Parametric Quantified SAT Solving

Thomas Sturm
Departamento de Matemáticas, Estadística y Computación, Universidad de Cantabria
39071 Santander, Spain
sturmt@unican.es

Christoph Zengler
Symbolic Computation Group
Wilhelm-Schickard-Institut, Universität Tübingen
72076 Tübingen, Germany
zengler@informatik.uni-tuebingen.de

ABSTRACT

We generalize successful algorithmic ideas for quantified satisfiability solving to the parametric case where there are parameters in the input problem. The output is then not necessarily a truth value but more generally a propositional formula in the parameters of the input. Since one can naturally embed propositional logic into first-order logic over Boolean algebras, our work amounts from a model-theoretic point of view to a quantifier elimination procedure for initial Boolean algebras. Our work is completely and efficiently implemented in the logic package Redlog contained in the open source computer algebra system Reduce. We describe this implementation and discuss computation examples pointing at possible applications of our work to configuration problems in the automotive industry.

Categories and Subject Descriptors

F.2.2 [**Analysis of Algorithms and Problem Complexity**]: Nonnumerical Algorithms and Problems—*Computations on Discrete Structures*; F.4.1 [**Mathematical Logic and Formal Languages**]: Mathematical Logic—*Computational Logic*; G.4 [**Mathematical Software**]: Algorithm design and analysis; I.1.2 [**Symbolic and Algebraic Manipulation**]: Algorithms

General Terms

Algorithms, Performance, Theory

Keywords

Propositional Logic, SAT, QSAT, Parameters, Generalization, Quantifier Elimination

1. INTRODUCTION

Satisfiability solving (SAT) and quantified satisfiability solving (QSAT) for propositional logic are canonical complete problems for the complexity classes NP and PSPACE, respectively [5, 10]. Especially in the last 15 years there has been considerable research in these areas [16, 17, 27, 8]. This is motivated on the one hand by the increasing number of practical applications like bounded model checking [4, 3]. On the other hand, finding efficient heuristics for these two problems provides via polynomial reduction efficient algorithms for all problems in NP and PSPACE, respectively. Furthermore algorithmic ideas from SAT and QSAT have been successfully transferred to dedicated algorithms for e.g., graph coloring or constraint satisfaction problems.

In an earlier paper [21], the first author and others have discussed how to formally integrate SAT and QSAT into their first-order computer logic system Redlog [7]. Their approach is essentially to consider first-order formulas in the language of Boolean algebras over the theory of initial Boolean algebras, i.e., Boolean algebras freely generated by the empty set. Then first-order formulas can be brought into a normal form, where every atomic formula is of the form $v = 0$ or $v = 1$ for variables v. This way, quantifier-free formulas can be directly interpreted and presented to the user as propositional formulas. SAT then amounts to deciding the existential closure of a given formula, and QSAT corresponds to the decision of an arbitrary first-order sentence.

Decision procedures for first-order theories have historically often been in fact quantifier elimination procedures, even long before this term was formally introduced. Prominent examples are Presburger's completeness proof for the additive theory of the integers or Tarski's decision procedure for real closed fields [19, 24]. Based on the virtual substitution approach [26] it was straightforward to devise in [21] a quantifier elimination procedure for initial Boolean algebras. Since variable-free atomic formulas are obviously decidable this quantifier elimination procedure covers in particular both SAT and QSAT. Additionally, it admits to consider formulas, where only some variables are quantified while others remain free and are considered parameters of the problem. One then obtains via quantifier elimination a quantifier-free formula exclusively in the parameters that is equivalent to the input formula. We refer to this procedure as *parametric quantified satisfiability solving* (PQSAT). Elsewhere this has been referred to as *open QBF* [1].

The asymptotic worst-case time complexity of the procedure in [21] is bounded by a single exponential function in the input length, where the input problem needs not be in any Boolean normal form. It is not hard to see that this bound is tight for the considered problem. However, the existing successful research on SAT and QSAT teaches us that such crude complexity considerations are not sufficient to draw conclusions about the practical applicability of an al-

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
ISSAC 2010, 25–28 July 2010, Munich, Germany.
Copyright 2010 ACM 978-1-4503-0150-3/10/0007 ...$10.00.

gorithm. In fact, from the benchmarks in [21] it is quite clear that this work could not compete with state-of-the art SAT or QSAT checkers. From a SAT solving point of view the procedure applied to sentences was roughly an implementation of the classical DLL algorithm [6] without learning and without non-chronological backtracking facilities.

In the present paper, we are now going to generalize to PQSAT more sophisticated and practically successful approaches to SAT and QSAT solving, viz. DLL with conflict driven clause learning and non-chronological backtracking. We have implemented our algorithm PQSAT including the underlying QSAT solver in the logic package Redlog contained in the open-source computer algebra system Reduce[1]. Our PQSAT uses QSAT mostly as a black box, which can easily be replaced by alternative QSAT solvers. On the basis of our description here, existing QSAT solvers can be extended to PQSAT in such a way that their performance on regular QSAT problems is not affected.

The plan of this paper as follows: In Section 2 we summarize the design and the properties of our underlying implementations of SAT and QSAT. Basic definitions and concepts introduced there are going to be reused for the description of PQSAT (and its specialization PSAT) in Section 3. We prove correctness and termination of our procedure and give some upper bounds for the asymptotic worst-case complexity. Section 4 presents an application example in the area of product configuration in the automotive industry and analyzes the performance of our methods and implementations by means comprehensive benchmarks taken from the literature as well as from industrial cooperation projects. Section 5 finally points at some future research directions.

2. REVISION OF SAT AND QSAT

In this section we are going to summarize the design and the properties of DLL SAT solvers and QSAT solvers to the extent necessary to describe in the following section our extensions of this approach to the parametric case. We also make clear, which design decisions we have taken for the implementation of our own underlying QSAT solver.

