A Formal Language and Tool for QBF Family Definitions

Noel Arteche

Thesis Defence
KU Leuven

July 1, 2020

Contents

Contents

Context of this thesis

- Erasmus+ exchange student from the University of the Basque Country, Spain

Context of this thesis

- Erasmus+ exchange student from the University of the Basque Country, Spain
- This work is a 12 ECTS thesis for my Bachelor's... in the form of a small Master's thesis at DTAI

Context of this thesis

- Erasmus+ exchange student from the University of the Basque Country, Spain
- This work is a 12 ECTS thesis for my Bachelor's... in the form of a small Master's thesis at DTAI
- Supervised by Marc Denecker and mentored by Matthias van der Hallen, co-supervised back in Spain by Montserrat Hermo

Context of the work

- General context: complexity theory, proof complexity and (empirical) research on QBF solvers

Context of the work

- General context: complexity theory, proof complexity and (empirical) research on QBF solvers
- After the generalized success of SAT solvers, research now looks beyond NP: PSPACE and the TQBF problem

Quantified Boolean Formulas I

A Quantified Boolean formula is a propositional formula where all the propositional variables are quantified ($\forall$ or $\exists$).
Usually, we work with *prenex* QBF:

$$\Phi = Q_1 x_1 \dots Q_n x_n : \varphi(x_1, \dots, x_n)$$

where:

- $Q_1, \dots, Q_n \in \{\forall, \exists\}$
- $x_1, \dots, x_n \in \{0, 1\}$
- $\varphi(x_1, \dots, x_n)$ is a propositional formula, $\varphi : \{0, 1\}^n \to \{0, 1\}$

Quantified Boolean Formulas II

- The True Quantified Boolean Formula (TQBF) problem: *is* Φ *satisfiable?*
- TQBF is PSPACE-complete
- Recall: SAT $\in$ NP-complete $\subseteq$ PH $\subseteq$ PSPACE.

Context of the work

- General context: complexity theory, proof complexity and (empirical) research on QBF solvers
- After the generalized success of SAT solvers, research now looks beyond NP: PSPACE and the TQBF problem

Context of the work

- General context: complexity theory, proof complexity and (empirical) research on QBF solvers
- After the generalized success of SAT solvers, research now looks beyond NP: PSPACE and the TQBF problem
- QBF solvers can be considered to output proofs of (un)satisfiability in given formal proof systems

Context of the work

- General context: complexity theory, proof complexity and (empirical) research on QBF solvers
- After the generalized success of SAT solvers, research now looks beyond NP: PSPACE and the TQBF problem
- QBF solvers can be considered to output proofs of (un)satisfiability in given formal proof systems
- Lower bounds on proof systems and running times on solvers are closely related

What and why

- Empirical side of QBF solving, common practice: see how solving times scale for formulas belonging to the same family

What and why

- Empirical side of QBF solving, common practice: see how solving times scale for formulas belonging to the same family
- What we need: a tool that can read definitions and, given values of the parameters, output files with the instances of the QBF

What and why

- Empirical side of QBF solving, common practice: see how solving times scale for formulas belonging to the same family
- What we need: a tool that can read definitions and, given values of the parameters, output files with the instances of the QBF
- Why? Need for more flexible editor, independent of format, aimed at proof complexity

What and why

- Empirical side of QBF solving, common practice: see how solving times scale for formulas belonging to the same family
- What we need: a tool that can read definitions and, given values of the parameters, output files with the instances of the QBF
- Why? Need for more flexible editor, independent of format, aimed at proof complexity
- Presented solution: QBDef

Contents

Formula Families

Formula families

A *formula family* is just a set of parameterized formulae. We will focus on sets of parameterized QBF.

Formula Families

Formula families

A *formula family* is just a set of parameterized formulae. We will focus on sets of parameterized QBF.

For example (in the context of SAT),

$$\mathcal{I}(G, k) : \text{graph } G \text{ has an independent set of size } k$$

is a parameterised propositional formula.

