QBF Programming
with the Modeling Language Bule

Jean Christoph Jung
Universität Hildesheim, Germany

Valentin Mayer-Eichberger
Universität Potsdam, Germany

Abdallah Saffidine
University of New South Wales, Sydney, Australia

Abstract

We introduce Bule, a modeling language for problems from the complexity class PSPACE via quantified Boolean formulas (QBF) — that is, propositional formulas in which the variables are existentially or universally quantified. Bule allows the user to write a high-level representation of the problem in a natural, rule-based language, that is inspired by stratified Datalog. We implemented a tool of the same name that converts the high-level representation into DIMACS format and thus provides an interface to arbitrary QBF solvers, so that the modeled problems can also be solved. We analyze the complexity-theoretic properties of our modeling language, provide a library for common modeling patterns, and evaluate our language and tool on several examples.

2012 ACM Subject Classification Theory of computation → Constraint and logic programming; Hardware → Theorem proving and SAT solving

Keywords and phrases Modeling, QBF Programming, CNF Encodings

Digital Object Identifier 10.4230/LIPIcs.SAT.2022.31

Supplementary Material Software (Source Code): https://github.com/vale1410/bule

1 Introduction

Quantified Boolean formulas (QBF) extend propositional formulas with explicit quantification (∃, ∀) over propositional variables. QBF are universal for the complexity class PSPACE in the sense that every problem in PSPACE can be efficiently reduced to the validity problem for QBF. That is, QBF and their validity play the same central role for PSPACE that propositional formulas and their satisfiability play for NP. Since PSPACE contains many reasoning problems (beyond NP) of practical interest from diverse areas, such as logic, games, verification, and planning, it is well possible that QBF is the next killer-app of the SAT community [28]. In the past years, the community has made tremendous progress in automatically solving QBF, as evident in the yearly QBF competition [21]. However, to date, there is no established high-level language for elegantly modeling such PSPACE problems in terms of QBF. We attempt to fill this gap by proposing the tool Bule (homophone with “booleh”), which provides exactly this functionality: it gets as input a concise mathematical description of a QBF and converts it to DIMACS1 format, which can be fed to standard QBF solvers. We envision Bule to be used for rapid prototyping of different QBF encodings which can then be evaluated in terms of their performance in solving.

1 http://www.qbflib.org/qdimacs
Conceptually, we model problems using Bule as follows. We represent inputs to the problem as sets of facts. This is very general, since every problem can be modeled in this way: graphs are represented by the set of their edges, circuits by their netlist, and so on. Then a Bule program $\Pi$ for problem $P$ transforms an arbitrary input $D$ of the problem at hand into a quantified Boolean formula $\varphi_{\Pi,D}$ such that

$$D \text{ is a “yes”-instance of problem } P \iff \varphi_{\Pi,D} \text{ is valid.}$$

The following picture describes the overall architecture of Bule.

Thus, we follow the common practice of separating the model $\Pi$ from the data $D$. The design of Bule is driven by the observation that the process of modeling a problem in terms of QBF usually involves applying clause templates to groups of domain individuals. This is reflected in the fact that the Bule program $\Pi$ is partitioned into an extensional part $\Pi_{\text{ext}}$ and an intensional part $\Pi_{\text{int}}$. The extensional part can be thought of as a powerful preprocessing step that identifies relevant groups of domain individuals. It is formulated in a variant of Datalog with stratified negation, which supports order, integer arithmetic, and uninterpreted function symbols, sufficiently restricted so that the evaluation of $\Pi_{\text{ext}}$ on $D$ is finite and uniquely determined. The intensional rules in $\Pi_{\text{int}}$ can be thought of as templates for the quantifier prefix and the clauses in $\varphi_{\Pi,D}$. Clause generation is done via grounding the templates to the groups of domain elements identified in the first phase and results in the target QBF $\varphi_{\Pi,D}$. On the pure clause level, a similar approach has been taken in [19].

Let us consider as an example the following two-player game. Given a set of $k$-bit numbers $W$, two players $E$ and $A$ determine in $k$ rounds a $k$-bit number $N$ by alternatingly picking the bits of $N$, say, in order of increasing significance. Player $E$ starts the game and wins if $N \in W$; otherwise, $A$ wins. We are interested in the existence of a winning strategy for $E$ – that is, a strategy for how to react to all possible moves of $A$ and win in all cases. Although the existence of a winning strategy in these games can be decided in PTime, central aspects of modeling with Bule can be illustrated in this example, and in fact, many other games, such as Geography and Tic Tac Toe, can be modeled in a similar fashion [24].