2.1 SAT

There has been considerable research on stochastic local search algorithms for SAT solving [22, 20]. This led to considerable improvements (1.324^n instead of 2^n) of the upper bound for the asymptotic worst-case complexity [12]. While these probabilistic algorithms perform very well on random input, the vast majority of SAT solvers successfully applied to real-world problems uses the *Davis–Logemann–Loveland (DLL) approach* [6]. All the winners of the last years' SAT Races fall into this category including RSat [18], MiniSAT [8], and PicoSAT [2]. Since we are ourselves interested in practical applicability at the first place, we are going to focus on DLL here.

DLL is basically a complete search in the search space of all 2^n variable assignments with early cuts in the search tree when an unsatisfiable branch is detected. Based on this approach Silva and Sakallah have introduced a concept referred to as *clause learning* [16]. Their approach extends the classical DLL approach by *non-chronological backtracking* and *automatic learning of new clauses* in case of conflicts. Therefore this approach is referred to as *conflict-driven clause learning* (CDCL). All successful solvers mentioned above are in fact CDCL solvers. CDCL solvers are CNF-only solvers, meaning the input formula must be converted into CNF before solving.

We use the standard notation of propositional logic with propositional variables from a suitable infinite set $\mathcal{V}$, Boolean operators $\neg$, $\vee$, $\wedge$, and Boolean constants true and false. An *assignment* is a partial function $\alpha : \mathcal{V} \to \{\top, \bot\}$ mapping variables to truth values. We write $x \leftarrow b$ for $\alpha(x) = b$, we denote by $\mathrm{dom}(\alpha) \subseteq \mathcal{V}$ the variables that have been assigned, and we follow the convention to write $\alpha \models \varphi$ when some formula φ holds with respect to α. We write $\mathrm{vars}(\varphi)$ to denote the finite set of variables occurring in a formula φ. A *conjunctive normal form* (CNF) is a conjunction of clauses. A *clause* is a disjunction $(\lambda_1 \vee \cdots \vee \lambda_n)$ of literals. Each *literal* λ_i is either a variable $x_i \in \mathcal{V}$ or its logical negation $\neg x_i$. It is convenient to identify a CNF with the set of all clauses contained in it. An *empty clause* is a clause where all x_i have been assigned truth values in such a way that all corresponding λ_i evaluate to false and therefore the whole clause is false. It is obvious that once reaching an empty clause, no extension of the corresponding assignment can satisfy the CNF formula containing that clause. We call the occurrence of an empty clause a *conflict*. A *unit clause* is a clause where all but one x_i have been assigned, all λ_j for $j \neq i$ evaluate to false, and therefore in order to satisfy the clause the remaining *unit variable* x_i must be assigned such that λ_i becomes $\top$. The process of detecting unit clauses and fixing the corresponding values for the unit variables is called *unit propagation*, which plays an important role in modern SAT solvers.

On each *decision level* CDCL assigns variables and performs unit propagation until either an empty clause arises or the formula is satisfied. If the formula is satisfied CDCL returns true. In the case of an empty clause a resolution-based learning process is started at the end of which CDCL learns a new clause and backtracks to a certain decision level. If an empty clause arises at level 0, CDCL returns false. There are various strategies how to learn the new clause in the conflict case. In Redlog we use the firstUIP strategy described in [17]. Since modern SAT solvers spend up to 90% of their time performing unit propagation it is crucial to implement this efficiently. In Redlog we use the common concept of *watched literals* [17]. The idea is to observe two literals in each clause. As long as these two have no assigned truth value, the clause cannot be unit. When one of the literals gets assigned and evaluates to false, another unassigned literal is chosen to be guarded instead. If there is no other literal, the clause is unit, and we can perform unit propagation. The last important adjusting screw is the heuristics how to choose the next variable to be assigned. Our implementation offers various choices of selection heuristics. The default is a variation of the MOM heuristic [9], which prefers literals with a maximum occurrence in clauses of minimal size. MOM turned out to perform very well on our benchmarks discussed in Subsection 4.2.

2.2 QSAT

While SAT implicitly assumes all variables to be existentially quantified, QSAT more generally expects for each variable an explicit quantification, either existential or universal. We assume here that formulas are in *prenex normal form*, where all quantifiers precede a CNF. Whenever using non-

[1]`http://reduce-algebra.sourceforge.net`

CNF formulas, we consider this an abbreviated notation for some equivalent formula in CNF. For detailed definitions of empty and unit clauses in the context of QSAT, see [27].

In the SAT case backtracking is only necessary when an empty clause is detected. For QSAT, in contrast, backtracking has to be performed possibly also in the case that one branch of the search tree is satisfiable: For each universally quantified variable x_i both branches, $x_i \leftarrow \top$ and $x_i \leftarrow \bot$, must be satisfiable. Therefore after assigning $x_i \leftarrow \bot$, there is a backtrack performed assigning $x_i \leftarrow \top$. We give the CDCL Algorithm for QSAT [27, 15, 11], which reflects the backtracking for universally quantified variables in lines 11–15.

Input: (φ, α), where φ is a fully quantified propositional formula, and α is an optional assignment for variables existentially quantified in the outermost block of φ, $\emptyset$ by default
Output: $(\tau, \varphi', \alpha')$, where $\tau \in \{\text{true}, \text{false}\}$, $\varphi' \longleftrightarrow \varphi$ with additional learned clauses, and α' the final variable assignment

```
1  label all bindings in α with level −1
2  level := 0
3  while true do
4      unitPropagation()
5      if φ has an empty clause then
6          level := analyseConflict()
7          if level = 0 then
8              return (false, φ, α)
9          backtrack(level)
10     else
11         if α ⊨ φ then
12             level := analyseSAT()
13             if level = 0 then
14                 return (true, φ, α)
15             backtrack(level)
16         else
17             level := level + 1
18             choose x ∈ vars(φ) \ dom(α) wrt. to the
               quantification level
19             α := α ∪ {(x ← ⊥, level)}
```

Algorithm 1: The QSAT algorithm QSAT(φ, α)

The procedure analyseConflict() learns a new clause and returns a suitable non-negative backtrack level. During the procedure analyseSAT() the value of the last universally quantified variable is flipped in order to search both branches in the search tree. Notice that when replacing lines 12–15 by "return true" we obtain the CDCL SAT algorithm as described in Subsection 2.1. In fact if there are no universally quantified variables then QSAT proceeds exactly like CDCL.

