Potassco: The Potsdam Answer Set Solving Collection

Martin Gebser, Benjamin Kaufmann, Roland Kaminski, Max Ostrowski, Torsten Schaub, and Marius Schneider
Universität Potsdam, Potsdam, Germany
E-mail: {gebser,kaufmann,kaminski,ostrowsk,torsten,manju}@cs.uni-potsdam.de

This paper gives an overview of the open source project *Potassco*, the Potsdam Answer Set Solving Collection, bundling tools for Answer Set Programming developed at the University of Potsdam.

Keywords: Answer Set Programming, Declarative Problem Solving

1. Introduction

Answer Set Programming (ASP; [5]) has become a popular approach to declarative problem solving in the field of Knowledge Representation and Reasoning (KRR; [79]). This is mainly due to its appealing combination of a rich yet simple modeling language with high-performance solving capacities.

ASP has its roots in

- Knowledge Representation and (Nonmonotonic) Reasoning,
- Logic Programming (with negation),
- Databases, and
- Boolean Constraint Solving.

The basic idea of ASP is to represent a given computational problem by a logic program[1] whose answer sets correspond to solutions, and then to use an ASP solver for finding answer sets of the program. This approach is closely related to the one pursued in propositional Satisfiability Testing (SAT; [9]), where problems are encoded as propositional theories whose models represent the solutions to the given problem. Even though, syntactically, ASP programs resemble Prolog programs, they are treated by rather different computational mechanisms. Indeed, the usage of model generation instead of query evaluation can be seen as a recent trend in the encompassing field of KRR but also more remote areas such as Automated Planning and Computer-aided Verification.

More formally, ASP allows for solving all search problems in NP (and NP^{NP}) in a uniform way [93, 14], offering more succinct problem representations than available in SAT [77]. Meanwhile, ASP has been used in many application areas, among them, product configuration [96], decision support for NASA shuttle controllers [86], composition of Renaissance music [10], synthesis of multiprocessor systems [71], reasoning tools in systems biology [30,55], (industrial) team-building [64], and many more.[2]

The success story of ASP has its roots in the early availability of ASP solvers, beginning with the *smodels* system [95], followed by *dlv* [75], SAT-based ASP solvers, like *assat* [80] and *cmodels* [61], and the conflict-driven learning ASP solver *clasp* [48], demonstrating the performance and versatility of ASP solvers by winning first places at international competitions like ASP'09, PB'09, and SAT'09.

In fact, *clasp* is a salient part of the open-source project *Potassco*, the Potsdam Answer Set Solving Collection, bundling tools for ASP developed at the University of Potsdam. In what follows, we summarize the various tools concentrating on their features and underlying motivations.

Our paper presupposes a certain familiarity with the syntax and semantics of logic programs under stable model semantics [58]. For details on semantics, we refer the reader to [58,5,57]. Likewise, first-order rep-

[1] In view of ASP's quest for declarativeness, the term *program* is of course a misnomer but historically too well established to be dropped.

[2] See `http://www.cs.uni-potsdam.de/~torsten/asp` for an extended listing of ASP applications.

AI Communications
ISSN 0921-7126, IOS Press.

resentations, commonly used to encode problems in ASP, are informally introduced by need in the remainder of this paper. See [38] for detailed descriptions along with various examples of the input languages of the grounder *gringo*.

2. ASP Solving

As with traditional computer programming, the ASP solving process amounts to a closed loop. Its steps can be roughly classified into

1. Modeling,
2. Grounding,
3. Solving,
4. Visualizing, and
5. Software Engineering.

We have illustrated this process in Figure 1 by giving the associated components. It all starts with a modeling phase, which results in a first throw at a representation of the given problem in terms of logic programming rules. The resulting program is usually formulated by means of first-order variables, which are systematically replaced by elements of the Herbrand universe in a subsequent grounding phase. This yields a finite propositional program that is then fed into the actual ASP solver. The output of the solver varies depending on the respective reasoning mode. Often, it consists of a textual representation of a sequence of answer sets. Depending on the quality of the resulting answer, one then either refines the (last version of the) problem representation or not.

As pointed out in the introductory section, the strongholds of ASP are usually regarded to be its rich modeling language as well as its high-performance solving capacities. Moreover, ASP distinguishes itself by highly optimized yet domain-independent grounding systems. In what follows, we concentrate on ASP solving and grounding systems, thereby sketching ASP's modeling language. For issues related to software engineering in ASP, the interested reader is referred to the dedicated workshop series, SEA [16,17]. A first approach to visualization can be found in [13].

3. *gringo*

The basic approach to writing programs in ASP follows a *generate-and-test* methodology (cf. [76]), inspired by intuitions on *NP* problems. That is, a "generating" part is meant to non-deterministically provide solution candidates, while a "testing" part eliminates candidates violating some requirements. (Note that this decomposition is only a methodological one; it is neither syntactically enforced nor computationally relevant.) In addition, one may specify *optimization* criteria via lexicographically ordered objective functions.

To illustrate this, let us consider an ASP solution to the *Traveling Salesperson* problem, given in Table 1 and 2. The first table describes the problem instance; this and all following programs are formulated in the input language of the ASP grounder *gringo* [45]. The predicates `node/1`, `edge/2`, and `cost/3` specify a directed graph with weighted edges. A statement like `node(1..6).` abbreviates the definition of six facts, viz. `node(1).`,...,`node(6).` Similarly, the expression `edge(1,2;3;4)` stands for `edge(1,2).`,...,`edge(1,4).` The costs of the edges are given unabbreviated as a collection of facts. The interested reader is referred to [38] for a detailed description of *gringo*'s input language.

```
node(1..6).

edge(1,2;3;4).  edge(2,4;5;6).  edge(3,1;4;5).
edge(4,1;2).    edge(5,3;4;6).  edge(6,2;3;5).

cost(1,2,2).  cost(1,3,3).  cost(1,4,1).
cost(2,4,2).  cost(2,5,2).  cost(2,6,4).
cost(3,1,3).  cost(3,4,2).  cost(3,5,2).
cost(4,1,1).  cost(4,2,2).
cost(5,3,2).  cost(5,4,2).  cost(5,6,1).
cost(6,2,4).  cost(6,3,3).  cost(6,5,1).
```

