Non-CNF QBF Solving with QCIR

Charles Jordan
Graduate School of
Information Science and Technology
Hokkaido University

Will Klieber
Software Engineering Institute
Carnegie Mellon University

Martina Seidl
Institute for Formal Models
and Verification
Johannes Kepler University Linz

Abstract

While it is empirically confirmed folklore that conjunctive normal form (CNF) is not the ideal input format for QBF solvers, most tool developers and therefore also the users focus on formulas in this restricted structure.

One important factor for establishing non-CNF solving is the input format. To overcome drawbacks of available formats, the QCIR format has recently been presented. The QCIR format is a circuit-based input format for quantified Boolean formulas which supports structure sharing. In contrast to previous formats, the representation is very compact, yet still easy to parse and to read for the human user.

In this paper, we analyze the QCIR format in detail and provide tools and benchmarks which, we hope, will make its usage attractive and motivate tool developers to support this format as well as users to formulate their encodings in this format.

Introduction

SAT solvers have made tremendous advances in recent years, and are now widely used in applications as general-purpose NP solvers (Biere et al. 2009). However, there are many applications where PSPACE-complete problems must be solved and (assuming $\text{PSPACE} \neq \text{NP}$) these problems cannot be represented as compact SAT encodings. Analogous to SAT solvers and NP, QBF solvers can serve as general-purpose PSPACE solvers.

Quantified Boolean formulas (QBF) extend propositional formulas by allowing explicit quantification ($\exists, \forall$) over the propositional variables (Kleine Büning and Bubeck 2009). The problem of determining whether a particular QBF holds is complete for the complexity class PSPACE, making them a promising host language for encoding and solving many important problems from verification and artificial intelligence (see (Benedetti and Mangassarian 2008) for a survey). Recently, there has been significant progress in QBF in developing fast, practical QBF solvers (see for example, the recent report on the QBF Gallery (Lonsing, Seidl, and Van Gelder 2015)), but QBF solvers have not reached the same level of adoption as SAT solvers so far.

QBF tools have inherited the tradition of usually requiring formulas to be in conjunctive normal form (CNF) from SAT solvers. A propositional formula is in CNF if it is a conjunction of clauses. A clause is a disjunction of literals and a literal is a variable or a negated variable. Additionally, QBFs have a *quantifier prefix* which fixes the quantifier type of the variables occurring in the CNF. This formula structure is called *prenex conjunctive normal form* (PCNF). Any formula of arbitrary structure can be efficiently encoded in PCNF by introducing auxiliary variables—called Tseitin variables—which avoid an exponential blowup (Tseitin 1983).

However, this tradition of using normal forms limits the information available to the QBF solvers which might negatively impact the solving process. In contrast to SAT, where satisfiability checking problems have to be solved, a QBF solver has to solve validity checking problems (depending on the variable quantification) as well which is biased by restrictive structure of the CNF.

Therefore, the QCIR format (QBF Gallery 2014) was developed during the last QBF Gallery events. However, tools and benchmarks are not yet widely available—leading to the cycle that tools and applications do not support QCIR because it is not yet in common use. In this paper we present application benchmarks and synthetic benchmarks in QCIR, as well as tools that utilize QCIR. We hope that this serves to establish QCIR as the standard format for non-CNF QBF.

This paper is structured as follows. First, we introduce the QCIR format in detail, compare it to older formats, and give some motivation for why it is important to have a non-CNF format for QBF. Next we present a set of benchmarks stemming from reduction finding tasks. Then we introduce a fuzzer for the QCIR format which randomly generates QCIR formulas. In the context of CNF solving, such formulas are successfully used in the solver development process, and this practice be easily lifted to the non-CNF case. We consider some first experiments with our newly generated benchmarks before concluding with an outlook.

The QCIR Format

Currently, the standard input format for QBF is QDIMACS[1], which has been used since the very earliest QBF

---

[1] http://www.qbflib.org/qdimacs.html

property	QDIMACS	boole	qpro	QCIR
prefix operators	–		✓	✓
non-CNF		✓	✓	✓
non-NNF		✓		✓
non-PNF		✓	✓	✓
structure sharing				✓
xor				✓
ite				✓
resource alloc.	✓		~	
textual var. names		✓		✓

Table 1: Comparison of QBF input formats

competitions organized a decade ago (e.g., (Narizzano, Pulina, and Tacchella 2006)) and is still the input format of the main tracks of the recent competitions. It is supported by most state-of-the-art QBF solvers and therefore most applications which aim at using a QBF solver generate formulas in QDIMACS format.

