

# Boolean Quantification in a First-Order Context

Andreas Seidl<sup>1</sup> and Thomas Sturm<sup>2</sup>

<sup>1</sup> University of Passau, Germany

seidl@fmi.uni-passau.de

<http://www.fmi.uni-passau.de/~seidl/>

<sup>2</sup> University of Passau, Germany

sturm@uni-passau.de

<http://www.fmi.uni-passau.de/~sturm/>

**Abstract.** We establish a framework to integrate propositional logic with first-order logic. This is done in such a way that it optionally appears either as first-order logic over a Boolean algebra or as propositional logic including Boolean quantification. We describe and prove complexity bounds for extended quantifier elimination by virtual substitution for our theory. This extended quantifier elimination is, besides many other mathematical algorithms and utilities, implemented in a new context IBALP of the REDUCE package REDLOG. We demonstrate the capabilities of this new context by means of numerous application and benchmark examples including QSAT problems.

## 1 Introduction

Propositional logic is a fundamental and versatile concept that is well-established in mathematics, computer science, and engineering. Compared to other more sophisticated logical theories, propositional logic is extremely simple. At the first glance it appears like a restricted and simplified variant of first-order logic, where propositional variables have taken over the place of atomic formulas. Still, it is compatible with this first-order logic in various natural ways.

It is the simplicity that makes propositional logic on the one hand that popular but on the other hand considerably limits its expressiveness. In many applications, the idea arises naturally to enlarge expressivity by copying the concept of quantification from first-order logic. Doing so straightforwardly, however, destroys the above-mentioned compatibility. This is unsatisfactory from a theoretical point of view, but also causes problems from a practical point of view: It inhibits integration of propositional logic with first-order logic within a single software system.

With this very aim of software integration, we motivate our approach to bring propositional logic in terms with first-order logic: We extend the computer logic system REDLOG by a component for dealing with propositional logic. This gives access to first-order results, methods, concepts, and already existing generic implementations [DS97a].

We summarize the contributions of this paper:

1. We analyze various ways of embedding propositional logic into first-order logic. We offer a solution that is compatible with the first-order theory of Boolean algebra and thus allows first-order quantification. Throughout this paper, we mean by *Boolean algebra* the Boolean algebra with two elements, which is uniquely determined up to isomorphisms. Optionally, formulas are presented to the user like propositional logic including Boolean quantification [Pap94].
2. We develop a quantifier elimination procedure for Boolean algebra, and analyze its complexity, which is optimal provided that  $P \neq NP$ . We give sequences of benchmark examples from Boolean circuit design, on which the procedure appears to perform linearly. Quantifier elimination in particular covers QSAT problems. It is actually much more general: It can find *uniform* solutions for *parametric* QSAT problems.
3. We explain how to heuristically improve our theoretically optimal quantifier elimination method by sophisticated simplification techniques, variable selection strategies, and elimination of superfluous case distinctions.
4. From our quantifier elimination procedure we can straightforwardly derive an extended quantifier elimination procedure. This yields for existential questions in addition to the necessary

and sufficient conditions in the parameters also *sample solutions*. Correspondingly, for universal questions it possibly yields *counterexamples*.

5. The methods described here are implemented within the current development version of the REDLOG component of the computer algebra system REDUCE. They will thus become publicly available with the next release of REDUCE. They are well-documented and professionally maintained. As a part of this integrated system, they benefit from the work done for other first-order theories, and vice versa.
6. The implementation provides two choices for the user front end: First, traditional first-order logic over Boolean algebras. Second, a *propositional wrapper*. This wrapper makes first-order logic appear to the user like traditional propositional calculus including Boolean quantification. The user can switch between these views according to his needs at any time during a session.
7. We give a variety of application examples for our methods from mathematics, computer science, and engineering. Our focus is on digital circuit design and testing. The examples include sequences of benchmark examples to fathom the limits in problem size for the applicability of our methods. Due to the apparently particularly favorable structure of Boolean algebras, we can cope with problem sizes that have not been accessible to quantifier elimination in any other context so far. In one circuit design example, we are able to eliminate more than 15 000 quantifiers.

## 2 Propositional vs. Predicate Logic

*Propositional logic* combines *propositional variables* from a set  $\{V_i\}_{i \in I}$  to formulas by means of Boolean operations like true, false,  $\neg$ ,  $\wedge$ ,  $\vee$ ,  $\implies$ ,  $\iff$ . A semantic for these formulas is obtained by interpreting the propositional variables as either “true” or “false” and then deriving a truth value for a formula according to the usual interpretation of the operations.

*First-order logic* works similarly with atomic formulas over some fixed language  $\Sigma$  instead of propositional variables. Atomic formulas are either equations or predicates constructed from relation symbols in  $\Sigma$ . Here a semantic is obtained by interpreting these atomic formulas in a  $\Sigma$ -structure  $\mathbb{S}$  with universe  $S$  at some point in  $S$ . From the truth values obtained for the atomic formulas this way, a truth value for the considered formula is derived exactly as for propositional logic. In addition, first-order logic allows quantification  $\exists x, \forall x$ , where  $x$  is a variable ranging over elements of the corresponding universe  $S$ .

### 2.1 Pure First-Order Approach

First-order logic covers propositional logic in a very natural way as follows: Choose the language  $\{V_i\}_{i \in I}$  where all these “propositional variables” are symbols for relations of arity 0. If one then refrains from using equations and quantification, then the corresponding subset of first-order formulas is identical to the propositional formulas discussed above. Semantic is compatible as well: The interpretation of the variables in the former corresponds to the choice of the structure  $\mathbb{S}$ , which gives a constant interpretation for each relation symbol  $V_i$  as either “true” or “false.”

Note that it is formally *not* possible with this approach to make quantifications like  $\exists V_i$  or  $\forall V_i$ . Such quantifications, however, have a quite straightforward semantic and are most desirable as we are going to demonstrate throughout this article. We therefore take a different approach to embed propositional calculus into first-order logic, which we can later extend to allow quantification.

### 2.2 Algebraic Propositional Approaches

From an algebraic point of view, propositional formulas are terms over a suitable language, such as the following one for Boolean algebras:

$$\Sigma = (0^{(0)}, 1^{(0)}, \sim^{(1)}, \&^{(2)}, |^{(2)}, \rightarrow^{(2)}, \leftrightarrow^{(2)}).$$

Terms  $\mathcal{T}$  are recursively defined to be variables from an infinite set  $\mathcal{V}$ , one of the constants 0,  $1 \in \Sigma$ ,  $\sim t$ , where  $t \in \mathcal{T}$ , or  $t_1 \varrho t_2$ , where  $\varrho \in \{\&, |, \rightarrow, \leftrightarrow\}$  and  $t_1, t_2 \in \mathcal{T}$ . These syntactic

terms are semantically interpreted in the Boolean algebra  $\mathbb{B}$  with universe  $\{0, 1\}$  considered as a  $\Sigma$ -structure. For this,  $\sim$  is interpreted as “not,”  $\&$  is “and,”  $|$  is “or,”  $\rightarrow$  is implication, and  $\leftrightarrow$  is biimplication. Consider an equation  $(t = t')(x_1, \dots, x_n)$ . The equation is *satisfiable* if there exist  $b_1, \dots, b_n \in \{0, 1\}$  such that  $\mathbb{B} \models (t = t')(b_1, \dots, b_n)$ . The equation is *valid* if  $\mathbb{B} \models t = t'$ , i.e., for all  $b_1, \dots, b_n \in \{0, 1\}$  we have  $\mathbb{B} \models (t = t')(b_1, \dots, b_n)$ . In the sequel we refer to this approach as the *pure algebraic approach*.