Table 1

A directed graph with weighted edges in ASP facts

Table 2 gives the encoding of the actual problem. The predicate `cycle/2` is meant to capture the resulting itinerary of the salesperson. The first two rules generate possible solution candidates. The first one makes sure that for each node exactly one of its outgoing edges belongs to the solution; similarly, the second rule deals with incoming edges. This is modeled with so-called cardinality constraints [95]. Their functioning is best explained by regarding their instantiated form. To this end, consider the grounding of the first rule when taking node `1` along with its outgoing edges:

```
1 { cycle(1,2),cycle(1,3),cycle(1,4) } 1.
```

Note that grounding simplifies the rule by eliminating true components. The above constraint stipulates

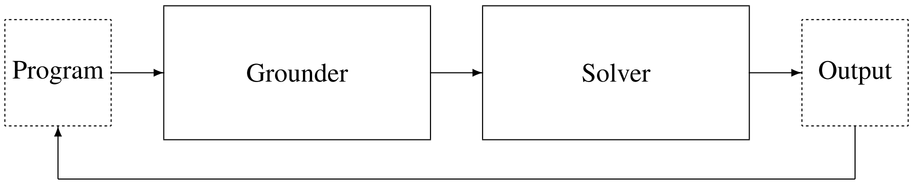

Fig. 1. ASP Solving Process

```
1 { cycle(X,Y) : edge(X,Y) } 1 :- node(X).
1 { cycle(X,Y) : edge(X,Y) } 1 :- node(Y).

reached(Y) :- cycle(1,Y).
reached(Y) :- cycle(X,Y), reached(X).

:- node(Y), not reached(Y).

#minimize [ cycle(X,Y) : cost(X,Y,C) = C ].
```

Table 2
An ASP encoding of the Traveling Salesperson problem

that each solution must contain exactly one of the three instances of the predicate `cycle/2`.

The predicate `reach/1` captures all nodes reachable from (starting) node `1` through the edges distinguished by predicate `cycle/2`. The fifth rule is an integrity constraint requiring that each node in the graph must be reachable in the aforementioned way. The ease of expressing such reachability constraints is a major feature of ASP.

Finally, the last statement instructs the ASP solver to search for answer sets comprising instances of the predicate `cycle/2` that yield a minimum sum of associated costs, expressed via predicate `cost/3`.

Invoking *gringo* with the two files in Table 1 (`graph`) and Table 2 (`tsp`) results in an intermediate format [72]. A human-readable format is obtained by invoking *gringo* with the option `--text` (or `-t` for short), e.g.:

```
$ gringo -t graph tsp
```

Another alternative format is obtained via option `--reify`, yielding a reified representation of the grounding (in terms of facts) that can then be used together with appropriate meta-programs. Of particular interest are also the options `--verbose[=<n>]` and `--gstats` providing the user with information about the proceeding of the grounding process and statistics of the grounding process, respectively. Further options can be consulted via the `--help` option.

In fact, the input language of *gringo* is Turing-complete, as exemplified below by an encoding of a universal Turing Machine. A particular instance, a machine solving the 3-state Busy Beaver problem, is represented by the facts in Table 3; its graphical specification is given in Figure 2.

The facts `start(a).` and `blank(0).` specify the starting state `a` and the blank symbol `0`, respectively, of the 3-state Busy Beaver machine. Furthermore, `tape(n,0,n).` provides the initial tape contents, where `0` indicates a blank at the initial position of the read/write head and the `n`'s represent infinitely many blanks to the left and to the right of the head. Finally, the predicate `trans/5` captures the transition function of the Busy Beaver machine. A fact of the form `trans(S,A,AN,SN,D).` describes that, if the machine is in state `S` and the head is on tape symbol `A`, it writes `AN`, changes its state to `SN`, and moves the head to the left or right as given by `D` $\in$ {`l`,`r`}.

```
start(a).  blank(0).  tape(n,0,n).
trans(a,0,1,b,r). trans(a,1,1,c,l).
trans(b,0,1,a,l). trans(b,1,1,b,r).
trans(c,0,1,b,l). trans(c,1,1,h,r).
```

Table 3
A 3-state Busy Beaver machine in ASP facts

Table 4 shows an encoding of a universal Turing Machine. It defines the predicate `conf/4` describing the configurations of the machine (e.g., the one specified in Table 3) it runs. The rule in the first line determines the starting configuration in terms of a state `S`, the tape symbol `A` at the initial position of the read/write head, and the tape contents `L` and `R` on its left and right, respectively. The remaining four rules derive successor configurations relative to the transition function (given

```
conf(S,L,A,R) :- start(S), tape(L,A,R).

conf(SN,l(L,AN),AR,R) :- conf(S,L,A,r(AR,R)),      trans(S,A,AN,SN,r).
conf(SN,l(L,AN),AR,n) :- conf(S,L,A,n), blank(AR), trans(S,A,AN,SN,r).
conf(SN,L,AL,r(AN,R)) :- conf(S,l(L,AL),A,R),      trans(S,A,AN,SN,l).
conf(SN,n,AL,r(AN,R)) :- conf(S,n,A,R), blank(AL), trans(S,A,AN,SN,l).
```

Table 4
An ASP encoding of a universal Turing Machine

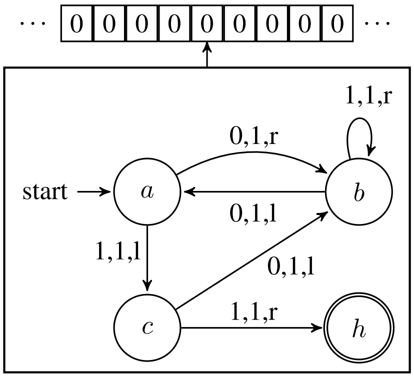

Fig. 2. A 3-state Busy Beaver machine

by facts over `trans/5`). The first two of these rules model movements of the head to the right, thereby distinguishing the cases that the tape contains some (explicit) symbol `AR` on the right of the head or that its right-hand side is fully blank (`n`). In the former case, the symbol `AN` to write is appended to the tape contents on the left of the new head position, represented by means of the functional term `l(L,AN)`, while `AR` becomes the symbol at the new head position and `R` the residual contents on its right. Unlike this, the rule dealing with a blank tape on the right takes a blank as the symbol at the new head position and `n` to represent infinitely many remaining blanks. Similarly, the last two rules specify the symmetric cases obtained for movements to the left. Note that, by using functions, the encoding in Table 4 allows for representing runs of machines without limiting the tape space that can be investigated. Hence, whether *gringo* halts depends on the machine to run. Notably, infinite loops in finite tape space are (implicitly) detected, since repeated configurations do not induce new ground rules.