The input $D$ consists of facts $\text{number}[n,\text{pos},\text{bit}]$ with the intended meaning that $n$ is a number in $W$ that has value $\text{bit}$ in position $\text{pos}$ in its binary representation. The existence of a winning strategy for $E$ is modeled with the Bule program depicted in Listing 1. Lines 2–4 form the extensional part of the program; the remaining lines constitute the intensional part, responsible for the generation of $\varphi_{\Pi,D}$. Lines 2–3 declare that player $E$ (modeled by the constant $e$) is responsible for the even positions, while $A$ (constant $a$) is responsible for the odd positions. Line 4 expresses that every input number is winning. Lines 7–9 declare variables via grounding as follows: Line 7 says that for each turn, say $i$, of player $E$, there is an existentially quantified variable $\text{set}_\text{bit}(i)$ in level $i$ of the quantifier prefix of $\varphi_{\Pi,D}$; that is, ground facts act as the propositional variables of the QBF formula. Line 8 is analogous for player $A$. Line 9 declares variables $\text{chosen}(n)$ for every winning number $n$. The missing level in the quantifier prefix indicates that this variable is existentially quantified in the innermost level. Finally, Lines 12–14 generate the clauses in $\varphi_{\Pi,D}$. Line 12 expresses that one of the winning numbers has to be chosen, and Lines 13–14 express that the bits set so far are compatible with the chosen number ($\sim$ and $|$ refer to negation and disjunction).
Listing 1 Modeling the Number Game in Bule.

```plaintext
% extensional program
number[_,P,_], P # mod 2 = 0 :: # ground turn[e,P].
number[_,P,_], P # mod 2 = 1 :: # ground turn[a,P].
number[N,_,_] :: # ground winning[N].

% variable declaration
winning[N] :: # exists chosen(N).

% clause generation
:: winning[N] : chosen(N).
number[N,B,1] :: -chosen(N) | set_bit(B).
number[N,B,0] :: -chosen(N) | -set_bit(B).
```

Summarizing again, the extensional part of \( \Pi \) computes relevant groups of such domain elements (see `winning` in the example above) and the intensional phase grounds the respective clauses. The ultimate goal is that only one line per clause template is required in the Bule model (see Lines 12–14 in Listing 1). To support modeling in Bule, we deliver a standard library, which is itself written in Bule and which provides commonly used modeling patterns, such as cardinality constraints and reachability encodings. We envision that, in the future, more patterns will be included so that QBF programming with Bule will be further facilitated.

The paper is organized as follows. In Section 2, we define syntax and semantics of Bule and conduct a complexity analysis. In Section 3, we describe the standard library, give more examples of Bule programs, and provide an evaluation. We conclude in Section 4.

Related Work. The current standard input language for QBF, QDIMACS, is propositional and thus not appropriate for high-level modeling of problems. As a modeling language, Bule draws inspiration from languages that have been proposed for describing propositional formulas in conjunctive normal form (CNF) [19, 25, 26], from logic programming [10, 14], and from answer set programming (ASP) [16, 18]. Bule’s approach to solving the model given some input data is also not new: in many of the mentioned works, solving is done by grounding the specified model with the relevant individuals from the input data [16, 18, 19, 25, 26]. As a difference (besides the possible arbitrary quantification using \( \exists, \forall \)), we stress that Bule’s grounding semantics does not resort to well-founded models (which is standard in ASP). A grounding approach to modeling and solving is also proposed in [9, 31] in which the model is given in first-order logic, possibly with inductive definitions, and is grounded to CNF clauses. In the imperative programming paradigm, the standard modeling tool is probably PySAT [22], a Python framework that allows for the convenient creation and solution of SAT instances. Finally, a language that allows for both high-level modeling and arbitrary quantification has recently been introduced in quantified ASP [15]. Quantified ASP relates to QBF in the same way that ASP relates to SAT.

2 The Bule Language: Syntax, Semantics, and Complexity

We assume familiarity with quantified Boolean formulas. In a nutshell, a quantified Boolean formula (QBF) is of shape \( \exists \bar{x}_1 \forall \bar{y}_1 \ldots \exists \bar{x}_n \forall \bar{y}_n . \varphi \), with \( \varphi \) being a propositional formula and a quantifier prefix \( \exists \bar{x}_1 \forall \bar{y}_1 \ldots \exists \bar{x}_n \forall \bar{y}_n \) which mentions all propositional variables in \( \varphi \). Validity of a QBF is the prototypical \( \text{PSPACE} \)-complete problem and is defined as usual [4].
2.1 Syntax and Semantics

We formally introduce the language Bule, starting with its syntax. For the sake of conforming to standard notions, we slightly deviate in the representation of the language from what Bule programs look like. In Section 2.3 below, we discuss these differences and give more details on the system actually implemented.