It is common practice not to make the  $\Sigma$ -Structure  $\mathbb{B}$  explicit. Instead, one prefers to speak of *truth assignments* on the variables, which extend to truth assignments on the terms. The terms are called *Boolean expressions* or *Boolean functions*. Consider a Boolean expression  $t(x_1, \dots, x_n)$ . *Satisfiability* of  $t$  is defined as the existence of a truth assignment for  $x_1, \dots, x_n$  such that  $t$  becomes “true.” This corresponds to the satisfiability of the equation  $t = 1$  in the pure algebraic approach. *Validity* of  $t$  is defined as  $t$  being “true” for all truth assignments for  $x_1, \dots, x_n$ . This corresponds to the validity of the equation  $t = 1$  in the pure algebraic approach. Often a congruence relation  $\approx \subseteq \mathcal{T} \times \mathcal{T}$  called *semantic equivalence* is established, where  $t \approx t'$  corresponds to the validity of the equation  $t = t'$  in the pure algebraic approach. We are going to refer to this second approach as the *common algebraic approach*.

The common algebraic approach is *compatible* with the pure algebraic approach in the sense of the following two observations:

1. The common algebraic approach does not introduce any syntactic objects or notions that are not existent in the pure algebraic approach. It rather relaxes semantics by defining notion in terms of Boolean expressions instead of corresponding equations.
2. As for all concepts that we have introduced so far, both these approaches can be straightforwardly translated into each other.

The algebraic propositional approaches, in either form, provide an excellent framework for dealing with many issues arising in computer science and engineering. This applies to questions that can be reduced to either satisfiability or validity.

The question for satisfiability of a given term  $t$  is actually the famous SAT problem, which is one of the most prominent examples for NP-complete problems. Validity comprises, as discussed above, semantic equivalence  $t \approx t'$ .

As we see, with the algebraic propositional approaches existential as well as universal quantification are implicitly present. They remain, however, in the meta-language in contrast to becoming part of the formal system. This implies an important restriction: It is impossible to tackle problems, where arbitrary quantification information is part of the input, in a uniform way. This is, however, most desirable as we shall see in Section 6 on application examples.

### 2.3 First-Order Logic Approach

On the basis of the pure algebraic approach, we consider first-order formulas over our language  $\Sigma$ . *First-order formulas*  $\mathcal{F}$  are recursively defined to be **true**, **false**, equations,  $\neg(\varphi)$ , where  $\varphi \in \mathcal{F}$ ,  $(\varphi_1) \sigma (\varphi_2)$  where  $\sigma \in \{\wedge, \vee, \implies, \iff\}$  and  $\varphi_1, \varphi_2 \in \mathcal{F}$ , or  $Qx(\varphi)$ , where  $Q \in \{\exists, \forall\}$ ,  $x \in \mathcal{V}$ , and  $\varphi \in \mathcal{F}$ . We allow ourselves to refer to first-order formulas simply as *formulas*. A formula is called a *quantifier-free formula* if it contains neither  $\exists$  nor  $\forall$ .

As an example consider the *quantified satisfiability problem* QSAT, also known as QBF (*quantified Boolean formula*). QSAT plays an important role by being complete for PSPACE in the same manner as SAT is complete for NPTIME: Given a term  $t(x_1, \dots, x_n)$ , the question is whether

$$\mathbb{B} \models \exists x_1 \forall x_2 \exists x_3 \dots Q_n x_n (t = 1),$$

where  $Q_n \in \{\exists, \forall\}$  depending on the parity of  $n$ .

For the common algebraic approach the situation is not quite that simple. Using first-order formulas would suddenly make visible the atomic formulas that had been so carefully hidden. As a solution, one simply continues playing the game, and writes  $\exists x_1 \forall x_2 \exists x_3 \dots Q_n x_n (t)$ . This is then called *Boolean quantification*. This, however, introduces syntactic objects that are not present in the pure algebraic approach thus invalidating observation no. 1 of our discussion on compatibility in the previous section.

Since our plan is to integrate propositional logic into a system based on pure algebra, this syntactic compatibility is in fact crucial. So we decide for the pure algebraic approach.

In addition, we optionally provide the view of the common algebraic approach by means of a *propositional wrapper*. The basis for this is the existence of normal forms for first-order formulas such that atomic formulas can be considered propositional variables. We are going to discuss the underlying theory for the wrapper in the following section. Later on, in Section 5 on the implementation of REDLOG, we shall revisit the wrapper in form of an optional software module that combines implicit normal form computations with certain I/O-features.

### 3 Quantified-Propositional Formulas

A first-order formula  $\varphi$  is called *propositional* if it is quantifier-free, and all contained equations are of the form  $x = 1$ , where  $x \in \mathcal{V}$ . Correspondingly, we then call these contained equations *propositional variables*. These propositional variables admit Boolean quantification by means of quantifying the contained formal variables  $x$  like  $\exists x$  or  $\forall x$ . Formulas obtained from propositional formulas by such quantifications are called *quantified-propositional* formulas.

**Theorem 1 (Quantified-Propositional Normal Form).** *Let  $\varphi$  be a formula.*

- (i) *There is a quantified-propositional formula  $\varphi'$  such that  $\mathbb{B} \models \varphi' \iff \varphi$ .*
- (ii)  *$|\varphi'| = O(|\varphi|)$ , where  $|\cdot|$  denotes the word length.*
- (iii)  *$\varphi'$  can be computed from  $\varphi$  in time and space bounded by  $O(|\varphi|)$ . More precisely constant space is sufficient.*

*Proof.* For Part (i) observe that an equation  $b_1 = b_2$  with  $b_1, b_2 \in \mathcal{T}$  is equivalent to  $(b_1) \leftrightarrow (b_2) = 1$ . We may thus w.l.o.g. assume that all equations in our considered formula  $\varphi$  are of such a form. We show our claim by induction on the number  $n$  of occurrences of function symbols from  $\Sigma$  in  $\varphi$ . If  $n = 0$ , then the claim is trivial. Let there be  $n + 1 > 0$  such occurrences. Then we can pick a corresponding equation  $\eta$ , and equivalently replace it with a formula  $\psi$  as follows: If  $\eta$  is  $0 = 1$  or  $1 = 1$ , then we set  $\psi := \text{false}$  or  $\psi := \text{true}$ , respectively. If  $\eta$  is of the form  $\sim(b) = 1$  with  $b \in \mathcal{T}$ , then  $\psi := \neg(b = 1)$ . Similarly, if  $\eta$  is of the form  $(b_1) \varrho (b_2) = 1$  with  $b_1, b_2 \in \mathcal{T}$  and  $\varrho \in \{\&, |, \rightarrow, \leftrightarrow\}$ , then we set  $\psi := (b_1 = 1) \sigma (b_2 = 1)$ , where  $\sigma$  is straightforwardly derived from  $\varrho$  by mapping  $\&$  to  $\wedge$ ,  $|$  to  $\vee$ ,  $\rightarrow$  to  $\implies$ , and  $\leftrightarrow$  to  $\iff$ . In either case, this yields one fewer function symbol in the result of our replacement. We can thus apply our induction hypothesis.

