Preprocessing Techniques for QBFs

Enrico Giunchiglia[1], Paolo Marin[1], and Massimo Narizzano[1]

DIST - Università di Genova
Viale Causa 13, 16145 Genova, Italy
`name.lastname@unige.it`

Abstract

In this paper we present *sQueezeBF*, an effective preprocessor for QBFs that combines various techniques for eliminating variables and/or clauses. In particular *sQueezeBF* combines (i) variable elimination by *Q-resolution* and equality reduction, and (ii) clause simplification via subsumption and self-subsumption resolution. The experimental analysis shows that *sQueezeBF* can produce significant reductions in the number of clauses and/or variables — up to the point that some instances are solved directly by *sQueezeBF* — and that it can significantly improve the efficiency of a range of state-of-the-art QBF solvers — up to the point that some instances cannot be solved without *sQueezeBF* preprocessing.

1 Introduction

Quantified Boolean Formulas are a powerful extension of the Satisfiability (SAT) problem in which the variables are universally as well as existentially quantified. Adding the quantification makes QBF a more expressively compact language with respect to SAT, but this comes with a price: QBF are believed to be in practice much harder to solve than SAT formulas. Many different problems can be efficiently encoded as QBF instances, and recently there has been a great interest and progress in solving such instances efficiently, such as in Verification [1, 2], Planning (Synthesis) [3, 4], and Reasoning about Knowledge [5].

Preprocessing formulas has been proven to be very effective for solving SAT instances since it can reduce their size considerably and decrease the solving time substantially, even taking into account the time required to perform the preprocessing. Recently two preprocessors have been presented in the QBF literature. *preQuel* [6, 7] tries to simplify the formula deriving as many binary clauses as possible, then it applies binary equality reduction to eliminate variables and clause subsumption to eliminate clauses. *preQuel* iterates these operations until no other simplification is possible, decreasing substantially the size of the formula. *proverbox* [8], instead, attempts to reduce the number of variables by selectively applying expansion on universal variables and resolution on the existential ones. This approach is similar to the quantifier expansion presented in *Quantor* [9] first and in *Nenofex* later [10]. It is well known that expansion can increase dramatically the size of the formula, but in *proverbox* only a conveniently selected subset of universally quantified variables with bounded expansion costs, is expanded.

In this paper we present *sQueezeBF*, an effective preprocessor for QBFs that combines various techniques for eliminating variables and/or clauses. In particular *sQueezeBF* combines variable elimination by *Q-resolution* and equality reduction, and clause simplification via subsumption and self-subsumption resolution. Some of the techniques implemented in *sQueezeBF* have been first proposed in SAT (see for example [11], [12]), however their extension to QBF poses many issues, one for all the variable ordering in the prefix has to be taken into account during the variable elimination. The experimental analysis shows that *sQueezeBF* significantly improves the efficiency of a range of state-of-the-art QBF solvers. In particular *sQueezeBF*: (*i*) reduces the size of the preprocessed formula, (*ii*) resolves by itself some instances and (*iii*) when coupled with a QBF solver, is able to improve the solver efficiency significantly. On the other side the experimental analysis also shows that in some cases *sQueezeBF* is not able to simplify the formula and thus to positively affect the performances of the coupled QBF solver. Finally, comparing *sQueezeBF* with *preQuel* and *proverbox* we see that most of the time *sQueezeBF* outperforms both the preprocessors in terms of size reduction and number of problem resolved when considering various *state-of-the-art* QBF solvers.

This paper is organised as follows. First we review the basics of QBF satisfiability. Then we discuss the algorithm of *sQueezeBF*. We end the paper with the experimental analysis and the conclusions.

2 Basic Definitions

Consider a set P of propositional letters. A *variable* is an element of P. A *literal* is a variable or the negation of a variable. In the following, for any literal l,

- $|l|$ is the variable occurring in l; and
- $\bar{l}$ is $\bar{l}$ if l is a variable, and is $|l|$ otherwise.

A *clause* C is an n-ary ($n \geq 0$) disjunction of literals such that, for any two distinct disjuncts l, l' in C, it is not the case that $|l| = |l'|$. A *propositional formula* is a k-ary ($k \geq 0$) conjunction of clauses.

A *QBF* is an expression of the form

$$Q_1 z_1 \dots Q_n z_n \Phi \qquad (n \geq 0) \tag{1}$$

where

- every Q_i ($1 \leq i \leq n$) is a quantifier, either existential $\exists$ or universal $\forall$,

- $z_1, \dots, z_n$ are distinct variables in P, and
- Φ is a propositional formula in the variables $z_1, \dots, z_n$.

In (1), $Q_1 z_1 \dots Q_n z_n$ is the *prefix*, Φ is the *matrix*, and Q_i is the *bounding quantifier* of z_i. Further, we say that a literal l is *existential* if $\exists |l|$ belongs to the prefix, and is *universal* otherwise. In the following, we use TRUE and FALSE as abbreviations for the empty conjunction and the empty disjunction respectively.