We assume (countably) infinite sets of variables $X$, function symbols $f$ (each having a fixed arity), and predicate symbols $p$ (also with an arity); function symbols of arity 0 are called constants. Terms are defined as usual based on variables and function symbols. Atoms are of the form $p(t_1,\ldots,t_n)$ for a predicate symbol $p$ of arity $n$ and terms $t_1,\ldots,t_n$. Literals are atoms $p(t_1,\ldots,t_n)$ and negated atoms $\neg p(t_1,\ldots,t_n)$. We distinguish two kinds of predicate symbols: extensional predicate symbols and intensional predicate symbols.

A Bule program is a pair $\Pi = (\Pi_{\text{ext}},\Pi_{\text{int}})$ of an extensional program $\Pi_{\text{ext}}$ and an intensional program $\Pi_{\text{int}}$. Recall that a Bule program $\Pi$ provides a template for generating, from a given instance $D$, a QBF $\varphi_{\Pi,D}$ such that Condition (*), from the introduction, is satisfied. We start with the description of the extensional program.

The extensional program $\Pi_{\text{ext}}$ consists of rules in the form

$$L_1,\ldots,L_k :: H$$

where $L_1,\ldots,L_k$ are extensional literals and $H$ is an extensional atom. As usual, we call $L_1,\ldots,L_k$ the body and $H$ the head of the rule. We make several standard assumptions regarding $\Pi_{\text{ext}}$ to ensure that $\Pi_{\text{ext}}$ is well-behaved. More specifically, we require that $\Pi_{\text{ext}}$ is range-restricted (every variable that occurs in the head or a negative body atom, occurs in a positive body atom), argument-restricted [23], and stratified (no recursion over negations) [1]. Formally, the semantics of an extensional program $\Pi_{\text{ext}}$ is defined via minimal models as follows. A term or an atom is ground if it does not contain a variable; ground atoms are also called facts. A substitution is a map $\theta$ from variables into ground terms. We denote with $A\theta$ the application of the substitution $\theta$ to atom $A$ – that is, the replacement of every variable $X$ with $\theta(X)$. A substitution $\theta$ is a match for literals $L_1,\ldots,L_k$ in a set of facts $D$ if $A\theta \in D$ for every positive literal $A$ in $L_1,\ldots,L_k$ and $A\theta \notin D$ for every literal $\neg A$ in the $L_1,\ldots,L_k$.

Let $D$ be a set of facts. A set of facts $M$ is a model of $\Pi_{\text{ext}}$ and $D$ if $D \subseteq M$ and for every rule $L_1,\ldots,L_k :: H \in \Pi_{\text{ext}}$ and every match $\theta$ of its body in $M$, we have $H\theta \in M$. A model $M$ is minimal if there is no $M' \subseteq M$ that is a model of $\Pi_{\text{ext}}$ and $D$. It is known that if $\Pi_{\text{ext}}$ is range restricted, argument restricted, and stratified, then for every finite set of facts $D$, there is a unique and finite minimal model $\Pi_{\text{ext}}(D)$ of $\Pi_{\text{ext}}$ and $D$ [23].

The intensional part $\Pi_{\text{int}}$ of a Bule program consists of rules of the form

$$L_1,\ldots,L_k :: Q[T] H$$

$$L_1,\ldots,L_k :: C_1,\ldots,C_n$$

where $L_1,\ldots,L_k$ are extensional literals as before, $H$ is an intensional atom, $Q \in \{\exists,\forall\}$, and $T$ is a non-negative integer. Moreover, $C_1,\ldots,C_n$ are conditional literals of the form $M_1,\ldots,M_m : B$ where $M_1,\ldots,M_m$ are extensional literals and $B$ is an intensional literal. We allow $m = 0$ in which case the conditional literal is just a single intensional literal. Rules of the form (2) are templates for variable declarations and rules of the form (3) are templates for clause generators. Intuitively, the $L_i$ and the $M_j$ act as guards in that they specify conditions under which variables / clauses are generated.

