# 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 aribtrary 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 ($\exists, \forall$) 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 DIMACS[1] 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.

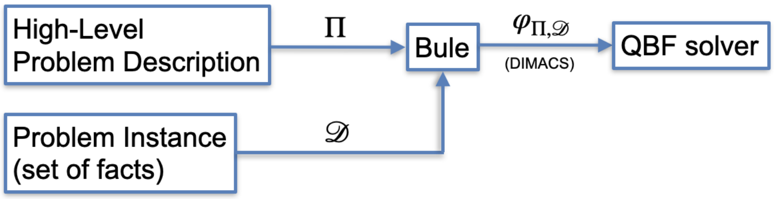
[1] `http://www.qbflib.org/qdimacs`

© Jean Christoph Jung, Valentin Mayer-Eichberger, and Abdallah Saffidine;
licensed under Creative Commons License CC-BY 4.0
25th International Conference on Theory and Applications of Satisfiability Testing (SAT 2022).
Editors: Kuldeep S. Meel and Ofer Strichman; Article No. 31; pp. 31:1–31:14
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

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 $\mathcal{D}$ of the problem at hand into a quantified Boolean formula $\varphi_{\Pi,D}$ such that

$$\mathcal{D} \text{ is a "yes"-instance of problem } P \quad \text{iff} \quad \varphi_{\Pi,\mathcal{D}} \text{ is valid.} \tag{$*$}$$

The following picture describes the overall architecture of *Bule*.




Thus, we follow the common practice of separating the model $\Pi$ from the data $\mathcal{D}$. 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 $\mathcal{D}$ 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 $\varphi_{\Pi,\mathcal{D}}$. 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 $\varphi_{\Pi,\mathcal{D}}$. 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 $\mathcal{D}$ consists of facts `number[n,pos,bit]` with the intended meaning that `n` is a number in $W$ that has value `bit` in position `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 $\varphi_{\Pi,\mathcal{D}}$. 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 `set_bit(i)` in level `i` of the quantifier prefix of $\varphi_{\Pi,\mathcal{D}}$; that is, ground facts act as the propositional variables of the QBF formula. Line 8 is analogous for player A. Line 9 declares variables `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 $\varphi_{\Pi,\mathcal{D}}$. 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*.

```
1  % extensional program
2  number[_,P,_], P #mod 2 = 0 :: #ground turn[e,P].
3  number[_,P,_], P #mod 2 = 1 :: #ground turn[a,P].
4  number[N,_,_] :: #ground winning[N].
5
6  % variable declaration
7  turn[e,P]  :: #exists[P] set_bit(P).
8  turn[a,P]  :: #forall[P] set_bit(P).
9  winning[N] :: #exists chosen(N).
10
11 % clause generation
12 :: winning[N] : chosen(N).
13 number[N,B,1] :: ~chosen(N) | set_bit(B).
14 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 $\exists \vec{x}_1 \forall \vec{y}_1 \ldots \exists \vec{x}_n \forall \vec{y}_n . \varphi$, with $\varphi$ being a propositional formula and a *quantifier prefix* $\exists \vec{x}_1 \forall \vec{y}_1 \ldots \exists \vec{x}_n \forall \vec{y}_n$ which mentions all propositional variables in $\varphi$. *Validity* of a QBF is the prototypical 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* $\Pi_{est}$ and an *intensional program* $\Pi_{\text{int}}$. Recall that a *Bule* program $\Pi$ provides a template for generating, from a given instance $\mathcal{D}$, a QBF $\varphi_{\Pi,\mathcal{D}}$ 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

$$L_1, \ldots, L_k :: H \tag{1}$$

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* $\mathcal{D}$ if $A\theta \in \mathcal{D}$ for every positive literal $A$ in $L_1, \ldots, L_k$ and $A\theta \notin \mathcal{D}$ for every literal $\neg A$ in the $L_1, \ldots, L_k$.

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

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

$$L_1, \ldots, L_k :: Q[T]\, H \tag{2}$$

$$L_1, \ldots, L_k :: C_1, \ldots, C_n \tag{3}$$

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 $\mathcal{D}$ be a set of facts. The QBF $\varphi_{\Pi,\mathcal{D}} = \exists\vec{x}_1\forall y_1 \ldots \exists\vec{x}_n\forall y_n.\varphi$ 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.

| complexity measure | combined complexity | | program complexity | |
|---|---|---|---|---|
| arbitrary quantification | Yes | No | Yes | No |
| *Bule* | EXPSPACE | NEXPTIME | PSPACE | NP |
| Horn *Bule* | EXPTIME | EXPTIME | P | P |

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

Now, $\varphi_{\Pi,\mathcal{D}}$ 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 $L_1, \ldots, L_k :: Q[T]\, H \in \Pi_{\mathrm{int}}$ and every match match $\theta$ of its body in $\Pi_{\mathrm{ext}}(\mathcal{D})$, 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 $L_1, \ldots, L_k :: C_1, \ldots, C_n \in \Pi_{\mathrm{int}}$ and every match $\theta$ of its body in $\Pi_{\mathrm{ext}}(\mathcal{D})$, $\varphi$ contains the clause

   $$\bigvee_{\ell \in \widehat{C}_1} \ell \vee \ldots \vee \bigvee_{\ell \in \widehat{C}_n} \ell$$

   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 $\mathrm{lit}(B\theta_0\theta_1)$, for every match $\theta_1$ of $M_1\theta_0, \ldots, M_m\theta_0$ in $\Pi_{\mathrm{ext}}(\mathcal{D})$.

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

## 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 $\mathcal{D}$
- **Question:** Is $\varphi_{\Pi,\mathcal{D}}$ 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 $\mathcal{D}$ 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.*

**Listing 2** *Bule* program for vertex cover.