For Part (ii), we first consider the initial step in Part (i) of normalizing equations. Denote the output of this by  $\varphi_1$ . Since for each equation in  $\varphi$ , we obtain at most 6 extra characters, we have  $|\varphi_1| \leq |\varphi| + 6|\varphi| = 7|\varphi|$ . In the subsequent step, we obtain at most 2 extra characters for each occurrence of a function symbol in  $\varphi_1$ . Hence,

$$|\varphi'| \leq |\varphi_1| + 2|\varphi_1| = 3|\varphi_1| \leq 21|\varphi|.$$

For showing Part (iii) we give a more algorithmic version of the induction proof above. Copy  $\varphi$ , but write for a variable  $x$ , if it is not preceded by a quantifier, the equation  $x = 1$  and for characters listed in the first line of the following table the character underneath:

0	1	$\sim$	$\&$		$\rightarrow$	$\leftrightarrow$	=
false	true	$\neg$	$\wedge$	$\vee$	$\implies$	$\iff$	$\iff$

We consider true and false to be one character each. Hence, length is only increased when rewriting variables. We have  $|\varphi'| \leq 3|\varphi_1|$ , and  $\varphi'$  is computed this way in linear time and constant space.  $\square$

In order to emphasize the point of view that equations  $x = 1$  in quantified-propositional formulas are propositional variables, we allow ourselves to use capital letters  $X$  as an alternative notation for  $x = 1$ . Note that occurrences of variables directly after  $\exists$  and  $\forall$  always remain lower case.

As an example, consider the formula

$$\forall x \forall y (x = \sim y \implies \forall a (a \& y = a \& \sim x)).$$

The following is a possible quantified-propositional normal form of this:

$$\forall x \forall y ((x = 1 \iff \neg y = 1) \implies \forall a (a = 1 \wedge y = 1 \iff a = 1 \wedge \neg x = 1)).$$

According to our convention to possibly view the equations as propositional variables, this formula would read as follows:

$$\forall x \forall y ((X \iff \neg Y) \implies \forall a (A \wedge Y \iff A \wedge \neg X)).$$

In the sequel, we will relax our notion of (quantified-)propositional formulas a bit by representing negated propositional variables like  $\neg X$  as  $x = 0$ . Then for every propositional formula there is an equivalent *positive* propositional formula, i.e., an  $\wedge$ - $\vee$ -combination of equations.

## 4 Simplification and Quantifier Elimination

In the past it has turned out that powerful simplification techniques are a crucial prerequisite for efficient implementation of quantifier elimination. In [DS97b] *smart* simplification, *deep* simplification and simplification *with theory* are introduced in the context of ordered fields. All these techniques and concepts are applicable as well to the framework discussed here.

### 4.1 Regular Quantifier Elimination

The Boolean Algebra  $\mathbb{B}$  admits quantifier elimination by virtual substitution [Wei88]: Let  $\psi$  be a quantifier-free formula. Then

$$\mathbb{B} \models \left( \bigvee_{t \in \{0,1\}} \psi[t/x] \right) \iff \exists x \psi \quad \text{and} \quad \mathbb{B} \models \left( \bigwedge_{t \in \{0,1\}} \psi[t/x] \right) \iff \forall x \psi.$$

Several quantifiers are successively eliminated from a prenex input formula starting with the innermost quantifier.

**Theorem 2 (Complexity of Quantifier Elimination).** *Consider a first-order formula of the form  $\varphi = Q_1 x_1 \dots Q_n x_n \psi$  where  $Q_1, \dots, Q_n \in \{\exists, \forall\}$  and  $\psi$  is a quantifier-free formula. Let  $\varphi'$  denote a quantifier-free formula with  $\mathbb{B} \models \varphi' \iff \varphi$  obtained by virtual substitution. Then*

$$|\varphi'| = O(2^n |\psi|) = 2^{O(|\varphi|)},$$

where  $|\cdot|$  denotes the word length. The bound  $2^{O(|\varphi|)}$  is also a bound on the time and space of the computation.

*Proof.* We show by induction on  $n$  that after the elimination of  $n$  quantifiers, the size of the output is bounded by  $2^n(|\psi| + 5) - 5$ . For  $n = 0$  this is trivial. Consider now the elimination of the  $n + 1$ -st quantifier from  $Q_{n+1} x_{n+1} \varphi_n$ : By the induction hypothesis we have  $|\varphi_n| \leq 2^n(|\psi| + 5) - 5$ . Elimination yields  $\varphi_{n+1} = (\varphi_n[0/x]) \sqcup (\varphi_n[1/x])$  with  $\sqcup \in \{\vee, \wedge\}$  depending on  $Q_{n+1}$ . We have

$$|\varphi_{n+1}| = 2|\varphi_n| + 5 = 2(2^n(|\psi| + 5) - 5) + 5 = 2^{n+1}(|\psi| + 5) - 5.$$

The time for this computation is asymptotically dominated by successively producing the intermediate results for each quantifier  $n$  times where  $n \leq |\varphi|$ . This imposes the bound  $|\varphi| \cdot 2^{O(|\varphi|)} = 2^{O(|\varphi|)}$ . The same observation holds for the space.  $\square$

Quantifier elimination by virtual substitution might appear after all like a brute force method, which simply tries all possible combinations of values for the quantified variables. In the worst-case this actually happens. In our practical computations, however, it performs much better for various heuristic reasons:

- It is not always necessary to substitute both values 0 and 1 for a variable. Suppose the considered matrix formula  $\psi$  is propositional and *positive*. Then only those values in  $\{0, 1\}$  have to be substituted for  $x$ , with which  $x$  actually occurs. For general formulas it is easy to compute with which values  $x$  *would* occur in an equivalent positive propositional formula. Such decisions on the basis of special forms without explicitly computing them is generally a useful technique with virtual substitution.
- After each single substitution, there is sophisticated simplification performed. As a consequence, in non-artificial decision problems, it often happens that the formula breaks down to a truth value after elimination of only a fraction of the variables. One such example is the multiplexer series discussed in Section 7.1.
- Within blocks of like quantifiers, the quantifiers can be equivalently interchanged. On the basis of this observation, quantifier elimination can greatly be supported by choosing the “right” variable to be eliminated next. We choose a strategy that has been successful with virtual substitution in several contexts: We choose the variable that occurs most frequently within the formula. The idea is that this increases the chance for simplification. Such selection strategies can be evaluated by applying also the dual strategy to benchmark examples for comparison. We have done so for most of our examples, which we are going to discuss in Section 6 and Section 7.

The special case of quantifier elimination applied to the existential closure of an equation  $t = 1$  with  $t \in \mathcal{T}$  is the satisfiability problem for Boolean formulas. This problem is known to be NP-complete. From this point of view Theorem 2 states that virtual substitution is—in terms of classical complexity theory—an optimal choice as a quantifier elimination procedure.

Similarly, the special case of quantifier elimination on the universal closure of an equation  $t = 1$  with  $t \in \mathcal{T}$  is a decision procedure for validity of  $t$ . Quantifier elimination on the universal closure of an equation  $t = t'$  with  $t, t' \in \mathcal{T}$  is a decision procedure for semantic equivalence.

Next, quantifier elimination straightforwardly comprises QSAT and other variants of this where the quantifiers are not strictly alternating.

In addition, there is a most interesting aspect of quantifier elimination, which conceptually exceeds the range of possible applications listed above: Not all the variables in the input formula have to be quantified. We give an example: On input of

$$\exists y((x \& \sim y) \mid \sim z) \leftrightarrow (y \& z) = 1$$

quantifier elimination yields  $x = 0 \wedge z = 1$ . This is a necessary and sufficient condition on the *parameters*  $x$  and  $z$  for the existence of such a  $y$ .

## 4.2 Extended Quantifier Elimination

With quantifier elimination by virtual substitution it is straightforward to keep track of the particular test points that are substituted for the variables in the subformulas of the final result. This idea is particularly interesting for the outermost block of like quantifiers. It leads to a technique called *extended quantifier elimination* [Wei97] or *quantifier elimination with answer*.

