Janhunen, Tomi

Answer Set Programming

Published in:
KUENSTLICHE INTELLIGENZ

DOI:
10.1007/s13218-018-0543-y

Published: 09/06/2018

Document Version
Peer reviewed version

Please cite the original version:
Janhunen, T. (2018). Answer Set Programming: Related with Other Solving Paradigms. *KUENSTLICHE INTELLIGENZ*, *32*(2-3), 125-131. https://doi.org/10.1007/s13218-018-0543-y

Künstliche Intelligenz manuscript No.
(will be inserted by the editor)

Answer Set Programming

Related with Other Solving Paradigms

Tomi Janhunen

Received: September 1, 2017 / Accepted: date

Abstract Answer set programming (ASP) is a declarative programming paradigm based on an interpretation of logical rules as constraints. In this article, we relate answer set programming with other constraint-based solving paradigms: Boolean satisfiability checking, satisfiability modulo theories, mixed integer programming, and constraint programming. We illustrate the relationship of ASP with these alternative paradigms in terms of simple examples, and identify the main primitives and characteristics of the constraint-based languages under consideration.

Keywords Declarative programming · Satisfiability checking · Constraint satisfaction · Theory reasoning

1 Introduction

Answer set programming (ASP) is an approach to declarative programming offering rich rule-based languages for modeling and efficient algorithms for the search of answer sets [5, 17]. These features enable one to solve a wide variety of challenging knowledge-intensive reasoning problems. In this article, it is assumed that the reader already understands the basics of ASP, as described in the other articles of this special issue or other introductory articles. The goal of this article is to relate ASP with other solving paradigms of similar nature. They resemble ASP in the sense that a problem is first formalized as a logical theory and after finding satisfying assignments for the theory, solutions to the problem can be extracted (see Figure 1) if satisfying assignments were found. The net effect is that the problem gets solved.

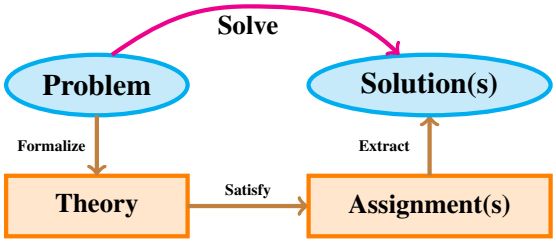

Fig. 1 Conceptual Model for Declarative Solving Paradigms

The rest of this article is organized as follows. In Section 2, we contrast ASP with *Boolean satisfiability* (SAT) checking which is perhaps the solving paradigm closest to ASP in a number of respects. Most importantly, the two paradigms are essentially based on two-valued (Boolean) variables and logic. The other paradigms of interest involve variables with larger domains such as the set of all integers. Traditional *constraint programming*, to be explained in Section 3, features variables with multi-valued domains and explicit constraints for the exclusion of incompatible variable assignments. On the other hand, the paradigms of *linear programming* (LP) and *mixed integer programming* (MIP), to be addressed in Section 4, are based on linear (in)equalities over infinite domains such as integers or reals. Yet another solving paradigm, *SAT modulo theories* (SMT),

The support from the Finnish Centre of Excellence in Computational Inference Research (COIN) funded by the Academy of Finland (under grant #251170) is gratefully acknowledged.

T. Janhunen
Department of Computer Science, Aalto University, Finland
E-mail: Tomi.Janhunen@aalto.fi

will be covered in Section 5. This approach extends SAT in terms of theories typically involving non-Boolean variables and, interestingly, analogous extensions to ASP have been proposed. Finally, this article is concluded by Section 6.

2 Boolean Satisfiability Checking

Boolean satisfiability checking (SAT) is a solving paradigm [3] highly analogous to ASP, but it is based on different syntax and semantics. In SAT checking, the syntax is essentially based on *formulas* in *propositional logic* whereas PROLOG-style rules are used in ASP. Propositional formulas are expressions formed from atomic propositions, also known as Boolean variables, using propositional connectives $\neg$ (*negation* "not"), $\wedge$ (*conjunction* "and"), $\vee$ (*disjunction* "or"), $\rightarrow$ (*implication* "if... then"), and $\leftrightarrow$ (*equivalence* "equals").

