A Formal Language for QBF Family Definitions

Noel Arteche Echeverría

University of the Basque Country / KU Leuven
`noel.artecheecheverria@student.kuleuven.be`

Abstract. In this work we propose a formal language to write definitions of classes of Quantified Boolean Formulae (QBF) in terms of —potentially— any type of parameters. A class of formulae provides an encoding in logic terms of some computational problem belonging to PSPACE. These definitions of families of formulae usually depend on some parameters determining the size, structure, alternation patterns of quantifiers and complexity of the described formulae. These parameters and their relation to the structure of formulae can be easily encoded in this language. Additionally, we present a computer tool capable of parsing these definitions and outputting the formulae in either the QCIR or QDIMACS formats to be fed to a QBF-solver. This aims to be both a framework and a tool for future empirical research in these topics.

Keywords: QBF · Proof Complexity · QCIR · QDIMACS

1 Introduction

QBF-solvers are tools built on top of formal proof systems, extended with some additional heuristics and preprocessing techniques. Consequently, studying the proof complexity of these systems provides insights into what are their strengths and weaknesses and, by extension, those of the QBF-solvers implemented on top. The theoretical research in this field often proceeds by defining classes of formulae or *formula families* to then show proof-theoretic lower bounds on them.

In practice, the definition of a formula family declares the structure of a particular set QBF formulae. One can think of a formula family definition as the encoding of any PSPACE problem into QBF.

To check whether the theoretical results on proof systems apply to the QBF-solvers built on top, a proper empirical framework has been missing. In this extended abstract, we present the work in progress for a formal language to define formula families and a computer tool capable of reading them and, given some values, output a file that can be fed to a QBF-solver.

Another desirable property of this language is the support for parameters others than natural numbers. Although most of the existing formula families in the literature depend on a single natural value (as this makes proofs and reasoning on them easier), other problems require other data types as parameters (think, for instance, of decision problems on graphs). The language presented in this paper supports virtually any data structure for parameter types via the use of embedded Python.

2   Formula Families

In this context, a *formula family* or *class of formulae* is a set of Quantified Boolean Formulae (QBF) all presenting the same syntactic patterns and semantics. In practice, one interacts with a formula family through its *definition*. One can think of the definition of a formula family as the encoding of any PSPACE problem into QBF.

The formula families used in the proof complexity literature are usually rather artificial. Such a case is that of the QPARITY formulae, first introduced in [2] and later used in [1] to show an exponential separation between *extended Q-Resolution* and *weak extended Q-Resolution*. These formulae have a single parameter, a natural number n, which determines both their size and structure. We present the more succinct version of circuits for this family.

Definition 1 (QParity circuits [1]). *Let $n \in \mathbb{N}^*$ and let $x_1, \ldots, x_n$ and z be Boolean variables. We define the quantifier prefix $P_n = \exists x_1 \ldots \exists x_n \forall z$. We define an auxiliary circuit t_2 as $t_2 = x_1 \oplus x_2$ and for $i \in \{3, \ldots, n\}$ we define auxiliary t-circuits as $t_i = t_{i-1} \oplus x_i$ and the complete matrix as $\rho_n = (z \vee t_n) \wedge (\neg z \vee \neg t_n)$. The QBF instance will be $\mathrm{QPARITY}_n = P_n : \rho_n$.*

The formulae express that there exists an assignment to the x-variables such that $x_1 \oplus \cdots \oplus x_n$ is neither 0 nor 1, an obvious contradiction.

For an extra, more natural example, of formula families and their translation into our language, the appendix contains the case study of the *Chromatic Formulae*, a QBF encoding of the Chromatic Number Problem.

3   The Formal Language

We now give a formalism in which to write these definitions. This formal language relies on the basic concept of *blocks*. A block is a sequence of literals (input variables that may be negated) or references to other blocks (which might in turn also be negated). A block can then be assigned a single *attribute*, which is either a *quantifier* (universal or existential) or a *logical operator* (conjunction, disjunction or exclusive disjunction).

Example 1 (Basic use of blocks). The formula $\varphi(x, y, z) = (x \vee y) \wedge z$ can be defined in our language using two blocks:

```
define block B1 := x, y;
define block B2 := B1, z;
```

Inasmuch blocks are just sequences of literals or blocks, they only declare ordering of variables. The meaning of these blocks is later given through an attribute (in this case, the *and* and *or* operators). These operate between them all the *bricks* present in the block. Using our formal syntax:

```
block B1 operated with OR;
block B2 operated with AND;
```

Now the formula φ is represented by referring to the block `B2`.

The structure of blocks captures simultaneously both the idea of gates on a Boolean circuit as well as the intricate nested patters of quantifier prefixes. Imagine that the previous formula is quantified as follows:

$$\forall x \exists y \exists z : \varphi(x, y, z)$$

We can define some blocks to obtain the structure of the quantifier prefix:

```
define block Q1 := x;
define block Q2 := y, z;
define block Q := Q1, Q2;

block Q1 quantified with A;
block Q2 quantified with E;
```

Finally, we can combine the quantifier prefix block `Q` with the block `B2` representing φ and indicate that this is our *output block* (similar to the idea of *output gates* in Boolean circuits as encoded in the QCIR format).

```
define block Phi := Q, B2;
output block: Phi;
```

To showcase the language in a more realistic scenario, we present the formal version of the QPARITY formulae from the previous section.