Consider quantifier elimination for a formula  $\varphi = \exists x_1 \dots \exists x_n \psi$ , where  $\psi$  possibly contains further quantifiers. Let  $\psi'$  be a quantifier-free equivalent for  $\psi$ , i.e.  $\psi'$  is an intermediate result of our elimination procedure. Then the final result  $\varphi'$  of the elimination is of the form

$$\varphi' = \bigvee_{i \in I} \psi'_i,$$

where each  $\psi'_i$  is obtained from  $\psi'$  by substituting either 0 or 1 for each of the  $x_1, \dots, x_n$ , say  $\psi'_i = \psi'[b_{i1}/x_1, \dots, b_{in}/x_n]$  with  $b_{i1}, \dots, b_{in} \in \{0, 1\}$ . Extended quantifier elimination now outputs a set of *guarded points* instead of  $\varphi'$ :

$$\{(\psi'_i, \{x_1 = b_{i1}, \dots, x_n = b_{in}\}) \mid i \in I\}.$$

It is obvious that  $\varphi'$  can be straightforwardly constructed from this. In addition, we obtain the following information: Whenever an interpretation of the parameters satisfies some  $\psi'_i$ , then the

**Table 1.** REDLOG quantifier elimination, simplification, and normal forms (left) and utilities (right).

Command	Short Description	Command	Short Description
<code>rlqe</code>	quantifier elimination	<code>sub</code>	syntactic substitution
<code>rlqea</code>	extended quantifier elimination	<code>rlmatrix</code>	formula matrix (drop prenex quantifiers)
<code>rlsimpl</code>	simplification	<code>rlall</code>	universal closure
<code>rlnnf</code>	negation normal form	<code>rllex</code>	existential closure
<code>rlpnf</code>	prenex normal form	<code>rlat1</code>	list of atomic formulas
<code>rlcnf</code>	conjunctive normal form	<code>rlatml</code>	multiplicity list of atomic formulas
<code>rldnf</code>	disjunctive normal form	<code>rlterm1</code>	list of terms
<code>rltab</code>	semantic tableau	<code>rltermml</code>	multiplicity list of terms
<code>rlatab</code>	automated semantic tableau	<code>rlfvar1</code>	list of free variables
<code>rlitab</code>	iterative semantic tableau	<code>rlbvar1</code>	list of bound variables
		<code>rlvar1</code>	list of variables
		<code>rlatnum</code>	number of atomic formulas
		<code>rlqnum</code>	number of quantifiers

original  $\exists$ -quantified formula  $\varphi$  holds for this interpretation, and the corresponding point  $x_1 = b_{i_1}, \dots, x_n = b_{i_n}$  is *one* possible choice of values for the quantified variables.

If the outermost block of quantifiers is a universal one, then we obtain the dual information: Whenever some interpretation of the parameters does *not* satisfy some  $\psi'_i$ , then the original  $\forall$ -quantified formula  $\varphi$  does *not* hold for this interpretation. The corresponding point  $x_1 = b_{i_1}, \dots, x_n = b_{i_n}$  is then *one* possible choice for the quantified variables to obtain a counterexample.

## 5 Implementation

### 5.1 REDLOG and IBALP

REDLOG stands for “REDUCE logic” system [DS97a]. It provides an extension of the computer algebra system REDUCE to a computer logic system implementing symbolic algorithms on first-order formulas w.r.t. temporarily fixed first-order languages and theories. Such a choice of language and theory is called a *context*. The work discussed here establishes a new such context IBALP. So far, the following REDLOG contexts are available:

- OFSF (Ordered fields, standard form representation of terms). These are real closed fields such as the real numbers with ordering. This context was the original motivation for REDLOG. It is still the most important and sophisticated one.
- ACFSF (Algebraically closed fields, standard form representation of terms). These are algebraically closed fields such as the complex numbers.
- DVFSF (Discretely valued fields, standard form representation of terms). The most prominent example for discretely valued fields are  $p$ -adic numbers for some prime  $p$  with abstract divisibility relations.
- IBALP (Initial Boolean Algebras, Lisp prefix representation of terms). These are Boolean algebras with two elements, which are uniquely determined up to isomorphisms.

The idea of REDLOG is to combine methods from computer algebra with logic thus extending the computer algebra system REDUCE to a computer logic system. In this extended system both the algebraic side and the logic side greatly benefit from each other in numerous ways.

We summarize the REDLOG features and commands currently available for IBALP in Table 1. The left hand side shows the mathematical core including (extended) quantifier elimination according to Theorem 2, simplification, tableau methods, and normal form computations [DS97b]. The right hand side collects numerous useful utilities. For more detailed information we refer the reader to the REDLOG user manual [DS99].

The current version REDLOG 2.0 is an integral part of the computer algebra system REDUCE 3.7. The work described here is part of the current development version of REDUCE and will be included into the next release.

```

REDUCE 3.7, 15-Apr-1999, patched to 3-Feb-2003 ...

1: load redlog;

*** turned off switch raise

2: c1 := ex({h1,h2,h3},h1=~(a & b) and h2=~(a & h1) and h3=~(b & h1) and
2:   f=~(h2 & h3));

c1 := ex h1 ex h2 ex h3 (h1 = ~ (a & b) and h2 = ~ (a & h1)
and h3 = ~ (b & h1) and f = ~ (h2 & h3))

3: rlqe c1;

(a = 1 and b = 1 and f = 0) or (a = 1 and b = 0 and f = 1)
or (a = 0 and b = 1 and f = 1) or (a = 0 and b = 0 and f = 0)

```

**Fig. 1.** Same quantifier elimination as in Figure 2 but not using the propositional wrapper.

```

REDUCE 3.7, 15-Apr-1999, patched to 3-Feb-2003 ...

1: load redlog;

*** turned off switch raise

2: on rlpccprint,rlsimpl;

3: rlpccvar H1,H2,H3;

4: c1 := ex({h1,h2,h3},h1=~(a & b) and h2=~(a & h1) and h3=~(b & h1) and
4:   f=~(h2 & h3));

c1 := ex h1 ex h2 ex h3 (((H1 and (not(A) or not(B))) or (A and B and not(H1)))
and ((H2 and (not(A) or not(H1))) or (A and H1 and not(H2)))
and ((H3 and (not(B) or not(H1))) or (B and H1 and not(H3)))
and ((F and (not(H2) or not(H3))) or (not(F) and H2 and H3)))

5: rlqe c1;

(A and B and not(F)) or (A and not(B) and F) or (not(A) and B and F)
or (not(A) and not(B) and not(F))

```

**Fig. 2.** Same quantifier elimination as in Figure 1 but using the propositional wrapper.

## 5.2 The Propositional Wrapper

Our new context IBALP contains a *propositional wrapper* as a component that can be optionally activated. This propositional wrapper is in fact the implementation of Theorem 1 plus the convention discussed thereafter to write the equations of quantified propositional formulas as capital propositional variables.

The wrapper is activated by turning on two global switches: `rlsimpl` and `rlpcprint`. The former causes all input and output to implicitly pass the standard simplification routine (also called `rlsimpl`). This simplification includes, but is not limited to, turning the input into quantified-propositional form. The switch `rlpcprint` causes the output routine to print the capital X for an equation `x=1`. Accordingly, `x=0` is translated to `not(x)`. This works for variables `a`, `...`, `z`. Further propositional variables can be declared using the keyword `rlpcvar`. On the other hand, whenever a capital propositional variable is input, there is internally the corresponding equation created, but hidden from the user via `rlpcprint`.

Figures 1 and 2 show the computation of a quantifier elimination in REDLOG with and without propositional wrapper. The choice whether to use the wrapper or not depends on the particular application and on the preferences of the user.

## 6 Application Examples

We treat a variety of questions in mathematics, digital logic design and, later in the following Section, 2-person games, to give an impression of the possible application range of our work. All computations in this and the following Section have been carried out on a 2 GHz Intel Pentium 4 using 128 MB of RAM.