We define

- the *level of a variable* z_i, to be $1+$ the number of expressions $Q_j z_j Q_{j+1} z_{j+1}$ in the prefix with $j \geq i$ and $Q_j \neq Q_{j+1}$;
- the *level of a literal* l, to be the level of $|l|$;
- the *level of the formula* (1), to be the level of z_1.

If φ is a QBF and l is a literal, φ_l is the QBF

1. whose matrix Φ is obtained from the matrix of φ by deleting the clauses C such that $l \in C$, and removing $\bar{l}$ from the others, and
2. whose prefix is obtained from the prefix of φ by deleting each variable and corresponding bounding quantifier not occurring in Φ.

The semantics of a QBF φ can be defined recursively as follows. If the prefix is empty, then φ's satisfiability is defined according to the truth tables of propositional logic. If φ is $\exists x \psi$ (respectively $\forall x \psi$), φ is satisfiable if and only if φ_x or (respectively and) $\varphi_{\overline{x}}$ are satisfiable. If $\varphi = Qx\psi$ is a QBF and l is a literal, φ_l is the QBF obtained from ψ by substituting l with TRUE and $\bar{l}$ with FALSE . It is easy to see that if φ is a QBF without universal quantifiers, the problem of deciding the satisfiability of φ reduces to SAT.

In (1), a literal l is

- *Unit* if l is existential, and, for some $m \geq 0$,
  - a clause $(l \vee l_1 \vee \dots \vee l_m)$ belongs to Φ, and
  - each literal l_i $(1 \leq i \leq m)$ is universal and has a lower level than l.
- *Monotone* or *pure* if
  - either l is existential, $\bar{l}$ does not belong to any clause in Φ, and l occurs in Φ;
  - or l is universal, l does not belong to any clause in Φ, and $\bar{l}$ occurs in Φ.

```
0 function sQueezeBF($\varphi$)
1   do
2     $\varphi$' = $\varphi$
3     $\varphi$ = Simplify($\varphi$)
4     $\varphi$ = EquivalenceCheck($\varphi$)
5     $\varphi$ = Q-resolution($\varphi$)
6     if $\varphi \equiv$ TRUE  return $\varphi$
7     if $\varphi \equiv$ FALSE  return $\varphi$
8   while $\varphi$' $\neq \varphi$
9   return $\varphi$
```

Figure 1: The algorithm of *sQueezeBF*.

Given a set of clauses S we define $|S|$ (size of S) as the number of literals contained in S. Finally given a QBF φ, a literal l and a set of clauses α, $\varphi(l/\alpha)$ is the QBF obtained from φ by substituting each occurrence of l with alpha [1].

3 *sQueezeBF*

In Figure 1 is presented the main algorithm of *sQueezeBF*. The algorithm takes in input a QBF, and it returns a simplified QBF, that can either be empty (true) or contain an empty clause (false formula). It starts saving the current state of the formula at line 2, and then it applies three operations sequentially:

- *Simplify*(line 3) gets in input the formula and simplifies the formula propagating all the unit and pure literals. Given a formula φ and a literal l unit or pure in φ, φ is equivalent to φ_l.
- *EquivalenceCheck* (line 4) first discovers all the equivalences of the type
$$l \Leftrightarrow l_1 \circ l_2 \ \ldots\ l_{n-1} \circ l_n$$
where $\circ \in \{\vee, \wedge\}$ and $n \geq 1$, or $\circ \in \{\Leftrightarrow\}$ and $n = 2$. Notice that if $n = 1$ we obtain a binary equivalence. Once the equivalence is found, then it substitutes every occurrences of l with its definition, if and only if the substitution does not increase the size of the formula in terms of literals.
- *Q-resolution*(line 5) can eliminate the variable by *Q-resolution*. The algorithm for each variable z checks if it is possible to eliminate z

[1]strictly speaking the result is not a QBF. We assume that the resulting expression is suitably converted to a QBF without introducing additional variables. For instance, each clause $C \vee l$ gets substituted by the clauses in $\{C \vee C' : C' \in \alpha\}$

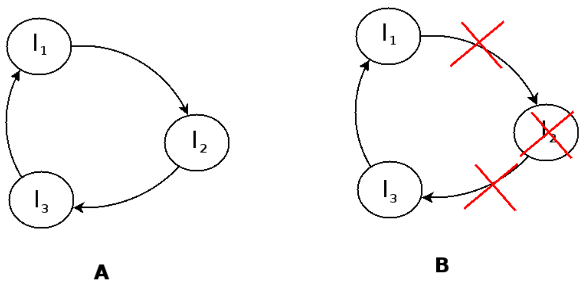

Figure 2: Dependency graph

by Q-resolving all the clauses where z occurs positively with all the clauses where z occurs negatively. If the formula does not increase in size then the variable is eliminated.