We are now in a position to provide the semantics for Bule programs $\Pi = (\Pi_{\text{ext}},\Pi_{\text{int}})$. Let $D$ be a set of facts. The QBF $\varphi_{\Pi,D} = \exists x_1 \forall y_1 \ldots \exists x_n \forall y_n . \varphi$ that we are going to construct uses propositional variables of the form $x_A$, for ground intensional atoms $A$. Given a ground literal $A$, we define the corresponding propositional literal $\text{lit}(A)$ as expected:
Table 1 Complexity of Bule evaluation. All results are completeness results.

<table>
<thead>
<tr>
<th>complexity measure</th>
<th>combined complexity</th>
<th>program complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>arbitrary quantification</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Bule</td>
<td>EXPSPACE</td>
<td>NEXPTIME</td>
</tr>
<tr>
<td></td>
<td>EXPTIME</td>
<td>PSPACE</td>
</tr>
<tr>
<td></td>
<td>NP</td>
<td></td>
</tr>
<tr>
<td>Horn Bule</td>
<td>EXPTIME</td>
<td>EXPTIME</td>
</tr>
<tr>
<td></td>
<td>P</td>
<td>P</td>
</tr>
</tbody>
</table>

\[
\text{lit}(A) = \begin{cases} 
  x_A & \text{if } A \text{ is an atom,} \\
  \neg x_{A'} & \text{if } A = \neg A' \text{ is a negated atom.}
\end{cases}
\]

Now, \( \varphi_{\Pi, D} \) is defined as the result of two steps. Step 1 defines the quantifier prefix and Step 2 defines the clauses. We recommend to read Steps 1 and 2 with Listing 1 in mind.

1. For every rule \( L_1, \ldots, L_k :: Q[T] \mathcal{H} \in \Pi_{\text{int}} \) and every match \( \theta \) of its body in \( \Pi_{\text{ext}}(D) \), variable \( x_H \theta \) is among \( \vec{x}_T \) in the prefix if \( Q = \exists \) and among \( \vec{y}_T \) if \( Q = \forall \).
2. For every rule \( L_1, \ldots, L_k :: C_1, \ldots, C_n \in \Pi_{\text{int}} \) and every match \( \theta \) of its body in \( \Pi_{\text{ext}}(D) \), \( \varphi \) contains the clause

\[
\bigvee_{\ell \in \widehat{C}_i} \ell \lor \ldots \lor \bigvee_{\ell \in \widehat{C}_n} \ell
\]

where \( \widehat{C}_i \) is a set of propositional literals defined as follows. Let \( \theta_0 \) be the restriction of \( \theta \) to variables that occur in \( L_1, \ldots, L_k \). If \( C_i \) is \( M_1, \ldots, M_m : B \), then \( \widehat{C}_i \) contains the literal \( \text{lit}(B \theta_0 \theta_1) \), for every match \( \theta_1 \) of \( M_1 \theta_0, \ldots, M_m \theta_0 \) in \( \Pi_{\text{ext}}(D) \).

For instance, Line 9 of Listing 1 declares a variable \( \text{chosen}(N) \) for every match of \( \text{winning}[N] \) in \( \Pi_{\text{ext}}(D) \), that is, for every number \( N \) in the dataset \( D \). Further, \( \text{winning}[N] : \text{chosen}[N] \) in Line 12 is a conditional literal that creates a disjunction containing one literal \( \text{chosen}(N) \) for every \( N \) in the input \( D \).

### 2.2 Complexity

Bule programs provide a succinct way of specifying QBF. We here investigate the complexity of Bule as a modeling language; that is, we study the following problem Bule evaluation:

- **Input:** Bule program \( \Pi \), set of facts \( D \)
- **Question:** Is \( \varphi_{\Pi, D} \) valid?

Throughout the section, we assume that there are no function symbols of arity \( > 0 \), but we conjecture that our results also hold in general. We consider two forms of complexity: combined complexity and program complexity. In the former, we consider both \( \Pi \) and \( D \) as input, whereas in the latter \( \Pi \) is assumed to be fixed. We consider two further dimensions. First, we study the influence of the presence of universally quantified variables, and second, we investigate Horn Bule programs. As expected, a Bule program is Horn if in rules of type (3), each generated clause contains at most one positive literal. The proof of the following is relatively standard and provided in the appendix.

**Theorem 1.** For Bule evaluation, the complexity results in Table 1 hold.
Bule System

We provide a prototype which can read Bule programs and solve them using the specified QBF solver. Since Bule grounds its input to DIMACS, in principle, any QBF solver that supports DIMACS as input language can be used. The system is implemented in OCaml and available for download at github.com/vale1410/bule with binaries for Linux and MacOS. This paper refers to release tag 4.0.1. The input language of the Bule system slightly deviates from the language given above. There are two extensions that facilitate modeling.