### 6.1 Laws of Boolean Algebra

We can prove the following laws in less than 10 ms: identity, boundedness, commutativity, associativity, distributivity, complement, uniqueness of complement, the laws of involution, idempotency, absorption, and de Morgan's laws.

For  $m, n \in \mathbb{N}$ , the distributivity law can be generalized as follows:

$$\varphi_{\text{dist},m,n} := \bigwedge_{i=1}^m A_i \vee \bigwedge_{j=1}^n B_j \iff \bigwedge_{i=1}^m \bigwedge_{j=1}^n (A_i \vee B_j)$$

From this we build two sequences for benchmarking:

$$\{\forall(\varphi_{\text{dist},n,n})\}_{n \in \{1, \dots, 100\}} \quad \text{and} \quad \{\forall(\varphi_{\text{dist},n,100})\}_{n \in \{1, \dots, 100\}},$$

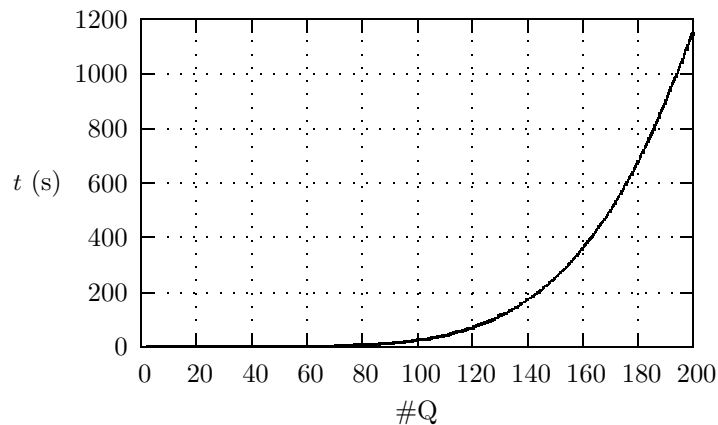
where  $\forall$  denotes the universal closure. The timings can be found in Figure 3 and Figure 4, respectively.

### 6.2 Digital Logic

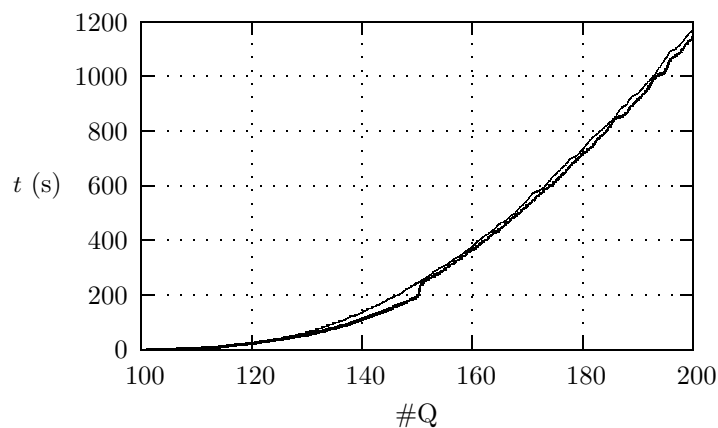
*Logical circuits* are built of *logical gates* like the ones listed in Figure 5. For all gates there except for the NOT-gate there are variants with more than two input lines.

In a *combinational circuit*, the outputs only depend on the current input signals. There is no information about previous input available within the circuit. Such a circuit can be straightforwardly encoded as a first-order formula: We denote the output of each gate  $G_i$  with  $h_i$ . In the special case that our considered  $h_i$  is an output line of the circuit, we use instead of  $h_i$  the designated variable. For each such  $h_i$  or output variable we write down an equation describing it in terms of the input lines of  $G_i$ , which are either input lines of the circuit or other  $h_j$ . A description of the entire circuit is then obtained as follows:

1. Construct a conjunction of all these equations.
2. Existentially quantify all inner nodes  $h_i$ .



**Fig. 3.** Timings for  $\varphi_{\text{dist},n,n}$  for  $n \in \{1, \dots, 100\}$  with the standard variable selection strategy. The dual strategy has about the same timings.



**Fig. 4.** Timings for  $\varphi_{\text{dist},n,100}$  for  $n \in \{1, \dots, 100\}$  with the standard variable selection strategy (thick) and the dual strategy (thin).

The described process can be automatized, e.g., on the basis of *net list* representations of circuits.

As an example, consider the circuit consisting of four NAND's in Figure 6. With the described method we derive the following description of our circuit:

$$\varphi_{c_1} := \exists h_1 \exists h_2 \exists h_3 (h_1 = \sim(a \& b) \wedge h_2 = \sim(a \& h_1) \wedge h_3 = \sim(b \& h_1) \wedge f = \sim(h_2 \& h_3)).$$

It is a typical textbook exercise ([LKM88], p.264, 5.4(b)) to show that this circuit is equivalent to XOR. We can automatically prove this in less than 10 ms by applying quantifier elimination to the following universal closure:

$$\forall (\varphi_{c_1} \iff f = (a | b) \& \sim(a \& b)).$$

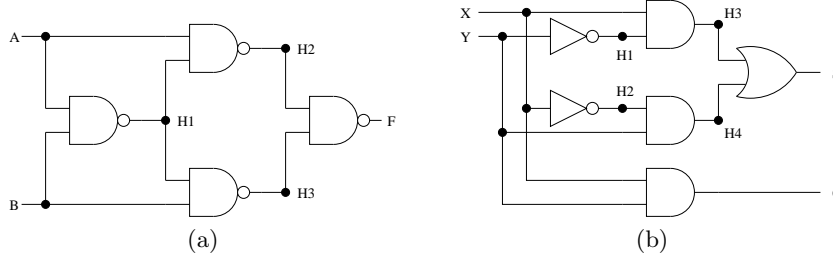
**Surjectivity of a Circuit** We consider the half-adder circuit shown in Figure 6. This circuit can be described with the following formula:

$$\varphi_{\text{ha}} := \exists h_1 \exists h_2 \exists h_3 \exists h_4 (h_1 = \sim y \wedge h_2 = \sim x \wedge h_3 = x \& h_1 \wedge h_4 = h_2 \& y \wedge s = h_3 | h_4 \wedge c = x \& y).$$

Our half-adder has two inputs  $x$ ,  $y$  and two outputs  $s$ ,  $c$ . Our question is now whether  $s$  and  $c$  can take all possible combinations of values: For finding this out, we apply extended quantifier elimination to the following formula:  $\forall x \forall y \exists s \exists c (\varphi_{\text{ha}})$ . This yields **false** plus the answer  $c = 1$ ,  $s = 1$ . Hence we know that these output values cannot be obtained by any input values for  $x$  and  $y$ .