```
vertex[X] :: #exists var(X).                 % one prop. variable per vertex
vertex[X] :: #ground cardinality_var[X].  % register counter
size[K]   :: #ground cardinality_constraint[leq,K,counter].   % at most K
edge[X,Y] :: var(X) | var(Y).                % covering condition
```

## 2.3 *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 $X + Y$, $X \div Y$, $X^Y$, $X \bmod Y$, ..., and comparisons such as $X < Y$. Correspondingly, we allow integers in $\mathcal{D}$ 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 $\Pi_{\text{ext}}$. We also allow integer variables as level indicator $T$ in variable delarations 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 $<_{\text{tot}}$ that can be used on arbitrary terms. This predicate is interpreted as a total order over the Herbrand base of $\Pi_{\text{ext}} \cup \mathcal{D}$ (recall: this is the set of all ground terms that can be formed using the function symbols in $\Pi_{\text{ext}} \cup \mathcal{D}$), 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 $<_{\text{tot}}$.

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 `#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 `#exists` and `#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.

# 3 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 / at most $k$ have to be true, are ubiquitous in modeling. For instance, consider the vertex cover problem: given a graph and a number $k$, decide whether

**Listing 3** STRIPS planning in *Bule*.

```
1  maxtime[T]            :: #ground time[0..T].
2  time[T], time[T+1]    :: #ground succ[T,T+1].
3
4  time[T], fluent[F]    :: #exists[0] true(T,F).
5  time[T], action[A]    :: #exists[0] do(T,A).
6
7  fluent[F],  init[F] ::  true(0,F).
8  fluent[F], ~init[F] :: ~true(0,F).
9  maxtime[T], goal[F] ::  true(T,F).
10
11 time[T]               :: action[A] : do(T,A).
12 time[T], action[A1], action[A2], A1 < A2 :: ~do(T,A1) | ~do(T,A2).
13
14 time[T], pre[A,F]     :: do(T,A) ->  true(T,F).
15 succ[T,U], neg[A,F]   :: do(T,A) -> ~true(U,F).
16 succ[T,U], pos[A,F]   :: do(T,A) ->  true(U,F).
17 succ[T,U], action[A], fluent[F], ~neg[A,F], ~pos[A,F] ::
18            do(T,A) & true(T,F) ->  true(U,F).
19 succ[T,U], action[A], fluent[F], ~neg[A,F], ~pos[A,F] ::
20            do(T,A) & ~true(T,F) -> ~true(U,F).
```

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 1000000ms.

| family | #instances | ∅var | ∅clauses | ∅ground. time (ms) | ∅solv. time (ms) |
|---|---|---|---|---|---|
| gttt | 180 | 468 | 1690 | 19 | 13145 |
| gttt-iterative | 27 | 400 | 1587 | 23 | 14701 |
| hex-hein | 34 | 1552 | 17687 | 42757 | 22613 |
| strips-direct | 325 | 1291 | 103172 | 751 | 242 |
| strips-binary | 668 | 648 | 41638 | 336 | 103671 + 30 TO |
| argumentation | 240 | 5160 | 14744 | 3853 | 1474 + 90 TO |

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 $\Pi_2^p$-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 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


1 Krzysztof R. Apt, Howard A. Blair, and Adrian Walker. Towards a theory of declarative knowledge. In *Foundations of Deductive Databases and Logic Programming*, pages 89–148. Morgan Kaufmann, 1988. doi:10.1016/b978-0-934613-40-8.50006-3.

2 Fahiem Bacchus. The AIPS '00 planning competition. *AI Mag.*, 22(3):47–56, 2001. URL: https://ojs.aaai.org/index.php/aimagazine/article/view/1571.

3 Olivier Bailleux and Yacine Boufkhad. Efficient CNF Encoding of Boolean Cardinality Constraints. In F. Rossi, editor, *9th International Conference on Principles and Practice of Constraint Programming*, volume 2833 of *Lecture Notes in Computer Science*, pages 108–122. Springer, 2003.

