Proceedings of the Sixteenth International Conference on
Principles of Knowledge Representation and Reasoning (KR 2018)

SOGrounder: Modelling and Solving Second-Order Logic

Matthias van der Hallen, Gerda Janssens
KU Leuven: Department of Computer Science
Celestijnenlaan 200A, 3001 Heverlee (Leuven)
Belgium
firstname.lastname@kuleuven.be

Abstract

KR languages based on logic have repeatedly proven their usefulness, e.g. Horn clauses, ASP, or FO logic. As the demand for more powerful and rich KR languages grows, we propose a KR language based on Second-Order (SO) Logic. As SO Logic is more expressive than SAT or ground (disjunctive) ASP, we propose Quantified Boolean Formulas (QBF) as a target language for our specifications. In this paper, we describe SOGrounder, a system that can ground SO Logic specifications to QBF. We start with a basic approach and suggest further techniques for reducing grounding, for example by introducing *Binary Quantification* as a language construct that takes two formules: One formula generating variable instantiations and one formula that must be instantiated. Finally, we show how to model a real world problem that is not reducible to first-order, and evaluate the performance of SOGrounder w.r.t. grounding time, grounding size and solving time as compared to existing encodings.

Introduction

In the spirit of Kowalski's seminal paper *Predicate Logic as a Programming Language* (Kowalski 1974), many KR Languages are based on logic: some are propositional, some First-Order. In recent years, ASP has clearly shown the applicability of modelling problems in logic and handing them to a solver on real-world business use cases: e.g. planning robot tasks in warehouses (Gebser et al. 2018).

Yet, there is a demand for more powerful and richer KR Languages. To this end, we propose Second-Order (SO) Logic as a modelling language. SO Logic allows many more problems to be expressed than ASP, FO or propositional logic; it has a descriptive complexity of PH (Immerman 1999). *SAT* and *ground (disjunctive) ASP* however have a descriptive complexity of NP and Σ_2^p respectively. As such, it is infeasible to ground SO to SAT or ground (disjunctive) ASP. Instead, we propose to ground to *Quantified Boolean Formulas* (QBF), which have a descriptive complexity of PSPACE. To our knowledge, the only other ground-and-solve tool with SO as a modelling language is SAT-TO-SAT (Bogaerts, Janhunen, and Tasharrofi 2016) and its grounder SO2GROUNDER. Our system improves on SAT-TO-SAT by:

- Lifting a syntactical restriction: After the first FO quantification $\mathcal{Q}x$, SO2GROUNDER no longer allows SO quantifications of a different quantifier $\overline{\mathcal{Q}}$.
- Interfacing with arbitrary QBF solvers using the widely accepted QDimacs format.

Second-Order Logic

In this section we will introduce SO logic, and show how we intend to use it as a modelling language.

Variables are symbols representing either an element of the domain $\mathcal{D}$, a predicate p/n over n domain elements, or a function f/n mapping n domain elements to a domain element. We call n the *arity* of the predicate or function.

When a variable represents a domain element, we call it a *first-order (FO)* variable. Variables representing either a predicate or function are called *second-order (SO)* variables.

Terms are either first-order variables, or the application of an n-ary function symbol f or a variable representing such a function to n terms.

Atoms are the application of an n-ary predicate symbol p or a variable representing such a predicate to n terms.

Formulas are defined inductively using the following rules:

- All atoms are formulas
- The negation ($\neg$) of a formula is a formula.
- If ϕ and ψ are formulas, the conjunction ($\wedge$), disjunction ($\vee$), implication ($\Rightarrow$) and equivalence ($\Leftrightarrow$) of ϕ and ψ are formulas, with $\phi \Rightarrow \psi$ and $\phi \Leftrightarrow \psi$ shorthand for $\neg\phi \vee \psi$ and $(\neg\phi \vee \psi) \wedge (\phi \vee \neg\psi)$ respectively.
- If x is a variable, and ϕ is a formula, then $\exists x : \phi$ and $\forall x : \phi$ are formulas. Note that no restrictions are imposed w.r.t. whether x is a first-order or second-order variable.

An example of a second order formula is $\exists f : \forall x : \forall y : x \neq y \Rightarrow f(x) \neq f(y)$, where $\neq$ (and $=$) is a preinterpreted predicate. For modelling convenience however, we can introduce *types* to second-order logic. Types are introduced by associating with each domain element a certain base type $\mathcal{T}_i$. Predicates and functions are associated with types $(\mathcal{T}_0, \ldots, \mathcal{T}_n)$ and $(\mathcal{T}_0, \ldots, \mathcal{T}_n) \mapsto \mathcal{T}$ respectively. Variables are typed corresponding to the symbol they represent, and we write $x :: \mathcal{T}$ to mean x is typed as $\mathcal{T}$.