**Fig. 5.** NOT, AND, OR, NAND and NOR gate.



**Fig. 6.** A circuit consisting of four NAND's (a) and a half-adder (b).

**Boolean Output Function** We consider circuits with inputs  $i_0, \dots, i_n$  and one output  $f$ . A *Boolean output function* for such a circuit is a quantifier-free formula  $\varphi'$  with free variables being a subset of  $\{i_0, \dots, i_n\}$  and  $F \iff \varphi'$ . As an example, consider the circuit in Figure 7 ([LKM88], p.265, 5.8). The following formula describes this circuit:

$$\begin{aligned} \varphi_{c_2} := \exists h_1 \dots \exists h_9 ( & h_1 = \sim x \wedge h_2 = \sim z \wedge h_3 = x \& y \wedge h_4 = \sim y \wedge h_5 = h_1 \mid h_2 \mid h_3 \wedge \\ & h_6 = h_3 \mid h_2 \mid h_4 \wedge h_7 = x \& h_5 \wedge h_8 = h_5 \& z \& h_6 \wedge h_9 = h_6 \& y \wedge \\ & f = h_7 \mid h_8 \mid h_9). \end{aligned}$$

By substituting 1 for  $f$  and applying quantifier elimination plus DNF computation, we derive the following formula as Boolean output function for this circuit (10 ms):

$$(x = 1 \wedge z = 0) \vee (x = 0 \wedge y = 1 \wedge z = 0) \vee (x = 1 \wedge y = 1 \wedge z = 1) \vee (x = 0 \wedge y = 0 \wedge z = 1).$$

In the more general case that a Boolean circuit described by a formula  $\varphi$  has several outputs  $f_1, \dots, f_m$ , the Boolean output function for the  $i$ -th output  $f_i$  is obtained by applying quantifier elimination to

$$\exists f_1 \dots \exists f_{i-1} \exists f_{i+1} \dots \exists f_m (\varphi[1/f_i]).$$

**Analysis of Faulted Circuits** A well-established fault model for digital logic testing [Mic86] is the following: By a manufacturing defect, some input line or the output line of a gate has constant value 0 or 1. Such faults are called stuck-at-0 and stuck-at-1, or shortly S-A-0 and S-A-1. This model dates back to the early 1960s [Eld59, Arm66]. The question is as follows: Given a certain gate and a specification of a possible fault, what are input values, such that a faulted value is visible at an output line?

We assume we have obtained the logical description  $\varphi$  of a circuit as described above. To model an output S-A- $k$  fault ( $k \in \{0, 1\}$ ) at gate  $G_i$ , we drop the quantifier  $\exists h_i$  and the one component of the conjunction corresponding to  $G_i$  in  $\varphi$ . We substitute  $k$  for  $h_i$  and call the result  $\varphi_{S-A-k}$ . Then we apply extended quantifier elimination to the following formula:

$$\forall (\varphi \iff \varphi_{S-A-k}).$$

If a circuit has  $n$  gates and a total of  $m$  input and output lines, then after prenex normal form computation such a formula has  $m + 2(n + (n - 1))$  quantifiers. We either obtain **true**, which means that the fault does not change the behavior of the circuit for any input, or **false**, in which case we get values for the input lines, such that the faulted circuit behaves differently from the good one. Again, it is clear that this test can be automatized.

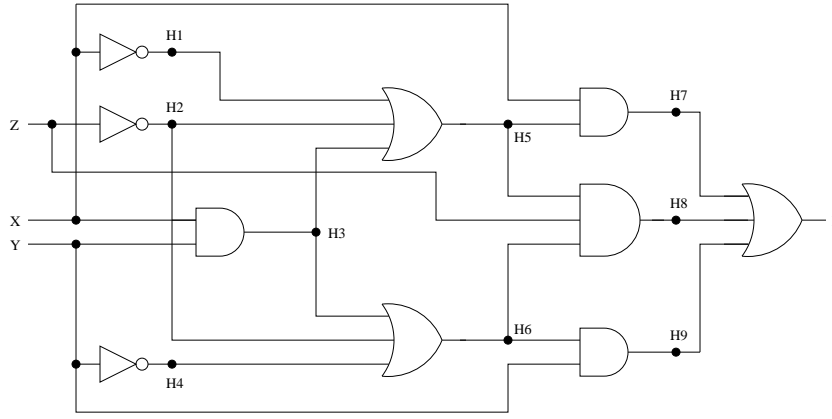


Fig. 7. Circuit with 10 gates.

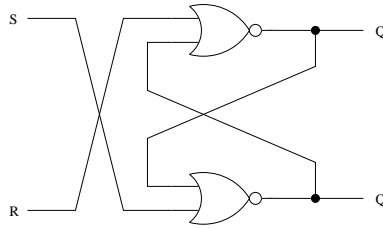


Fig. 8. Active-HIGH SR-latch.

We demonstrate this technique using the circuit from Figure 7. Let us consider an output S-A-1 fault at gate 5. This is the upper 3-input OR gate. The modified formula reads as follows:

$$\begin{aligned} \varphi_{c_2, S-A-1} := \exists h_1 \dots \exists h_4 \exists h_6 \dots \exists h_9 ( & h_1 = \sim x \wedge h_2 = \sim z \wedge h_3 = x \& y \wedge h_4 = \sim y \wedge \\ & h_6 = h_3 \mid h_2 \mid h_4 \wedge h_7 = x \& 1 \wedge h_8 = 1 \& z \& h_6 \wedge \\ & h_9 = h_6 \& y \wedge f = h_7 \mid h_8 \mid h_9). \end{aligned}$$

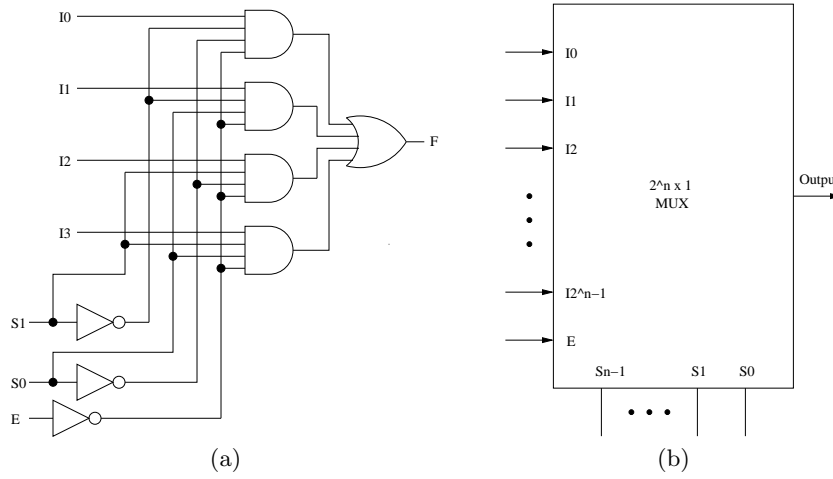
Extended quantifier elimination applied to  $\forall(\varphi_{c_2} \iff \varphi_{c_2, S-A-1})$  yields **false** and values  $x = 1$ ,  $y = 0$ , and  $z = 1$ . Testing these input values, we indeed verify that the good circuit outputs  $f = 0$ , but the circuit with S-A-1 fault at gate 5 outputs  $f = 1$ . Hence  $x = 1$ ,  $y = 0$  and  $z = 1$  is a suitable test pattern to detect an S-A-1 fault at gate 5.

**Active-HIGH SR-Latch** SR-latches are an example for an *asynchronous sequential circuit* in contrast to the combinational circuits considered so far. The characteristic difference is that in sequential circuits there are feedback connections from the outputs to the inputs. One consequence of this is that such circuits have a *memory*. In contrast to combinational circuits the outputs are not longer a function of the inputs but also depend on the current internal state of the device.

An SR-latch has two inputs  $s$  (set) and  $r$  (reset) and two outputs  $q$  and  $q'$ . A possible realization would be with two NOR gates as shown in Figure 8. The input  $r = s = 1$  is not admissible. The internal memory of the SR-latch is the state of  $q$ . At any time, we have  $q' = \sim q$ . If  $r = s = 0$ , then  $q$  remains in its current state, which can be either 0 or 1. If  $s = 1$ , then we get  $q = 1$ , and if  $r = 1$ , then we get  $q = 0$ , not matter what the previous state of  $q$  was.

The following formula describes the latch in the same way as just seen for combinational circuits. In addition, it contains the above-mentioned restriction on  $s$  and  $r$ :

$$\varphi_{\text{SRL}} := q = \sim(r \mid q') \wedge q' = \sim(s \mid q) \wedge \neg(r = 1 \wedge s = 1).$$



**Fig. 9.** A  $4 \times 1$ -multiplexer and the block diagram of a  $2^n \times 1$  multiplexer.

For  $\varphi_{\text{SRL}}$  we can immediately prove that  $q = \sim q'$  since quantifier elimination applied to the following formula yields true in less than 10 ms:  $\forall q \forall q' \forall r \forall s (\varphi_{\text{SRL}} \implies q = \sim q')$ .