For the variable selection in line 18 there are essentially the same heuristics used as with SAT. There is, however, one important restriction: Successive quantifiers of the same type are grouped like

$$Q_1 x_1 \dots Q_1 x_{k_1} Q_2 x_{k_1+1} \dots Q_2 x_{k_2} \dots \varphi,$$

where $Q_i \in \{\exists, \forall\}$, $Q_{i+1} \neq Q_i$, and $x_j \in \text{vars}(\varphi)$. The index $i \in \mathbb{N}$ of Q_i is the *quantification level* of the corresponding quantified variables $x_{k_{i-1}+1}, \dots, x_{k_i}$. The variable selection heuristics must choose a variable from the smallest quantification level where there are still unassigned variables. Notice that in the worst case like $\exists x_1 \forall x_2 \exists x_3 \forall x_4 \dots \varphi$ one must successively pick x_1, x_2, x_3, x_4, ..., i.e., there is no choice at all.

It is noteworthy that in contrast to most existing implementations of QSAT we do not only return τ but also φ' and α. This additional information is required for the PQSAT algorithm described in the next section. We have verified for several existing QSAT solvers that the code can be easily adapted to meet our requirements. Similarly existing solvers can easily be adapted to accept the additional input parameter α, in fact some already do so.

3. PARAMETRIC QSAT

PQSAT generalizes QSAT by admitting *free*, i.e. non-quantified, variables. It is important to understand that these free variables are, *not* assumed to be implicitly existentially quantified but *parameters* of the described problem. Consequently, PQSAT does in general not *decide* its input problem by returning true or false. Instead the output is a *disjunctive normal form (DNF)* establishing necessary and sufficient conditions on the free variables for the existence of a satisfying assignment. In the special case that for all possible assignments of the free variables the corresponding instances of QSAT yield $\tau = \text{true}$, PQSAT will return true as well. Analogously, PQSAT yields false if all QSAT instances return $\tau = \text{false}$.

Algorithm 2 states the general PQSAT algorithm. For an input formula φ with free variables, we split the set vars(φ) of all variables into a set bvars(φ) of bound (quantified) variables and a set fvars(φ) of free variables .

Input: (φ, α), where φ is a quantified propositional formula possibly containing free variables, and α is an optional partial assignment for fvars(φ), $\emptyset$ by default
Output: (τ, φ'), where τ is a quantifier-free formula with vars$(\tau) = $ fvars(φ), and $\varphi' \longleftrightarrow \varphi$ with additional learned clauses

```
1  if fvars(φ) \ dom(α) = ∅ then
2      (σ, φ', β) := QSAT(∃φ, α)
3      if σ = true then
4          return (form(α), φ')
5      else
6          return (false, φ')
7  else
8      x := choose a variable from fvars(φ) \ dom(α)
9      α' := α ∪ {x ← ⊥}
10     ψ1 := false
11     if no conflict is reached after unit propagation then
12         (ψ1, φ) := PQSAT(φ, α')
13     α'' := α ∪ {x ← ⊤}
14     ψ2 := false
15     if no conflict is reached after unit propagation then
16         (ψ2, φ) := PQSAT(φ, α'')
17     return (simplify(ψ1 ∨ ψ2), φ)
```

Algorithm 2: The PQSAT algorithm PQSAT(φ, α)

In the degenerate case that there are no free variables at all, our PQSAT algorithm will reduce to one call to QSAT. Recall from the previous section that the QSAT algorithm

in turn reduces to a CDCL SAT algorithm in the more degenerate case of a purely existential problem.

The algorithm is recursive in both its input parameters. It is noteworthy that for the initial call α can be used to fix in advance the assignment for some subset of free variables when experimenting with a problem. Notice that the return value α of QSAT is not essentially used here; it will be in our optimization for PSAT discussed in Subsection 3.3

The main idea of PQSAT is to use essentially the classical DLL algorithm for the free variables. Whenever in that course all free variables have been assigned, we have got a QSAT subproblem, for which we call $\mathrm{QSAT}(\varphi)$ and obtain either $(\mathrm{true}, \varphi', \alpha)$ or $(\mathrm{false}, \varphi', \alpha)$.

In line 2 we construct the existential closure of φ in order to meet the specification of QSAT. The existential quantifiers introduced this way are actually semantically irrelevant as all corresponding variables are already assigned by α.

Observe in line 12 that we save in φ the original input formula augmented by clauses additionally learned during the first recursive PQSAT call. This is propagated in line 16 to the second recursive PQSAT call. This leads to the effect that we *transport* learned clauses from one QSAT call to the next and thus avoiding repeatedly arriving at the same conflicts. It is not hard to see that since learning happens via resolution and resolution is compatible with substitution of truth values for variables the learned clauses in fact remain valid. The idea is visualized in Figure 3. In order to limit the blow up of φ in that course we use *activity heuristics* after each run of QSAT to delete learned clauses that have not significantly produced new conflicts in the past.

If ψ_1 or ψ_2 is false in line 17 then $\mathrm{simplify}(\psi_1 \vee \psi_2)$ eliminates one superfluous false.

When QSAT returns (true, φ) we add a Boolean representation $\mathrm{form}(\alpha)$ of the current variable assignment α to the output formula and proceed with the algorithm. Following the usual convention that empty conjunctions are true this Boolean representation is defined as

$$\mathrm{form}(\alpha) = \bigwedge_{\substack{v \in \mathrm{dom}(\alpha) \\ v(\alpha) = \top}} v \wedge \bigwedge_{\substack{v \in \mathrm{dom}(\alpha) \\ v(\alpha) = \bot}} \neg v.$$

As an example consider

$$\mathrm{form}(\{x \leftarrow \top, y \leftarrow \bot, z \leftarrow \bot\}) = x \wedge \neg y \wedge \neg z.$$

It is easy to see that for any assignments α, α' we have $\mathrm{form}(\alpha) = \mathrm{form}(\alpha')$ iff $\alpha = \alpha'$. Furthermore:

Lemma 1 (Universal Property). *Let α be an assignment. Then the following hold:*

(i) $\mathrm{vars}(\mathrm{form}(\alpha)) = \mathrm{dom}(\alpha)$ *and* $\alpha \models \mathrm{form}(\alpha)$

(ii) If γ is a conjunction of literals and $\mathrm{vars}(\gamma) = \mathrm{dom}(\alpha)$ *and $\alpha \models \gamma$, then $\gamma = \mathrm{form}(\alpha)$ up to commutativity.* $\square$

Consider α with $\mathrm{dom}(\alpha) = \mathrm{fvars}(\varphi)$. Then up to commutativity $\mathrm{form}(\alpha)$ is the unique conjunction of literals with $\mathrm{vars}(\mathrm{form}(\alpha)) = \mathrm{fvars}(\varphi)$ and $\alpha \models \mathrm{form}(\alpha)$.