Invoking *gringo* with files containing the rules in Table 3 (`beaver`) and Table 4 (`turing`) yields:

```
$ gringo -t beaver turing
...
state(a,n,0,n).
state(b,l(n,1),0,n).
state(a,n,1,r(1,n)).
state(c,n,0,r(1,r(1,n))).
state(b,n,0,r(1,r(1,r(1,n)))).
state(a,n,0,r(1,r(1,r(1,r(1,n))))).
state(b,l(n,1),1,r(1,r(1,r(1,n)))).
state(b,l(l(n,1),1),1,r(1,r(1,n))).
state(b,l(l(l(n,1),1),1),1,r(1,n)).
state(b,l(l(l(l(n,1),1),1),1),1,n).
state(b,l(l(l(l(l(n,1),1),1),1),1),0,n).
state(a,l(l(l(l(n,1),1),1),1),1,r(1,n)).
state(c,l(l(l(n,1),1),1),1,r(1,r(1,n))).
state(h,l(l(l(l(n,1),1),1),1),1,r(1,n)).
```

In fact, the Turing Machine is completely evaluated by *gringo* that prints all feasible configurations in the same order as a Turing Machine would process them. This means that the last line contains the configuration in which the machine reaches the final state. Here, the 3-state Busy Beaver machine terminates after writing six times the symbol `1` to the tape.

The expressive power of Turing-computability should not mislead to the idea that the grounder is meant to address computable problems completely by itself. Rather, it provides the most general setting for deterministic computations. In particular, this allows for eliminating many external preprocessing steps involving imperative programming languages.

Finally, let us highlight some features of the latest construction series of *gringo*, starting with version 3.0. A comprehensive documentation is found in *gringo*'s manual [38]; previous versions are described in [54,45]. First of all, the 3.0 series only stipulates rules to be *safe* (cf. [1]) rather than to be domain-restricted through additional domain predicates, as in previous versions of *gringo*. As a consequence, programs are no longer subject to any restriction guaranteeing a finite grounding (like λ-restrictedness [54]). Rather, this responsibility is left with the user in order to provide her with the greatest flexibility. To see this, consider the λ-restricted logic program in Table 5, needing the domain predicate `p/1` for delineating the

```
q(1,2).   q(2,3).   q(3,1).   p(1;2;3).
q(X,Z) :- q(X,Y), q(Y,Z), p(X;Y;Z).
```

Table 5

A λ-restricted logic program for the transitive closure of `q/2`

instantiation of the last rule. Unlike this, the safe variant of this program, accepted by the recent *gringo* version, makes such predicates obsolete, as seen in Table 6. This general setting is supported by a grounding algorithm based on semi-naive database evaluation (cf. [1]), closely related to that of *dlv* [75].

```
q(1,2).   q(2,3).   q(3,1).
q(X,Z) :- q(X,Y), q(Y,Z).
```

Table 6

A safe logic program for the transitive closure of `q/2`

As with previous versions of *gringo*, its language supports various aggregates. In fact, the cardinality constraints in Table 2 are abbreviated *count* aggregates. In full, they had to be written as

```
1 #count { cycle(X,Y) : edge(X,Y) } 1.
```

In addition, *gringo* supports the aggregate functions `#sum`, `#min`, `#max`, `#avg`, `#even`, and `#odd` with their obvious meanings. An interesting language extension of the 3.0 series are its optimize statements with priorities, indicated by an `@`, e.g.:

```
#minimize { a = 4@7, b = 2@1, c = 3@1 }.
```

Here, `a` has weight `4` and priority `7`. Priorities allow for representing a sequence of lexicographically ordered minimization objectives, where greater levels are more significant than smaller ones.[3]

Another powerful feature of *gringo* is its integrated scripting language, viz. *lua* [70]. *lua* provides an alternative means for deterministic computations and is very useful when things would get messy in logic programming. A typical example is interfacing to databases or numeric computations. Both are easier expressed and computed in *lua* and then passed to *gringo* in terms of sets of facts. See [38] for details and exemplary use cases. The interested reader may also consult [44] on the most recent advances in *gringo*.

[3]Explicit priority levels avoid a dependency of priorities on input order, as considered by *lparse* [98] if several minimize statements are provided. Priority levels are also supported by *dlv* [75] in weak constraints.

4. *clasp*

clasp is originally designed and optimized for conflict-driven ASP solving, as described in [48]. To this end, it features a number of sophisticated reasoning and implementation techniques, some specific to ASP and others borrowed from CDCL-based SAT solvers (cf. [9]).[4] The basic search procedure of CDCL-based solvers can be outlined by means of the loop [27] given in Figure 3. At first, the closure under deterministic consequence operations is computed. This operation is of course different for SAT and ASP solvers. Then, four cases are distinguished. In the first one, a non-conflicting complete assignment is returned. In the second case, an unassigned variable is non-deterministically chosen and assigned. Or at last, a conflict is encountered. All assignments made before the first non-deterministic choice constitute the *top-level*. Hence, a top-level conflict indicates unsatisfiability. Otherwise, the conflict is analyzed and learned in form of a conflict constraint. Then, the algorithm backjumps by undoing a maximum number of successive assignments so that exactly one literal of the constraint is unassigned.

clasp has been purposefully designed as a highly configurable system, and thus many of these features are subject to user control via command line options (try `clasp --help` for an overview). Moreover, *clasp* can be used as a full-fledged SAT or Pseudo-Boolean solver, accepting propositional CNF formulas in *dimacs* format and Pseudo-Boolean formulas in *opb* format, respectively.[5] The remainder of this section, however, is devoted to ASP solving, detailing some selected features of *clasp*.