In order to establish non-CNF formats, several efforts have been taken. One of the first attempts was the boole[2] format. It offers an infix representation which is very natural to the human user, but imposes some challenges with the processing. To overcome some of the issues the qpro[3] format was introduced which restricted itself to negation normal form (NNF) but allowed arbitrary positions of quantifiers within the formula tree. Neither of these formats, however, supports structure sharing and special gates like *if-then-else* (ite) or xor. For an overview see Table 1.

To overcome the drawbacks of the available formats, the QCIR (Quantified CIRcuit) format was introduced as a joint community activity in the context of the QBF Gallery 2013. It was then refined and proposed as input format for the structural track of the QBF Gallery 2014, but unfortunately this track was canceled as not enough participating solvers were submitted to organize an interesting competition.

The format supports different variants of QBFs. Basically, gates are defined which are then used in the definitions of other gates. Various kinds of gates like and, or, ite, and xor gates are supported. Furthermore, there are special quantifier gates which allows to position quantifiers at any position in the formula. The definition of the gates also enables structure sharing, because a label of a gate may occur in arbitrarily many other definitions of gates. The detailed grammar of QCIR formulas is shown in Figure 1. The first line of a QCIR file is the format identifier ("#QCIR-14"), and any other line that starts with "#" is a comment.

As non-CNF formulas in prenex format suffice for many applications, the grammar in Figure 1 includes a convenient syntax (*qblock-prefix*) for specifying a quantifier prefix without the explicit introduction of quantifier gates. Figure 2 shows the general form of a formula in prenex form. An example of a prenex formula is shown in Figure 3.

The QCIR format requires that gate variables be defined before they are used in the definition of another gate. E.g., if

$$
\begin{aligned}
\textit{qcir-file} &::= \textit{format-id qblock-prefix output-stmt} \\
&\quad (\textit{gate-stmt nl})^* \\
\textit{format-id} &::= \texttt{\#QCIR-14}\ \textit{nl} \\
\textit{qblock-prefix} &::= (\texttt{free}\,(\textit{var-list})\,\textit{nl})?\ \textit{qblock-quant}^* \\
\textit{qblock-quant} &::= \textit{quant}\,(\textit{var-list})\ \textit{nl} \\
\textit{var-list} &::= (\textit{var},\,)^*\ \textit{var} \\
\textit{lit-list} &::= (\textit{lit},\,)^*\ \textit{lit}\ |\ \epsilon \\
\textit{output-stmt} &::= \texttt{output}\,(\textit{lit})\ \textit{nl} \\
\textit{gate-stmt} &::= \textit{var} = \texttt{and}\,(\textit{lit-list}) \\
&\quad |\quad \textit{var} = \texttt{or}\,(\textit{lit-list}) \\
&\quad |\quad \textit{var} = \texttt{xor}\,(\textit{lit},\ \textit{lit}) \\
&\quad |\quad \textit{var} = \texttt{ite}\,(\textit{lit},\ \textit{lit},\ \textit{lit}) \\
&\quad |\quad \textit{var} = \textit{quant}\,(\textit{var-list};\ \textit{lit}) \\
\textit{quant} &::= \texttt{exists}\ |\ \texttt{forall} \\
\textit{var} &::= (\text{A string of ASCII letters, digits,} \\
&\quad \text{and underscores}) \\
\textit{lit} &::= \textit{var}\ |\ \texttt{-}\textit{var} \\
\textit{nl} &::= \text{newline}
\end{aligned}
$$

Figure 1: Grammar of QCIR input format

```
#QCIR-14
quant(var, ..., var)
:
quant(var, ..., var)
output(lit)
var = gate_exp
:
var = gate_exp
```

Figure 2: Structure of QCIR formula in prenex form

```
#QCIR-14
forall(v1)
exists(v2, v3)
output(g3)
g1 = and(v1, v2)
g2 = and(-v1, -v2, v3)
g3 = or(g1, g2)
```

$$\forall v_1.\exists v_2.\exists v_3.\ \underbrace{\underbrace{(v_1 \wedge v_2)}_{g_1} \vee \underbrace{(\neg v_1 \wedge \neg v_2 \wedge v_3)}_{g_2}}_{g_3}$$