4 Olaf Beyersdorff, Mikolás Janota, Florian Lonsing, and Martina Seidl. Quantified boolean formulas. In Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, editors, *Handbook of Satisfiability*, chapter 31, pages 1177–1221. IOS Press, 2021.

5 Stefano Bistarelli, Lars Kotthoff, Francesco Santini, and Carlo Taticchi. Summary report for the third international competition on computational models of argumentation. *AI Mag.*, 42(3):70–73, 2021. doi:10.1609/aimag.v42i3.15109.

6 Uwe Bubeck and Hans Kleine Büning. Models and quantifier elimination for quantified horn formulas. *Discret. Appl. Math.*, 156(10):1606–1622, 2008. doi:10.1016/j.dam.2007.10.005.

7 Tom Bylander. The computational complexity of propositional STRIPS planning. *Artif. Intell.*, 69(1-2):165–204, 1994. doi:10.1016/0004-3702(94)90081-7.

8 Michael Cashmore, Maria Fox, and Enrico Giunchiglia. Planning as quantified boolean formula. In *ECAI 2012 - 20th European Conference on Artificial Intelligence.*, volume 242 of *Frontiers in Artificial Intelligence and Applications*, pages 217–222. IOS Press, 2012. doi:10.3233/978-1-61499-098-7-217.

9 Broes De Cat, Bart Bogaerts, Maurice Bruynooghe, Gerda Janssens, and Marc Denecker. Predicate logic as a modeling language: the IDP system. In *Declarative Logic Programming: Theory, Systems, and Applications*, pages 279–323. ACM / Morgan & Claypool, 2018. doi:10.1145/3191315.3191321.

10 Stefano Ceri, Georg Gottlob, and Letizia Tanca. What you always wanted to know about datalog (and never dared to ask). *IEEE Trans. Knowl. Data Eng.*, 1(1):146–166, 1989. doi:10.1109/69.43410.

11 Evgeny Dantsin, Thomas Eiter, Georg Gottlob, and Andrei Voronkov. Complexity and expressive power of logic programming. *ACM Comput. Surv.*, 33(3):374–425, 2001. doi:10.1145/502807.502810.

12 Phan Minh Dung. On the acceptability of arguments and its fundamental role in nonmonotonic reasoning, logic programming and n-person games. *Artif. Intell.*, 77(2):321–358, 1995. doi:10.1016/0004-3702(94)00041-X.

13 Paul E. Dunne and Trevor J. M. Bench-Capon. Coherence in finite argument systems. *Artif. Intell.*, 141(1/2):187–203, 2002. doi:10.1016/S0004-3702(02)00261-8.

14 Thomas Eiter, Georg Gottlob, and Heikki Mannila. Disjunctive datalog. *ACM Trans. Database Syst.*, 22(3):364–418, 1997. doi:10.1145/261124.261126.

15 Jorge Fandinno, François Laferrière, Javier Romero, Torsten Schaub, and Tran Cao Son. Planning with incomplete information in quantified answer set programming. *Theory Pract. Log. Program.*, 21(5):663–679, 2021. doi:10.1017/S1471068421000259.

16 Paolo Ferraris and Vladimir Lifschitz. Mathematical foundations of answer set programming. In *We Will Show Them! Essays in Honour of Dov Gabbay, Volume One*, pages 615–664. College Publications, 2005.

17 Richard E. Fikes and Nils J. Nilsson. Strips: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2:189–208, 1971.

18 Martin Gebser, Amelia Harrison, Roland Kaminski, Vladimir Lifschitz, and Torsten Schaub. Abstract gringo. *Theory Pract. Log. Program.*, 15(4-5):449–463, 2015. `doi:10.1017/S1471068415000150`.

19 Martin Gebser, Tomi Janhunen, Roland Kaminski, Torsten Schaub, and Shahab Tasharrofi. Writing declarative specifications for clauses. In *15th European Conference on Logics in Artificial Intelligence JELIA*, volume 10021 of *Lecture Notes in Computer Science*, pages 256–271, 2016. `doi:10.1007/978-3-319-48758-8_17`.

