. King and H. Søndergaard. Automatic Abstraction for Congruences. In *VMCAI*, volume 5944 of *LNCS*, pages 281–293. Springer, 2010.
21. D. E. Knuth. Sorting and Searching. In *The Art of Computer Programming*, volume 3. Addison-Wesley, 1997.
22. D. Kroening and O. Strichman. *Decision Procedures*. Springer, 2008.
23. D. Le Berre. SAT4J: Bringing the power of SAT technology to the Java platform, 2010. http://www.sat4j.org/.
24. V. M. Manquinho, P. F. Flores, J. P. M. Silva, and A. L. Oliveira. Prime implicant computation using satisfiability algorithms. In *International Conference on Tools with Artificial Intelligence*, pages 232–239. IEEE Press, 1997.
25. K. McMillan. Interpolation and SAT-based model checking. In *CAV*, volume 2725 of *LNCS*, pages 1–13. Springer, 2003.
26. K. L. McMillan. Applying SAT methods in unbounded symbolic model checking. In *CAV*, volume 2404 of *LNCS*, pages 250–264. Springer, 2002.
27. D. Monniaux. Quantifier Elimination by Lazy Model Enumeration. In *CAV*, volume 6174 of *LNCS*, pages 585–599. Springer, 2010.
28. D. A. Plaisted and S. Greenbaum. A structure-preserving clause form translation. *Journal of Symbolic Computation*, 2(3):293–304, September 1986.
29. W. V. Quine. A Way to Simplify Truth Functions. *American Mathematical Monthly*, 62(9):627–631, 1995.
30. T. Reps, M. Sagiv, and G. Yorsh. Symbolic Implementation of the Best Transformer. In *VMCAI*, volume 2937 of *LNCS*, pages 252–266. Springer, 2004.
31. E. W. Samson and B. E. Mills. Circuit minimization: Algebra and Algorithms for new Boolean canonical expressions. Technical Report TR 54-21, United States Air Force, Cambridge Research Lab, 1954.
32. B. Schlich. Model checking of software for microcontrollers. *ACM Trans. Embedded Comput. Syst.*, 9(4), 2010. Article Number 36.
33. G. S. Tseitin. On the complexity of derivation in the propositional calculus. In A. O. Slisenko, editor, *Studies in Constructive Mathematics and Mathematical Logic*, volume Part II, pages 115–125, 1968.
34. C. Umans. The Minimum Equivalent DNF Problem and Shortest Implicants. In *FOCS*, pages 556–563. IEEE Press, 1998.
35. J. Whittemore, J. Kim, and K. Sakallah. SATIRE: a new incremental satisfiability engine. In *Design Automation Conference*, pages 542–545. ACM, 2001.
36. C. M. Wintersteiger, Y. Hamadi, and L. de Moura. Efficiently solving quantified bit-vector formulas. In *FMCAD*, 2010. To appear.