Similar to DLL after each assignment of a free variable in line 9 or 13 we use unit propagation with watched literals to propagate the current assignment. If we encounter an empty clause, we cut the search tree.

To conclude this subsection Figure 3 visualizes a computation of PQSAT with $\varphi = \exists x \forall y((x \vee y \vee \neg u) \wedge (\neg x \vee \neg y \vee w))$ and $\alpha = \emptyset$. Since QSAT yields true for the assignments

$$\{u \leftarrow \bot, w \leftarrow \bot\}, \quad \{u \leftarrow \bot, w \leftarrow \top\}, \quad \{u \leftarrow \top, w \leftarrow \top\}.$$

one finally obtains $\tau = (\neg u \wedge \neg w) \vee (\neg u \wedge w) \vee (u \wedge w)$.

3.1 Correctness and Termination

In this section we assume the correctness and termination of the CDCL QSAT procedure as proved in [27].

Lemma 2. *Let (τ, φ') be the return value of some PQSAT call. Then τ is in DNF.*

Proof. If $|\mathrm{fvars}(\varphi)| = 0$, then $\tau = \mathrm{form}(\alpha)$. By definition this is either true or a conjunction of literals, both of which are in DNF. For $|\mathrm{fvars}(\varphi)| = n + 1$ we obtain essentially $\tau = \psi_1 \vee \psi_2$ with $|\mathrm{fvars}(\psi_1)| = |\mathrm{fvars}(\psi_2)| = n$. According to the induction hypothesis both ψ_1 and ψ_2 are in DNF and so is $\psi_1 \vee \psi_2$. $\square$

Lemma 3. *Let φ be a quantified formula. Consider*

$$\mathrm{A} = \{\, \alpha \mid \mathrm{fvars}(\varphi) = \mathrm{dom}(\alpha),\ \mathrm{QSAT}(\varphi, \alpha) = (\mathrm{true}, \varphi', \alpha')\,\}.$$

Then $\varphi \longleftrightarrow \bigvee_{\alpha \in \mathrm{A}} \mathrm{form}(\alpha)$.

Proof. To start with, observe that for each $\alpha \in \mathrm{A}$ we have $\mathrm{fvars}(\mathrm{form}(\alpha)) = \mathrm{dom}(\alpha) = \mathrm{fvars}(\varphi)$ and thus

$$\mathrm{fvars}\Big(\bigvee_{\alpha \in \mathrm{A}} \mathrm{form}(\alpha)\Big) = \mathrm{fvars}(\varphi).$$

Let α_0 be an assignment with $\mathrm{dom}(\alpha_0) = \mathrm{fvars}(\varphi)$. Assume that $\alpha_0 \models \varphi$. Then $\mathrm{QSAT}(\varphi, \alpha_0) = (\mathrm{true}, \varphi', \alpha')$ for some φ', α'. It follows that $\alpha_0 \in \mathrm{A}$. Since $\alpha_0 \models \mathrm{form}(\alpha_0)$ we obtain $\alpha_0 \models \bigvee_{\alpha \in \mathrm{A}} \mathrm{form}(\alpha)$. Assume, vice versa, that $\alpha_0 \models \bigvee_{\alpha \in \mathrm{A}} \mathrm{form}(\alpha)$. Then $\alpha_0 \models \mathrm{form}(\alpha_1)$ for some $\alpha_1 \in \mathrm{A}$. It follows that

$$\mathrm{vars}(\mathrm{form}(\alpha_1)) = \mathrm{dom}(\alpha_1) = \mathrm{fvars}(\varphi) = \mathrm{dom}(\alpha_0).$$

By Lemma 1(ii) we obtain $\mathrm{form}(\alpha_1) = \mathrm{form}(\alpha_0)$, which in turn implies $\alpha_1 = \alpha_0$. On the other hand we know $\mathrm{QSAT}(\varphi, \alpha_1) = (\mathrm{true}, \varphi', \alpha')$ for some φ', α', and using the correctness of QSAT it follows that $\alpha_0 = \alpha_1 \models \varphi$. $\square$

Theorem 1 (Correctness of PQSAT). *Let φ be a quantified propositional formula and* $(\tau, \varphi') = \mathrm{PQSAT}(\varphi, \emptyset)$. *Then τ is quantifier-free and $\tau \longleftrightarrow \varphi$.*

Proof. By inspection of the algorithm we see that possible return values for τ are $\mathrm{form}(\alpha)$ (line 4) or false (line 6) or disjunctions of these (line 17), all of which are quantifier-free.

Consider the special case that for all assignments α with $\mathrm{dom}(\alpha) = \mathrm{fvars}(\varphi)$ there has been $\mathrm{QSAT}(\varphi, \alpha)$ called in line 2. Then it is easy to see that $\tau = \bigvee_{\alpha \in \mathrm{A}} \mathrm{form}(\alpha)$ as described in Lemma 3. Hence $\tau \longleftrightarrow \varphi$ by Lemma 3.