Integer variables and arithmetic. We allow for the use of integer variables \(X, Y\), standard integer arithmetic as in \(X + Y, X \div Y, XY, X \mod Y\), and comparisons such as \(X < Y\). Correspondingly, we allow integers in \(D\) and \(\Pi\). Formally, an integer \(n\) is represented as the \(n\)-fold application of a function symbol \(s\) to a constant \(0\), that is, \(s^n(0)\), and integer arithmetic is encoded in \(\Pi_{ext}\). We also allow integer variables as level indicator \(T\) in variable declarations of shape (2), as in Lines 7–8 from Listing 1.

Order predicate. Besides the mentioned comparisons between integer variables, we allow for a binary order predicate \(<_{tot}\) that can be used on arbitrary terms. This predicate is interpreted as a total order over the Herbrand base of \(\Pi_{ext} \cup D\) (recall: this is the set of all ground terms that can be formed using the function symbols in \(\Pi_{ext} \cup D\)), but it is not defined as to which one. The Bule user may assume, however, that the interpretation is compatible with the order \(<\) over the integers, which justifies that we write \(<\) in place of \(<_{tot}\).

Apart from these extensions, we use a slightly different syntax in Bule programs as input for our tool; most of them are already visible in Listing 1.

- We mark extensional rules of shape (1) by adding the prefix \(#\text{ground}\) to the head.
- We mark extensional atoms by writing \(p(t_1, \ldots, t_n)\) instead of \(p(t_1, \ldots, t_n)\).
- We write \(#\text{exists}\) and \(#\text{forall}\) instead of \(\exists\) and \(\forall\) in variable declarations of type (2).
- Conditional literals \(C_1, \ldots, C_n\) in rules of type (3) are separated with "|" instead of ",".
- Clauses can equivalently be written as implications using the symbol "\(\rightarrow\)" where the left side is a conjunction using "\&" and the right side is a disjunction.

Modeling in Bule

In this section, we showcase modeling in Bule by means of several examples, focusing on the library support. Providing libraries is motivated by the fact that certain patterns recur during modeling. In the current version, Bule provides libraries for cardinality constraints and for reachability.

Cardinality. Cardinality constraints, that is, constraints that specify that among a set of propositional variables at least \(\frac{k}{k}\) to be true, are ubiquitous in modeling. For instance, consider the vertex cover problem: given a graph and a number \(k\), decide whether
there is a subset of $k$ vertices that cover all edges. Listing 2 illustrates the use of cardinality constraints by providing a $Bule$ program. A great deal of research has been invested to find good CNF encodings for cardinality constraints. Currently, the user can choose between the encodings from $[3,29,30]$ via the last parameter of `cardinality_constraint`, see Line 3.

Reachability. In many $PSPACE$ problems, one needs to model graph reachability. From a complexity theoretical point of view, the interesting (that is, hard) case is when the graph is implicitly represented. We illustrate this using the example of STRIPS planning $[17]$. Recall that a STRIPS instance consists of an initial state, goal states, and set of actions with pre- and postconditions. A STRIPS instance can easily be cast as a set of facts using function symbols, e.g., via `#ground init(on(a,b))` one can specify that initially object $a$ is located on $b$ and via `pos(act,clear(t))` that a positive consequence of executing $act$ is that $clear(t)$ becomes true. A full encoding of a STRIPS instance is given in Listing 4 in the appendix. A STRIPS instance implicitly represents an (exponentially large) graph: the nodes are the possible states of the world and the edges are the possible actions. The STRIPS planning problem is to decide given a STRIPS instance whether there is a path from the initial state to a goal state in that graph. STRIPS planning is $PSPACE$-complete $[7]$.

$Bule$ currently supports two ways of encoding reachability:

- **direct**: This encoding uses propositional variables $pos(i,a)$, for every $i \leq \ell$, $\ell$ the maximal length of the sought path, and every node $a$ in the graph, which intuitively state that the $i$-th node on the path is $a$. The clauses enforce that there for every $i$ only one node is chosen, and that consecutive nodes are connected.

- **binary search** $[8]$: This encoding uses a standard technique that is underlying, e.g., the proof of Savitch’s Theorem: there is a path of length $2^\ell$ between $a$ and $b$ iff there is some $c$ such that there are paths of length $2^{\ell-1}$ from $a$ to $c$ and from $c$ to $b$, respectively.