Formula Families

Formula families

A *formula family* is just a set of parameterized formulae. We will focus on sets of parameterized QBF.

For example (in the context of SAT),

$$\mathcal{I}(G, k) : \text{ graph } G \text{ has an independent set of size } k$$

is a parameterised propositional formula.
The set

$$\{\mathcal{I}(G, k) : G \text{ is a graph and } k \in \mathbb{N}^*\}$$

is the formula family.

The QParity formulae

The QParity formulae

Definition (QParity circuits, [2, 1])

Let $n \in \mathbb{N}$, $n \geq 2$,

The QParity formulae

Definition (QParity circuits, [2, 1])

Let $n \in \mathbb{N}$, $n \geq 2$, and let $x_1, \ldots, x_n$ and z be Boolean variables.

The QParity formulae

Definition (QParity circuits, [2, 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 $P_n = \exists x_1 \ldots \exists x_n \forall z$.

The QPARITY formulae

Definition (QPARITY circuits, [2, 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 QParity formulae

Definition (QParity circuits, [2, 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 $\text{QParity}_n = P_n : \rho_n$.

The QParity formulae

Definition (QParity circuits, [2, 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 $\text{QParity}_n = P_n : \rho_n$.

- QParity_n is obviously false: it states that there exists an assignment for the x-variables such that $x_1 \oplus \cdots \oplus x_n$ is both 0 and 1.

The $\mathrm{QP{\scriptsize ARITY}}$ formulae

Definition ($\mathrm{QP{\scriptsize ARITY}}$ circuits, [2, 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 $\mathrm{QP{\scriptsize ARITY}}_n = P_n : \rho_n$.

- $\mathrm{QP{\scriptsize ARITY}}_n$ is obviously false: it states that there exists an assignment for the x-variables such that $x_1 \oplus \cdots \oplus x_n$ is both 0 and 1.
- This is a *prenex* quantified Boolean *circuit*.

The QParity formulae

Definition (QParity circuits, [2, 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 $\textsc{QParity}_n = P_n : \rho_n$.

- $\textsc{QParity}_n$ is obviously false: it states that there exists an assignment for the x-variables such that $x_1 \oplus \cdots \oplus x_n$ is both 0 and 1.
- This is a *prenex* quantified Boolean *circuit*.
- It has only one parameter: $n \in \mathbb{N}$.

The QPARITY formulae

Definition (QPARITY circuits, [2, 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 $\text{QPARITY}_n = P_n : \rho_n$.

- QPARITY_n is obviously false: it states that there exists an assignment for the x-variables such that $x_1 \oplus \cdots \oplus x_n$ is both 0 and 1.
- This is a *prenex* quantified Boolean *circuit*.
- It has only one parameter: $n \in \mathbb{N}$.
- Used to show an exponential separation between proof system in [1].

The Chromatic formulae

The Chromatic formulae

The CHROMATIC NUMBER problem

Given a graph G and a number $k \in \mathbb{N}$, decide whether G has *chromatic number* k, i.e. k is the smallest number such that G is k-colorable.

The Chromatic formulae

The CHROMATIC NUMBER problem

Given a graph G and a number $k \in \mathbb{N}$, decide whether G has *chromatic number* k, i.e. k is the smallest number such that G is k-colorable.

- The CHROMATIC NUMBER problem is DP-complete: it is the intersection of an NP-complete and a coNP-complete problem.

The Chromatic formulae

The Chromatic Number problem

Given a graph G and a number $k \in \mathbb{N}$, decide whether G has *chromatic number* k, i.e. k is the smallest number such that G is k-colorable.

- The Chromatic Number problem is DP-complete: it is the intersection of an NP-complete and a coNP-complete problem.
- We can encode it in QBF (see [3])

Other formula families

1. Chen Formulae of Type 1
2. Chen Formulae of Type 2
3. QParity Formulae
4. Chromatic Formulae
5. Janota Formulae
6. KBKF Formulae
7. Geography Formulae

Contents

The Formal Language

We want a formal language in which we can encode definitions like the ones before.