Consider now a particular assignment α_0 with $\mathrm{dom}(\alpha_0) = \mathrm{fvars}(\varphi)$ for which there has not been $\mathrm{QSAT}(\varphi, \alpha_0)$ called. According to lines 11 and 14 of PQSAT then there exists a partial assignment $\alpha_0' \subseteq \alpha_0$ causing a conflict, that is $\alpha_0' \models \varphi \longleftrightarrow \mathrm{false}$. It follows that $\alpha_0 \models \varphi \longleftrightarrow \mathrm{false}$ and accordingly $\mathrm{QSAT}(\varphi, \alpha_0) = (\mathrm{false}, \varphi', \alpha')$. Hence that missing QSAT call is irrelevant for the semantics of τ. $\square$

Theorem 2 (Termination of PQSAT). *PQSAT terminates.*

Proof. We have to show that there is no infinite recursion. Since $|\mathrm{fvars}(\varphi) \setminus \mathrm{dom}(\alpha)| \in \mathbb{N}$ decreases with every recursive call either in line 12 or line 16 due to the assignments in line 9 or line 13, respectively, the condition $\mathrm{fvars}(\varphi) \setminus \mathrm{dom}(\alpha) = \emptyset$ in line 1 finally becomes true, and the algorithm returns in line 4 or line 6. $\square$

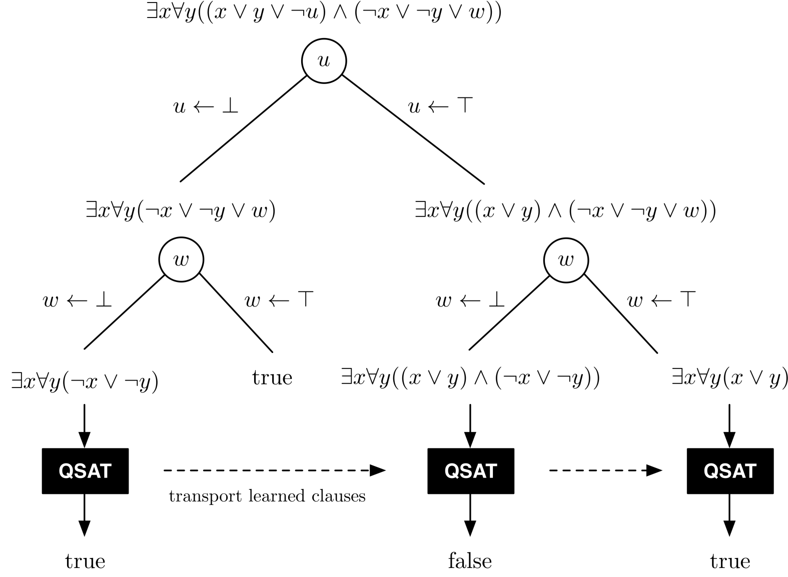

Figure 1: Example PQSAT computation

3.2 Complexity

THEOREM 3 (COMPLEXITY OF PQSAT). *Consider as complexity parameters $f = |\operatorname{fvars}(\varphi)|$ and $b = |\operatorname{bvars}(\varphi)|$. Then the asymptotic time complexity of* PQSAT *is bounded by 2^{f+b} in the worst case. In particular this complexity is bounded by $2^{\operatorname{length}(\varphi)}$.*

PROOF. Consider an input formula φ and let f and b be as above. QSAT is obviously bounded by 2^b. In PQSAT the QSAT algorithm is called at most 2^f times. We hence obtain $2^f \cdot 2^b = 2^{f+b}$. □

3.3 PSAT and Local Search

We consider the special case that the input formula of PQSAT does not contain any universal quantifier but possibly existential quantifiers and free variables. We are going to refer to such formulas as existential formulas. Naturally we refer to PQSAT for existential formulas as PSAT. For PSAT problems we use ideas from probabilistic SAT [22] to improve our PQSAT algorithm. The key idea is the following: When we have found a satisfying assignment we may expect that there are further satisfying assignments "close" to the found one. We are now going to make precise this idea.

The *Hamming distance* between two assignments is defined as the number of variables for which the assignments differ:

$$d(\alpha, \alpha') = |\{\, x \in \operatorname{dom}(\alpha) \mid \alpha(x) \neq \alpha'(x) \,\}|.$$

The *Hamming circle* $H(\alpha, r)$ with center α and radius r is the set of all assignments α' with $d(\alpha, \alpha') \leq r$.

Let φ be an existential formula without universal quantifiers. Denote by $\hat{\varphi}$ the *matrix* of φ, i.e. the formula without quantifiers. Let α be an assignment with $\operatorname{dom}(\alpha) = \operatorname{fvars}(\varphi)$, and β be an assignment with $\operatorname{dom}(\beta) \subseteq \operatorname{bvars}(\varphi)$ such that $\alpha \cup \beta \models \hat{\varphi}$. It is going to turn out *a posteriori* that for our industrial application problems discussed in Subsection 4.1 there is a significant probability that there exist further $\alpha' \in H(\alpha, r)$ with $r = 25$ such that $\alpha' \cup \beta \models \hat{\varphi}$ as well. We have determined that number $r = 25$ heuristically.

We are now going to optimize our PQSAT algorithm to *locally search* the Hamming circle whenever finding a satisfying assignment. In the successful cases where $\alpha' \in H(\alpha, r)$ with $\alpha' \cup \beta \models \hat{\varphi}$ this saves expensive calls to $\operatorname{QSAT}(\varphi, \alpha')$. In the unsuccessful cases, however, we cannot draw any conclusions: If for an assignment β we have $\alpha' \cup \beta \models \hat{\varphi} \longleftrightarrow \text{false}$, then there can be a different β' with $\alpha' \cup \beta' \models \hat{\varphi}$.

Since assignments are now checked at two different places, viz. local search and calls to QSAT, we maintain a set s of hashes of already checked assignments α in order to avoid duplicate checking.

Algorithm 3 states the "if" part of the PSAT algorithm. Input, output and the "else" part are literally as in Algorithm 2 (lines 8–17).