Example 1 Suppose that we have n processes to schedule and let r_i with $1 \leq i \leq n$ be an atomic proposition meaning that process i is scheduled for execution. Then the formula $r_1 \vee \ldots \vee r_n$ states that at least one process is running. The mutual exclusion of scheduling is expressible with formulas $\neg(r_i \wedge r_j)$ for each pair $\langle i, j \rangle$ of processes where $i < j$. □

For the sake of efficient implementation, a variety of normal forms have been developed for propositional formulas. Many of them are based on the connectives $\wedge$, $\vee$, and $\neg$ such as the two basic normal forms, namely the *conjunctive normal form* (CNF) and the *disjunctive normal form* (DNF). The input of a SAT solver is effectively a propositional formula in CNF: a conjunction $C_1 \wedge \ldots \wedge C_n$ of *clauses* C_i that are disjunctions $l_1 \vee \ldots \vee l_m$ of *literals*, i.e., atomic propositions a or their negations $\neg a$. Any propositional formula can be transformed into this form. Moreover, if the introduction of new atoms is allowed, the transformation is feasible in a compact (linear) way using the so-called Tseitin transformation [28]: subformulas are recursively defined using new atoms as their names and the actual normal form is derived as clauses resulting from the respective definitions.

The link between ASP and SAT relies on the fact that any propositional formula can be expressed by rules in ASP and we illustrate the case of CNF and DNF, or any *negation normal form* (NNF) based on literals, conjunction, and disjunction [3]. In classical logic, every propositional atom a is subject to a choice formalized by a choice rule

```
{a}.
```

where a was simply translated as an atom `a` into ASP. Any negative literal $\neg a$ can be expressed using default negation as `not a`. A logical disjunction $f_1 \vee \ldots \vee f_n$ of propositional formulas $f_1, \ldots, f_n$ can be captured by rules

```
f :- f1.  ...  f :- fn.
```

where `f` is a new atom acting as a name for the disjunction in question and $\mathtt{f}_1, \ldots, \mathtt{f}_n$ in the rule bodies act as names for the disjuncts. If a subformula f_i happens to be a literal then $\mathtt{f}_i$ can be replaced by its translation in the rule above. Any conjunction $f_1 \wedge \ldots \wedge f_n$ can be expressed with just one rule

```
f :- f1, ..., fn.
```

The forms of rules described above are sufficient for the evaluation of (arbitrary) propositional formulas. If `f` is the name expressing a formula f with rules, the satisfaction of f or $\neg f$ can be enforced using the respective ASP constraints:

```
:- not f.    :- f.
```

Example 2 The CNF $(p \vee \neg q) \wedge (q \vee r)$ translates into

```
{p;q;r}.  d1 :- p.  d1 :- not q.
d2 :- q.  d2 :- r.  c :- d1,d2.  :- not c.
```

However, it is not necessary to give new names for the subformulas of a formula that is already in CNF.

Example 3 The CNF from Example 2 is also captured by:

```
{p;q;r}.  :- not p,q.  :- not q,not r.
```

It should be clear by now how the basic propositional connectives $\neg$, $\vee$, and $\wedge$ are available in ASP. Since we cannot write more complex formulas based on them as such, we have to follow the guidelines of the translation presented above. In practice, when modeling with ASP, the intuitive readings of atomic propositions are picked on the fly and their logical relationships are described in terms of rules.

