A Formal Language and Tool for QBF Family Definitions

Noel Arteche
Matthias van der Hallen

QBF Workshop 2020

July 10, 2020

Contents

Contents

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.

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 $\mathrm{QParity}_n = P_n : \rho_n$.

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 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 $\mathrm{QPARITY}_n = P_n : \rho_n$.

- $\mathrm{QPARITY}_n$ 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*.

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 $\mathrm{QParity}_n = P_n : \rho_n$.

- $\mathrm{QParity}_n$ 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: $n \in \mathbb{N}$.

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* ($\forall$, $\exists$) or a *logical operator* ($\wedge$, $\vee$, $\oplus$, $\rightarrow$,).

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 $P_n = \exists x_1 \dots \exists x_n \forall z \dots$

```
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* $i \in \{3, \dots, n\}$ *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 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. graphs.

Embedded Python features

We want parameters with non-scalar data-types and operations between them, e.g. graphs. We allow Python expressions enclosed in backticks: ‘ . . . ‘

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

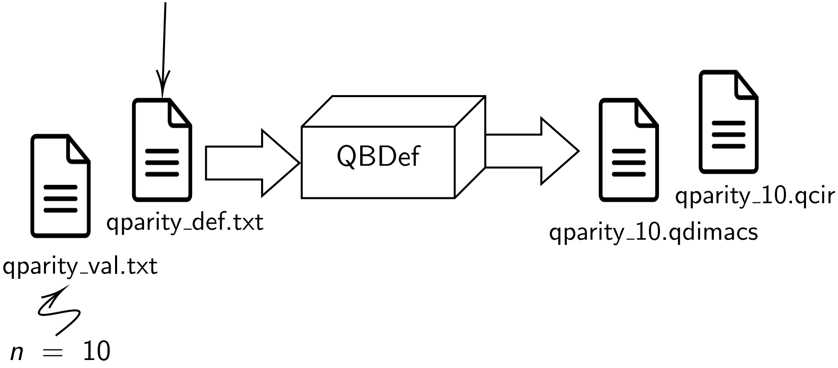
Contents

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 and current version available at `https://github.com/alephnoell/QBDef`

Contents

Future work

1. Further work *on* QBDef
   - Improved support for non-prenex formulae
   - Support for external Python packages

Future work

1. Further work *on* QBDef
    - Improved support for non-prenex formulae
    - Support for external Python packages
2. Further research *with* QBDef

More related research

All of this belong to my Bachelor's thesis.

More related research

All of this belong to my Bachelor's thesis.

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

More related research

All of this belong to my Bachelor's thesis.

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

More related research

All of this belong to my Bachelor's thesis.

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

More related research

All of this belong to my Bachelor's thesis.

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

noel.artecheecheverria@student.kuleuven.be
matthias.vanderhallen@kuleuven.be

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.