1 **if** $\operatorname{fvars}(\varphi) \setminus \operatorname{dom}(\alpha) = \emptyset$ **then**
2   **if** $\operatorname{hash}(\alpha) \notin s$ **then**
3 $(\sigma, \varphi', \beta) := \operatorname{QSAT}(\exists\varphi, \alpha)$
4 $s := s \cup \{\operatorname{hash}(\alpha)\}$
5   **if** $\sigma = \text{true}$ **then**
6 $\psi_1 := \operatorname{form}(\alpha)$
7 $(\psi_2, s) := \operatorname{localSearch}(\hat{\varphi}, \alpha, r, \beta, s)$
8   **return** $(\operatorname{simplify}(\psi_1 \vee \psi_2), \varphi')$
9   **else**
10   **return** (false, φ')
11   **else**
12   **return** (false, φ)

Algorithm 3: The relevant part of $\operatorname{PSAT}(\varphi, \alpha)$

Note that in contrast to PQSAT we cannot use α to fix an assignment in advance. This would require some slight modifications. The procedure localSearch() used there is

going to be given as Algorithm 4.

Input: $(\hat{\varphi}, \alpha, r, \beta, s)$ where $\hat{\varphi}$ is a quantifier-free formula, $\alpha \cup \beta$ is an assignment with $\operatorname{dom}(\alpha \cup \beta) \subseteq \operatorname{vars}(\hat{\varphi})$, $r \in \mathbb{N}$, and s is a set of hashes
Output: (ψ, s') where ψ is a quantifier-free formula with $\operatorname{vars}(\psi) = \operatorname{dom}(\alpha)$, and s' is a set of hashes

1 $\psi := \text{false}$
2 **foreach** $\alpha' \in H(\alpha, r)$ **do**
3 | **if** $\alpha' \cup \beta \models \hat{\varphi}$ **then**
4 | | $\psi := \psi \vee \operatorname{form}(\alpha')$
5 | | $s := s \cup \{\operatorname{hash}(\alpha')\}$
6 **return** (ψ, s)

Algorithm 4: localSearch$(\hat{\varphi}, \alpha, r, \beta, s)$

To conclude this section we discuss why these ideas cannot be straightforwardly generalized to PQSAT. Recall that the starting point of our search is a satisfying assignment $\alpha \cup \beta \models \hat{\varphi}$. The role of β is to serve as a witness for the satisfiability wrt. α of the corresponding existentially quantified formula φ. Since in the general case there are possibly universally quantified variables such a witness cannot exist for principle reasons.

4. APPLICATIONS AND BENCHMARKS

Besides the application examples for PQSAT mentioned in [21] we are going to present an application for PSAT originating from a cooperation with the automotive industry. We will describe this application in Subsection 4.1 before we go on to benchmarks and comparisons to other systems in Subsection 4.2.

4.1 Configurations in the Automotive Industry

In the automotive industry, the compilation and maintenance of correct product configuration data is a complex task. In [23, 14] there is described how valid configurations of constructible vehicles can be expressed using quantifier-free propositional formulas. We are now going to present a simplified version of this method in order to explain our new application which is based on such descriptions.

For many positions in a vehicle one can choose between various parts to fill that position. Each part p, e.g. engine, steering wheel, or left mirror, of a vehicle is mapped to an equipment code x_p. Many parts correspond to *customer options*. It is important to understand however that there are thousands of parts, the vast majority of which is not directly selected by the customer. For our discussion here we refer to those parts as *hidden parts*. The set of all constructible vehicles is described by one formula referred to as *product overview formula* (POF). A POF is a conjunction of rules. A single rule is either a *constructibility condition* (CC) or a *supplementary code* (SC). A CC is an implication $x_p \to \varphi$ where φ is an arbitrary quantifier-free propositional formula. It must hold when $x_p \leftarrow \top$. An SC is an implication $\varphi \to x_p$ which must hold when a certain condition φ holds.

Consider as a toy example a vehicle where the customer options are three different engines with equipment codes e_1, e_2, e_3, and three additional features a_1, a_2, a_3. As hidden parts we consider two different gearboxes g_1, g_2. There is exactly one engine in a vehicle (cc_1–cc_4) and exactly one gearbox (cc_5–cc_7):

$$\mathrm{cc}_1 = \text{true} \to e_1 \vee e_2 \vee e_3, \qquad \mathrm{cc}_5 = \text{true} \to g_1 \vee g_2,$$
$$\mathrm{cc}_2 = e_1 \to \neg e_2 \wedge \neg e_3, \qquad \mathrm{cc}_6 = g_1 \to \neg g_2,$$
$$\mathrm{cc}_3 = e_2 \to \neg e_1 \wedge \neg e_3, \qquad \mathrm{cc}_7 = g_2 \to \neg g_1.$$
$$\mathrm{cc}_4 = e_3 \to \neg e_1 \wedge \neg e_2,$$

Engine e_1 must be combined with gearbox g_1, e_2 must be combined with g_2, and e_3 can be combined with g_1 or g_2:

$$\mathrm{cc}_8 = e_1 \to g_1, \quad \mathrm{cc}_9 = e_2 \to g_2, \quad \mathrm{cc}_{10} = e_3 \to g_1 \vee g_2.$$

Feature a_2 must not be combined with a_3, the combination of e_3 and g_1 must be combined with a_2, and the combination of e_3 and g_2 must be combined with a_3:

$$\mathrm{cc}_{11} = a_2 \to \neg a_3, \quad \mathrm{sc}_1 = e_3 \wedge g_1 \to a_2, \quad \mathrm{sc}_2 = e_3 \wedge g_2 \to a_3.$$

The POF is the conjunction of all CCs and SCs:

$$\mathrm{POF} = \bigwedge_{i=1}^{11} \mathrm{cc}_i \wedge \bigwedge_{j=1}^{2} \mathrm{sc}_j\,.$$

When a customer chooses a certain option p, the currently used configuration tool adds $x_p \leftarrow \top$ to an assignment α. For each customer option p' that is not chosen it adds $x_{p'} \leftarrow \bot$. At the end there runs an automatic assignment process, which iteratively adds $x_q \leftarrow \top$ to α for all SCs $\varphi \to x_q$ with $\alpha \models \varphi$. For hidden parts $x_{q'}$ it also adds $x_{q'} \leftarrow \bot$ for all SCs $\varphi \to \neg x_q$ with $\alpha \models \varphi$. Notice that customer options can be flipped from $\bot$ to $\top$ but not vice versa. A car is considered constructible if and only if its POF is satisfiable wrt. to the final α. In the positive case α encodes the configuration of the car. Notice that this configuration cannot be obtained straightforwardly by pure SAT solving.