Listing 1 Encoding the Scheduling Condition

```
1  % Domain
2  proc(1..n).
3
4  % Rules
5  { run(P) } :- proc(P).
6  lb :- run(P), proc(P).
7  ub :- run(P1), run(P2),
8        proc(P1), proc(P2), P1<P2.
9
10 % Constraints
11 :- not lb.      :- ub.
```

For instance, Listing 1 provides an encoding of the formulas in Example 1. Line 2 defines the domain of processes parameterized by their number `n`. Running processes `P` are chosen (Line 5). The lower bound (atom `lb`) is inferred if at least one of the processes is running (Line 6), formalizing a disjunction. The upper bound (atom `ub`) is reached if any two processes `P1` and `P2` are running simultaneously (Lines 7–8), i.e., we have a disjunction of conjunctions. These rules work regardless the number of processes `n`. Line 11 enforces the satisfaction of constraints. For each `n`, the program of Listing 1 has exactly `n` answer sets so the number of answer sets grows linearly in `n`. The scaling of the program is affected by the grounding step where `P1` are `P2` are replaced by ordered pairs of numbers in the range `1..n`. This results in a quadratic blow-up in the length of the resulting ground program, which may degrade the performance of solving for larger values of `n`. However, such a blow-up can be avoided by introducing an explicit bound in order to limit the cardinality of the choice being made:

```
{ run(P): proc(P) } = 1.
```

Translations from ASP to SAT are of interest in their own right, since they enable one viable way to implement the search for answer sets using SAT solvers. Surprisingly, such translations are non-trivial, which can be understood from the semantic perspective. While the satisfaction of formulas is only of interest in SAT checking, the answer set semantics of ASP sets further requirements on top of satisfying rules as implications. In fact, there are formal results [16, 21] establishing the higher expressive power of rules under answer set semantics compared to propositional formulas subject to satisfaction by truth assignments. On one hand, this goes back to the *non-monotonicity* of reasoning in ASP, i.e., adding new pieces of information may give rise to retraction of the previous conclusions. For instance, the following rule *alone* prohibits all answer sets:

```
p :- not p.
```

However, the addition of the atom `p` as a fact cancels the rule above and establishes an answer set where `p` is true. Such effects cannot be perceived in SAT which is essentially based on classical *monotonic* logic: the consistency of a *theory* cannot be restored by adding formulas as above. Any faithful embedding of ASP into SAT is necessarily *non-modular* [16], i.e., it is impossible to translate logic programs rule-by-rule. Rules that mention a particular atom `a` in their head *define* `a` and they are possibly relevant when reasoning about `a`. Even further, any positively interdependent atoms must be treated together by translations into SAT. The next code snippet illustrates a typical use case of positive recursion.

Listing 2 Recursive Encoding for Deadlock Detection

```
 1  % Domain
 2  proc(1..n).
 3
 4  % Rules
 5  { wait(P1,P2): proc(P2), P1!=P2 } :-
 6    proc(P1).
 7  dep(P1,P2) :- wait(P1,P2),
 8    proc(P1), proc(P2).
 9  dep(P1,P3) :- wait(P1,P2), dep(P2,P3),
10    proc(P1), proc(P2), proc(P3).
11
12  % Constraints
13  :- run(P1), run(P2), dep(P1,P2),
14     proc(P1), proc(P2).
```

In Listing 2, we introduce `n` processes as before (Line 2). Each process may be waiting for any other processes to finish (Lines 5–6). This possibility creates dependencies between processes formalized by the recursive definition in Lines 7–8 and 9–10. The former gives the base case in terms of `wait/2` while the latter defines `dep/2` as the *transitive closure* of the `wait/2` relation. Finally, the constraint (Lines 13–14) excludes the possibility of a *deadlock*. For the sake of strong typing, we used the domain predicate `proc/2` systematically in the encoding although modern grounders are able to infer domains of variables on the fly. In this respect, we could omit the domains of variables specified in Lines 8, 10, and 14, since the required domain information can be propagated from the definition of `wait/2`. The definition of `dep/2` gives rise to strongly connected components (in terms of positive recursion) whose sizes grow quadratically in `n`. If such components are translated into SAT by the most compact translations [16], a logarithmic blow-up in `n` seems unavoidable, illustrating the difference in the expressive power of ASP and SAT. The blow-up is deemed exponential if new atoms are not allowed [21].