Figure 3: Example of QCIR formula in prenex form

$$\llbracket x \rrbracket = \begin{cases} x & \text{if } x \text{ is a quantified or free variable} \\ \neg \llbracket z \rrbracket & \text{if } x \text{ is } -z \\ \llbracket z_1 \rrbracket \wedge \ldots \wedge \llbracket z_n \rrbracket & \text{if } x \text{ is defined as "} x \text{ = and}(z_1, \ldots, z_n)\text{"} \\ \llbracket z_1 \rrbracket \vee \ldots \vee \llbracket z_n \rrbracket & \text{if } x \text{ is defined as "} x \text{ = or}(z_1, \ldots, z_n)\text{"} \\ (\llbracket z_1 \rrbracket \wedge \neg \llbracket z_2 \rrbracket) \vee (\neg \llbracket z_1 \rrbracket \wedge \llbracket z_2 \rrbracket) & \text{if } x \text{ is defined as "} x \text{ = xor}(z_1, z_2)\text{"} \\ (\llbracket z_1 \rrbracket \wedge \llbracket z_2 \rrbracket) \vee (\neg \llbracket z_1 \rrbracket \wedge \llbracket z_3 \rrbracket) & \text{if } x \text{ is defined as "} x \text{ = ite}(z_1, z_2, z_3)\text{"} \\ \exists z_1 \ldots \exists z_n. \llbracket g \rrbracket & \text{if } x \text{ is defined as "} x \text{ = exists}(z_1, \ldots, z_n; g)\text{"} \\ \forall z_1 \ldots \forall z_n. \llbracket g \rrbracket & \text{if } x \text{ is defined as "} x \text{ = forall}(z_1, \ldots, z_n; g)\text{"} \end{cases}$$

Figure 5: Semantics of QCIR format

```
#QCIR-14
output(g5)
g1 = xor(x, z)
g2 = exists(x; g1)
g3 = xor(z, g2)
g4 = and(g2, g3)
g5 = forall(z; g4)
```
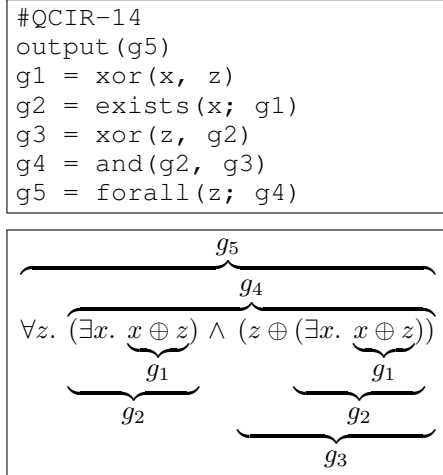

Figure 4: Example of a non-prenex QCIR formula

the gate definitions include "g1 = and(v1, v2)" and "g2 = or(g1, v3)", then the definition of g1 must come before the definition of g2. Note that this requirement ensures that the circuit graph is acyclic. A tool that accepts QCIR should abort with an error if a gate is used before being defined or if a gate is defined more than once.

It may be noted that QCIR allows and and or gates with zero arguments; as expected, these correspond to the truth values true and false, respectively.

The semantics of a QCIR formula is relatively straightforward. For a gate literal x, let $\llbracket x \rrbracket$ denote the formula that x represents. (If x is a non-gate variable, then $\llbracket x \rrbracket$ is just x itself.) Figure 5 gives the semantics for gate definitions. A QCIR file as a whole represents the formula represented by the literal specified on the output line, prefixed with the quantifier prefix specified by *qblock-prefix*.

An example of a formula in non-prenex form is shown in Figure 4. This formula has two features which, although not forbidden by the QCIR grammar, may be difficult for solvers to handle: (1) a variable (here x) is quantified more than once, and (2) a quantified gate occurs negatively in the formula (i.e., occurs inside a negation, inside an xor gate, or inside the first argument of an *if-then-else* gate). It is expected that some non-prenex solvers may declare that they

do not handle formulas with either of these features. Therefore, the more restricted *cleansed* variant of the QCIR format has been introduced.

In addition to *closed* formulas (in which all variables are bound by quantifiers), QCIR also supports *open* formulas which contain free variables. QCIR requires that all free variables be declared on the free line of the quantifier prefix. A solver supporting open formulas returns a propositional formula that is logically equivalent to the input QBF. The QCIR format can be used to encode this output propositional formula.