4.1. Interfaces and Preprocessing

For ASP solving, *clasp* reads ground logic programs provided by *gringo* (or *lparse* [98], alternatively). Choice rules, cardinality and weight constraints (cf. [38]) are either compiled into normal rules during parsing, configurable via option `--trans-ext`, or dealt with in an intrinsic fashion (by default; see Section 4.3 for details).

At the beginning, a logic program is subject to extensive preprocessing [49]. The idea is to simplify the program while identifying equivalences among its rele-

[4]CDCL stands for *Conflict-Driven Clause Learning* (cf. [15,81]).

[5]Both formats are automatically detected and handled by *clasp* series 1.3.

```
loop
  propagate                                    // compute deterministic consequences
  if no conflict then
    if all variables assigned then return variable assignment
    else decide                                // non-deterministically assign some variable
  else
    if top-level conflict then return unsatisfiable
    else
      analyze                                  // analyze conflict and add a conflict constraint
      backjump                                 // undo assignments until conflict constraint is unit
```

Fig. 3. Solving loop of CDCL-based solvers

vant constituents. These equivalences are then used for building a compact program representation (in terms of Boolean constraints). Notably, preprocessing is sometimes able to turn a non-tight program into a tight one (cf. [31,3]).[6] Logic program preprocessing is configured via option `--eq`, taking an integer value fixing the number of iterations.

Once a program has been transformed into Boolean constraints, they can be subject to further preprocessing, primarily based on resolution [25]. Such SAT-oriented preprocessing is invoked with option `--sat-prepro` and further parameters.

A major yet internal feature of *clasp* is that it can be used in a stateful way. That is, *clasp* may keep its state, involving program representation, recorded nogoods, heuristic values, etc., and be invoked under additional (temporary) assumptions and/or by adding new atoms and rules. The corresponding interfaces are fundamental for supporting incremental ASP solving as realized in *iclingo* ([39]; cf. Section 8), a combination of *gringo* and *clasp* for incremental grounding and solving. Also, they allow for solving under assumptions [26], an important feature that is, for example, used in our parallel ASP solver *claspar* ([28]; cf. Section 6).

4.2. Reasoning Modes

Although *clasp*'s primary use case is the computation of answer sets, it also allows for computing supported models[7] of a logic program (via command line option `--supp-models`).[8] In either case, *clasp* provides a number of reasoning modes, determining how to proceed when a model is found.

To begin with, different ways of enumerating models are supported by *clasp*. In fact, solution enumeration is non-trivial in the context of backjumping and conflict-driven learning. A popular approach consists in recording solutions as nogoods and exempting them from deletion. Although *clasp* supports this via option `--solution-recording`, it is prone to blow up in space in view of an exponential number of solutions in the worst case. Unlike this, the default enumeration algorithm of *clasp* runs in polynomial space [47]. Both enumeration approaches also allow for projecting models to a subset of atoms [51], invoked with `--project` and configured via the well-known directives `#hide` and `#show` of *gringo*. This option is of great practical value whenever one faces overwhelmingly many answer sets, involving solution-irrelevant variables having proper combinatorics. For example, the program consisting of the choice rule `{a,b,c}.` has eight (obvious) answer sets. When augmented with directive `#hide c.`, still eight solutions are obtained, yet including four duplicates. Unlike this, invoking *clasp* with `--project` yields only four answer sets differing on `a` and/or `b`, respectively.

As regards implementation, it is interesting to note that *clasp* offers a dedicated interface for enumeration. This allows for abstracting from how to proceed once a model is found and thus makes the search algo-

[6] Informally, tightness [31] indicates that a program is free of recursion through positive literals.

[7] The models of the Clark completion [12] of a program are called *supported models* [2]. On tight programs, supported models and answer sets coincide [31].

[8] To be more precise, this option disables unfounded set checking. Sometimes, the grounder or preprocessing may already eliminate some supported models such that they cannot be recovered later on.

rithm independent of the concrete enumeration strategy. Further reasoning modes implemented via the enumeration interface admit computing the intersection or union of all answer sets of a program (via `--cautious` and `--brave`, respectively). Rather than computing the whole set of (possibly) exponentially many answer sets, the idea is to compute a first answer set, record a constraint eliminating it from further solutions, then compute a second answer set, strengthen the constraint to represent the intersection (or union) of the first two answer sets, and to continue in this way until no more answer set is obtained. This process involves computing at most as many answer sets as there are atoms in the input program. Either the cautious or the brave consequences are then given by the atoms captured by the final constraint.

Another application-relevant feature is optimization. As already mentioned in Section 3, an objective function is specified via a sequence of `#minimize` or `#maximize` statements. For finding optimal solutions, *clasp* offers several options. First, *clasp* allows for computing one or all (`--opt-all`) optimal solutions. Second, the objective function can be initialized via `--opt-value`. The latter turns out to be useful when one is interested in computing consequences belonging to all optimal solutions (in combination with `--cautious`). To this end, one starts with searching for an (arbitrary) optimal answer set and then relaunches *clasp* by bounding its search with the obtained optimum. Doing the latter with `--cautious` yields the atoms that belong to all optimal answer sets. On applications, it turned out to be very helpful to optimize using the option `--restart-on-model` (making *clasp* restart after each (putatively) optimal solution) in order to ameliorate convergence to the optimum. Moreover, option `--opt-heu` can be used to alter default sign selection (see below) for atoms subject to the objective function towards a better function value. Optimization is implemented via the aforementioned enumeration interface. When a solution is found, an optimization constraint is updated with the corresponding objective function value. Furthermore, it is worth mentioning that *clasp* also propagates optimization constraints, that is, they can imply (and provide reasons for) literals upon unit propagation. Finally, if optimization is actually undesired and all solutions ought to be inspected instead, the option `--opt-ignore` is available to make modifying the input (by removing optimize statements) obsolete.

Prediction under inconsistency in an application to bioinformatics [36] is an interesting use case of *clasp*'s manifold reasoning modes.