Example 2 (Formal version of the QPARITY *formulae).* Firstly, the language asks some general information about the name and format of the family. Then we declare *parameters*, followed by their type as well as possible constraints (in this case, $n \geq 1$). We then declare the variables. Variables x_i are denoted `x(i)` and the range of indices must be specified.

```
name: QParity;
format: circuit-prenex;

parameters: {
    n : int, 'n >= 1';
}

variables: {
    x(i)    where i in 1..n;
    z;
}
```

We now declare the blocks. We have blocks for quantifiers and blocks for gates, but they use the same space and syntax (this will allow to define non-prenex circuits). When quantifying a block, the block is expanded, assigning the quantifier to variables in a depth-first manner.

```
blocks: {
    define blocks {
        X := x(i);
    } where i in 1..n;

    define block Z := z;

    define block Q := X, Z;

    block X quantified with E;
    block Z quantified with A;
```

In the same section, we define the blocks used to build the matrix of the formula. An important feature is that of *groupings*: a set of blocks grouped under the same name, so that the can all be simultaneously operated or quantified.

```
    define blocks grouped in T {
        T(2) := x(1), x(2);
        T(i) := T(s), x(i);
    } where i in 3..n, s = 'i-1';

    define blocks grouped in Ro {
        Ro1 := z, T(n);
        Ro2 := -z, -T(n);
    };

    define block F := Ro1, Ro2;

    all blocks in T operated with XOR;
    all blocks in Ro operated with OR;
    block F operated with AND;

    define block Phi := Q, F;
}

output block: Phi;
```

For an extra example, the Appendix contains the formal version of the Chromatic Formulae presented in Definition 2, which showcases the use of a graph as a parameter of the definition.

4   The Tool

Based on the language described above, a computer tool has been developed in Python to parse definitions and, given values for the parameters, output files containing the defined formulae to be fed to a QBF solver.

The tool translates the parsed definition and values building an internal representation of the formula into a Python object and then outputs it into a valid format. If the input definition declares prenex Conjunctive Normal Form (CNF) formula, the tool can print either a QDIMACS or a QCIR file. If the input is a circuit, it can natively output a QCIR file or convert it to CNF and output a QDIMACS file using William Klieber's conversion tool developed in the context of the GhostQ QBF-solver[1]. Finally, if the formulae are non-prenex, they can be printed in QCIR or they can be prenexed to then output them as the prenex circuits described above.

Complex arithmetic expressions present in the definitions must be written in Python syntax and enclosed in backticks. This is because they are interpreted and executed by Python itself. Embedded Python gives our language an immense expressive power, as virtually any condition or structured object can be written in the definitions using built-in Python data types and functions.

5 Ongoing Work

Currently only a prototype version exists and requires polishing and extensive testing. Besides, some features are not implemented yet, such as prenexing of formulae and a friendlier user interface.

Additionally, some extra features are planned for the tool regarding conversion between output formats. In [1] it was shown that when converting a circuit to CNF, the placement of quantifiers of auxiliary variables can greatly affect solving performance. Currently, the auxiliary tool used to convert from circuits to CNF places all existential quantifiers at the end of the prefix. If possible, we would like to explore modifications of this conversion tool where these auxiliary variables are quantified in potentially more efficient positions.

References

1. Beyersdorf, O., Chew, L., Janota, M.: Extension variables in QBF resolution. In: Workshops at the Thirtieth AAAI Conference on Artificial Intelligence (2016)
2. Beyersdorff, O., Chew, L., Janota, M.: Proof complexity of resolution-based QBF calculi. In: LIPI Symposium on Theoretical Aspects of Computer Science (STACS'15). vol. 30, pp. 76–89. Schloss Dagstuhl-Leibniz International Proceedings in Informatics (2015)
3. Sabharwal, A., Ansotegui, C., Gomes, C.P., Hart, J.W., Selman, B.: QBF modeling: Exploiting player symmetry for simplicity and efficiency. In: International Conference on Theory and Applications of Satisfiability Testing. pp. 382–395. Springer (2006)

---

[1] https://www.wklieber.com/ghostq/qcir-converter.html

Appendix: Chromatic Formulae

For a more natural example on QBF encodings, formula families and their translation to our language, take the *Chromatic Number Problem*, a well-known NP-complete problem: given a graph $G = (V, E)$ and a natural number $k \geq 1$, decide whether k is the *chromatic number* of G, the minimum k such that G is k-colorable. Although this is an NP-complete problem and, as such, can be encoded into a SAT formula, a more natural encoding is also possible using quantification: there *exists* a coloring of G with k colours and *for all* other coloring of size $k-1$, these are not valid colorings for G.

Let us now define a formula family for the problem, depending on two parameters: the graph G and the number k. The following definition for this problem was given by Sabharwal, Ansotegui et al. in [3].