At the moment, there are two solvers natively supporting the QCIR format: the solver GhostQ[4] and the non-CNF variant of the CEGAR-based solver RAReQS[5].

Benefit of Non-CNF Representation

CNF is the standard for SAT solvers, and it works rather well in that domain. However, for QBF solvers, converting from a circuit representation to CNF can harm the performance of QBF solvers. The reason for this is that the Tseitin transformation for converting to CNF introduces only existentially quantified variables (not universally quantified variables) for the gates of the circuit. These existential Tseitin variables are effective in efficiently pruning the search space when searching for a satisfying assignment, but they provide less help when searching for a falsifying assignment. In other words, the Tseitin transformation makes it difficult for a solver to discover when no falsifying assignments exists. A SAT solver is untroubled by this, because it never cares whether any falsifying assignments exist. As an example of how CNF is harmful in QBF, consider the prenex formula

$$\forall X. \exists y. \, y \vee \underbrace{\psi(X)}_{g_1} \quad (1)$$

This formula is trivially true. Assigning y to be true immediately makes the matrix of the formula true, regardless of ψ. Under the Tseitin transformation, Equation 1 becomes:

$$\forall X. \exists y. \exists \{g_1, ..., g_n\}. \, (y \vee g_1) \wedge (\text{clauses defining gate vars})$$

---

[4] https://www.cs.cmu.edu/~wklieber/ghostq/
[5] http://sat.inesc-id.pt/~mikolas/sw/rareqs-nn/

Setting y to be true no longer immediately makes the matrix true. Instead, an assignment needs to include the gate variables and the universal variables X in order to satisfy the matrix. Experimental results (Ansótegui, Gomes, and Selman 2005; Zhang 2006) indicate that purely CNF-based QDPLL QBF solvers would, in the worst case, require time exponential in the number of variables in X to solve the CNF formula, even though the original problem (before translation to CNF) is trivial.

With a circuit representation, QBF solvers can introduce two variables for each gate, one existential and one universal. The existential variables are used in the normal Tseitin tranformation to CNF, whereas the universal variables are used in the dual transformation to disjunctive normal form (DNF). This dual representation enables the solver to immediately detect that no falsifying assignments exist for the above formula. Experimental results indicate that this dual representation can greatly improve a QBF solver's performance on real-world problems (Zhang 2006; Goultiaeva and Bacchus 2010; Klieber et al. 2010).

Application: Reduction-Finding

Recent QBF Gallery events included a class of application benchmark formulas that encode the problem of searching for certain complexity-theoretic reductions. These benchmark formulas are now available in QCIR format, and can be produced using the generator described in (Jordan and Kaiser 2013a) with the new option -qcir to produce QCIR formulas without CNF conversion.

First we give a short introduction of the problem. See (Jordan and Kaiser 2013b) for details, definitions and comparisons of various approaches to this question.

Given decision problems P and Q, a reduction from P to Q is a (relatively easy-to-compute) function r satisfying

$$\exists r \forall x : x \in P \iff r(x) \in Q. \qquad (2)$$

Of course, the general problem of determining whether a reduction exists between two problems is undecidable – as are most related questions. However, we can restrict attention to a limited (but sufficiently interesting) class of reductions and relax (2) to hold only for structures x of size at most a given finite n. This results in a problem that is (in a sense) in Σ_2^p.

The formulas we use encode the problem of searching for simple quantifier-free reductions between the 2304 pairs of 48 different decision problems contained in NL. Some of the resulting problems are trivial and useful only for basic testing, but some seem quite hard – e.g., searching for simpler reductions for the Immerman-Szelepcsényi Theorem.