4.3. Propagation and Search

Propagation in *clasp* relies on an interface *Boolean constraint*; it is thus not limited to (clausal representations of) nogoods (cf. [27]). However, dedicated data structures are used for binary and ternary nogoods (cf. [90]), accounting for the many short nogoods stemming from Clark completion [12]. More complex constraints are accessed via two *watch lists* for each variable (cf. [84]), storing the Boolean constraints that need to be updated if the variable becomes true or false, respectively. While propagation over long nogoods is based on the well-known two-watched-literal algorithm, a counter-based approach is used for propagating cardinality and weight constraints [40].

During unit propagation, binary nogoods are handled before ternary ones, which are in turn inspected before other Boolean constraints. As detailed in [48], our propagation procedure is distinct in giving a clear preference to unit propagation over unfounded set computations. Unfounded set detection aims at small and "loop-encompassing" rather than greatest unfounded sets. As detailed in [40], native treatment of cardinality and weight constraints augments the source-pointer-based unfounded set algorithm, while still aiming at lazy unfounded set checking and backtrack-freeness. The creation and representation of loop nogoods is controlled via option `--loops`. In the default setting, loop nogoods are created for individual unfounded atoms, as shown in [40].

clasp's primary decision heuristics, selectable via option `--heuristic`, use *look-back* strategies derived from corresponding CDCL-based approaches in SAT, viz., *vsids* [84], *berkmin* [62], and *vmtf* [90]. The main goal of such heuristics is selecting variables that contributed to recent conflicts. To this end, they maintain an activity score for each variable, which is primarily influenced by conflict resolution and decayed periodically. The major difference between the approaches of *berkmin* and *vsids* lies in the scope of variables considered during decision making. While *vsids* selects the free variable that is globally most active, *berkmin* restricts the selection to variables belonging to the most recently recorded but yet unsatisfied conflict nogood. Although the look-back heuristics implemented in *clasp* are modeled after the corresponding CDCL-based approaches, one important difference is that *clasp* optionally also scores variables contained in loop nogoods. In case of *berkmin*, it may also select a free variable belonging to a recently recorded loop nogood. Finally, we note that *clasp*'s heuristic can also be based upon *look-ahead* strategies (that extend unit propagation by *failed-literal detection* [32]). This makes sense whenever *clasp* is run without conflict-driven learning, operating similar to *smodels*.

Once a decision variable has been selected, a sign heuristic decides about its truth value. The main criterion for look-back heuristics is to satisfy the greatest number of conflict nogoods. Initially and also for tie-breaking, *clasp* does sign selection based on a variable's type: atom variables are preferentially set to false, while body variables are made true. This aims at maximizing the number of resulting implications. Another sign heuristic implemented in *clasp* is *progress saving* [87]. The idea is as follows: upon backjumping (or restarting), the recent truth values of retracted variables are saved, except for those assigned at the last decision level. These saved values are then used for sign selection. The intuition behind this strategy is that the assignments made prior to the last decision level did not lead to a conflict and may have satisfied some subproblems. Hence, re-establishing them may help to avoid solving subproblems multiple times. Progress saving is invoked with option `--save-progress`; its computational impact, however, depends heavily on the structure of the application at hand.

The robustness of *clasp* is boosted by multiple advanced restart strategies, namely, geometric, fixed-interval, Luby-style, or a nested policy (see [46,50] for details), configurable via option `--restarts`. Usually, restart strategies are based on the global number of conflicts. Beyond that, *clasp* features local restarts [91], which can be activated with `--local-restarts`. Here, one counts the number of conflicts per decision level in order to measure the difficulty of subproblems locally. Furthermore, a bounded approach to restarting (and backjumping) is used when enumerating answer sets, as described in [47]. To complement its more determined search, *clasp* also allows for initial randomized runs [27], typically with a small restart threshold, in the hope to extract putatively interesting nogoods. Finally, it is worth noting that, despite the fact that recent SAT solvers use rather aggressive restart strategies, *clasp* still defaults to a more conservative geometric policy (cf. [27]) because it performs better on ASP-specific benchmarks.

To limit the number of nogoods stored simultaneously, recorded nogoods are periodically subject to deletion. Complementing look-back heuristics, *clasp*'s nogood deletion strategy associates an activity with each recorded nogood, which is incremented whenever the nogood is used for conflict resolution. Borrowing ideas from *minisat* [27] and *berkmin* [62], the initial threshold on the number of stored nogoods is calculated from the size of an input program and increased by a certain factor upon each restart. (The defaults for the maximum size of *clasp*'s dynamic nogood database and its growth can be overridden via `--deletion`.) As soon as the current threshold is exceeded, deletion is initiated and removes up to 75% of the recorded nogoods. Nogoods that are currently locked (because they serve as antecedents) or whose activities significantly exceed the average activity are exempt from deletion. However, the nogoods that are not deleted have their activities decayed, which gives preference to those used in the future. All in all, *clasp*'s nogood deletion strategy aims at limiting the overall number of stored nogoods, while keeping the relevant and recently recorded ones. This likewise applies to conflict and loop nogoods.

5. *claspD*

In fact, many important problems in KRR have an elevated degree of complexity, calling for expressive solving paradigms being able to capture problems at the second level of the polynomial hierarchy (cf. [92] for a survey). One possibility to deal with such a problem consists in expressing it as a Quantified Boolean Formula (QBF) and then to use some QBF solver to compute its solutions. Another approach is furnished by ASP solvers dealing with disjunctive logic programs, that is, logic programs allowing for disjunction in the heads and (default) negation in the bodies of rules.

For addressing NP^{NP}-problems, we built an extension of *clasp* dealing with disjunctive logic programs. The resulting ASP solver is called *claspD* [20]. It inherits many features from *clasp*, such as conflict-driven learning, lookback-based decision heuristics, restart policies, watched literals, etc.

The actual search for answer sets can be further distinguished into a generating part, providing answer set candidates, and a testing part, verifying the provided candidates. Since both of these tasks can be computationally complex, they are performed by associated inference engines, implemented in *claspD* by feeding the core search module from *clasp* with particular Boolean constraints. While the generator traverses the search space for answer sets, communicating its current state through an assignment to the tester, the latter checks for unfounded sets and reports them back via nogoods.