In summary, ASP and SAT share a lot when it comes to logical modeling and the typical use cases are much alike. Since ASP was established later than SAT checking, it is not a surprise that ASP solvers exploit many ideas from SAT solvers in particular. But there are already examples of knowledge transfer in the other direction as well, e.g., the answer set projection and enumeration techniques have been recently adopted for *space-efficient* model enumeration in SAT [10], a task for which SAT solvers were not originally designed. Rules deployed in ASP enable knowledge representation in a declarative and a symbolic way which can

be considered as a notable strength when modeling application problems. In contrast to this, the procedural generation of formulas or clauses is still prevalent in the area of SAT checking, potentially hiding the declarative reading of formulas. Moreover, the *closed world assumption* (CWA) plays an essential role in ASP grounders, also laying the foundation for efficient instantiation of answer set programs. The effects of CWA are two-fold. First, only objects explicitly mentioned by a program need to be considered when instantiating variables. Second, atomic statements are false by default, which means that the extensions of domain predicates used to control variable instantiation are as small as possible. In SAT checking, the premises for schematic programming are weaker since variable instantiation is controlled procedurally and the choice of values remains completely on the programmer's own responsibility unless the declarative approach of [13] is adopted. The answer set semantics has also been generalized for the full propositional language [9] but the approach has not gained popularity in ASP modeling so far. On the other hand, such nested structures can be translated away [26] if appropriate. As regards SAT checking, if the full propositional language is preferred in modeling, then the formulas generated can be readily translated into CNF using the Tseitin transformation [28]. For further insights into the interconnection of ASP and SAT, we refer the reader to a recent survey [20] as well as the historical translations from SAT to ASP [25] and back [22, 16].

3 Constraint Programming

In this section, we turn our attention to *constraint programming* (CP) [27] and discuss its relationship with SAT and ASP. The CP paradigm centers around solving *constraint satisfaction problems* (CSPs). Such a problem involves a finite collection of *variables* $X_1,\ldots,X_n$, a *domain* $D(X_i)$ for each variable X_i, and a set of *constraints* C.

Two things have to be defined for each constraint $c \in C$. The *scope* S_c of c is simply a subset of $\{X_1,\ldots,X_n\}$ and it defines the variables whose values are constrained by c. In addition, it is necessary to define when c is satisfied. More formally, the *satisfaction relation* associated with a constraint c is any subset of the Cartesian product $\prod_{X_i \in S_c} D(X_i)$. A *valuation* ν is a function that maps each variable X_i to an element in $D(X_i)$. The valuation ν is a *solution* to the CSP in question, if for each constraint c and the variables $Y_1,\ldots,Y_k$ in S_c, the tuple $\langle \nu(Y_1),\ldots,\nu(Y_k)\rangle$ belongs to R_c. A wide variety of general-purpose constraints have been introduced in the CP literature. Due to limited space, we will cover just a couple.

Listing 3 Encoding for Resource Allocation

```
1  % Domains
2  proc(1..n). core(1..m).
3
4  % Rules
5  { assign(P,C): core(C) } = 1 :- proc(P).
6
7  % Constraints
8  :- assign(P1,C), assign(P2,C),
9     proc(P1), proc(P2), P1<P2, core(C).
```

Example 4 Recalling the n processes from our preceding examples, let us introduce variables $E_1,\ldots,E_n$ to model the allocation of the processes on m cores denoted $c_1,\ldots,c_m$. Thus, let the domain $D(E_i) = \{c_1,\ldots,c_m\}$ for each E_i. For the sake of resource allocation, it would be natural to impose the *all-different* constraint $\textsc{AllDiff}(E_1,\ldots,E_n)$. This is to express that each process is assigned to a distinct core for execution. For instance if $n = 2$ and $m = 3$, then we have $D(E_1) = D(E_2) = \{c_1,\ldots,c_3\}$. The valuation ν such that $\nu(E_1) = c_3$ and $\nu(E_2) = c_2$ gives one solution to the respective CSP, since the constraint $\textsc{AllDiff}(E_1,E_2) =$

$$\{\langle c_1,c_2\rangle,\langle c_1,c_3\rangle,\langle c_2,c_1\rangle,\langle c_2,c_3\rangle,\langle c_3,c_1\rangle,\langle c_3,c_2\rangle\}.$$