If the formula is neither satisfiable (line 7) nor unsatisfiable (line 6) then the algorithm iterates the operations until no further simplification is possible (line 8).

3.1 Variable Elimination via equivalence checking

The variable elimination via equivalence checking was introduced for SAT in [12]. It is an algorithm that works in two steps: (i) identification of *definitions* and (ii) variable substitution. In the first step *EquivalenceCheck* looks for the following set of clauses in the formula:

$$(\bar{l} \vee l_2 ... \vee l_{n-1} \vee l_n) \wedge (l \vee \bar{l}_1) \wedge ... \wedge (l \vee \bar{l}_n) \tag{2}$$

$$(l \vee l_1 \vee l_2) \wedge (l \vee \bar{l}_1 \vee \bar{l}_2) \wedge (\bar{l} \vee \bar{l}_1 \vee l_2) \wedge (\bar{l} \vee l_1 \vee \bar{l}_2) \tag{3}$$

where in both the formulas (4) and (5) the level of $|l|$ is the lowest among the level of each $|l_i|$ occurring in the equivalence. Notice that the set (2) corresponds to the definition:

$$l \Leftrightarrow l_1 \circ l_2 ... \circ l_{n-1} \circ l_n \qquad \text{where } \circ \in \{\vee, \wedge\} \ \mathit{and} \ n \geq 1 \tag{4}$$

and the set (3) corresponds to the definition:

$$l \Leftrightarrow l_1 \Leftrightarrow l_2 \tag{5}$$

Once that *EquivalenceCheck* finds the equivalences, then it substitutes each variable with its definition in the formula. Notice that after a substitution of a variable z with its definition α in a formula φ, the new formula $\varphi(z/\alpha)$ is not guaranteed to be in CNF, so a conversion step may be necessary. In order to avoid the introduction of new variables with the commonly used translation techniques, we apply $DeMorgan$ rules. It is well known that *DeMorgan* rules can increase the size of the resulting QBF. However, by eliminating redundant literals or clauses in the produced set of clauses, such increase often does not happen. In the cases in which it does, we discard the changes and substitution does not take place. This ensures that the formula never increases in size.

Finally, notice that not all the defined variables can be substituted. Take for example:

$$\varphi = \alpha \wedge (l_1 \Leftrightarrow l_2 \vee l_4) \wedge (l_2 \Leftrightarrow l_3 \vee l_4) \wedge (l_3 \Leftrightarrow l_1 \vee l_5)$$

if l_1 is eliminated first, then when l_3 is substituted in the formula, l_1 is reintroduced, and this is true no matter which variable is substituted first. This problem arises since at least a variable occurs in the definitions of the others. In order to solve the problem of circularity, *EquivalenceCheck* constructs a dependency graph where each node represents a defined variable, and each edge, connecting the defined variables, represents the dependency between the definitions. So for example, the Figure 2:A represents the dependencies graph of the example above, where the edge from the node l_3 and the node l_1 represents the fact that l_1 occurs in the definition of l_3. Notice that the edges have a direction, representing the fact that the pointed node is in the definition of the non pointed one. In Figure 2:A, l_1 occurs in the definition of l_3. After the graph is created then the algorithm looks for circular path and if any is found then the path is cut eliminating one of the definition. Looking at Figure 2:A in order to eliminate the circular path the algorithm deletes one of the three definitions, for example l_2 (Figure 2:B). At this point we can substitute first l_3 and then l_1 and the formula is simplified eliminating 2 variables. Notice that the order matters, since if we substitute l_1 first, then once l_3 is substituted l_1 is reintroduced.

3.2 Variable Elimination via *Q-resolution*

Variable elimination via resolution is a technique used in many state-of-the-art SAT solvers, first introduced in [13] during the search and also used in [14] as preprocessor. In QBF variable elimination via *Q-resolution* has been first introduced by *Quantor* [9]: during the search it eliminates the existential variables with the higher prefix level by *Q-resolution*.

Given two clauses $C_1 = \{z, l_1, ..., l_n\}$ and $C_2 = \{\overline{z}, l'_1, ..., l'_m\}$ the implied clause $C = \{l_1, ..., l_n, l'_1, ..., l'_m\}$ is called the *resolvent* of the two original clauses by performing *resolution* on the variable z. We write $C = C_1 \otimes C_2$. We can also define the *Q-resolution* on set of clauses: given a set of clauses S_z, meaning that all contain z and a set of clauses $S_{\overline{z}}$ (all containing $\overline{z}$) we define the resolvent between set of clauses as:

$$S_z \otimes S_{\overline{z}} : \{C_z \otimes C_{\overline{z}} | C_z \in S_z, C_{\overline{z}} \in S_{\overline{z}}\} \tag{6}$$

The elimination of a variable z from a QBF can be computed by pairwise resolving each clause in S_z with every clause in $S_{\overline{z}}$. The produced resolvent

$S' = S_z \otimes S_{\overline{z}}$ replaces the original clauses containing z and $\overline{z}$ resulting in an equivalent problem.

