A Formal Language for QBF Family Definitions

Noel Arteche Echeverría[1,2] and Matthias van der Hallen[2]

[1] University of the Basque Country, Faculty of Computer Science
Manuel Lardizabal 1, 20018 Donostia / San Sebastián, Spain
narteche002@ikasle.ehu.eus
[2] KU Leuven, Department of Computer Science
Celestijnenlaan 200A, 3001 Heverlee (Leuven), Belgium
noel.artecheecheverria@student.kuleuven.be
matthias.vanderhallen@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, and 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 QBDef, 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 · Formal language · Proof complexity · PSPACE · QCIR · QDIMACS

1 Introduction

Many QBF solvers can be considered to perform a heuristic search for a proof in some proof system. Consequently, studying the proof complexity of these systems provides insights into their strengths and weaknesses. The theoretical research 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 the formulae contained in that set. One can think of a formula family definition as the encoding of a PSPACE problem into QBF. Problems outside PSPACE might be encoded in QBF too, but (probably) not in polynomial time.

To check whether the theoretical results on proof systems apply to the QBF solvers built on top, a tool to ease the generation of instances from formula families has been missing. In this extended abstract, we present the work in progress for a formal language to define formula families and QBDef, a computer tool capable of reading them and, instantiating concrete parameter values, output a file that can be fed to a QBF solver.

Although most of the existing formula families in the literature depend on a single scalar value, the language presented in this paper supports virtually any data structure for parameter types via the use of embedded Python, e.g. graphs.

Naturally, any programming language could be used for this same purpose, by means of a script that generated instances of specific formula families. However, these family-specific scripts work directly with the formulae written in the final formats. Although this is acceptable if we are interested only in a particular family, the tool presented in this work (QBDef) lets users focus on the formulae, without having to worry about lower-level details and formats, and encourages to playfully come up with inventive hard-to-prove formulae while, at the same, brings QBF modelling tools closer to the non-QBF-expert.

2 Formula Families

In this context, a *formula family* or *class of formulae* is a set of QBF all presenting the same structure and meaning. In particular, we are interested in dealing with a formula family's *definition*.

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 proof systems. These formulae have a single parameter $n \in \mathbb{N}$. We present the more succinct version of circuits for this family, as an example of what a formula family definition looks like.

Definition 1 (QParity circuits [1]). *Let $n \in \mathbb{N}$, $n \geq 2$, 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 $\rho_n = t_n \oplus z$. The* QBF *instance will be* $\text{QParity}_n = P_n : \rho_n$.

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 *bricks*, which are literals (input variables that may be negated) or references to other blocks (also possibly negated). A block can then be assigned a single *attribute*, i.e. a *quantifier* 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;
```

Blocks only declare ordering of bricks. Meaning is later given through an attribute (e.g. the *and* and *or* operators). These operate between them all the bricks in the block.

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

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;             block Q1 quantified with A;
define block Q2 := y, z;          block Q2 quantified with E;
define block Q := Q1, Q2;
```

Finally, we can combine the quantifier prefix block `Q` with the block `B2` representing φ and indicate that this is our *output block*.

```
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, we declare *parameters*, followed by their type as well as possible constraints (in this case, $n \geq 2$). We then declare the variables. Variables x_i are denoted `x(i)` and the range of indices must be specified.

```
parameters: {                     variables: {
    n : int, `n >= 2`;                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 allows to define non-prenex circuits). When quantifying a block, the block is expanded, assigning the quantifier to variables in a depth-first manner.

```
define blocks {                   define block Q := X, Z;
    X := x(i);
} where i in 1..n;                block X quantified with E;
                                  block Z quantified with A;
define block Z := z;
```

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 they 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 block Ro := T(n), z;
all blocks in T operated with XOR;
block Ro operated with XOR;
define block Phi := Q, F;
```

For an extra, less artificial example, the Appendix contains the case study of the *Chromatic Formulae*, a QBF encoding of the Chromatic Number Problem where the use of a graph as a parameter of the definition is showcased.

4 The Tool

Based on the language described above, QBDef, a computer tool to parse definitions and output files to be fed to a QBF solver has been developed in Python.

If the input definition declares a PCNF 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[3]. If the formulae are non-prenex, they can be printed in the specific QCIR format for non-prenex formulae.

Complex arithmetic expressions must be written in Python syntax and enclosed in backticks. This is because they are interpreted and evaluated 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 of QBDef exists and requires polishing and extensive testing[4].

Additionally, ongoing work includes the study of other families that could exploit the features of this language, such as encodings of PSPACE games.

As a long term goal, this work could lead to future empirical research in the performance of QBF solvers by easily translating both existing and new formula families into the language presented here.

Acknowledgements

Thanks to Marc Denecker and Montserrat Hermo for co-promoting the Master's thesis to which this work belongs. Thanks to Florian Lonsing for his useful comments and insights.

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

[4] The current prototype is available at `https://github.com/alephnoell/QBDef`. The final version of the tool and its source code will be made available in coming months through this same channel.

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)

Appendix

Example 3 (Definition and formal version of the Chromatic Number Problem). The *Chromatic Number Problem* is a well-known DP-complete problem: given a graph G and a natural number $k \geq 1$, decide whether k is the *chromatic number* of G, i.e. 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.

We need to define a formula family for the problem, depending on two parameters: the graph G and the number k. The following definition or encoding for this problem was given by Sabharwal, et al. in [**?**].

In what follows we denote by n the number of nodes in the graph $G = (V, E)$. 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.

We now define a subformula Γ that is true whenever the x-variables form a legal k-coloring,

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

and another subformula, Δ, which is true only when the y-variables do not form a legal $(k-1)$-coloring

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

Clearly, k will be the chromatic number of G if there exists an assignment for the x-variables that makes Γ true and for any assignment to the y-variables, Δ

is true. This gives us the full encoding of the Chromatic Number Problem into a QBF, that we call the *Chromatic Formula*, $K(G, k)$:

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

We can now give the formal version of this definition in the syntax of our language. 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][j-1] == 1‘` (using Python syntax). The full code is given below.

Formal version of the Chromatic Formulae

```
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;
```