An alternative formulation as a CSP could introduce the binary constraint $\textsc{Diff}(E_i,E_j)$ for each pair of variables E_i and E_j such that $i < j$, i.e., a quadratic number in total. Hence the *global* all-different constraint may provide more compact representation of our resource allocation problem and, therefore, better premises for efficient implementation. □

Any satisfiability problem can be turned into a CSP in a highly modular way (clause-by-clause). Boolean variables occurring in the CNF become the variables of the respective CSP, ranging over a Boolean domain $\{0,1\}$. Each clause in the CNF can be treated as a constraint scoped by its variables. Since CSPs admit larger domains for variables, they can be viewed as generalizations of SAT problems. One notable way of solving CSPs is to *translate* them into SAT and to use SAT solvers for actual computations [15]. Depending on the constraints involved, such Booleanization may result in substantial blow-ups in the problem representation caused by the encodings of finite-domain variables with Boolean variables. E.g., a variable with four values could be encoded by two Boolean variables using a binary representation for the possible values. Thus, it is fair to say that CSPs properly generalize SAT problems.

To address the relationship with ASP, let us first encode the CSP from Example 4 in ASP. The resulting program is

Listing 4 ASP Extended by Constraints

```
1  % Domains
2  proc(1..n).
3  &dom {1..m} = core(P) :- proc(P).
4
5  % Constraints
6  &distinct {core(P) : proc(P)}.
```

given as Listing 3. An additional domain of `m` cores is introduced (Line 2). The choice of the assignment (Line 5) means that a unique core `C` is associated with each process `P`. Finally, the constraint (Lines 8–9) makes sure that each process is assigned to a different core. Obviously, our program encodes the ALLDIFF constraint through DIFF constraints as discussed above. Until very recently, ASP languages have lacked primitives for the aggregation of the ALLDIFF constraint as well as an implementation. However, the latest extensions of the CLINGO system [14] enable the source-level representation of such aggregates and their seamless transmission to the solving phase using an intermediate format, viz. *aspif*, for ground rules and constraints.

Both ASP and CP languages have primitives that are difficult to express in the other, making these languages somewhat incomparable in terms of expressive power. First, since each SAT problem can be encoded modularly as a CSP, it follows immediately that certain answer set programs are not translatable into CSPs in a modular way. To this end, the formal counter-examples [16] lift easily to the case of CP. Second, the blow-ups incurred by the Booleanization of CSPs are analogously perceived when expressing CSPs with rules. There is existing work on the respective translation of CSPs into ASP [8, 7]. For instance, the way in which domains and constraints are encoded in Listing 3 corresponds to the *direct encoding* of [8] where the constraint in Lines 8–9 would be turned into a rule capturing the *violations* of ALLDIFF. The other potential embeddings of ALLDIFF into ASP considered in [8] vary in the compactness of representation as well as the efficiency of propagation and solving.

There are many ASP solvers that natively support constraint satisfaction: CLINGCON [2], DLVHEX [6], EZCSP [1], and INCA [8]. Latest CP-style extensions of ASP [14] are illustrated in Listing 4. In detail, Line 3 introduces an integer variable `core(P)` for each process `P`. These variables are supposed to have *distinct* values as declared in Line 6. In this way, it is possible to express CP-style constraints directly within an answer set program. However, rules and constraints can be combined in far more versatile ways [11].