As an example of how the algorithm works, take the formula φ containing a variable x and two set of clauses

$$S_x = \{\{x \vee \overline{a}\}, \{x \vee a \vee c\}, \{x \vee \overline{d}\}\} \text{ and } S_{\overline{x}} = \{\{\overline{x} \vee a\}, \{\overline{x} \vee b \vee d\}\}.$$

then the resolvent between the two set of clauses is

$$S_x \otimes S_{\overline{x}} = \{\{\overline{a} \vee b\}, \{\overline{a} \vee b \vee d\}, \{a \vee c \vee b\}, \{a \vee c \vee b \vee d\}, \{b \vee \overline{d}\}\}.$$

In order to eliminate the variable x from φ, all the clauses in $S_x \cup S_{\overline{x}}$ have to be deleted from φ, adding all the clauses in $S_x \otimes S_{\overline{x}}$ and obtaining the formula φ'. Notice that in the example, the size of φ' is greater then the size of φ, since $|S_x \cup S_{\overline{x}}| = 12$, and $|S_x \otimes S_{\overline{x}}| = 14$. In order to avoid an increase in the size of the formula, *Q-resolution* does not eliminate all the variables, but only the ones that don't increase the size. In particular a variable z is eliminated from a formula φ if and only if

$$|S_z \cup S_{\overline{z}}| \geq |S_z \otimes S_{\overline{z}}| \tag{7}$$

Notice that the dimension of the resolvent can not be calculated in advance since many trivial clauses, that have to be discarded, can be generated during the *Q-resolution* process. For this reason the algorithm first calculates the dimension of the original set of clauses, and then it computes the resolvent, discarding it if (7) is not satisfied, and the variable is not deleted.

3.3 Clause Elimination via Self/Backward/Forward Subsumption

A clause C_1 is said to *subsume* a clause C_2 if the literals contained in C_1 are a subset of the literals contained in C_2, and abusing with notation we write that $C_1 \subseteq C_2$. A subsumed clause is redundant and can be discarded without changing the QBF it represents. Since redundant clauses are memory consuming, and slow down the search process, it is desirable to make a QBF subsumption free, by detecting and removing the subsumed clauses.

Whenever a new clause is added to the formula, for example as consequence of a variable elimination, *sQueezeBF* checks all the existing clauses in the database to see whether they are subsumed. This check is usually called as *backward subsumption* [15]. The newly added clause is also checked against all the existing clauses to see if it is subsumed by any existing clauses. This check is usually called as *forward subsumption* [15].

Moreover the subsumption algorithm can be expanded: suppose to have a clause $C_1 = \alpha \vee l$ and a clause $C_2 = \beta \vee \overline{l}$, where the sub-clause $\alpha \subseteq \beta$. We could say that C_2 almost subsumes C_1 except for the literal l. Applying resolution on the two clauses we obtain a new clause $C = \alpha \vee \beta$, but since α subsume β the derived clause becomes $C = \beta$, that subsume C_2. Thus after adding C to the database we can delete C_2, in essence eliminating one literal. In this case we say that C_2 is self subsumed by C and we call this technique *self-subsuming resolution* (see for more details [12]).

Even if eliminating clauses may reduce the size of the formula, the common perception seems to be that subsumption and removal are expensive, especially for solvers based on search. However, subsumption removal is important for solvers or preprocessors based on resolution like for example the solver *Quantor* [9] and the preprocessor NiVER [14]. Resolution based solvers are usually memory limited, and the resolution operation often generates large number of clauses that are subsumed by existing ones. Given a clause C, the naive algorithm for discovering subsumption consists in comparing C with each clause containing at least one literal occurring in C. This algorithm can be very expensive especially if the literals in the clauses are not ordered and/or if the clause database is large. In [9], the author proposed a signature based algorithm for backward subsumption detection, that significantly reduces the number of checks. For each clause is attached a signature, which in practice is a number computed with an hash function based on the literals in the clause. If the clause C_1 subsumes the clause C_2, they have the same hash number, i.e. the same signature. It is not always true that if C_1 has the same signature of C_2 then C_1 subsume C_2 or vice-versa. The efficiency of this algorithm depends from the hashed function used to compute the signature, i.e. if many clauses have the same signature then the algorithm converges to the naive one, on the other side, if we have a unique signature for each clause, then it may blow up in space. Implementing a good hash function has to be a trade off between efficiency in time and efficiency in space.

sQueezeBF implements subsumption removal based on the occurrences of the literals. For each clause we define a counter indicating how many times the clause is visited, always initialised to zero. When a new clause C is introduced, then all the occurrences of each literal of C are visited and the corresponding counter is incremented. If in any time a counter reach the size of the clause visited, then it subsumes C, and C is not learned (forward subsumption). On the other hand, if a clause counter reach the size of C, then C subsumes the one checked. The new clause is learned and the one subsumed is eliminated (backward subsumption). The following example explains better the algorithm: suppose that we want to introduce the following clause $C = l_1 \vee l_2 \vee l_3$, into a database containing 3 clauses : $C_1 = l_1 \vee l_2$, $C_2 = l_3 \vee \bar{l}_4$, $C_1 = \bar{l}_2 \vee l_3$. Starting from l_1 we visit C_1 and we increment its counter from 0 to 1, then we check l_2 visiting again C_1 and incrementing its counter from 1 to 2. Since the counter associated to C_1 is equal to its size, this means that C_1 subsumes C, and thus *sQueezeBF* does not learn C.