As shown in [20], an approximative unfounded-set detecting procedure is integrated into propagation and thus continuously applied during the generation of answer set candidates. In contrast, exhaustive checks for so-called non-head-cycle-free components (cf. [20]), are performed only selectively, e.g., if an assignment is total, due to their high computational cost.

The input language of *claspD* consists of logic programs in *gringo*'s output format. Like *clasp*, also *claspD* supports answer set enumeration [47] and optimization. It also handles cardinality and weight constraints [95], currently through compilation.

Given that *claspD* evolved from an earlier branch of *clasp*, it is planned to re-merge it into *clasp* in the mid-future.

6. *claspar*

Despite the progress of sequential ASP Solving technology, only little advancement is observed in the parallel setting.[9] This is deplorable in view of the rapidly growing availability of clustered, multi-processor, and/or multi-core computing devices. We addressed this shortcoming by building a distributed version of *clasp*, focusing on the parallelization of search. The resulting distributed ASP solver is called *claspar* [28,94]. Our approach builds upon the Message Passing Interface (MPI; [67]), realizing communication and data exchange between computing units via message passing. Interestingly, MPI abstracts from the actual hardware and lets us execute our system on clusters as well as multi-processor and/or multi-core machines.

We aimed at a simple and transparent approach in order to be able to take advantage of the high performance offered by modern off-the-shelf ASP solvers such as *clasp*. To this end, we have chosen simple master-worker architectures, in which each worker consists of an ASP solver along with an attached communication module. The solver is linked to its communication module via an elementary interface requiring only marginal modifications to the solver. All major communication is initiated by the workers' communication modules, exchanging messages with the master in an asynchronous way. The specific communication structure can be configured via the option `--topology`, allowing for flat and more complex hierarchical architectures.

[9] Earlier attempts include [88,4,65,66].

Although we tried to keep our design generic, we took advantage of some design features of *clasp*, as outlined in the previous section. In fact, *clasp* extends the static concept of a top-level by additionally providing a dynamic variant referred to as *root-level* [27]. As with the top-level, conflicts within the root-level cannot be resolved given that all of its variable assignments are precluded from backtracking. We build upon this feature for splitting the search space. Splitting is accomplished according to a so-called *guiding path* [100], the sequence of all non-deterministic choices. Given a root-level $i-1$, a guiding path $(v_1, \dots, v_{i-1}, v_i, \dots, v_n)$ can be divided into a prefix $(v_1, \dots, v_{i-1})$ of non-splittable variables and a postfix $(v_i, \dots, v_n)$ of splittable variables. We can split the search space at the first splittable variable by incrementing the root-level by one and dissociating a guiding path composed of the first $i-1$ variables and the complement of the ith variable, yielding $(v_1, \dots, v_{i-1}, \overline{v_i})$. Note that the local assignment remains unchanged, and only the root-level is incremented to i. We have chosen to split at the first splittable variable because, first, this results in cutting off the largest part of the search space and, second, this way backjumping is least restricted.

Alternatively *claspar* allows for running different configurations that may either split the search space among each other or compete against each other by addressing the very same search space. To this end, a portfolio of different *clasp* configurations is supplied to *claspar* via option `--portfolio-file`. The different configurations are then assigned either randomly or in a round-robin fashion (via `--portfolio-mode=<mode>`).

Upon enumerating answer sets, (locally) using the scheme in [47], the assignment can contain complements of choices from previously enumerated answer sets. Such complements $\overline{u_1}, \dots, \overline{u_j}$ indicate that the search spaces for answer sets containing $(v_1, \dots, v_{i-1})$ and at least one of $u_1, \dots, u_j$ have already been explored. In order to avoid repetitions, it is thus important to pass guiding path $(v_1, \dots, v_{i-1}, \overline{u_1}, \dots, \overline{u_j}, \overline{v_i})$ in response to a split request. This refinement for repetition-free answer set enumeration is implemented in *claspar*.

As of now, *claspar* supports the reasoning modes enumeration and optimization. Optimization involves the exchange of putative optima between solver instances. This allows *claspar* to abandon futile search whenever the local value of the objective function is worse than the value of solutions found by other solver instances. A more elaborate exchange of information is that of conflict nogoods. The latter is controlled by two options:

`--clause-sharing` allows for configuring different strategies for clause exchange, for instance, depending upon different selection criteria of nogoods to be exchanged and the number of nogoods per communication.

`--clause-distribution` specifies the communication architecture for nogood exchange. This can be local, depending on the master/worker topology, organized as a hypercube (work nodes are arranged in a hypercube and clauses are exchanged along the edges), and all to all as well as no exchange at all.

See [42] for more details on the most recent advances in *claspar*.

7. *clingo*

For ASP solving, a program is first grounded by *gringo* and the resulting propositional program is then passed to *clasp*. This is usually done via a UNIX pipeline:

```
$ gringo myprogram | clasp
```

An alternative to this is offered by *clingo*, combining *gringo* and *clasp* in a monolithic system. The above call then reduces to:

```
$ clingo myprogram
```

clingo supports all features and options of *gringo* and *clasp*.

8. *iclingo*

Many real-world applications, like Planning or Model Checking, have associated PSPACE-decision problems. For instance, the plan existence problem of deterministic planning is PSPACE-complete [11]. But the problem of whether a plan having a length *bounded* by a given polynomial exists is in NP. In the setting of ASP, such problems can thus be dealt with in a bounded way by considering in turn one problem instance after another, gradually increasing the bound on the solution size.