Listing 5 Expressing Linear Constraints in ASP

```
1  &sum{ duration(P) } >= 1 :- proc(P).
2  &sum{ duration(P) } <= d :- proc(P).
3  &sum{ duration(P) : proc(P)} <= limit.
```

In fact, rules enable meta-level control over CP-style constraints and the logical combination of traditional constraints in non-trivial ways. For instance, ALLDIFF constraints can be made conditional by adding an appropriate body to the constraint, such as the one in Line 6 in Listing 4. Then it is possible to encode, e.g., *disjunctions* of ALLDIFF constraints. Yet it is unclear how to express such combinations as a single constraint in traditional CP in a modular way.

4 Linear and Mixed Integer Programming

Constraint satisfaction problems generalize SAT and ASP in a fundamental way by allowing for arbitrary finite-domain constraints. *Linear programming* (LP) is a related paradigm based on fixed linear (in)equalities of a form

$$c_1x_1 + \ldots + c_nx_n \; ? \; b$$

where $x_1,\ldots,x_n$ are variables, $c_1,\ldots,c_n$ are their coefficients, b is a constant, and ? is typically one of the comparison operators $=$, $\leq$, and $\geq$. The actual collection of operators may depend on the domains of variables and the implementation. Certain operators are also expressible with others such as $=$ in terms of $\leq$ and $\geq$. As regards the domains $D(x_i)$ of variables x_i, typical options are either reals or rationals. If also integer and Boolean variables are of interest, the paradigm is called *mixed integer programming* (MIP). The satisfaction of linear constraints is defined in analogy to CSPs (see Section 3) and a *solution* v is an assignment of variables such that all linear (in)equalities are satisfied. In *linear optimization* based on LP or MIP, a linear function $b_1y_1+\ldots+b_my_m$ is used as an *objective function* whose value is to be either minimized or maximized. The eventual goal is to find an *optimal solution* v in this respect. Similar optimization of answer sets is supported by native answer set solvers and the respective problems in SAT checking are known as maximum satisfiability (MaxSAT) problems, see [3].

In MIP, integer and real variables enable a linear-size translation of ground logic programs into linear constraints [23] but such a translation is necessarily non-modular like any other translation towards a formalism based on classical satisfaction. Given the translation, it is possible to harness

MIP solvers, such as CPLEX by IBM, for the computation of answer sets. Perhaps more conveniently from the user's perspective, the syntax of ASP has also been extended by linear constraints. Such generalizations enable highly maintainable encodings, combining the strengths of rule-based representations with the expressiveness of linear constraints over integers and reals [19]. Listing 5 shows a glimpse of a CLINGO encoding featuring linear constraints. In Lines 1–2, we associate a duration from `1` to `d` units with each of the `n` processes introduced in our previous examples. Assuming the sequential execution of processes, the sum of all durations should not exceed a predetermined limit (Line 3). Both `d` and `limit` are parameters of the encoding and their values can be set using the command-line options of CLINGO.

5 SAT Modulo Theories

The *SAT modulo theories* (SMT) [3] paradigm extends satisfiability checking by theory reasoning. In addition to propositional atoms, dedicated *theory atoms* t are introduced in the language. Theory atoms may have additional structure such as statements $x - y < k$ in *integer difference logic* (IDL). Accordingly, a satisfying assignment ν of a formula has an additional requirement: the collection of theory literals $\{t \mid \nu(t) = \mathbf{true}\} \cup \{\neg t \mid \nu(t) = \mathbf{false}\}$ must be consistent in a background theory. Overall, there is a rich variety of SMT-style extensions of SAT stretching its applicability to domains where pure propositional logic seems insufficient and non-Boolean variables become inevitable.