4 Experimental Analysis

To evaluate the effectiveness of our preprocessor, we compare *sQueezeBF* with *preQuel* and *proverbox* on a selected pool of fixed-structure QBF instances taken from QBFLIB [16]. We first compare their effectiveness computing the size of each formula before and after the preprocessing. The results showing this comparison are reported in Table 1, where for each family (listed in the first column) we report (*i*) the average number of clauses (C), (*ii*) variables (V), and (*iii*) literals per clause (L/C) of the original formula (column "original"). Then, for each family we show (*i*) the average variation in the number of clauses (C%) and (*ii*) variables (V%), with respect to the column "original", and (*iii*) the average literals per clause (L/C) obtained using each preprocessor (in the columns *proverbox*, *preQuel*, *sQueezeBF*). For example taking the first row, Blocks, and the first column, the three values (C,V,L/C) = (6810,485,2.96) represent respectively the number of clauses in average (6810), the number of variables in average (485) and the literal per clause in average (2.96). The other columns state the variation with respect to the first one, i.e. for example the column *proverbox* shows the values (C%,V%,L/C) =(-39,-24,3.06) representing the fact that running *proverbox* makes the formula smaller, having 39% less clauses and 21% less variables with respect to the original problem. In the limit cases in which: (*i*) a preprocessor has failed while working out a formula (i.e. due to Timeouts or Memory outs), the features of the output formula are considered to be the same of the input one [2]; (*ii*) a preprocessor has solved a formula, we consider the output formula to have: 0 variables and 1 clause in case the preprocessor has returned false; or 0 variables and 0 clauses in case the preprocessor has returned true. All the experiments have been run on a farm of PCs, each one equipped with a PIV 3.2 GHz processor, 1GB of RAM, and running Linux Ubuntu 6.10; the time limit has been set to 600s. Counting the number of families in which each preprocessor has performed best:

- *sQueezeBF* is the most effective preprocessor always reducing the number of clauses and variables: the reduction ratio is the highest with the exception of the families Blocks and comp; excluding the FPGA families, the average reduction ratio is more than 50%;

- *preQuel* and *proverbox* are the most effective in shortening the clauses, but usually the latter is the least effective in reducing clauses and variables;

- all the preprocessors are effective reducing the size of the problem preprocessed. However, two families do not follow this trend: the FPFF and FPFS, where *preQuel* and *sQueezeBF* do not change the size of the QBFs, whereas *proverbox* double the size of each formula

[2] *proverbox* is the only preprocessor that has failed on some formulas.

in average. This is probably due to the fact that the problems are already written in a compact CNF, and applying preprocessor without size bound can only increase the size.

It can also be observed how some families have an average reduction, in terms of number of clauses and variables, equal to 100%: all the instances of these families have been solved by the preprocessor.

Overall,

- *proverbox* has solved 20 instances (10 DFlipFlop, 1 wmiforward, 3 SzymanskiP, and 6 VonNeumann),
- *preQuel* has solved 20 instances (10 DFlipFlop and 10 VonNeumann), and
- *sQueezeBF* has solved 29 instances (10 DFlipFlop, 9 wmiforward and 10 VonNeumann).

Family	original			*proverbox*			*preQuel*			*sQueezeBF*		
	C	V	L/C	C%	V%	L/C	C%	V%	L/C	C%	V%	L/C
Blocks *(13)*	6810	485	2.96	-39	-24	3.06	**-56**	**-41**	**2.93**	-53	**-41**	3.01
comp *(8)*	815	300	2.36	-13	-17	2.41	-45	**-70**	**2.39**	**-48**	-58	2.76
DFlipFlop *(10)*	82655	62096	2.32	**-100**	**-100**	**0.00**	**-100**	**-100**	**0.00**	**-100**	**-100**	**0.00**
EvPr4x4lg *(7)*	12973	1961	3.72	-13	-14	3.87	-12	-14	3.87	**-23**	**-26**	**3.83**
EvPr4x4s *(7)*	67639	7454	3.25	-23	-10	3.61	-11	-4	**3.35**	**-38**	**-24**	3.53
FPFF *(5)*	658	69	6.38	+109	+57	**5.34**	0	**-1**	6.39	**-2**	**-1**	5.83
FPFS *(3)*	522	72	6.50	+129	+57	**5.49**	0	**-1**	6.52	**-1**	**-1**	6.06
s499 *(6)*	24534	9053	2.62	-20	-18	**2.64**	-19	-17	2.70	**-61**	**-61**	2.76
SN *(84)*	5935	2902	2.57	-4	-19	2.88	-2	-7	**2.58**	**-15**	**-24**	2.72
SzymanskiP *(12)*	90271	74313	2.60	-1	-1	**2.29**	0	0	2.60	**-49**	**-60**	3.17
VN *(10)*	559528	381580	2.34	-20	-20	0.93	**-100**	**-100**	**0.00**	**-100**	**-100**	**0.00**
wmiforward *(72)*	3204	983	2.48	0	0	**2.45**	-15	-38	2.48	**-57**	**-66**	2.82