We show, in Listing 3, a $Bule$ program that models STRIPS planning via the direct encoding. Indeed, after the generation of all relevant time points in Lines 1–2, the variables `true(t,p)` and `do(t,a)` declared in Lines 3–4 correspond directly to the variables mentioned
Table 2 Overview of evaluation. Timeout (TO) was set to 100000ms.

<table>
<thead>
<tr>
<th>family</th>
<th>#instances</th>
<th>$\mathcal{O}$var</th>
<th>$\mathcal{O}$clauses</th>
<th>$\mathcal{O}$ground. time (ms)</th>
<th>$\mathcal{O}$solv. time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>gtt</td>
<td>180</td>
<td>468</td>
<td>1690</td>
<td>19</td>
<td>13145</td>
</tr>
<tr>
<td>gtt-iterative</td>
<td>27</td>
<td>400</td>
<td>1587</td>
<td>23</td>
<td>14701</td>
</tr>
<tr>
<td>hex-hein</td>
<td>34</td>
<td>1552</td>
<td>17687</td>
<td>42757</td>
<td>22613</td>
</tr>
<tr>
<td>strips-direct</td>
<td>325</td>
<td>1291</td>
<td>103172</td>
<td>751</td>
<td>242</td>
</tr>
<tr>
<td>strips-binary</td>
<td>668</td>
<td>648</td>
<td>41638</td>
<td>336</td>
<td>103671 + 30 TO</td>
</tr>
<tr>
<td>argumentation</td>
<td>240</td>
<td>5160</td>
<td>14744</td>
<td>3853</td>
<td>1474 + 90 TO</td>
</tr>
</tbody>
</table>

in the description of the direct encoding. Lines 7–9 describe the initial and goal states. Lines 10–11 express that at every time point exactly one action has to be executed. Finally, Lines 14-20 describe the update of states and the applicability of the chosen action.

The implementation of the binary search encoding of STRIPS planning is much more tedious and error-prone. Bule provides an interface that allows the user to specify where the encoding is needed and automatically generates the relevant clauses. Thus, the binary search encoding can be used in the same way as the direct encoding, see Listing 5 in the appendix.

Evaluation. To evaluate our language and tool, we created Bule programs for problems from three different domains: positional games, STRIPS planning, and argumentation. All programs and instances are available from https://github.com/vale1410/bule.

Positional games are two-player games played on a board; the players alternatingly occupy a field of their choosing, and the player who can first occupy one of their winning regions wins. Examples of positional games are (generalized) Tic Tac Toe and Hex. We used programs from [24] that were already written in Bule and evaluated three sets of benchmarks, which can be found in Lines 1–3 of Table 2.

For STRIPS planning, we used the two programs provided in Listing 3 and Listing 5; that is, one based on the direct encoding of reachability and one based on the binary encoding. We used instances from the IPC BlocksWorld benchmark set [2].

Finally, we considered an inference problem over abstract argumentation frameworks (AAF), a central and popular knowledge representation formalism [12]. There are several different semantics for these AAF, which can, in principle, all be easily modeled in Bule. We chose sceptical inference over preferred semantics (see appendix for details on the semantics and the Bule program), which is a $\Pi^p_2$-complete problem [13]. The benchmark set is taken from the 2019 edition of the argumentation competition [5].

The results displayed in Table 2 demonstrate that the current implementation grounds small- to medium-sized instances within reasonable time. We solved the instances with different solvers and report the results of the best solver. It can be seen that the solving time mostly dominates the grounding time, and it is of the same order of magnitude in the remaining cases. From a usability perspective, it is important to stress that the Bule program for the argumentation problem was developed within a very short time; it can easily be adapted to other semantics, resulting in a prototype for solving argumentation problems.

4 Conclusion and Future Work

We have presented the language Bule for conveniently modeling problems in \textsc{pspace} as QBF problems and a prototype implementation. In the future, we want to extend the library with more functionality, e.g., Pseudo-Boolean Constraints or alternative encodings.
of reachability [27]. We also want to improve our prototype in terms of grounding speed, using experiences from the ASP community [20]. Finally, it will be interesting to investigate high-level tasks such as pre-processing at the level of the modeling language (in contrast to inference on the level of the induced QBF), debugging, and the creation of (possibly high-level) validity certificates in case the formula is valid.

References


31:10 QBF Programming with Bule

A Proof of Theorem 1

Theorem 1. For Bule evaluation, the complexity results in Table 1 hold.