We provide three sets of problems[6]:

red_1133 2304 formulas with very simple parameters ($k = 1, c = 1, n = 3$),

red_1344 QCIR encodings of the benchmarks from the QBF Gallery ($k = 1, c = 3, n = 4$),

---

[6]See (Jordan and Kaiser 2013b) for the meaning of parameters. Formulas available in QCIR and QDIMACS at
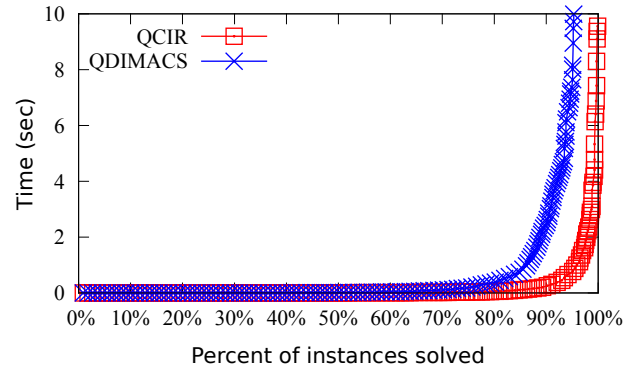https://www-alg.ist.hokudai.ac.jp/~skip/qcir

Figure 6: Cactus plot for GhostQ on the red_1133 family

red_hard a small set of more challenging formulas.

The formulas are produced using (2). First, there are leading existential quantifiers corresponding to the possible contents of a parameterized reduction. Then, the universal variables correspond to the appropriate structure (e.g., graphs) on which the decision problem P is defined. The leading existential variables occur only on the right-side of the iff – they are used only to determine whether $r(x) \in Q$. Depending on P, Q and n, additional variables may be used to encode transitive closures. The structure of the inner formula depends on the formulas defining P and Q.

As a concrete example, the QCIR formula nreachq_reachqu_1133.qcir corresponds to searching for an extremely simple reduction for the Immerman-Szelepcsényi Theorem but restricts attention to only graphs of size 3. In QCIR this has 75 quantified variables (including encoded transitive closures), while CNF conversion introduces additional variables – for a total of 1759 in the QDIMACS encoding of this problem.

We compared the new QCIR encoding of red_1133 to the existing QDIMACS encoding of these problems, running GhostQ on both encodings with a timeout of 10 seconds. As discussed in detail later in this paper, GhostQ has a preprocessor that attempts to reverse-engineer QDIMACS back into circuit form. However, this capability is limited, and it turned out to be largely ineffective for the QDIMACS version of the red_1133 benchmarks. On the QCIR encodings, GhostQ timed out on only 3 instances, whereas it timed out on 108 instances for the QDIMACS encodings. If the timeout is lowered to 1 second, GhostQ times out on 96 QCIR instances and 316 QDIMACS instances. Figure 6 show the relation between time limit and number of instances solved.

Randomly Generated Benchmarks

Randomly generated formulas play an important role in the development process of CNF-based SAT and QBF solvers (Brummayer, Lonsing, and Biere 2010). Such formulas are the basis for fuzz testing and model-based testing in order to automatically detect various kinds of defects in solvers. Especially corner cases which are easily overlooked
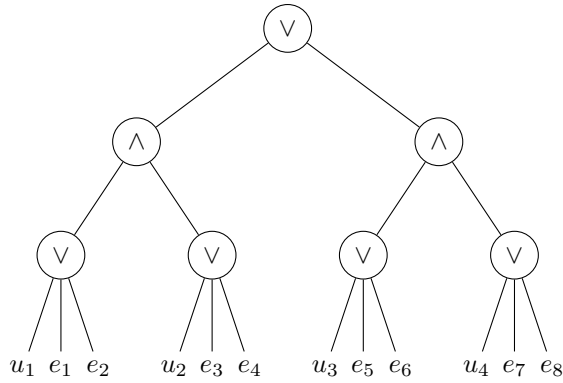
Figure 7: Subformula structure

by manually established tests are easily found by random tests. In contrast to application benchmarks, which are often very hard to solve and require therefore a lot of resources for the solver to terminate, random formulas can be generated according to random models which provide well understood control mechanisms to produce formulas of a reasonable difficulty, size, and structure. Especially for formulas in CNF, the random models are well understood and characterized and depend only on a few parameters like number of variables, number of clauses, ratio between clauses and variables, negation probability of literals, etc. With giving out the structural restrictions of a CNF, the number of parameters to be considered for generating formulas increases because many different formulas structures are then possible.

For building non-PCNF solvers the testing infrastructure available for CNF solvers have to be adopted. Therefore, random models are required for generating formulas in the novel input formats. We have proposed a framework for generating such formulas and showed how to instantiate it for formulas in the qpro format (Creignou, Egly, and Seidl 2012). In this work, the *fixed shape model* was considered. We implemented now a generator[7] producing QCIR instances. In particular, we consider formulas of the structure $\forall X \exists Y \phi$, where $|X| = n$, $|Y| = m$, and ϕ is a conjunction of $L = c * n$ subformulas as depicted in Figure 7. We conducted experiments with $m = 1000$ and $n \in \{40, 45, 50\}$ and $2 \leq c \leq 4$. The results are shown in Figure 8 where we used the QCIR supporting solvers RAReQS and GhostQ to solve the generated formulas.