Table 1: Size-reduction comparison between different preprocessors. "C", "V" and "L/C" denote the average number of clauses, variables, and literals per clause. Best values are written in bold. *sQueezeBF* is the preprocessor with all the techniques enabled. In the column "Family" , EvPr4x4lg = evader-pursuer-4x4-logarithmic, EvPr4x4s = evader-pursuer-4x4-standard, FPFF = FPGA_PLB_FIT_FAST, FPFS = FPGA_PLB_FIT_SLOW, SN = Sorting_networks, VN = VonNeumann; the number written in parentheses represents the number of instances in the family.

The Table 1 clearly states that *sQueezeBF* is the preprocessor with the biggest impact on the formulas, almost halving the size of each formula, on average.

However, *sQueezeBF* is a mix of different techniques and, in order to evaluate the impact that each one has on the preprocessor, *sQueezeBF* has been run disabling each technique one at a time. In Table 2 we show the results, organised as the Table 1, where in each column, instead of a preprocessor, is presented *sQueezeBF* with a technique disabled. In particular, the first column (*sQueezeBF*) represents the full-featured preprocessor (i.e. no technique disabled); the suffixes "– *Eq*", "– *Qr*" and "– *Ss*" represent a version of *sQueezeBF* featuring all the techniques but variable elimination via Equivalence Checking, variable elimination by Q-Resolution or clause simplification via Self-Subsumption respectively. For example taking the row Blocks and the column *sQueezeBF*, the three values (C,V,L/C) = (3211,287,3.01) represent respectively the number of clauses in average (3211), the number of variables in average (287), and the literal per clause in average (3.01). The other columns state the variation with the first one, i.e. looking to the first row (Blocks), the column *Eq* shows the values (C%,V%,L/C) =(+28,+21,3.01) representing the fact that running the preprocessor without variable elimination via equivalence checking makes the formula larger, having 28% more clauses (i.e. almost 4110 clauses on average) and 21% more variables (i.e. almost 347 variables on average).

Table 2 shows that disabling a technique always leads to a bigger formula, i.e. that all the techniques are necessary to obtain the formula with the lowest size. In the table this is highlighted from the absence of negative numbers. Then, we see that the reduction of clauses is mostly affected by disabling Equivalence Checking, the reduction of variables is mostly affected by Equivalence Checking and Q-Resolution, and the reduction of literals per clause is mostly affected by Self-Subsumption.

In the table, the limit case, in which *sQueezeBF* solves an instance and another preprocessor doesn't, is indicated with the symbol "**- -**": the family DFlipFlop hasn't been solved by *sQueezeBF* disabling the Q-Resolution technique. On average, the number of clauses and variables of the instances of this family has been reduced to 77135 and 57940 respectively.

It can be noticed that, even disabling Equivalence Checking, some of the variables that were eliminated because of a definition of equivalence are still eliminated by Q-Resolution, as witnessed from the numbers in the table.

In order to evaluate the behaviour of the different preprocessors coupled with a QBF solver, we have run *QuBE* [3][17], *yQuaffle* [4][18, 19], and *sSolve* [5][20] as search-based solvers; *Quantor* [6][9] as a resolution based solver; and *sKizzo* [7][21] as a symbolic skolemization based solver. Table 3 reports the results where on each box there is number of problems solved,

[3]Release QuBE6.4: its own preprocessor has been disabled.

[4]Version 021006.

[5]Version *sSolve*C from QBFEVAL 08.

[6]Version 3.0.

[7]Version *sKizzo*-0.10-qck from QBFEVAL 07.