20 Martin Gebser, Roland Kaminski, Benjamin Kaufmann, and Torsten Schaub. *Answer Set Solving in Practice*. Synthesis Lectures on Artificial Intelligence and Machine Learning. Morgan & Claypool Publishers, 2012.

21 E. Giunchiglia, M. Narizzano, L. Pulina, and A. Tacchella. Quantified Boolean Formulas satisfiability library (QBFLIB), 2005. URL: `www.qbflib.org`.

22 Alexey Ignatiev, Antonio Morgado, and Joao Marques-Silva. PySAT: A Python toolkit for prototyping with SAT oracles. In *International Conference on Theory and Applications of Satisfiability Testing (SAT)*, pages 428–437, 2018.

23 Yuliya Lierler and Vladimir Lifschitz. One more decidable class of finitely ground programs. In *25th International Conference on Logic Programming (ICLP)*, volume 5649 of *Lecture Notes in Computer Science*, pages 489–493. Springer, 2009. `doi:10.1007/978-3-642-02846-5_40`.

24 Valentin Mayer-Eichberger and Abdallah Saffidine. Positional games and QBF: the corrective encoding. In Luca Pulina and Martina Seidl, editors, *International Conference on Theory and Applications of Satisfiability Testing (SAT)*, volume 12178 of *Lecture Notes in Computer Science*, pages 447–463. Springer, 2020. `doi:10.1007/978-3-030-51825-7_31`.

25 David G. Mitchell and Eugenia Ternovska. A framework for representing and solving NP search problems. In *Twentieth National Conference on Artificial Intelligence (AAAI)*, pages 430–435. AAAI Press / The MIT Press, 2005. URL: `http://www.aaai.org/Library/AAAI/2005/aaai05-068.php`.

26 Ruzica Piskac, Leonardo de Moura, and Nikolaj Bjørner. Deciding effectively propositional logic with equality. Technical Report MSR-TR-2008-181, Microsoft Research, December 2008. URL: `https://www.microsoft.com/en-us/research/publication/deciding-effectively-propositional-logic-with-equality/`.

27 Irfansha Shaik and Jaco van de Pol. Classical planning as QBF without grounding (extended version). *CoRR*, abs/2106.10138, 2021. `arXiv:2106.10138`.

28 Ankit Shukla, Armin Biere, Luca Pulina, and Martina Seidl. A survey on applications of quantified boolean formulas. In *31st IEEE International Conference on Tools with Artificial Intelligence, ICTAI*, pages 78–84. IEEE, 2019. `doi:10.1109/ICTAI.2019.00020`.

29 Carsten Sinz. Towards an optimal CNF encoding of boolean cardinality constraints. In *11th International Conference on Principles and Practice of Constraint Programming (CP)*, pages 827–831, Sitges, Spain, October 2005.

30 Joost P. Warners. A Linear-Time Transformation of Linear Inequalities into Conjunctive Normal Form. *Information Processing Letters*, 68(2):63–69, 1998.

31 Johan Wittocx, Maarten Mariën, and Marc Denecker. Grounding FO and FO(ID) with bounds. *J. Artif. Intell. Res.*, 38:223–269, 2010. `doi:10.1613/jair.2980`.

# 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 $\mathsf{A} = (\mathsf{Arg}, \mathsf{Att})$, where $\mathsf{Arg}$ is a set of *arguments* and $\mathsf{Att} \subseteq \mathsf{Arg} \times \mathsf{Arg}$ 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* $\mathsf{A}$ is a set $E \subseteq \mathsf{Arg}$ such that:

- $E$ is *conflict-free*, that is, there is no $(e, f) \in \mathsf{Att}$ with $e, f \in E$;
- $E$ is *admissible*, that is, for every $e \in E$, and every $(f, e) \in \mathsf{Att}$, there is an $g \in E$ with $(g, f) \in \mathsf{Att}$;
- $E$ 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 $\mathsf{A} = (\mathsf{Arg}, \mathsf{Att})$ and $a \in \mathsf{Arg}$
- **question:** Is $a \in E$ for all preferred extensions $E$ of $\mathsf{A}$?

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).