Note that these models were only a slight generalization of the CNF random models and their properties are therefore quite well understood. Extending them with structure sharing and other operators like xor and *if-then-else* is subject to future work.

Outlook

We conclude this paper with two topics which are subject to ongoing research: (i) reverse engineering structure if only a CNF is given, and (ii) certification.

---

[7] available at http://fmv.jku.at/qcir

Reverse Engineering CNF

At the present time, most of the benchmarks available for QBF are in the QDIMACS format. In order to make the QCIR format more attractive to solver developers, we have developed a reverse-engineering tool QCIR-CONV that take a QDIMACS file and converts it to the QCIR format. For QDIMACS files that were converted to CNF from a circuit, this tool attempts to reconstruct the circuit. QCIR-CONV is based on the preprocessor originally developed as part of GhostQ. Recent related work for extracting structure from a CNF representation includes (Goultiaeva and Bacchus 2013). QCIR-CONV is available at:

http://www.cs.cmu.edu/%7ewklieber/qcir-conv.py

Let us discuss how QCIR-CONV works. At a high level, QCIR-CONV looks for patterns in the QDIMACS file that reveal gate definitions. For example, the following clauses may correspond to the gate definition $g = x_1 \vee ... \vee x_n$:

$$(\neg g \vee x_1 \vee ... \vee x_n)$$
$$(g \vee \neg x_1)$$
$$\vdots$$
$$(g \vee \neg x_n)$$

In order for this be a valid gate definition, g must not be upstream (in the quantifier prefix) of any of the literals $x_1, ..., x_n$. Additionally, if any of the literals $x_1, ..., x_n$ (or their negations) had previously been committed as defining a gate, we must check that g does not appear in the subformula represented by such literals, to avoid constructing a circuit with a cycle. Furthermore, for a single literal g, there might be multiple sets of clauses that could be used to define g as a gate. In this case, QCIR-CONV arbitrarily picks one of them (except that it gives lower priority to single-input gates, i.e., equivalences such as $g = x$).

QCIR-CONV also looks for *if-then-else* gates. Consider a gate $g = \texttt{ite}(s, x, y)$, where s is the 'selector' of the ite gate. Such a gate can be encoded in CNF as follows:

Implication	Clause
$(s \wedge x) \Rightarrow g$	$(\neg s \vee \neg x \vee g)$
$(s \wedge \neg x) \Rightarrow \neg g$	$(\neg s \vee x \vee \neg g)$
$(\neg s \wedge y) \Rightarrow g$	$(s \vee \neg y \vee g)$
$(\neg s \wedge \neg y) \Rightarrow \neg g$	$(s \vee y \vee \neg g)$

QCIR-CONV looks for four consecutive clauses of the above form to detect possible ite gates. (Within the group of four clauses, the clauses may appear in any order with respect to each other, and the literals in each clause may also appear in any order.) XOR gates are also found by the above process for finding ITE gates: $\texttt{xor}(x, y) = \texttt{ite}(x, \neg y, y)$.