We say a formula is properly typed if all of its subformulas are properly typed. An application of a predicate (function) p (f) is properly typed if the predicate (function) has type $(\mathcal{T}_0, \dots, \mathcal{T}_n)$ ($(\mathcal{T}_0, \dots, \mathcal{T}_n) \mapsto \mathcal{T}$) and its arguments x_1 to x_n have types $\mathcal{T}_1$ to $\mathcal{T}_n$. With typing, we could introduce the types *Color* and *Country*, and the above example formula could become: $\exists f :: ((Country) \mapsto Color) : \forall x :: Country : \forall y :: Country : x \neq y \Rightarrow f(x) \neq f(y)$.

SO Logic as a Modelling Language

Every Second-Order Logic modelling consists of three main parts: The *vocabulary*, the *structure* and the *theory*.

Vocabulary The *vocabulary* defines the types used in the remainder of the modelling. It also allows the declaration of predicates and functions. If these predicates or functions are not subsequently defined in the structure, they are understood to be implicitly quantified existentially on the outermost level, being in scope for the entire *theory*.

Structure The *structure* can define certain predicates and functions that were declared in the *vocabulary*. As such, it is the perfect place to put instance-specific information. Currently, only *two-valued* structures are allowed, i.e., a structure must always fully specify which tuples are true/false for the predicates and functions it defines.

Theory The *theory* consists of a set of properly typed SO formulas with no *free* (unquantified) *variables* . We call these formulas *sentences*. For a theory to be satisfied, every *sentence* in the theory must be satisfied.

Strategic Companies We now define the *Strategic Companies* problem, a well known example of a Σ_2^p-hard (Cadoli, Eiter, and Gottlob 1997) problem. It models the conundrum of a large holding owning multiple companies, forced to downsize by selling off a company. Of course, the company wants to minimize the impact of selling this company. Specifically, two conditions have to be met:

1. The sale should not impact the holdings portfolio, i.e. it should produce the same set of goods.
2. Some sets of companies within the holding together own another company, this is called a *controlling set*. As such, it is possible to sell of a company while retaining control of it through a controlling set. In this case, the holding is not allowed to 'downsize' by selling this company.

The Strategic Companies problem was featured in the ASP Competition 2013[1], where it was presented in the following way: We call a set of companies C controlled by the holding a *strategic set* if it is a subset-minimal set that (1) produces all goods and (2) is closed under ownership through controlling sets. Given two distinct companies c, c', determine whether they form a *strategic pair*, i.e., whether a strategic set containing both c and c' exists. One additional restriction imposes that every controlling set contains at most 4 companies.

Listing 1 shows our modelling of the Strategic Companies problem. In lines 1-6 we specify the *vocabulary*, containing the `Company` and `Good` types, together with predicates to represent the *controlling set*s (`cont`), who produces what (`prod`), the strategic set (`ss`) and the strategic pair (`sp`). Lines 8-10 continue by specifying the *structure*, which contains instance specific data. We conclude with lines 12-15, specifying the *theory*: The strategic set must contain the strategic pair (line 12), it must produce all goods (line 13), be closed under ownership through controlling sets (line 14) and it must be subset-minimal (line 15): no *strict subset* of `ss` must exist for which conditions (1) and (2) hold. Note that this model contains two second order quantifications: $\exists$`ss` (implicitly) in the vocabulary and $\neg\exists$`ss'` on Line 15.

[1]https://www.mat.unical.it/aspcomp2013/OfficialProblemSuite

QBF

A *quantified boolean formula* (QBF) is a formula in propositional logic where variables can be quantified either existentially or universally. Quantifiers can alternate indefinitely and it is this property that makes deciding satisfiability for QBF PSPACE-complete, whereas deciding satisfiability for (unquantified) SAT formulas is NP-complete.

When each variable is quantified at the *beginning* of the formula, we say that it is in *prenex* form. Such formulas can be written as $\mathcal{Q}_1\overline{x}_1 \dots \mathcal{Q}_n\overline{x}_n.\phi$ with $\mathcal{Q}_1, \mathcal{Q}_n$ quantifiers. We generally call $\mathcal{Q}_1\overline{x}_1 \dots \mathcal{Q}_n\overline{x}_n$ the *quantifier prefix*, also written $\hat{\mathcal{Q}}$. We group subsequent quantifications of the same quantifier into quantifier *blocks* and define *level($\mathcal{Q}_i$)* to be the number of quantifier blocks preceding it. As such, the *highest* level or *innermost* quantification block is the one most to the right in the formula.