Proof. The upper bounds are a consequence of the well-known fact that, given a range-restricted and stratified program $\Pi_{\text{ext}}$ without function symbols, we can compute $\Pi_{\text{ext}}(\mathcal{D})$ in time polynomial in $m^a$, where $m$ denotes the size of (a representation of) $\mathcal{D}$ and $a$ denotes the maximal arity of a predicate symbol in $\Pi_{\text{ext}}$ [11]. It follows that the generated quantified propositional formula $\varphi_{\Pi,\mathcal{D}}$ can be computed in time polynomial in $m^a$ as well. Note that in data complexity $a$ is constant, so $\varphi_{\Pi,\mathcal{D}}$ can be computed in time polynomial in $m$ in this case. The upper bounds now follow from known results for QBF, SAT, and Horn QBF [6], and Horn SAT.

The lower bounds follow from the fact that appropriate Turing machines can be simulated. For example, it is well-known that we can simulate exponentially time bounded deterministic Turing machines in Datalog [11]. Since Datalog is a fragment already of the extensional part of Bule programs and the mentioned simulation relies on predicate symbols of arbitrary arity, this yields EXPTIME-hardness in combined complexity without universal quantification. The other entries without universal quantification are similar.

B Background on Abstract Argumentation

An abstract argumentation framework (AAF) is a directed graph $A = (\text{Arg}, \text{Att})$, where $\text{Arg}$ is a set of arguments and $\text{Att} \subseteq \text{Arg} \times \text{Arg}$ is the attack relation. The semantics of AAF is based on the notion of extension. Intuitively, an extension is a set of arguments (that is, a subset of the vertices) which is defended against arguments not in the set and possibly satisfies some more properties depending on the use case. Each variant of extension gives rise to a different semantics. We use here the notion of a preferred extension. A preferred extension of $A$ is a set $E \subseteq \text{Arg}$ such that:

- $E$ is conflict-free, that is, there is no $(e, f) \in \text{Att}$ with $e, f \in E$;
- $E$ is admissible, that is, for every $e \in E$, and every $(f, e) \in \text{Att}$, there is an $g \in E$ with $(g, f) \in \text{Att}$;
- $E$ is maximal (w.r.t. $\subseteq$) with the first two properties.

There might be multiple preferred extensions and we consider sceptical inference which is the following problem:

- input: AAF $A = (\text{Arg}, \text{Att})$ and $a \in \text{Arg}$
- question: Is $a \in E$ for all preferred extensions $E$ of $A$?

Listing 6 displays a rather direct Bule model for (the complement of) sceptical inference.

C Listings

The following pages contain the missing listings.
Listing 4 STRIPS instance as `Bule` input. The comments display the corresponding Planning Domain Description Language code (PDDL).

```plaintext
%% STRIPS instance
(define (problem blocks-ab-from-table1-to-stacked-ab-table3)
  (:domain blocksworld)
  (:objects a - block b - block t1 - table t2 - table t3 - table)
  (:init (on a b) (on b t1) (clear a) (clear t2) (clear t3))
  (:goal (on a b) (on a b) (on b t3)))

# ground objects [a, block], objects [b, block], objects [t1, table], objects [t2 , table], objects [t3, table].
# ground init [on(a,b)], init [on(b,t1)], init [clear(a)], init [clear(t2)],
# init [clear(t3)].
# ground goal [on(a,b)], goal [on(b,t3)].

# ground maxtime [4].

%% STRIPS instance
(define (domain blocksworld)
  (:requirements :strips :typing)
  (:types block - object table - object)
  (:predicates (on ?x - block ?y - object) (clear ?x - object))
  (:action move
    :parameters (?b - block ?x - table ?y - table)
    :precondition (and (on ?b ?x) (clear ?b) (clear ?y))
    :effect (and (not (on ?b ?x)) (not (clear ?y))
              (on ?b ?y) (clear ?x)))
  (:action stack
    :parameters (?a - block ?x - object ?b - block)
    :precondition (and (on ?a ?x) (clear ?a) (clear ?b))
    :effect (and (not (on ?a ?x)) (not (clear ?b))
              (on ?a ?b) (clear ?x)))
  (:action unstack
    :parameters (?a - block ?b - block ?y - object)
    :precondition (and (on ?a ?b) (clear ?a) (clear ?y))
    :effect (and (not (on ?a ?b)) (not (clear ?y))
              (on ?a ?y) (clear ?b) (clear ?a)))

# ground type [block, object], type [table, object].

objects [X, block], objects [Y, object] :: # ground fluent [on(X,Y)].
objects [X, object] :: # ground fluent [clear(X)].

objects [B, block], objects [X, table], objects [Y, table] :: # ground action [move(B,X,Y)],
pre [move(B,X,Y), on(B,X)], pre [move(B,X,Y), clear(Y)], pre [move(B,X,Y),
clear(B)],
eg [move(B,X,Y), on(B,X)], neg [move(B,X,Y), clear(Y)],
post [move(B,X,Y), on(B,Y)], post [move(B,X,Y), clear(X)].

objects [A, block], objects [X, object], objects [B, block] :: # ground action [stack(A,X,B)],
pre [stack(A,X,B), on(A,X)], pre [stack(A,X,B), clear(B)], pre [stack(A,X,B),
clear(A)],
eg [stack(A,X,B), on(A,X)], neg [stack(A,X,B), clear(B)],
post [stack(A,X,B), on(A,B)], post [stack(A,X,B), clear(X)].

objects [A, block], objects [B, block], objects [Y, object] :: # ground action [unstack(A,B,Y)],
pren [unstack(A,B,Y), on(A,B)], pre [unstack(A,B,Y), clear(Y)], pre [unstack(A,B,Y), clear(A)],
eg [unstack(A,B,Y), on(A,B)], neg [unstack(A,B,Y), clear(Y)],
post [unstack(A,B,Y), on(A,Y)], post [unstack(A,B,Y), clear(B)].
```
Listing 5 STRIPS planning via Bule reachability library. In Line 3, the user can specify the chosen encoding: direct or binary search. Bule will then generate the corresponding clauses from it.