If the output of a gate occurs in only one polarity in the original formula (for example, if the gate doesn't occur within the scope of a negation or similar gates such as XOR), then the Plaisted-Greenbaum transformation (Plaisted and Greenbaum 1986) can be used instead of the Tseitin transformation for converting a formula to CNF. QCIR-CONV currently does not attempt to reverse the Plaisted-Greenbaum transformation.
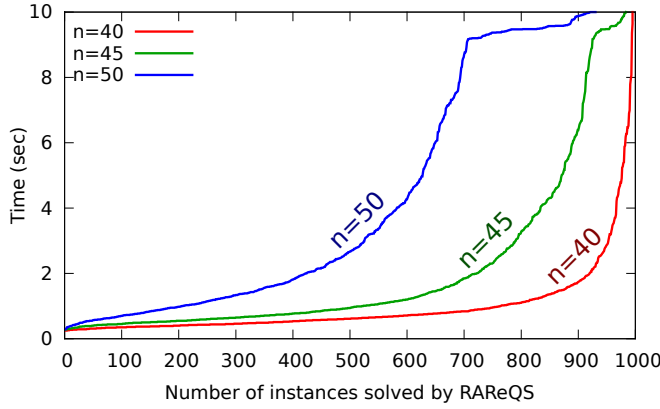
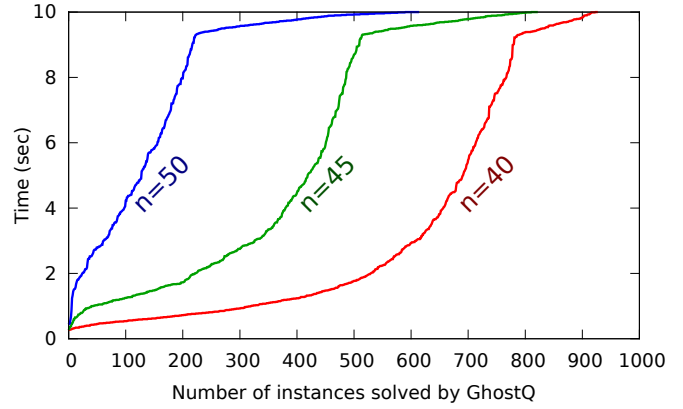**Erratum: Data above t = 9 sec is unreliable.**

Figure 8: Runtimes of RAReQS and GhostQ with $m = 1000$ and $n \in \{40, 45, 50\}$

Companions. Suppose that an existential variable g is always 'accompanied' by a set of literals C — i.e., every clause that contains the variable g (in either its positive or negative form) also contains C. (This pattern was observed in several QBFLIB benchmarks.) For example, consider a formula Φ where g occurs in exactly the following clauses:

$$(\neg s \vee \neg g \vee a \vee b), (\neg s \vee g \vee \neg a), (\neg s \vee g \vee \neg b),$$
$$(\neg s \vee \neg g \vee z), (\neg s \vee g \vee \neg z)$$

In this example, $C = \{\neg s\}$. The first three clauses, if $\neg s$ is removed from them, conform to the pattern of a gate definition $g = a \vee b$. Further note that the truth value of Φ is unchanged if $\neg s$ is removed from these three clauses. It is conjectured that these superfluous literals were included in benchmarks because they helped early CNF QBF solvers to detect satisfying assignments. QCIR-CONV identifies whether a potential gate variable is always 'accompanied' by other literals, and if so, it accounts for them when trying to identify clauses that would constitute a gate definition.

Strategy Extraction

Some QBF solvers (e.g., (Balabanov and Jiang 2012; Goultiaeva, Gelder, and Bacchus 2011; Jussila et al. 2007; Niemetz et al. 2012)) are able to provide a *strategy* for the winning player, i.e., Skolem functions for the existential variables if the formula is true, or Herbrand functions for the universal variables if the formula is false. (Balabanov et al. 2015) developed such a strategy-extraction algorithm for a long-distance resolution QBF solver. The strategy maps each variable of the winning player to a formula in terms of the upstream variables of the opposing player. A slightly-modified version of the QCIR format is able to represent such strategies. In the QCIR grammar, for the definition of *output-stmt*, we replace output (*lit*) with output (*varlist*). All the variables belonging to the winning player would be listed in this output statement and defined by appropriate *gate-stmt*s. Note that this allows subformulas to be defined once and shared over the strategies for multiple variables.

Acknowledgements

References

Ansótegui, C.; Gomes, C. P.; and Selman, B. 2005. The Achilles' Heel of QBF. In *Proc. of the 20th National Conference on Artificial Intelligence and the 17th Innovative Applications of Artificial Intelligence (AAAI/IAAI 2005)*, 275–281. AAAI Press / The MIT Press.

Balabanov, V., and Jiang, J. R. 2012. Unified QBF certification and its applications. *Formal Methods in System Design* 41(1):45–65.

Balabanov, V.; Jiang, J. R.; Janota, M.; and Widl, M. 2015. Efficient extraction of QBF (counter)models from long-distance resolution proofs. In *Proc. of the 29th Conference on Artificial Intelligence (AAAI 2015)*, 3694–3701. AAAI Press.

Benedetti, M., and Mangassarian, H. 2008. QBF-based formal verification: Experience and perspectives. *Journal on Satisfiability, Boolean Modeling and Computation* 5(1-4):133–191.