The Formal Language

We want a formal language in which we can encode definitions like the ones before. We need:

1. A powerful language feature that constrains the definition structure but is declarative in nature and expressively powerful enough.

The Formal Language

We want a formal language in which we can encode definitions like the ones before. We need:

1. A powerful language feature that constrains the definition structure but is declarative in nature and expressively powerful enough.
2. Support for non-scalar parameters e.g. graphs

The Formal Language

We want a formal language in which we can encode definitions like the ones before. We need:

1. A powerful language feature that constrains the definition structure but is declarative in nature and expressively powerful enough.
2. Support for non-scalar parameters e.g. graphs
3. Different formats: prenex and non-prenex, CNF, circuits

The *block* structure

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).

The *block* structure: example

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

`block B1 quantified with E;` $\longrightarrow B1 = \exists x \exists y$

`block B1 quantified with A;` $\longrightarrow B1 = \forall x \forall y$

The *block* structure: example

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

`block B1 quantified with E;` $\longrightarrow B1 = \exists x \exists y$

`block B1 quantified with A;` $\longrightarrow B1 = \forall x \forall y$

`block B1 operated with XOR;` $\longrightarrow B1 = x \oplus y$

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

`block B2 operated with OR;` $\longrightarrow B2 = x \vee \neg y \vee (x \oplus y)$

`block B2 operated with AND;` $\longrightarrow B2 = x \wedge \neg y \wedge (x \oplus y)$

Encoding the QParity formulae I

Encoding the QParity formulae II

Let $n \in \mathbb{N}$, $n \geq 2$, and let $x_1, \ldots, x_n$ and z be Boolean variables...

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

parameters: {
    n : int, `n >= 2`;
}

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

Encoding the QParity formulae III

We define the quantifier prefix $P_n = \exists x_1 \ldots \exists x_n \forall z \ldots$

```
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;
```

Encoding the QPARITY formulae IV

We define $t_2 = x_1 \oplus x_2$ *and for* $i \in \{3, \ldots, n\}$ *we define* $t_i = t_{i-1} \oplus x_i$ *and the complete matrix as* $\rho_n = t_n \oplus z$*.*

```
define block T(2) := x(1), x(2);
define blocks grouped in T {
    T(i) := T(s), x(i);
} where i in 3..n, s = `i-1`;

define block Rho := T(n), z;

block T(2) operated with XOR;
all blocks in T operated with XOR;
block Rho operated with XOR;
```

Encoding the $\textsc{QParity}$ formulae V

The QBF instance is $\textsc{QParity}_n = P_n : \rho_n$.

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

    output block: Phi;
```

Embedded Python features

We want parameters with non-scalar data-types and operations between them, e.g. the graph in the Chromatic Formulae.

Embedded Python features

We want parameters with non-scalar data-types and operations between them, e.g. the graph in the Chromatic Formulae. We allow Python expressions enclosed in backticks: ` ... `