As an example, let us consider simplistic STRIPS Planning. Table 7 gives a simple planning problem involving three fluents, `p`, `q`, and `r`, and two actions, `a` and `b`, having precondition `p` and `q` as well as effects `q`, ¬`p` and `r`, ¬`q`, respectively. The initial situation fulfills `p`, and the goal is to satisfy `r`.

```
fluent(p).      fluent(q).     fluent(r).

action(a).      action(b).
   pre(a,p).       pre(b,q).
   add(a,q).       add(b,r).
   del(a,p).       del(b,q).

init(p).        query(r).
```

Table 7
A simple STRIPS Planning problem in ASP facts

This planning problem can be solved by the ASP encoding in Table 8. First of all, observe that the length

```
time(1..t).

holds(P,0) :- init(P).

1 { occ(A,T) : action(A) } 1 :- time(T).
 :- occ(A,T), pre(A,F), not holds(F,T-1).

holds(F,T) :- holds(F,T-1), not ocdel(F,T),
              time(T).
holds(F,T) :- occ(A,T), add(A,F).
ocdel(F,T) :- occ(A,T), del(A,F).

 :- query(F), not holds(F,t).
```

Table 8
An ASP encoding of STRIPS Planning

of a plan is restricted to `t`, provided when calling the grounder. The truth of fluents at individual time steps is (partially) described by predicate `holds/2`. The cardinality constraint requires that exactly one action occurs at each time step. The subsequent integrity constraint stipulates that, if an action occurs at time `T`, its precondition must hold at `T-1`. The three following rules deal with progression over time. The first rule states that fluent values remain unchanged unless evidence to the contrary. The two remaining rules specify the effect of actions. Conflicts between the first and third rule are resolved in favor of the more specific rule, leading to the exception `not ocdel(F,T)` within the general rule. Finally, the last integrity constraint ensures that only plans are accepted that satisfy the goal at the last time step `t`.

An answer to this planning problem is usually found by appeal to iterative deepening search. That is, one first checks whether the program has an answer set for `t=1`, if not, the same is done for `t=2`, and so on. For a given `t`, this approach re-processes all rules parametrized with `T` multiple times, while the final integrity constraint is dealt with only once.

Unlike this, we aim at computing the answers sets in an incremental fashion, and thus providing an incremental approach to both grounding and solving in ASP. Our goal is to avoid redundancy by gradually processing the extensions to a problem rather than repeatedly re-processing the entire extended problem. To this end, we take advantage of *incremental logic programs* [39], consisting of a triple (B, P, Q) of logic programs, among which P and Q contain a (single) parameter k ranging over the natural numbers. In view of this, we also denote P and Q by $P[k]$ and $Q[k]$. The base program B is meant to describe static knowledge, independent of parameter k. The role of P is to capture knowledge accumulating with increasing k, whereas Q is specific for each value of k. Provided all programs are "modularly composable" (cf. [39]), we are interested in finding an answer set of the program $B \cup \bigcup_{1 \leq j \leq i} P[k/j] \cup Q[k/i]$ for some (minimum) integer $i \geq 1$.

For illustration, let us transform the above ASP planning encoding into an incremental logic program. Clearly, the problem instance in Table 7 belongs to the static knowledge in B as well as the `holds/2` definition concerning the initial situation. In practice, this is declared by the statement `#base.` In our simple example, the cumulative part consists of all rules possessing variable `T` in Table 8. As shown in Table 9, this part is indicated by `#cumulative t.`, declaring `t` as the corresponding parameter. Note that `t` replaces all occurrences of `T` and makes the predicate `time/1` obsolete. Finally, the volatile part is indicated by `#volatile t.` and applies to the query only. A comprehensive documentation is found in *gringo*'s manual [38].

Incremental programs are solved by the incremental ASP system *iclingo* [39], built upon the libraries of *gringo* and *clasp*. Unlike the standard proceeding, *iclingo* has to operate in a "stateful way". That is, it has to maintain its previous (grounding and solving) state for processing the current program slices. In this way, all components, B, $P[j]$, and $Q[i]$ are dealt with only once, and duplicated work is avoided when increasing i. As regards grounding, *iclingo* reduces efforts by avoiding reproducing previous ground rules.

```
#base.

holds(P,0) :- init(P).

#cumulative t.

1 { occ(A,t) : action(A) } 1.
 :- occ(A,t), pre(A,F), not holds(F,t-1).

holds(F,t) :- holds(F,t-1), not ocdel(F,t).
holds(F,t) :- occ(A,t), add(A,F).
ocdel(F,t) :- occ(A,t), del(A,F).

#volatile t.

 :- query(F), not holds(F,t).
```

Table 9

An incremental ASP encoding of STRIPS Planning

Regarding solving, it reduces redundancy, in particular, if a learning ASP solver such as *clasp* is used, given that previously gathered information on heuristics, conflicts, or loops, respectively, remains available and can thus be continuously exploited. In fact, the latter is configurable via options `--ilearnt` and `--iheuristic` that allow for either keeping or forgetting learned nogoods and heuristic values, respectively. The interested reader is referred to [39] for a detailed description of *iclingo*'s features, semantics, and implementation.

Meanwhile *iclingo* has been successfully employed in various settings. For instance, for implementing action description languages in *coala* ([35]; cf. Section 12) and PDDL-style planning in *plasp* ([74]; cf. Section 12). Also, we used it as back-end of *fmc2iasp* ([53]; cf. Section 12) for implementing a competitive system for finite model generation.

9. *clingcon*

Certain applications are more naturally modeled by mixing Boolean with non-Boolean constructs, e.g., accounting for resources, fine timings, or functions over finite domains. In other words, non-Boolean constructs make sense whenever the involved variables have large domains. This is addressed by the hybrid ASP solver *clingcon* [52], combining the Boolean modeling capacities of ASP with Constraint Processing (CP; [19, 89]).[10] To this end, *clingcon* adopts techniques from

[10]Groundbreaking work on enhancing ASP with CP techniques was conducted in [8,82,83].

the area of SAT-Modulo-Theories (SMT), combining conflict-driven learning with theory propagation by means of a CP solver. For the latter, we have chosen *gecode* [56] as black box constraint solver. *clingcon* follows the so-called lazy approach of advanced SMT solvers by abstracting from the constraints in a specialized theory [85]. The idea is as follows. The ASP solver passes the portion of its (partial) Boolean assignment associated with constraints to a CP solver, which then checks these constraints against its theory via constraint propagation. As a result, it either signals unsatisfiability or, if possible, extends the Boolean assignment by further constraint atoms. For conflict-driven learning within the ASP solver, however, each assigned constraint atom must be justified by a set of (constraint) atoms providing a "reason" for the underlying inference. As regards the language, this approach also follows the one taken by SMT solvers in letting the ASP solver deal with the atomic, that is, Boolean structure of the program, while a CP solver addresses the "sub-atomic level" by dealing with the constraints associated with constraint atoms.