In analogy to MIP, ASP can be translated into a number of SMT fragments such as IDL [18], *bit-vector logic* (BVL) [24], and *SAT modulo graphs* (SMG) [12]. These translations are all linear with respect to the length of the ground program, but the current implementations of BVL tend to reintroduce the logarithmic factor (the actual width of bit vectors) exhibited by the translations into SAT (see Section 2). The additional data structures used in IDL (integers) and SMG (the graph) enable implementations without this effect and the translations of ground programs remain linear. There are also theory-based extensions of ASP and the general framework has shaped up in a number of works, see [19] for a summary of existing approaches and tools. One notable theory concerns graph properties in analogy to SMG and, in particular, that of acyclicity [4]. Although many graph properties including acyclicity are expressible in ASP itself, it may be desirable to use graph-based extensions for the sake of compact representation or more efficient, explicit reasoning based on graphs. Moreover, acyclicity is so frequently occurring property in applications that a dedicated implementation, such as the one devised in [4], is justifiable.

Listing 6 Enforcing Acyclicity in ASP modulo Theories

```
1  % Directive
2  #edge (P1,P2): wait(P1,P2).
```

Our final example, given as Listing 6, illustrates the use of acyclicity in the context of our example in Listing 2. In lines 7–10 of the original encoding, we checked for circular dependencies using an explicit predicate `dep/2` for dependencies and defined it in terms of recursive rules. In contrast to this, according to Listing 6, whenever a process `P1` is waiting for another process `P2`, there is an arc `(P1,P2)` in the global graph associated with the program. This is accomplished by the directive in Line 2 and no further rules are needed. The induced graph is constantly subject to the acyclicity constraint enforced by the answer set solver. As a consequence, any answer set that gives rise to a cycle in the graph will be excluded. Besides directives, other kinds of theory atoms are available in the CLINGO system in the general aggregate syntax already present in Listings 4 and 5.

6 Conclusion

In this article, we provided an overview of ASP and related paradigms, which resemble ASP from the modeling perspective (cf. Figure 1) but are quite different when it comes to the actual syntax and semantics of logical expressions involved. The given examples try to illustrate in simplified settings how the main characteristics of other formalisms are available in the realm of ASP. The support for variables with large domains is perhaps the weakest point of standard ASP where only Boolean variables appear in the ground logic program forwarded to the solving phase. However, with extended support for non-Boolean variables, ASP is a promising formalism that is able to cater for the features of others. The *aspif* format may provide a basis for a uniform representation for constraints in the future systems. For instance the *minizinc* format used to represent CSPs can be readily translated[1] into respective representations in *flatzinc* and *aspif* formats. Yet another issue is to which extent the various fragments of SMT-LIB[2] can be supported in ASP.

[1] `https://github.com/potassco/fz2aspif`

[2] `http://smtlib.cs.uiowa.edu/`

Tomi Janhunen is a senior university lecturer at Aalto University in the Department of Computer Science. He holds the title of docent from Aalto University. Janhunen received his doctoral degree in theoretical computer science from Helsinki University of Technology in Finland in 1998. His main research interests are in knowledge representation and automated reasoning, especially in answer set programming, extensions of Boolean satisfiability, and translations between logical formalisms. He was the PC co-chair of LPNMR'17 and JELIA'10. In addition, he has served the program committees of 50 conferences and workshops in his research area and reviewed articles for scientific journals actively.