When the formula ϕ in $\hat{\mathcal{Q}}.\phi$ is in *Conjunctive Normal Form* (CNF) we say the QBF formula is in *Prenex Conjunctive Normal Form* (PCNF). QDimacs, the input encoding which most QBF solvers accept, is in fact a textual representation of a QBF formula in PCNF.

Implementation

We now discuss the implementation of a grounder from SO to QBF. Our initial approach proceeds, in order, as follows:

Push Negations Using the rules $\neg\exists x.\phi \rightsquigarrow \forall x.\neg\phi$, $\neg(\phi \land \psi) \rightsquigarrow \neg\phi \lor \neg\psi$, etc. we ensure that negations only appear in front of atoms.

Unnesting Using the rule $f(g(\overline{x})) = z \rightsquigarrow \exists y.f(y) = z \land g(\overline{x}) = y$, we remove function applications from positions where terms are expected.

Graphing Using the rules $f(\overline{x}) = y \rightsquigarrow f'(\overline{x}, y)$ and $\exists(\forall)f : \phi \rightsquigarrow \exists(\forall)f' : F(f') \land (\Rightarrow)\phi$ with $F(f') \equiv \forall\overline{x} : \exists y : f'(\overline{x}, y) \land \forall y' : (f'(\overline{x}, y') \Rightarrow y = y')$ we transform functions f/n into predicates $f'/n+1$ with existence and uniqueness constraints.

Normalization Using the rules $\phi \Rightarrow \psi \rightsquigarrow \neg\phi \lor \psi$, $\phi \Leftrightarrow \psi \rightsquigarrow (\neg\psi \lor \phi) \land (\neg\psi \lor \phi)$ and $\phi \Leftrightarrow \psi \rightsquigarrow (\phi \land \psi) \lor (\neg\phi \land \neg\psi)$, we eliminate $\Rightarrow$,$\Leftrightarrow$. We choose between the two rules for equivalences based on the type of the enclosing connective.

FO Grounding Using the rule $\exists(\forall)x : \phi \rightsquigarrow \bigvee_{t \in \mathcal{D}_x} (\bigwedge_{t \in \mathcal{D}_x}) : \phi[x/t]$ with x an FO variable with domain $\mathcal{D}_x$, we instantiate all *first order* quantifications.

Listing 1: SO model for Strategic Companies

```
type Company = {Barilla;Dececco;Callippo;Star}             # All companies
type Good    = {Pasta;Tonno}                               # All products
cont         :: (Company, Company, Company, Company, Company) # Controls: last is controlled
prod         :: (Company, Good)                            # Produces
ss           :: (Company)                                  # Strategic Set
sp           :: (Company, Company)                         # Strategic pairs

prod = {Barilla,Pasta; Dececco,Pasta; Callippo,Tonno; Star,Tonno}
cont = {Star,Star,Star,Star,Barilla; Barilla,Barilla,Barilla,Barilla,Dececco}
sp   = {Barilla,Callippo}

∀c :: Company : ∀c1 :: Company : sp(c,c1) ⇒ (ss(c) ∧ ss(c1)).
∀g :: Good : ∃p :: Company : ss(p) & prod(p, g).
∀c :: Company : (∃o1 :: Company : ∃o2 :: Company : ∃o3 :: Company : ∃o4 :: Company : ss(o1) ∧ ss(o2) ∧ ss(o3) ∧
     ss(o4) ∧ cont(o1,o2,o3,o4,c)) ⇒ ss(c).
¬(∃ss' :: (Company) : (ss' ≠ ss) ∧ (∀c :: Company : ss'(c) ⇒ ss(c)) ∧ (∀g :: Good : ∃p :: Company : ss'(p) &
     prod(p, g)) ∧ (∀c :: Company : (∃o1 :: Company : ∃o2 :: Company : ∃o3 :: Company : ∃o4 :: Company : ss'(o1)
     ∧ ss'(o2) ∧ ss'(o3) ∧ ss'(o4) ∧ cont(o1,o2,o3,o4,c)) ⇒ ss'(c))).
```

Unique Names We introduce unique names for every remaining (*SO*) quantification, e.g.: $\forall f : \phi \wedge \exists f : \forall g : f(x) \vee \psi \rightsquigarrow \forall f : \phi \wedge \exists f' : \forall g : f'(x) \vee \psi[f/f']$.