```
where i in 1..n;
where i in 1..`n**3 + 7`;
```

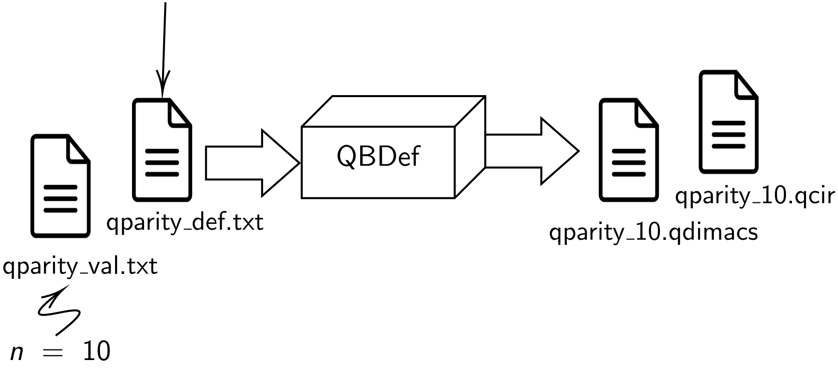
Contents

The Tool: QBDef I

QBDef is commnad-line tool written in Python that takes definitions written in the formal language as input and outputs files in QCIR or QDIMACS.

The Tool: QBDef II

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 $\text{QParity}_n = P_n : \rho_n$.*



Demo

Details and features

- Command-line tool written in Python.

Details and features

- Command-line tool written in Python.
  - Python 3 for the main code.

Details and features

- Command-line tool written in Python.
  - Python 3 for the main code.
  - Lark library for parsing the formal language.

Details and features

- Command-line tool written in Python.
  - Python 3 for the main code.
  - Lark library for parsing the formal language.
  - Python 2 for conversion to QDIMACS via William Klieber's QCIR-to-QDIMACS tool.

Details and features

- Command-line tool written in Python.
  - Python 3 for the main code.
  - Lark library for parsing the formal language.
  - Python 2 for conversion to QDIMACS via William Klieber's QCIR-to-QDIMACS tool.
- Supports:

Details and features

- Command-line tool written in Python.
  - Python 3 for the main code.
  - Lark library for parsing the formal language.
  - Python 2 for conversion to QDIMACS via William Klieber's QCIR-to-QDIMACS tool.
- Supports:
  - Prenex CNF

Details and features

- Command-line tool written in Python.
  - Python 3 for the main code.
  - Lark library for parsing the formal language.
  - Python 2 for conversion to QDIMACS via William Klieber's QCIR-to-QDIMACS tool.
- Supports:
  - Prenex CNF
  - Prenex circuits

Details and features

- Command-line tool written in Python.
  - Python 3 for the main code.
  - Lark library for parsing the formal language.
  - Python 2 for conversion to QDIMACS via William Klieber's QCIR-to-QDIMACS tool.
- Supports:
  - Prenex CNF
  - Prenex circuits
  - Non-prenex circuits (experimental)

Details and features

- Command-line tool written in Python.
  - Python 3 for the main code.
  - Lark library for parsing the formal language.
  - Python 2 for conversion to QDIMACS via William Klieber's QCIR-to-QDIMACS tool.
- Supports:
  - Prenex CNF
  - Prenex circuits
  - Non-prenex circuits (experimental)
- Ouputs:

Details and features

- Command-line tool written in Python.
    - Python 3 for the main code.
    - Lark library for parsing the formal language.
    - Python 2 for conversion to QDIMACS via William Klieber's QCIR-to-QDIMACS tool.
- Supports:
    - Prenex CNF
    - Prenex circuits
    - Non-prenex circuits (experimental)
- Ouputs:
    - QCIR (for prenex circuits)

Details and features

- Command-line tool written in Python.
  - Python 3 for the main code.
  - Lark library for parsing the formal language.
  - Python 2 for conversion to QDIMACS via William Klieber's QCIR-to-QDIMACS tool.
- Supports:
  - Prenex CNF
  - Prenex circuits
  - Non-prenex circuits (experimental)
- Ouputs:
  - QCIR (for prenex circuits)
  - QDIMACS (for PCNF)

Details and features

- Command-line tool written in Python.
  - Python 3 for the main code.
  - Lark library for parsing the formal language.
  - Python 2 for conversion to QDIMACS via William Klieber's QCIR-to-QDIMACS tool.
- Supports:
  - Prenex CNF
  - Prenex circuits
  - Non-prenex circuits (experimental)
- Ouputs:
  - QCIR (for prenex circuits)
  - QDIMACS (for PCNF)
  - Non-Prenex QCIR (experimental)

Details and features

- Command-line tool written in Python.
  - Python 3 for the main code.
  - Lark library for parsing the formal language.
  - Python 2 for conversion to QDIMACS via William Klieber's QCIR-to-QDIMACS tool.
- Supports:
  - Prenex CNF
  - Prenex circuits
  - Non-prenex circuits (experimental)
- Ouputs:
  - QCIR (for prenex circuits)