fluent[F] :: #ground reach_fluent[g,F].
maxtime[T] :: #ground reach_length[id,T].

    #ground reach_choose[id,direct]. % other option is "binary"

reach_init[ID,,T], fluent[F], init[F] ::
    reach_test(ID,init(T)) -> reach_state(ID,T,F).

reach_init[ID,,T], fluent[F], -init[F] ::
    reach_test(ID,init(T)) -> -reach_state(ID,T,F).

reach_goal[ID,,T], goal[F] ::
    reach_test(ID,goal(T)) -> reach_state(ID,T,F).

reach_succ[ID,Q,T,U], action[A] :: #exists[Q] do(T,A).


reach_succ[ID,,T,U], action[A1], action[A2], A1 < A2 ::
    -do(T,A1) | -do(T,A2).

reach_succ[ID,,T,U], pre[A,F] ::
    reach_test(ID,succ(T,U)) & do(T,A) -> reach_state(ID,T,F).

reach_succ[ID,,T,U], neg[A,F] ::
    reach_test(ID,succ(T,U)) & do(T,A) -> -reach_state(ID,U,F).

reach_succ[ID,,T,U], pos[A,F] ::
    reach_test(ID,succ(T,U)) & do(T,A) -> reach_state(ID,U,F).

reach_succ[ID,,T,U], action[A], fluent[F], -neg[A,F], -pos[A,F] ::
    reach_test(ID,succ(T,U)) & do(T,A) & reach_state(ID,T,F) ->
        reach_state(ID,U,F).

reach_succ[ID,,T,U], action[A], fluent[F], -neg[A,F], -pos[A,F] ::
    reach_test(ID,succ(T,U)) & do(T,A) & -reach_state(ID,T,F) ->
        -reach_state(ID,U,F).
Listing 6  Bule model for (the complement of) sceptical inference w.r.t. preferred semantics.

% Quantifier Declaration

% Clauses
target[X] :: ~e(X).
att[X,Y] :: e(X) -> attacked(Y).
arg[Y] :: attacked(Y) -> att[X,Y] : e(X).

% conflict free
arg[X] :: e(X) -> ~attacked(X).

% admissible
att[X,Y] :: e(Y) -> attacked(X).

% Now for f
att[X,Y] :: f(X) -> attackedF(Y).

% conflict free
arg[X] :: cheatCF(X) -> f(X).
arg[X] :: cheatCF(X) -> attackedF(X).

% admissible
att[X,Y] :: cheatA(X,Y) -> f(Y).
att[X,Y] :: cheatA(X,Y) -> ~attackedF(X).

% missing
arg[X] :: cheatM(X) -> e(X).
arg[X] :: cheatM(X) -> ~f(X).

cheat -> arg[X]:cheatCF(X) | arg[X]:cheatM(X) | att[X,Y]:cheatA(X,Y).
arg[X] :: ~cheat & f(X) -> e(X).