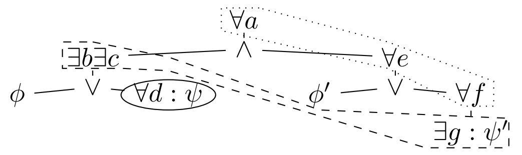
Prenex Form We pull all SO quantifiers to the front by applying the following two rules. Note that these rules only hold when variable α does not appear in ψ, a condition we have preemptively satisfied due to the **Unique Names** transformation and the restriction of sentences to properly typed SO formulas with *no free variables*.

$$(\exists\alpha : \phi) \wedge (\vee)\psi \rightsquigarrow \exists\alpha : \phi \wedge (\vee)\psi \qquad (1\exists)$$

$$(\forall\alpha : \phi) \wedge (\vee)\psi \rightsquigarrow \forall\alpha : \phi \wedge (\vee)\psi \qquad (2\forall)$$

To minimise the number of quantifier alternations, we switch between applying rule (1∃) and rule (2∀) only when we cannot further apply the active rule. As visualised below, $\forall a : (\exists b : \exists c : (\phi \vee \forall d : \psi)) \wedge (\forall e : \phi' \vee \forall f : \exists g : \psi')$ becomes $\forall a, e, f : \exists b, c, g : \forall d : (\phi \vee \psi) \wedge (\phi' \vee \psi')$.

Note how $\forall e$ and $\forall f$ are pulled to the level of $\forall a$ by rule (2∀), whereas $\forall d$ is *blocked* from being pulled to that level by the quantifications $\exists b$ and $\exists c$, as no rule allows switching the order of $\forall$ and $\exists$ quantifications.



Tseitinize The resulting prenex formula must be flattened to *Prenex Conjunctive Normal Form* (PCNF). We can do this by introducing a so called Tseitin variable for every nested subformula, e.g.: $Q_1 x : Q_2 y : \phi \wedge (\psi \vee (\chi \wedge \rho)) \rightsquigarrow Q_1 x : Q_2 y : \phi \wedge \exists T_i : (\psi \vee T_i) \wedge (T_i \Leftrightarrow (\chi \wedge \rho))$. After pulling the existential quantification of the Tseitin variables to the front and rewriting the equivalences using the **Normalization** rule, we obtain PCNF.

Grounding SO We replace every predicate atom $p(\overline{x})$ with a proposition $p_{\overline{x}}$ and quantifications $\forall(\exists)p$ with quantifications $\forall(\exists)p_{\overline{x_0}}, \ldots, \forall(\exists)p_{\overline{x_n}}$ with $\overline{x_i}$ in the domain $\mathcal{D}_p$ of p, e.g. $\forall p : \phi \wedge p(1)$ with the domain $\mathcal{D}_p = \{1, 2\}$ would become $\forall p_1 : \forall p_2 : \phi[p(1)/p_1, p(2)/p_2] \wedge p_1$.

Advanced grounding techniques

In this section we will detail some improvements on the grounding process sketched above.

In the grounding process above, our **FO Grounding** transformation grounds every formula. However, when we know the truth value of certain (sub)formulas such as predicate atoms $p(\overline{x})$, we can replace the (sub)formula by the truth value and propagate this, instead of grounding the (known) formula. This improvement is known as Reduced Grounding or RED (de Cat et al. 2013). Although this is a simple technique based on *substitution* and *simplification*, it can be powerful. This technique is implemented on top of the grounding process above by SOGrounder.

Grounding With Bounds (Wittocx, Mariën, and Denecker 2010), or GWB, further improves upon RED by maintaining a symbolic representation for every formula in the theory, e.g. using Binary Decision Diagrams or BDDs. These symbolic representations allow for querying the instances which are known to be true (false), known as the *CT* (*CF*) bound of the formula. With this knowledge, when grounding for example a universal quantifier, we only need to introduce ground formulas for instances which are not known to be true, i.e. instances not in *CT*. Although SOGrounder does not support this improvement, it allows users to specify a generating formula for (a set of) quantifications by hand. We call this language construct the *binary quantification*, as it introduces a quantification taking *two* formulas; one formula generating variable instantiations and one being instantiated. E.g., `∀(x,y)::[graph(x,y)]:`ϕ instantiates ϕ only for those `x`,`y` for which `graph` holds. Note that the types of `x`,`y` can be derived from the type of `graph`. For strategic companies, this can simplify the nested quantification of `o1`, `o2`, `o3`, and `o4` to a single binary quantification.