We are going to obtain a relation describing all possible combinations of inputs and output, thus resembling the Boolean output function of combinational circuits. For this, we eliminate the complementary counterpart  $q'$  of  $q$  treating it like an inner node. This corresponds to quantifier elimination on  $\exists q'(\varphi_{\text{SRL}})$ . This plus subsequent DNF computation yields after less than 10 ms altogether the following result:

$$\varphi' := (q = 1 \wedge r = 0) \vee (q = 0 \wedge s = 0).$$

The following table lists the values of this  $\varphi'$  for all choices of  $r$ ,  $s$ , and  $q$ . Whenever we have  $\varphi' = 1$ , then this indicates a possible state of the latch, and vice versa:

$r$	0	0	0	0	1	1	1	1
$s$	0	0	1	1	0	0	1	1
$q$	0	1	0	1	0	1	0	1
$\varphi'$	1	1	0	1	1	0	0	0

## 7 Benchmarking

We give some examples from which we can systematically construct sequences of increasingly difficult problems of the same shape for benchmarking.

### 7.1 A Sequence of Multiplexers

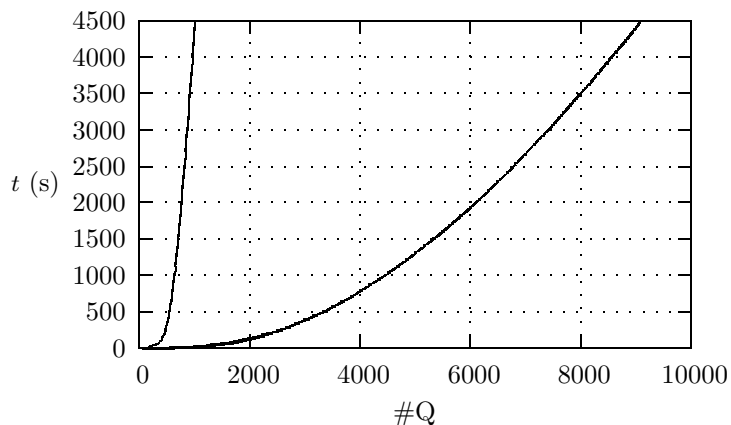
A  $2^n \times 1$ -multiplexer is a combinational circuit that selects data from one of  $2^n$  input lines  $i_0, \dots, i_{2^n-1}$  and routes it to a single output line  $f$ . To address the input line,  $n$  selection lines  $s_0, \dots, s_{n-1}$  are used. Furthermore, multiplexers usually have an additional input line  $e$  to enable or disable the unit. In this example, the convention is that a multiplexer is enabled if  $e = 0$ . See Figure 9 for a  $4 \times 1$ -multiplexer.

Let  $\sigma(k, l)$  be the  $l$ -th bit in the binary representation of  $k \in \mathbb{N}$  (counting starts with the least significant bit). Then the following formula is a possible logical description of a  $2^n \times 1$ -multiplexer, which can be easily obtained by elimination of interior nodes as described before:

$$\varphi_{\text{mux},n} := f \iff \bigvee_{k=0}^{2^n-1} \left( i_k = 1 \wedge e = 0 \wedge \bigwedge_{l=0}^{n-1} s_l = \sigma(k, l) \right) \quad (n > 0).$$

**Table 2.** Performance on the multiplexer benchmark series:  $n$  is the index,  $\#Q$  is the number of quantifiers,  $\#\varphi_{\text{muxcor},n}$  is number of equations of the quantified-propositional normal form,  $t_{\text{GEN}}$  is the time required for automatically generating this example,  $t_{\text{QE}}$  is the time for quantifier elimination using our variable selection strategy, dual  $t_{\text{QE}}$  is the time for quantifier elimination using the dual strategy for comparison.

$n$	$\#Q$	$\#\varphi_{\text{muxcor},n}$	$t_{\text{GEN}}$ (s)	$t_{\text{QE}}$ (s)	dual $t_{\text{QE}}$ (s)
1	5	19	0	0	0
2	8	43	0	0	0
3	13	99	0	0	0
4	22	227	0	0	0
5	39	515	0	0	0.1
6	72	1155	0	0	0.8
7	137	2563	0	0.2	4.9
8	266	5635	0	0.6	40.5
9	523	12291	0.1	2.6	359.0
10	1036	26627	0.4	16.9	4618.5
11	2061	57347	1.4	134.8	81384.8
12	4110	122883	5.6	826.1	—
13	8207	262147	22.0	3669.8	—
14	16400	557059	125.9	12674.2	—



**Fig. 10.** Interpolated computation times for the multiplexer series with standard variable selection strategy (thick) and the dual variant (thin).

The formula  $\varphi_{\text{mux},n}$  contains  $1 + 2^n(2 + n)$  equations. One possible *correctness condition* on such a multiplexer would be to ask, whether, assuming  $e = 0$ , the value of  $f$  corresponds to that of some input  $i_k$  ( $1 \leq k \leq 2^n$ ). This can be expressed as follows:

$$\varphi_{\text{muxcor},n} := \forall \left( \varphi_{\text{mux},n} \wedge e = 0 \implies \bigvee_{k=0}^{2^n-1} i_k = f \right) \quad (n > 0).$$

This provides us with a sequence  $(\varphi_{\text{muxcor},n})_{n>0}$  of benchmark formulas of increasing complexity. The formula  $\varphi_{\text{muxcor},n}$  has  $2^n + n + 2$  variables. The number of equations *after* computation of a quantified-propositional normal form can be found in Table 2, where we summarize the performance of our quantifier elimination on this benchmark sequence. The computation times for both our variable selection strategy and its dual variant are pictured in Figure 11. We see that our strategy appears almost linear, while its dual is clearly exponential.

The following two correctness conditions for a  $2^n \times 1$  multiplexer guarantee that it exactly meets its specification:

**Table 3.** Performance on the second multiplexer benchmark series. Here  $\perp$  indicates that the memory size of 128 MB was exceeded.

$n$	#Q	# $\varphi_{\text{muxcor}1,n}$	$t_{\text{QE}}$ (s)	dual $t_{\text{QE}}$ (s)	$n$	#Q	# $\varphi_{\text{muxcor}2,n}$	$t_{\text{QE}}$ (s)	dual $t_{\text{QE}}$ (s)
1	9	30	0	0	1	10	30	0	0
2	26	124	0	0	2	32	124	0	0
3	83	568	0.3	0.5	3	104	568	0.6	9.9
4	292	2672	148.3	405.3	4	352	2672	45.2	$\perp$
5	1093	12512	>6h	>6h	5	1248	12512	6404.6	$\perp$
6	4230	57792	—	—	6	4608	57792	$\perp$	$\perp$

1. For all possible values of the selection lines  $s_0, \dots, s_{n-1}$  there exists an input line  $k$  such that for all possible values for the input lines  $i_0, \dots, i_{2^n-1}$  and  $e$  and  $f$  we have  $i_k = f$  for the enabled multiplexer. The following formula, which has  $2^n \cdot (2^n + 2) + n$  quantifiers, states this:

$$\varphi_{\text{muxcor}1,n} := \forall s_0 \dots \forall s_{n-1} \bigvee_{k=0}^{2^n-1} \forall i_0 \dots \forall i_{2^n-1} \forall e \forall f (\varphi_{\text{mux},n} \wedge e = 0 \implies i_k = f) \quad (n > 0).$$

2. For all input lines  $k$  there exist values for the selection lines  $s_0, \dots, s_{n-1}$  such that for all values for the input lines  $i_0, \dots, i_{2^n-1}$  and  $e$  and  $f$  we have  $i_k = f$  for the enabled multiplexer. This is expressed by the following formula, which has  $(n + 2^n + 2) \cdot 2^n$  quantifiers:

$$\varphi_{\text{muxcor}2,n} := \bigwedge_{k=0}^{2^n-1} \exists s_0 \dots \exists s_{n-1} \forall i_0 \dots \forall i_{2^n-1} \forall e \forall f (\varphi_{\text{mux},n} \wedge e = 0 \implies i_k = f) \quad (n > 0).$$

Timings for these two series are collected in Table 3.