Biere, A.; Heule, M.; van Maaren, H.; and Walsh, T., eds. 2009. *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*. IOS Press.

Brummayer, R.; Lonsing, F.; and Biere, A. 2010. Automated testing and debugging of SAT and QBF solvers. In *Proc. of the 13th International Conference on Theory and Applications of Satisfiability Testing (SAT 2010)*, volume 6175 of *Lecture Notes in Computer Science*, 44–57. Springer.

Creignou, N.; Egly, U.; and Seidl, M. 2012. A framework for the specification of random SAT and QSAT formulas. In *Proc. of the 6th International Conference on Tests and Proofs (TAP 2012)*, volume 7305 of *Lecture Notes in Computer Science*, 163–168. Springer.

Goultiaeva, A., and Bacchus, F. 2010. Exploiting QBF duality on a circuit representation. In *Proc. of the 24th Conference on Artificial Intelligence (AAAI 2010)*. AAAI Press.

Goultiaeva, A., and Bacchus, F. 2013. Recovering and utilizing partial duality in QBF. In *Proc. of the 16th International Conference on Theory and Applications of Satisfiability Testing (SAT 2013)*, volume 7962 of *Lecture Notes in Computer Science*, 83–99.

Goultiaeva, A.; Gelder, A. V.; and Bacchus, F. 2011. A uniform approach for generating proofs and strategies for both true and false QBF formulas. In *Proc. of the 22nd International Joint Conference on Artificial Intelligence (IJCAI 2011)*, 546–553. IJCAI/AAAI.

Jordan, C., and Kaiser, Ł. 2013a. Benchmarks from reduction finding. In *QBF Workshop*. Available at `http://fmv.jku.at/qbf2013/`.

Jordan, C., and Kaiser, Ł. 2013b. Experiments with reduction finding. In *Proc. of the 16th Int. Conference on Theory and Applications of Satisfiability Testing (SAT 2013)*, volume 7962 of *Lecture Notes in Computer Science*, 192–207. Springer.

Jussila, T.; Biere, A.; Sinz, C.; Kröning, D.; and Wintersteiger, C. M. 2007. A first step towards a unified proof checker for QBF. In *Proc. of the 10th International Conference on Theory and Applications of Satisfiability Testing (SAT 2007)*, volume 4501 of *Lecture Notes in Computer Science*. Springer. 201–214.

Kleine Büning, H., and Bubeck, U. 2009. Theory of quantified Boolean formulas. In *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*. IOS Press. 735–760.

Klieber, W.; Sapra, S.; Gao, S.; and Clarke, E. M. 2010. A non-prenex, non-clausal QBF solver with game-state learning. In *Proc. of the 13th International Conference on Theory and Applications of Satisfiability Testing (SAT 2010)*, volume 6175 of *Lecture Notes in Computer Science*. Springer.

Lonsing, F.; Seidl, M.; and Van Gelder, A. 2015. The QBF Gallery: Behind the scenes. arXiv:1508.01045.

Narizzano, M.; Pulina, L.; and Tacchella, A. 2006. Report of the third QBF solvers evaluation. *Journal on Satisfiability, Boolean Modeling and Computation* 2(1-4):145–164.

Niemetz, A.; Preiner, M.; Lonsing, F.; Seidl, M.; and Biere, A. 2012. Resolution-based certificate extraction for QBF - (tool presentation). In *Proc. of the 15th International Conference on Theory and Applications of Satisfiability Testing (SAT 2012)*, volume 7317 of *Lecture Notes in Computer Science*, 430–435. Springer.

Plaisted, D. A., and Greenbaum, S. 1986. A structure-preserving clause form translation. *Journal of Symbolic Computation* 2(3):293–304.

QBF Gallery 2014. QCIR-G14: A non-prenex non-CNF format for quantified Boolean formulas. Available at `http://qbf.satisfiability.org/gallery/qcir-gallery14.pdf`.

Tseitin, G. 1983. On the complexity of derivation in propositional calculus. In *Automation of Reasoning*, Symbolic Computation. Springer. 466–483.

Zhang, L. 2006. Solving QBF by Combining Conjunctive and Disjunctive Normal Forms. In *Proc. of the 21st National Conference on Artificial Intelligence and the 18th Innovative Applications of Artificial Intelligence (AAAI/IAAI 2006)*. AAAI Press.