One last improvement implemented by SOGrounder is the way Tseitin variables are introduced. The **Tseitinize** transformation described above will introduce every Tseitin literal at the innermost quantification level (Giunchiglia, Marin, and Narizzano 2009). However, it is possible to quantify the Tseitin variable T at a lower quantification level, as long as every variable v_i within the Tseitinized formula of T is quantified in the same quantifier block or earlier as T. Recent QBF research suggests that this has a large impact on search efficiency (Beyersdorff, Chew, and Janota 2016). We also use *polarity optimization* (Plaisted and Greenbaum 1986) which takes the polarity of the Tseitinized formula into account to reduce the number of clauses introduced.

Experiments

To evaluate SOGrounder's performance, we use the model of the *Strategic Companies* problem (Listing 1)[2], modified to benefit from *binary quantification* as discussed above.

To generate problem instances parametrized by the number of companies, we have used the method described in Maratea et al. (2008). The resulting instances are comparable in size and difficulty to the benchmark set present in previous QBF solver competitions.[3] Using our model and SOGrounder, we have generated QBF encodings, for which the grounding times and sizes are reported in Table 1. We have also used the instantiation scheme (IS) from Maratea et al. to generate QBF encodings. We compare the resulting grounding size with that from SOGrounder by reporting the number of literals and clauses. Regarding solving, we compare both resulting QBF encodings, solved using DepQBF (Lonsing and Biere 2010) in Table 2. Note that as our tool generates a qDimacs file, we can use many other solvers instead of DepQBF. As a verification, and motivated to also compare with state-of-the-art solvers in other paradigms, we also report grounding and solving times for the ASP solver Clingo in the respective tables. For Clingo, we use an existing ASP encoding that employs *saturation* (Eiter and Gottlob 1995) to encode the Σ_2^p-hard parts of the problem in ASP.

#Companies	Grounding time (ms)		Lits		Clauses	
	SOGrounder	Clingo	SOGrounder	IS	SOGrounder	IS
14	220	8	282	436	1,054	1,319
29	266	16	582	901	2,179	2,729
38	322	22	762	1,180	2,854	3,575
54	409	38	1,082	1,676	4,054	5,079
77	626	65	1,542	2,389	5,779	7,241
82	684	67	1,642	2,544	6,154	7,711
94	952	83	1,882	2,916	7,054	8,839
100	867	92	2,002	3,102	7,504	9,403

Table 1: Overview of grounding times and sizes for strategic companies of size n.

It is clear from the results in Table 1 that SOGrounder produces smaller groundings than those produced by IS. One contributing factor is the advanced Tseitinization process described above. Another contributing factor are the tools used by IS, and their choice to model the dual problem s.t. only two quantifier blocks are introduced. It is this

[2]Binaries and experiments available at https://bit.ly/2rRckXi

[3]http://qbflib.org/suite_detail.php?suiteId=19 on 17/5/2018.

Solver	Solving time (ms)							
	14	29	38	54	77	82	94	100
DepQBF (QDimacs)	19	38	55	160	4,980	6,725	14,546	16,533
IS	24	80	161	1,291	20,078	15,388	31,714	42,311
Clingo	11	41	85	340	2,522	3,202	7,354	9,351
GhostQ (QCIR)	120	177	205	252	565	863	2,110	2,355

Table 2: Overview of solving times for SOGrounder and IS (QDimacs), Clingo (ASP), and GhostQ (QCIR).

smaller grounding that we identify as the main contributor for SOGrounder's better solving times w.r.t. IS. For Clingo, we expect its optimized *bottom-up* grounding and powerful lookback heuristics to account for its better ground and solving time.

To illustrate the effects of employing binary quantification, Table 3 reports grounding times (in ms) for SOGrounder with and without using binary quantifications (5 min. limit). Clearly, without such constructs, or techniques to derive them, grounding can scale very bad. Note that binary quantification does not affect the grounding size in this case.

Binary quantification	Grounding time (ms)						
	5	8	14	17	29	30	33
no	244	385	3329	8608	–	–	–
yes	204	228	220	230	266	270	285

Table 3: Overview of SOGrounder's grounding times (ms) with and without binary quantification, 5 min time limit.