The problem with the automatic assignment process is that is does not necessarily terminate and strongly depends on the order of adding assignments. Furthermore in a significant number of cases it delivers false negatives, i.e. turns the POF unsatisfiable via α although the vehicle is constructible in reality.

Our solution is to use PSAT as a less efficient but more powerful fallback option. In the case that the POF has turned out unsatisfiable for some order we proceed as follows: We start with the original input α of the automatic assignment process. For each assignment $x_p \leftarrow \top$ we conjunctively add x_p to the POF. Then we delete *all* assignments from α. Notice that we completely ignore assignments $x_p' \leftarrow \bot$ corresponding to customer options. Finally all codes corresponding to customer options are considered as free variables, while all codes corresponding to hidden parts are existentially quantified. The result of PSAT is then $\bigvee_i \tau_i$ where each conjunction τ_i of literals describes one possible way to render the vehicle constructible by specifying the absence and presence of customers options that were not mentioned in the original order.

We continue the example above. Assume that a customer chooses e_3 and a_1. We compute

$$(\tau_1 \vee \tau_2, \varphi') = \mathrm{PSAT}(\exists g_1 \exists g_2 (\mathrm{POF} \wedge e_3 \wedge a_1), \emptyset),$$

where

$$\tau_1 = \neg e_2 \wedge \neg e_1 \wedge a_3 \wedge \neg a_2, \quad \tau_2 = \neg e_2 \wedge \neg e_1 \wedge \neg a_3 \wedge a_2.$$

For each τ_i we are interested in the subset of positive literals, which specify additionally required customer options. These

subsets are presented to the user as the final output:

$$\{\{a_2\}, \{a_3\}\}.$$

We see that the either code a_2 or code a_3 must be chosen, but not both. Our final output can be processed by a human expert. Alternatively one can automatically select subsets by, e.g., minimizing the number of necessary changes or costs.

Our benchmarks discussed in the next section will demonstrate that our approach and our implementation of PSAT in Redlog is capable of solving such problems on real instances from current product lines of vehicles with thousands of variables and ten thousands of clauses.

4.2 Benchmarks

To start with, we would like to point out that all examples discussed so far take less than 10 ms CPU time, which corresponds to the accuracy of the timing facilities built into Reduce on our architecture. We have used PSL-based Reduce on an Apple Mac Pro with two 2.8 GHz Quad-Core Intel Xeon Processors using one core of one processor and 750 MB of memory.

Table 1 shows computations with three instances of configuration problems in the automotive industry as described in the previous subsection. We have taken these instances from the publicly available benchmark suite of DaimlerChrysler's Mercedes car lines.[2] The names of the instances in the table are followed by number of variables and number of clauses. We compare the computation times of PQSAT and PSAT for an increasing number of free variables. For PSAT we also give the *l.s. rate*, which is the percentage of QSAT calls saved by local search. One can clearly see that for all examples PSAT outperforms PQSAT. Up to 75% of calls to QSAT can be saved, and we observe significant speed up factors, in particular with many free variables.

In Table 2 we compare PQSAT with QE [21]. QE is implemented in Reduce as well such that the computation times are absolutely comparable. We use a set of standard benchmarks for SAT solvers and QSAT solvers. We restrict to quite small examples such that also QE finishes within reasonable time. We consider three SAT benchmarks:

`ii8` stems from the Boolean formulation of a problem of inductive interference [13].

`sinz` is a superset of the formulas considered in Table 1.[2]

`auto` is a set of product configuration formulas as described in Subsection 4.1. It is currently used in industry.

For QSAT we consider two benchmarks:

`toilet` is the *bomb in the toilet* planning problem [25].

`2player` has been introduced and discussed in [21].

The last section of Table 2 shows the results of our PQSAT benchmarks. In order to get large PQSAT benchmarks we have deleted some quantifiers from the *bomb in the toilet* [25] problem.

It is noteworthy that on these PQSAT benchmarks the performance of QE, in contrast to that of PQSAT, increases when increasing the number of free variables in a fixed formula. This observation is compatible with the complexity analysis in [21]: QE is single exponential in the number of quantifiers but only polynomial in all other reasonable complexity parameters. Recall that our PQSAT, in contrast, is single exponential in *all* variables. For all benchmarks considered here PQSAT clearly outperforms QE.

Finally we would like to remark that our current implementation of SAT and QSAT cannot compete with the current highly specialized solvers [17, 8, 18, 2]. Recall, however, that our approach is mostly generic and compatible with these solvers.

5. FUTURE WORK

Since PQSAT uses the general idea of the DLL approach our next research goal is to adapt more of the recent developments in SAT solving to our algorithm. This includes learning for free variables. There are interesting research perspectives concerning various heuristics used throughout this paper, e.g. the Hamming radius for the local search. Concerning applications our configuration technique discussed in Subsection 4.1 can of course be transferred to other product lines. A not so obvious but probably very interesting example is software configuration. As additional application areas we are considering to study bounded model checking and software verification.

[2]`http://www-sr.informatik.uni-tuebingen.de/~sinz/DC/DC_base.zip`

6. REFERENCES

[1] M. Benedetti and H. Mangassarian. QBF-based formal verification: Experience and perspectives. *JSAT*, 5:133–191, 2008.

[2] A. Biere. Picosat essentials. *JSAT*, 4:75–97, 2008.

[3] A. Biere, A. Cimatti, E. Clarke, O. Strichman, and Y. Zhu. Bounded model checking. In M. Zelkowitz, editor, *Highly Dependable Software*, volume 58 of *Advances in Computers*. Academic Press, San Diego, CA, 2003.

[4] E. Clarke, A. Biere, R. Raimi, and Y. Zhu. Bounded model checking using satisfiability solving. *Form. Methods Syst. Des.*, 19(1):7–34, 2001.

[5] S. A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the STOC '71*, pages 151–158. ACM Press, New York, NY, 1971.