```
%% 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 (and (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)],
  neg[move(B,X,Y), on(B,X)], neg[move(B,X,Y), clear(Y)],
  pos[move(B,X,Y), on(B,Y)], pos[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)],
  neg[stack(A,X,B), on(A,X)], neg[stack(A,X,B), clear(B)],
  pos[stack(A,X,B), on(A,B)], pos[stack(A,X,B), clear(X)].

objects[A,block], objects[B,block], objects[Y,object] :: #ground action[
    unstack(A,B,Y)],
  pre[unstack(A,B,Y), on(A,B)], pre[unstack(A,B,Y), clear(Y)], pre[
    unstack(A,B,Y), clear(A)],
  neg[unstack(A,B,Y), on(A,B)], neg[unstack(A,B,Y), clear(Y)],
  pos[unstack(A,B,Y), on(A,Y)], pos[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.

```
1  fluent[F]  :: #ground reach_fluent[g,F].
2  maxtime[T] :: #ground reach_length[id,T].
3                #ground reach_choose[id,direct]. % other option is "binary"
4
5  reach_init[ID,_,T], fluent[F],  init[F] ::
6       reach_test(ID,init(T)) ->  reach_state(ID,T,F).
7  reach_init[ID,_,T], fluent[F], ~init[F] ::
8       reach_test(ID,init(T)) -> ~reach_state(ID,T,F).
9
10 reach_goal[ID,_,T], goal[F] ::
11      reach_test(ID,goal(T)) -> reach_state(ID,T,F).
12
13 reach_succ[ID,Q,T,U], action[A] :: #exists[Q] do(T,A).
14
15 reach_succ[ID,_,T,U] :: action[A] : do(T,A).
16 reach_succ[ID,_,T,U], action[A1], action[A2], A1 < A2 ::
17      ~do(T,A1) | ~do(T,A2).
18
19 reach_succ[ID,_,T,U], pre[A,F] ::
20      reach_test(ID,succ(T,U)) & do(T,A) ->  reach_state(ID,T,F).
21 reach_succ[ID,_,T,U], neg[A,F] ::
22      reach_test(ID,succ(T,U)) & do(T,A) -> ~reach_state(ID,U,F).
23 reach_succ[ID,_,T,U], pos[A,F] ::
24      reach_test(ID,succ(T,U)) & do(T,A) ->  reach_state(ID,U,F).
25 reach_succ[ID,_,T,U], action[A], fluent[F], ~neg[A,F], ~pos[A,F] ::
26      reach_test(ID,succ(T,U)) & do(T,A) &  reach_state(ID,T,F) ->
27          reach_state(ID,U,F).
28 reach_succ[ID,_,T,U], action[A], fluent[F], ~neg[A,F], ~pos[A,F] ::
29      reach_test(ID,succ(T,U)) & do(T,A) & ~reach_state(ID,T,F) ->
30          ~reach_state(ID,U,F).
```

**Listing 6** *Bule* model for (the complement of) sceptical inference w.r.t. preferred semantics.

```

% Quantifier Declaration
arg[X]   :: #exists[1] e(X).
arg[X]   :: #exists[1] attacked(X).
arg[X]   :: #forall[2] f(X).
arg[X]   :: #exists[3] attackedF(X).
arg[X]   :: #exists[3] cheatCF(X).
att[X,Y] :: #exists[3] cheatA(X,Y).
arg[X]   :: #exists[3] cheatM(X).
         :: #exists[3] cheat.

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

% 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).
```