The possible impact of Tseitinization is well-known within QBF research. In fact, it is suggested that its impact is much higher for QBF than for SAT, and recently alternative transformations (Klieber et al. 2013) and encodings such as QCIR (Jordan, Klieber, and Seidl 2016) that do not require PCNF (and thus, Tseitinization) have been developed. As it is very much possible to add QCIR output to SOGrounder, we modified SOGrounder to output QCIR encodings for this exact SO model specifically, and used the QCIR compatible GhostQ solver to judge the impact of translating to a more high-level QBF representation (Table 2).

It is clear from these that the impact of using a more high-level QBF representation is non-negligible in this case. However, further experiments will show whether this holds in general.

Conclusion and Future Work

SOGrounder is a tool which accepts SO Logic modellings and grounds them to QBF. It is clear from experiments that this yields a viable option for specifying and solving problems of a higher computational complexity than FO or even ASP. Its performance can match or even beat existing handwritten QBF and state-of-the-art ASP encodings.

Nevertheless, to ensure performance on the large range of possible problems, of varying complexity, we must expand our benchmark set and investigate how advanced techniques such as GWB and Lazy Grounding interact with Second Order and the underlying QBF solvers. We also want to further investigate the effects of Tseitinization in QBF, and build support for QBF encodings which do not require CNF, such as QCIR. Lastly, we want to compare with other, non ground-and-solve modelling languages supporting SO, such as ProB (Leuschel and Butler 2003).

Acknowledgements

Matthias van der Hallen is supported by a Ph.D. fellowship from the Research Foundation - Flanders (FWO - Vlaanderen).

References

Beyersdorff, O.; Chew, L.; and Janota, M. 2016. Extension variables in QBF resolution. In *AAAI Workshop: Beyond NP*, volume WS-16-05 of *AAAI Workshops*. AAAI Press.

Bogaerts, B.; Janhunen, T.; and Tasharrofi, S. 2016. Solving QBF instances with nested SAT solvers. In *AAAI Workshop: Beyond NP*, volume WS-16-05. AAAI Press.

Cadoli, M.; Eiter, T.; and Gottlob, G. 1997. Default logic as a query language. *IEEE Trans. Knowl. Data Eng.* 9(3):448–463.

de Cat, B.; Bogaerts, B.; Devriendt, J.; and Denecker, M. 2013. Model expansion in the presence of function symbols using constraint programming. In *ICTAI*, 1068–1075. IEEE Computer Society.

Eiter, T., and Gottlob, G. 1995. On the computational cost of disjunctive logic programming: Propositional case. *Ann. Math. Artif. Intell.* 15(3-4):289–323.

Gebser, M.; Obermeier, P.; Otto, T.; Schaub, T.; Sabuncu, O.; Nguyen, V.; and Son, T. C. 2018. Experimenting with robotic intra-logistics domains. *CoRR* abs/1804.10247.

Giunchiglia, E.; Marin, P.; and Narizzano, M. 2009. Reasoning with quantified boolean formulas. In *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*. IOS Press. 761–780.

Immerman, N. 1999. *Descriptive complexity*. Graduate texts in computer science. Springer.

Jordan, C.; Klieber, W.; and Seidl, M. 2016. Non-cnf QBF solving with QCIR. In *AAAI Workshop: Beyond NP*, volume WS-16-05 of *AAAI Workshops*. AAAI Press.

Klieber, W.; Janota, M.; Marques-Silva, J.; and Clarke, E. M. 2013. Solving QBF with free variables. In *CP*, volume 8124 of *Lecture Notes in Computer Science*, 415–431. Springer.

Kowalski, R. A. 1974. Predicate logic as programming language. In *IFIP Congress*, 569–574.

Leuschel, M., and Butler, M. J. 2003. Prob: A model checker for B. In *FME*, volume 2805 of *Lecture Notes in Computer Science*, 855–874. Springer.

Lonsing, F., and Biere, A. 2010. Depqbf: A dependency-aware QBF solver. *JSAT* 7(2-3):71–76.

Maratea, M.; Ricca, F.; Faber, W.; and Leone, N. 2008. Look-back techniques and heuristics in DLV: implementation, evaluation, and comparison to QBF solvers. *J. Algorithms* 63(1-3):70–89.

Plaisted, D. A., and Greenbaum, S. 1986. A structure-preserving clause form translation. *J. Symb. Comput.* 2(3):293–304.

Wittocx, J.; Mariën, M.; and Denecker, M. 2010. Grounding FO and FO(ID) with bounds. *J. Artif. Intell. Res.* 38:223–269.