## 7.2 A Sequence of QSAT Problems

Consider a quantifier-free formula  $\psi$  in CNF, containing at most the Boolean variables  $x_1, \dots, x_n$ . Then the following formula is in QSAT:

$$\exists x_1 \forall x_2 \dots Q_n x_n (\psi).$$

The strict alternation of quantifiers is not a restriction, as one can introduce dummy variables that do not appear in  $\psi$ . Hence, up to equivalence of formulas, our implementation in particular solves QSAT-problems.

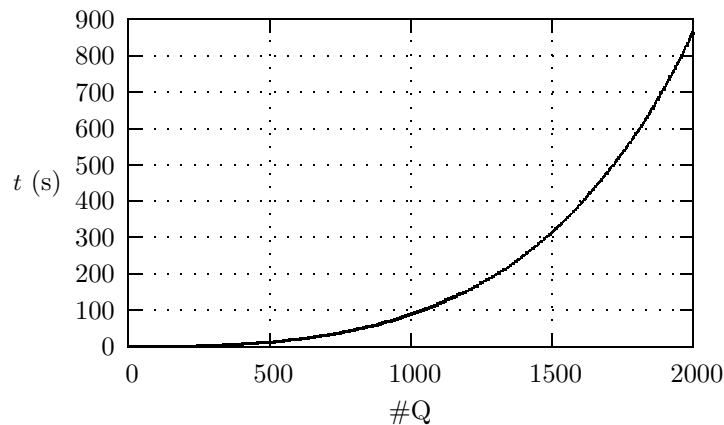
QSAT-problems can be interpreted as 2-person games [Pap94]. Two players, call them  $\exists$  and  $\forall$ , move alternatingly by choosing a value 0 or 1 for the current variable, and  $\exists$  moves first.  $\exists$  tries to make  $\psi$  “true,” while  $\forall$  tries to make  $\psi$  “false.” With this interpretation a QSAT formula is “true” iff  $\exists$  has a winning strategy.

We consider the following sequence of QSAT problems for  $n \in \{1, \dots, 2000\}$ :

$$\varphi_{\text{qsatnands},n} := \exists x_1 \forall x_2 \dots Q_n x_n \left( \bigwedge_{i=1}^{n-1} \sim(x_i \& x_{i+1}) = 1 \right)$$

The formula  $\varphi_{\text{qsatnands},n}$  contains  $n$  variables and  $2(n-1)$  equations after quantified-propositional normal form computation. Interpreting such a formula, for fixed  $n$ , as a game as just described,  $\exists$  has a winning strategy: By choosing 0 for every  $x_i$  ( $i$  odd) all NAND’s will become true, independently of  $\forall$ ’s choice for  $x_{i+1}$ .

We can verify this: Applying quantifier elimination to  $\varphi_{\text{qsatnands},n}$  yields true for any choice of  $n$ . See Figure 11 for timings for  $n \in \{1, \dots, 2000\}$ .



**Fig. 11.** Timings for  $\varphi_{\text{qsatnands},n}$  for  $n \in \{1, \dots, 2000\}$ . Variable selection plays no role here; all quantifier blocks have length 1.

### 7.3 Competing with SAT and QSAT Solving

Our present system can not at all compete with state-of-the art solvers that are specialized in SAT or QSAT [GNT01,MMZ<sup>+</sup>01,Rin99]. We have unsuccessfully tried to solve the first instance of `ii16` from the DIMACS benchmarks of SAT problems and Rintanen’s QSAT benchmark `BLOCKS3i.4.4`. The former did not finish within 6 hours. For the latter, the quantifier elimination exceeded the memory of 128 MB.

This is a consequence of the enormous generality of our approach: In contrast to the above mentioned specialized systems, we can cope with free variables. Moreover, the quantifier-free parts of our prenex input formulas need not be in conjunctive normal form. Note that our method does not exploit the conjunctive normal form in which the SAT and QSAT benchmarks are given, and that conjunctive normal form computation possibly blows up the size of a formula exponentially.

## 8 Conclusions

We have shown how to embed propositional logic into a first-order context in a syntactically clean way. This made it possible to apply first-order methods (e.g. quantifier elimination), concepts (e.g. quantification) and existing generic implementations (e.g. REDLOG’s generic quantifier elimination and simplification procedures) for first-order languages and theories. The class of problems that can be modeled and solved within our framework is a superset of QSAT and includes many interesting applications, e.g. from logical circuit design and testing. Using our implementation in REDUCE we have demonstrated the relevance of our approach and the capabilities of our system by various application and benchmark examples.

## Acknowledgment

Andreas Dolzmann has pointed us at the variable selection strategy to choose the variable of most frequent occurrence.

## References

- [Arm66] D. B. Armstrong. On finding a nearly minimal set of fault detection tests for combinational logic nets. In *IEEE Trans. Electron. Comput.*, volume EC-15, pages 66–73. February 1966.
- [DS97a] Andreas Dolzmann and Thomas Sturm. Redlog: Computer algebra meets computer logic. *ACM SIGSAM Bulletin*, 31(2):2–9, June 1997.

- [DS97b] Andreas Dolzmann and Thomas Sturm. Simplification of quantifier-free formulae over ordered fields. *Journal of Symbolic Computation*, 24(2):209–231, August 1997.
- [DS99] Andreas Dolzmann and Thomas Sturm. Redlog user manual. Technical Report MIP-9905, FMI, Universität Passau, D-94030 Passau, Germany, April 1999. Edition 2.0 for Version 2.0.
- [Eld59] R. D. Eldred. Test routines based on symbolic logic statements. In *Journal of the ACM*, volume 6, pages 33–36. January 1959.
- [GNT01] Enrico Giunchiglia, Massimo Narizzano, and Armando Tacchella. QuBE: A system for deciding quantified boolean formulas satisfiability. In *Proceedings of the International Joint Conference on Automated Reasoning (IJCAR'01), Siena, Italy*, June 2001.
- [LKM88] Gideon Langholz, Abraham Kandel, and Joe L. Mott. *Digital Logic Design*. Wm. C. Brown Publishers, Dubuque, Iowa, 1988.
- [Mic86] Alexander Miczo. *Digital Logic Testing and Simulation*. Harper & Row, Publishers, New York, 1986.
- [MMZ<sup>+</sup>01] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an efficient SAT solver. In *Proceedings of the 38th Design Automation Conference (DAC'01)*, June 2001.
- [Pap94] Christos H. Papadimitriou. *Computational Complexity*. Addison-Wesley, 1994.
- [Rin99] Jussi Rintanen. Improvements to the evaluation of quantified Boolean formulae. In *Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence (IJCAI'99), July 31-August 6, Stockholm, Sweden*, pages 1192–1197. Morgan Kaufmann, 1999.
- [Wei88] Volker Weispfenning. The complexity of linear problems in fields. *Journal of Symbolic Computation*, 5(1&2):3–27, February–April 1988.
- [Wei97] Volker Weispfenning. Simulation and optimization by quantifier elimination. *Journal of Symbolic Computation*, 24(2):189–208, August 1997. Special issue on applications of quantifier elimination.