[6] M. Davis, G. Logemann, and D. Loveland. A machine program for theorem-proving. *Commun. ACM*, 5(7):394–397, 1962.

[7] A. Dolzmann and T. Sturm. Redlog: Computer algebra meets computer logic. *ACM SIGSAM Bulletin*, 31(2):2–9, 1997.

[8] N. Eén and N. Sörensson. An extensible SAT-solver. In *SAT*, volume 2919 of *LNCS*, pages 502–518. Springer, 2003.

[9] J. W. Freeman. *Improvements to propositional satisfiability search algorithms*. PhD thesis, University of Pennsylvania, Philadelphia, PA, 1995.

[10] H. Fujiwara and S. Toida. The complexity of fault detection problems for combinational logic circuits. *IEEE Trans. Comput.*, 31(6):555–560, 1982.

[11] E. Giunchiglia, M. Narizzano, and A. Tacchella. Learning for quantified boolean logic satisfiability. In *Eighteenth National Conference on Artificial intelligence*, pages 649–654. American Association for Artificial Intelligence, Menlo Park, CA, 2002.

Table 1: Comparison of PQSAT and PSAT (all times in s)

	C168_FW (1909/7477)			C129_FR(1888/7404)			C211_FW(1665/5929)		
free variables	PQSAT	PSAT	l.s. rate	PQSAT	PSAT	l.s. rate	PQSAT	PSAT	l.s. rate
0	2.3	2.3	0%	2.8	2.8	0%	1.4	1.4	0%
5	2.6	2.4	0%	3.5	3.6	0%	1.9	1.9	0%
10	2.4	2.5	25%	4.8	4.1	74%	2.3	2.3	0%
15	3.1	2.5	25%	7.0	4.7	27%	2.3	2.3	6%
20	6.1	3.1	14%	17.0	6.4	30%	2.4	2.5	21%
25	11.5	3.2	10%	111.0	37.8	27%	4.1	2.4	65%
30	281.0	45.9	9%	—	—	—	8.3	3.5	56%
35	—	—	—	—	—	—	9.9	4.9	26%
40	—	—	—	—	—	—	50.5	24.5	30%

Table 2: Benchmark for PQSAT and the quantifier elimination procedure (QE) from [21]

Benchmark	instances	variables	free variables	clauses	QE time in s	PQSAT time in s
ii8	41	66–1068	0	186–821	3230.00	8.50
sinz	36	1411–1909	0	1982–11342	4227.00	99.67
auto	8	2291–4223	0	3006–16387	16650.00	61.20
toilet_a_02	5	18–90	0	39–408	310.00	< 0.01
toilet_a_04	9	32–140	0	129–894	4780.00	< 0.01
2player ($n = 250$)	1	500	0	998	5.10	0.03
2player ($n = 500$)	1	1000	0	1998	36.00	0.08
2player ($n = 750$)	1	1500	0	2998	115.90	0.18
toilet_a_04_01.4	2	60	20, 40	229	2.40	3.00
toilet_a_06_01.4	3	86	20, 40, 60	649	121.30	12.90
toilet_a_08_01.2	2	60	20, 40	2205	21.30	8.10

[12] K. Iwama and S. Tamaki. Improved upper bounds for 3-SAT. In *Proceedings of the SODA '04*, pages 328–328. SIAM, Philadelphia, PA, 2004.

[13] A. P. Kamath, N. K. Karmarkar, K. G. Ramakrishnan, and M. G. C. Resende. A continuous approach to inductive inference. *Mathematical Programming*, 57(1-3):215–238, 1992.

[14] W. Küchlin and C. Sinz. Proving consistency assertions for automotive product data management. *J. Autom. Reasoning*, 24(1-2):145–163, 2000.

[15] R. Letz. Lemma and model caching in decision procedures for quantified boolean formulas. In *Proceedings of the TABLEAUX '02*, pages 160–175. Springer, 2002.

[16] J. P. Marques-Silva, K. A. Sakallah, J. P. Marques, S. Karem, and A. Sakallah. Conflict analysis in search algorithms for propositional satisfiability. In *Proceedings of the IEEE International Conference on Tools with Artificial Intelligence*, 1996.

[17] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: engineering an efficient SAT solver. In *Proceedings of the DAC '01*, pages 530–535. ACM, New York, NY, 2001.

[18] K. Pipatsrisawat and A. Darwiche. Rsat 2.0: SAT solver description. Technical Report D-153, Computer Science Department, UCLA, 2007.

[19] M. Presburger. Über die Vollständigkeit eines gewissen Systems der Arithmetik ganzer Zahlen, in welchem die Addition als einzige Operation hervortritt. In *Comptes Rendus du premier congres de Mathematiciens des Pays Slaves*, pages 92–101, Warsaw, Poland, 1929.

[20] U. Schöning. New algorithms for k-SAT based on the local search principle. In *Proceedings of the MFCS '01*, pages 87–95. Springer, 2001.

[21] A. M. Seidl and T. Sturm. Boolean quantification in a first-order context. In *Proceedings of the CASC 2003*, pages 329–345. Technische Universität München, Munich, Germany, 2003.

[22] B. Selman, H. Kautz, and B. Cohen. Local search strategies for satisfiability testing. In *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, pages 521–532, 1995.

[23] C. Sinz, A. Kaiser, and W. Küchlin. Formal methods for the validation of automotive product configuration data. *Artif. Intell. Eng. Des. Anal. Manuf.*, 17(1):75–97, 2003.

[24] A. Tarski. A decision method for elementary algebra and geometry. Prepared for publication by J. C. C. McKinsey. RAND Report R109, RAND, Santa Monica, CA, 1948.

[25] R. Waldinger. The bomb in the toilet. *Computational Intelligence*, 3(1):220–221, 1987.

[26] V. Weispfenning. The complexity of linear problems in fields. *J. Symbolic Computation*, 5(1-2):3–27, 1988.

[27] L. Zhang and S. Malik. Conflict driven learning in a quantified boolean satisfiability solver. In *Proceedings of the ICCAD '02*, pages 442–449. ACM, New York, NY, 2002.