Definition 2 (General Chromatic Formulae [3]). *Let $G = (V, E)$ be a graph and let k be a positive natural number. Let $n = |V|$. We define variables $x_{i,j}$ for $i \in [n]$ and $j \in [k]$ and $y_{i,j}$ for $i \in [n]$ and $j \in [k-1]$ (semantically, any of these variables is set to 1 if and only if node i is set to have colour j). Then we define the following two subformulae:*

$$\Gamma = \bigwedge_{i \in [n]} (x_{i,1} \vee \ldots \vee x_{i,k}) \wedge \bigwedge_{\substack{i \in [n] \\ j \neq j' \in [k]}} (\neg x_{i,j} \vee \neg x_{i,j'}) \wedge \bigwedge_{\substack{(i,i') \in E \\ j \in [k]}} (\neg x_{i,j} \vee \neg x_{i',j})$$

$$\Delta = \bigvee_{i \in [n]} (\neg y_{i,1} \wedge \ldots \wedge \neg y_{i,k-1}) \vee \bigvee_{\substack{i \in [n] \\ j \neq j' \in [k-1]}} (y_{i,j} \wedge y_{i,j'}) \vee \bigvee_{\substack{(i,i') \in E \\ j \in [k-1]}} (y_{i,j} \wedge y_{i',j})$$

Semantically, Γ is true when the x-variables form a legal k-coloring, while Δ is true only when the y-variables do not form a legal $(k-1)$-coloring.

The Chromatic Formula *is*

$$K(G,k) = \exists x_{1,1} \ldots x_{1,k} \ldots x_{n,1} \ldots x_{n,k} \forall y_{1,1} \ldots y_{1,k-1} \ldots y_{n,1} \ldots y_{n,k-1} : \Gamma \wedge \Delta$$

We are now ready to give the formal version of this definion in the syntax of our language.

Example 3 (Formal version of the Chromatic formulae). The main new feature showcased by this example is that we need to check whether a certain edge (i, j) is in the graph, $(i, j) \in_? E$. For this purpose, we encode the graph as and adjacency matrix, `edges`, and then the condition can be written as `edges[i-1][i-1] == 1` (using Python syntax). The formal version is as follows:

```
name: Chromatic formulae;
format: circuit-prenex;
```

```
parameters: {
    n     : int, 'n >= 1';
    edges : list;
    k     : int, 'k >= 1';
}

variables: {
    x(i, j)    where i in 1..n, j in 1..k;
    y(i, j)    where i in 1..n, j in 1..'k-1';
}

blocks: {

    /* === blocks for quantifers === */

    define blocks grouped in X {
        X(i) := x(i, j);
    } where i in 1..n, j in 1..k;

    define blocks grouped in Y {
        Y(i) := y(i, j);
    } where i in 1..n, j in 1..'k-1';

    define block Q := all blocks in X, all blocks in Y;

    all blocks in X quantified with E;
    all blocks in Y quantified with A;

    /* ==== blocks for matrix ==== */

    define blocks grouped in AllColored {
        Colored(i) := x(i, j);
    } where i in 1..n, j in 1..k;

    define blocks grouped in NotColored {
        NotColored(i) := -y(i, j);
    } where i in 1..n, j in 1..'k-1';

    define block Gamma1 := all blocks in AllColored;

    define block Delta1 := all blocks in NotColored;

    define blocks grouped in SubGamma2 {
        SG2(i, j, l) := -x(i, j), -x(i, l);
    } where i in 1..n, j in 1..k, l in 1..k, 'j != l';
```

```
define blocks grouped in SubDelta2 {
    SD2(i, j, l) := y(i, j), y(i, l);
} where i in 1..n, j in 1..`k-1`, l in 1..`k-1`, `j != l`;

define block Gamma2 := all blocks in SubGamma2;

define block Delta2 := all blocks in SubDelta2;

define blocks grouped in SubGamma3 {
    SG3(i, j, l) := -x(i, l), -x(j, l);
} where i in 1..n, j in 1..n, `edges[i-1][j-1] == 1`, l in 1..k;

define blocks grouped in SubDelta3 {
    SD3(i, j, l) := y(i, l), y(j, l);
} where i in 1..n, j in 1..n, `edges[i-1][j-1] == 1`, l in 1..`k-1`;

define block Gamma3 := all blocks in SubGamma3;

define block Delta3 := all blocks in SubDelta3;

define block Gamma := Gamma1, Gamma2, Gamma3;

define block Delta := Delta1, Delta2, Delta3;

define block F := Gamma, Delta;

all blocks in AllColored operated with OR;
all blocks in NotColored operated with AND;

all blocks in SubGamma2 operated with OR;
all blocks in SubDelta2 operated with AND;

all blocks in SubGamma3 operated with OR;
all blocks in SubDelta3 operated with AND;

blocks Gamma1, Gamma2, Gamma3 operated with AND;
block Gamma operated with AND;

blocks Delta1, Delta2, Delta3 operated with OR;
block Delta operated with OR;

block F operated with AND;

/* define the output block */
```

```
    define block Phi := Q, F;
}

output block: Phi;
```