References

1. Balduccini M, Lierler Y (2017) Constraint answer set solver EZCSP and why integration schemas matter. TPLP 17(4):462–515
2. Banbara M, Kaufmann B, Ostrowski M, Schaub T (2017) Clingcon: The next generation. TPLP 17(4):408–461
3. Biere A, Heule M, van Maaren H, Walsh T (eds) (2009) Handbook of Satisfiability, Frontiers in Artificial Intelligence and Applications, vol 185, IOS Press
4. Bomanson J, Gebser M, Janhunen T, Kaufmann B, Schaub T (2016) Answer set programming modulo acyclicity. Fundamenta Informaticae 147(1):63–91
5. Brewka G, Eiter T, Truszczyński M (2011) Answer set programming at a glance. Communications of the ACM 54(12):92–103
6. De Rosis A, Eiter T, Redl C, Ricca F (2015) Constraint answer set programming based on HEX-programs
7. Drescher C (2015) Conflict-driven constraint answer set solving. PhD thesis, University of New South Wales, Sydney, Australia, 226 pages
8. Drescher C, Walsh T (2011) Translation-based constraint answer set solving. In: Proceedings of IJCAI 2011, pp 2596–2601
9. Ferraris P (2005) Answer sets for propositional theories. In: Proceedings of LPNMR'05, Springer, LNAI, vol 3662, pp 119–131
10. Gebser M, Kaufmann B, Schaub T (2009) Solution enumeration for projected Boolean search problems. In: Proceedings of CPAIOR 2009, Springer, LNCS, vol 5547, pp 71–86
11. Gebser M, Ostrowski M, Schaub T (2009) Constraint answer set solving. In: Proceedings of ICLP 2009, Springer, LNCS, vol 5649, pp 235–249
12. Gebser M, Janhunen T, Rintanen J (2014) Answer set programming as SAT modulo acyclicity. In: Proceedings of ECAI 2014, IOS Press, pp 351–356
13. Gebser M, Janhunen T, Kaminski R, Schaub T, Tasharrofi S (2016) Writing declarative specifications for clauses. In: Proceedings of JELIA'16, pp 256–271
14. Gebser M, Kaminski R, Kaufmann B, Ostrowski M, Schaub T, Wanko P (2016) Theory solving made easy with clingo 5. In: Technical Communications of ICLP 2016, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, OASICS, vol 52, pp 2:1–2:15
15. Huang J (2008) Universal Booleanization of constraint models. In: Proceedings of CP 2008, Springer, LNCS, vol 5202, pp 144–158
16. Janhunen T (2006) Some (in)translatability results for normal logic programs and propositional theories. Journal of Applied Non-Classical Logics 16(1-2):35–86
17. Janhunen T, Niemelä I (2016) The answer set programming paradigm. AI Magazine 37(3):13–24
18. Janhunen T, Niemelä I, Sevalnev M (2009) Computing stable models via reductions to difference logic. In: Proceedings of LPNMR 2009, Springer, LNCS, vol 5753, pp 142–154
19. Janhunen T, Kaminski R, Ostrowski M, Schaub T, Schellhorn S, Wanko P (2017) Clingo goes linear constraints over reals and integers. TPLP 17(5–6):872–888
20. Lierler Y (2017) What is answer set programming to propositional satisfiability. Constraints 22(3):307–337
21. Lifschitz V, Razborov A (2006) Why are there so many loop formulas? ACM Transactions on Computational Logic 7(2):261–268
22. Lin F, Zhao J (2003) On tight logic programs and yet another translation from normal logic programs to propositional logic. In: Proceedings of IJCAI 2003, Morgan Kaufmann, pp 853–858
23. Liu G, Janhunen T, Niemelä I (2012) Answer set programming via mixed integer programming. In: Proceedings of KR 2012, AAAI Press, pp 32–42
24. Nguyen M, Janhunen T, Niemelä I (2011) Translating answer-set programs into bit-vector logic. In: Proceedings of INAP 2011, Springer, LNCS, vol 7773, pp 95–113
25. Niemelä I (1999) Logic programs with stable model semantics as a constraint programming paradigm. Ann Math Artif Intell 25(3-4):241–273

26. Pearce D, Sarsakov V, Schaub T, Tompits H, Woltran S (2002) A polynomial translation of logic programs with nested expressions into disjunctive logic programs: Preliminary report. In: Proceedings of ICLP 2002, pp 405–420
27. Rossi F, van Beek P, Walsh T (eds) (2006) Handbook of Constraint Programming, Foundations of Artificial Intelligence, vol 2. Elsevier
28. Tseitin G (1983) On the complexity of derivation in propositional calculus. In: Siekmann J, Wrightson G (eds) Automation of Reasoning 2: Classical Papers on Computational Logic 1967–1970, Springer, pp 466–483