Family	*sQueezeBF*			– *Eq*			– *Qr*			– *Ss*		
	C	V	L/C	C%	V%	L/C	C%	V%	L/C	C%	V%	L/C
Blocks *(13)*	3211	287	3.01	**+28**	**+21**	3.01	+15	+10	2.88	+27	+18	**3.08**
comp *(8)*	428	124	2.76	**+34**	**+51**	2.57	+5	+3	2.80	+7	+8	**2.85**
DFlipFlop *(10)*	1	0	0.00	0	0	0.00	**- -**	**- -**	**2.32**	0	0	0.00
EvPr4x4lg *(7)*	9984	1452	3.83	0	+1	3.83	+5	**+16**	**3.86**	**+7**	0	3.83
EvPr4x4s *(7)*	42121	5676	3.53	0	0	**3.53**	**+27**	**+26**	3.40	+14	0	3.52
FPFF *(5)*	645	68	5.83	0	**0**	5.95	0	**0**	5.83	**+2**	**0**	**6.39**
FPFS *(3)*	518	71	6.06	0	**0**	6.06	0	**0**	6.06	**+1**	**0**	**6.52**
s499 *(6)*	9451	3567	2.76	**+112**	**+114**	2.79	0	0	2.76	+39	+44	**2.87**
SN *(84)*	5070	2212	2.72	+1	+1	2.69	**+12**	**+24**	2.62	0	0	**2.74**
SzymanskiP *(12)*	45738	29781	3.17	0	0	**3.17**	**+97**	**+150**	2.60	0	0	**3.17**
VN *(10)*	1	0	0.00	**0**	**0**	**0.00**	**0**	**0**	**0.00**	**0**	**0**	**0.00**
wmiforward *(72)*	1364	339	2.82	**+78**	**+101**	2.60	+21	+25	2.77	+46	+55	**2.89**

Table 2: Size-reduction comparison between different versions of *sQueezeBF*. The header has the same meaning of the one in Table 1, but the reference values are those of *sQueezeBF*. The highest values are written in bold, meaning that disabling the corresponding technique leads to the highest degradation.

and the cumulative solving time, for each solver (on the rows) when coupled with a particular preprocessor (on the columns). Notice that in the column original are presented the results for each solver without any preprocessing applied, while the last three columns represent the different versions of *sQueezeBF*. Table 3 witnesses that *sQueezeBF* is the only one able to constantly improve the efficiency of a range of *state-of-the-art* QBF Solvers. In particular the column "–" shows that the use of *sQueezeBF* improves of a factor two the number of problems solved by a given solver. Using *sQueezeBF* as preprocessor affects also the solving time of each solver, decreasing it substantially. Moreover, the Table 3 also shows which technique has more impact on which solver. For example, it looks like that disabling the variable elimination via equivalence checking, *QuBE* can not longer solve many problems, while the self subsumption is the least effective. Instead, looking at the other solvers, disabling both equivalence checking and clause elimination via self subsumption decrease the performance of the solver. For all the solvers the least effective technique seems to be variable elimination via *Q-resolution*. This is not so surprising since the bound limitation for the *Q-resolution* is very tight, and *sQueezeBF* spends a lot of time because there are many variables which are not eliminated because the size of the formula would increase. Notice that the combination of the three techniques always leads to the best performances for some solvers (*QuBE*, *sKizzo* and *Quantor*), but this is not true for others (*sSolve* and *yQuaffle*) that are able to solve more problems when *Q-resolution* is disabled. Finally it is interest-

				sQueezeBF			
Solver	original	*proverbox*	*preQuel*	–*Eq*	–*Qr*	–*Ss*	–
QuBE	98 (87k)	87 (111k)	111 (78k)	118 (79k)	138 (72k)	148 (70k)	156 (55k)
sSolve	95 (88k)	80 (112k)	114 (77k)	115 (77k)	135 (64k)	120 (77k)	133 (66k)
yQuaffle	82 (95k)	85 (109k)	88 (92k)	88 (86k)	104 (75k)	96 (81k)	101 (79k)
sKizzo	111 (78k)	110 (93k)	120 (73k)	116 (70k)	143 (56k)	120 (69k)	147 (53k)
Quantor	104 (80k)	106 (95k)	128 (67k)	106 (74k)	123 (64k)	106 (76k)	136 (58k)

Table 3: Number of instances solved and cumulative time (in parentheses) by using different preprocessors. Times include both preprocessing and solving, and is expressed in thousands. The first column refers to the solving without the help of any preprocessing. *sQueezeBF* names mean: *sQueezeBF* is the preprocessor with all its techniques enabled, "– *Eq*", "– *Qr*" and "– *Ss*" stand for *sQueezeBF* without equivalence checking, variable elimination by Q-Resolution or self-subsumption respectively.

ing to see in table 3 that the preprocessor, on this set of benchmarks, makes *QuBE* a better solver than *sKizzo*.

About the time needed by the different preprocessors, these are cumulatively presented in Table 3, where it is shown also the effect of the different preprocessors (*sQueezeBF*, *proverbox*, *preQuel*) when coupled with different solvers (*QuBE*, sSolve, yQuaffle, sKizzo and Quantor). In general, the preprocessing time is negligible wrt the whole task of preprocess and solve, but for some large instances it can be more onerous trying to simplify it rather than solving it by a solver. The cumulative preprocessing times are approximately 410s for *preQuel*, 1470s for *sQueezeBF*, and 15500s for *proverbox*.