  - QDIMACS (for PCNF)
  - Non-Prenex QCIR (experimental)
- Source code available at `https://github.com/alephnoell/QBDef`

Contents

Future work

1. Further work *on* QBDef

Future work

1. Further work *on* QBDef
    - More operators ($\rightarrow$, $\leftrightarrow$, if-then-else, ...)

Future work

1. Further work *on* QBDef
   - More operators ($\rightarrow$, $\leftrightarrow$, if-then-else, ...)
   - Improved support for non-prenex formulae

Future work

1. Further work *on* QBDef
    - More operators ($\rightarrow$, $\leftrightarrow$, if-then-else, ...)
    - Improved support for non-prenex formulae
    - Support for external Python packages

Future work

1. Further work *on* QBDef
   - More operators ($\rightarrow$, $\leftrightarrow$, if-then-else, ...)
   - Improved support for non-prenex formulae
   - Support for external Python packages
   - Tests based on certificates

Future work

1. Further work *on* QBDef
    - More operators ($\rightarrow$, $\leftrightarrow$, if-then-else, ...)
    - Improved support for non-prenex formulae
    - Support for external Python packages
    - Tests based on certificates
2. Further research *with* QBDef

Future work

1. Further work *on* QBDef
    - More operators ($\rightarrow$, $\leftrightarrow$, if-then-else, ...)
    - Improved support for non-prenex formulae
    - Support for external Python packages
    - Tests based on certificates
2. Further research *with* QBDef
3. Besides: More features based on feedback from the community (QBF Workshop 2020)

Conclusion

Main result: Developed QBDef, a tool to generate instances of parameterised QBF from definitions to use for proof complexity and solver testing.

Conclusion

Main result: Developed QBDef, a tool to generate instances of parameterised QBF from definitions to use for proof complexity and solver testing.

In the thesis:

1. In-depth study of some other formula families plus an encoding of a two-player PSPACE game.

Conclusion

Main result: Developed QBDef, a tool to generate instances of parameterised QBF from definitions to use for proof complexity and solver testing.

In the thesis:

1. In-depth study of some other formula families plus an encoding of a two-player PSPACE game.
2. More detailed discussion of the formal language and its features.

Conclusion

Main result: Developed QBDef, a tool to generate instances of parameterised QBF from definitions to use for proof complexity and solver testing.

In the thesis:

1. In-depth study of some other formula families plus an encoding of a two-player PSPACE game.
2. More detailed discussion of the formal language and its features.
3. Further discussion on the tool and its implementation.

Conclusion

Main result: Developed QBDef, a tool to generate instances of parameterised QBF from definitions to use for proof complexity and solver testing.

In the thesis:

1. In-depth study of some other formula families plus an encoding of a two-player PSPACE game.
2. More detailed discussion of the formal language and its features.
3. Further discussion on the tool and its implementation.
4. More applications and future lines of work.

Questions?

References

O. Beyersdorf, L. Chew, and M. Janota.
Extension variables in QBF resolution.
In *Workshops at the Thirtieth AAAI Conference on Artificial Intelligence*, 2016.

O. Beyersdorff, L. Chew, and M. Janota.
Proof complexity of resolution-based QBF calculi.
In *LIPI Symposium on Theoretical Aspects of Computer Science (STACS'15)*, volume 30, pages 76–89. Schloss Dagstuhl-Leibniz International Proceedings in Informatics, 2015.

A. Sabharwal, C. Ansotegui, C. P. Gomes, J. W. Hart, and B. Selman.
QBF modeling: Exploiting player symmetry for simplicity and efficiency.
In *International Conference on Theory and Applications of Satisfiability Testing*, pages 382–395. Springer, 2006.