5 Conclusions

In this paper we present *sQueezeBF*, a very effective preprocessor for QBF reasoning. We took into account many benchmarks from different families and two other different preprocessing tools, *preQuel* and *proverbox*. We have shown that *sQueezeBF* is much more effective in terms of formula reduction, since most of the times decreases the size of the formula preprocessed, and never increases the size of the formula, and this is not always true for *preQuel* and *proverbox*. We also compare five different *state-of-the-art* solvers: the proposed techniques offer robust improvements across the different solvers among all the tested benchmark families. To the best of our knowledge thanks to *sQueezeBF* the solvers are able to solve 34 problems that have never been resolved before. Finally, as future work, we would like to implement new techniques such as, for example, variable expansion and binary clause resolution.

References

[1] Scholl, C., Becker, B.: Checking equivalence for partial implementations. In: Proceedings of the 38th Design Automation Conference (DAC'01). (2001) 238–243

[2] Ayari, A., Basin, D.A.: Bounded model construction for monadic second-order logics. In: CAV '00: Proceedings of the 12th International Conference on Computer Aided Verification, London, UK, Springer-Verlag (2000) 99–112

[3] Rintanen, J.: Constructing conditional plans by a theorem prover. Journal of Artificial Intelligence Research **10** (1999) 323–352

[4] Castellini, C., Giunchiglia, E., Tacchella, A.: Improvements to SAT-based conformant planning. In: Proc. ECP. (2001)

[5] Baader, F., ed.: Automated Deduction - CADE-19, 19th International Conference on Automated Deduction Miami Beach, FL, USA, July 28 - August 2, 2003, Proceedings. In Baader, F., ed.: CADE. Volume 2741 of Lecture Notes in Computer Science., Springer (2003)

[6] Samulowitz, H., Davies, J., Bacchus, F.: QBF Preprocessor Prequel (2006) available at http://www.cs.toronto.edu/~fbacchus/sat.html.

[7] Samulowitz, H., Davies, J., Bacchus, F.: Preprocessing QBF. In: Principles and Practice of Constraint Programming, Springer-Verlag (2006)

[8] Bubeck, U., Büning, H.K.: Bounded universal expansion for preprocessing qbf. In Marques-Silva, J., Sakallah, K.A., eds.: SAT. Volume 4501 of Lecture Notes in Computer Science., Springer (2007) 244–257

[9] Biere, A.: Resolve and expand. In: Proc. SAT. (2004) 59–70

[10] Lonsing, F., Biere, A.: Nenofex: Expanding nnf for qbf solving. In Büning, H.K., Zhao, X., eds.: SAT. Volume 4996 of Lecture Notes in Computer Science., Springer (2008) 196–210

[11] Kleine-Büning, H., Karpinski, M., Flögel, A.: Resolution for quantified Boolean formulas. Information and Computation **117**(1) (1995) 12–18

[12] Eén, N., Biere, A.: Effective preprocessing in sat through variable and clause elimination. [22] 61–75

[13] Davis, M., Putnam, H.: A computing procedure for quantification theory. Journal of the ACM **7** (1960) 201–215

[14] Subbarayan, S., Pradhan, D.K.: Niver: Non increasing variable elimination resolution for preprocessing sat instances. In: SAT. (2004)

[15] Zhang, L.: On subsumption removal and on-the-fly cnf simplification. [22] 482–489

[16] Giunchiglia, E., Narizzano, M., Tacchella, A.: Quantified Boolean Formulas satisfiability library (QBFLIB) (2001) `www.qbflib.org`

[17] Giunchiglia, E., Narizzano, M., Tacchella, A.: Clause/term resolution and learning in the evaluation of quantified Boolean formulas. Journal of Artificial Intelligence Research (JAIR) **26** (2006) 371–416

[18] Zhang, L., Malik, S.: Towards a symmetric treatment of satisfaction and conflicts in quantified Boolean formula evaluation. In: Proceedings of the Eighth International Conference on Principles and Practice of Constraint Programming. (2002) 200–215

[19] Zhang, L., Malik, S.: Conflict driven learning in a quantified Boolean satisfiability solver. In: Proceedings of International Conference on Computer Aided Design (ICCAD'02). (2002)

[20] Feldmann, R., Monien, B., Schamberger, S.: A distributed algorithm to evaluate Quantified Boolean Formulae. In: Proceedings of the 7th Conference on Artificial Intelligence (AAAI-00) and of the 12th Conference on Innovative Applications of Artificial Intelligence (IAAI-00), Menlo Park, CA, AAAI Press (July 30– 3 2000) 285–290

[21] Benedetti, M.: skizzo: A suite to evaluate and certify qbfs. In: Proc. CADE. (2005) 369–376

[22] Bacchus, F., Walsh, T., eds.: Theory and Applications of Satisfiability Testing, 8th International Conference, SAT 2005, St. Andrews, UK, June 19-23, 2005, Proceedings. In Bacchus, F., Walsh, T., eds.: SAT. Volume 3569 of Lecture Notes in Computer Science., Springer (2005)