To illustrate this, consider the example in Table 10. This program describes a balance with two buckets, a and b, at each end. According to the cardinality constraint, we must pour a certain amount of water into exactly one of the buckets at each time point. The amount of added water may vary between 100 and 300. The balance is down at one bucket's side, if the bucket contains more water than the other; otherwise, it is up. Initially, bucket a is empty while b contains 100 units. The goal is to find sequences of *pour* actions making the side of bucket a be down after `t` time steps.

The program contains regular and constraint atoms. The latter type of predicates is denoted by relations, preceded with the symbol `$`. Hence, the amount of water is completely abstracted from the ASP solver and is exclusively handled by the constraint solver. Thus, the capacity can be modeled using any precision and any domain size without interfering with the grounder. In fact, after instantiation, the ASP solver does not distinguish between the regular atom `pour(b,1)` and the constraint atom

```
volume(b,2) $== volume(b,1)+amount(b,1).
```

It assigns Boolean values to both types of atoms. However, depending on the assigned truth value, the CP solver must assign integer values to the constraint variables, `volume(b,2)`, `volume(b,1)`, and `amount(b,1)`, such that the equation satisfies the assigned truth value.

10. *claspfolio*

As a matter of fact, advanced Boolean constraint technology, as used in *clasp*, is sensitive to parameter configuration. In fact, we are unaware of any true application on which *clasp* is run in its default settings. Inspired by *satzilla* [99], we address the parameter sensitivity in ASP solving by exploring a portfolio-based approach. To this end, we concentrate on *clasp* and map a collection of benchmark features onto an element of a portfolio of distinct *clasp* configurations. This mapping is realized by appeal to Support Vector Regression (SVR; [7]).

Given a logic program, the goal of *claspfolio* [41] is to automatically select a suitable configuration of *clasp*. In view of the huge configuration space, the attention is limited to some (manually) selected configurations belonging to a portfolio. Each configuration consists of certain *clasp* options. To approximate the behavior of such a configuration, *claspfolio* applies a model-based approach predicting solving performance from particular features of the input. The portfolio used by *claspfolio* (0.8.0) contains 12 *clasp* configurations, included because of their complementary performances on the training set. The options of these configurations mainly configure the preprocessing, the decision heuristic, and the restart policy of *clasp* in different ways. This provides us with a collection of solving strategies that have turned out to be useful on a range of existing benchmarks. In fact, the hope is that some configuration is (a) well-suited for a user's application and (b) automatically selected by *claspfolio* in view of similarities to the training set.

As shown in Figure 4, ASP solving with *claspfolio* consists of four parts.

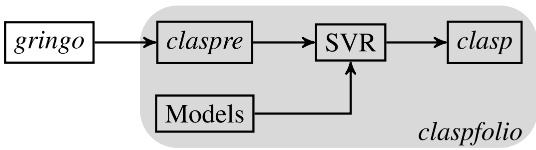

Fig. 4. Architecture of *claspfolio*

First, the ASP grounder *gringo* instantiates a logic program. Then, a light-weight version of *clasp*, called *claspre*, is used to extract features and possibly even solve (too simple) instances. If the instance was not solved by *claspre*, the extracted features are mapped to a score for each configuration in the portfolio. Finally, *clasp* is run for solving, using the configuration with the highest score.

```
$domain(0..10000).
time(0..t).
bucket(a).
bucket(b).

1 { pour(B,T) : bucket(B) } 1 :- time(T), T < t.

100 $<= amount(B,T) :- pour(B,T), T < t.
amount(B,T) $<= 300 :- pour(B,T), T < t.
amount(B,T) $== 0   :- not pour(B,T), bucket(B), time(T), T < t.

volume(B,T+1) $== volume(B,T) + amount(B,T) :- bucket(B), time(T), T < t.

down(B,T) :- volume(C,T) $< volume(B,T), bucket(B), bucket(C), time(T).
  up(B,T) :- not down(B,T), bucket(B), time(T).

volume(a,0) $== 0.
volume(b,0) $== 100.

:- up(a,t).
```

Table 10
Pouring into buckets on a balance

11. *coala*

Action languages provide a compact formal model for describing dynamic domains [59], being central to many applications like model checking, planning, robotics, etc. Moreover, action languages can be implemented rather efficiently through compilation to ASP or SAT. Our system *coala* takes advantage of this by offering a variety of different compilation techniques for several action languages.

coala originates from *al2asp*, constituting the heart of the *BioC* system [24] used for reasoning about biological models in action language $\mathcal{C}_{TAID}$ [23]: *al2asp* compiles $\mathcal{C}_{TAID}$ to $\mathcal{C}$, which is in turn mapped to ASP via the transformation in [78]. *coala* extends the capacities of *al2asp* in several ways. First, it adds certain features of $\mathcal{C}$+ [60] and provides full support of $\mathcal{B}$ [97] (and $\mathcal{A}_L$). Second, it offers different compilation schemes. Apart from a priori bounded encodings using standard ASP systems, *coala* furnishes incremental encodings that can be used in conjunction with the incremental ASP system *iclingo*. Moreover, *coala* distinguishes among forward and backward (incremental) encodings, depending on whether trajectories are successively extended from initial states or whether they are built backwards starting from final states. Third, *coala* supports all action query languages, $\mathcal{P}$, $\mathcal{Q}$, and $\mathcal{R}$, in [59]. Fourth, *coala* allows for posing LTL-like queries, following [68]. Finally, *coala* offers the usage of first-order variables that are treated by the underlying ASP grounder. Optionally, type checking for variables can be enabled. *coala* is implemented in `C++` and can also be used as a library.

12. Potassco Labs

The Potassco Labs suite comprises programs that are either small utilities, projects still under development, or not driven to the full maturity as the ones described above. Among them, we (currently) find:

dlvtogringo is a tool converting output generated by "`dlv -instantiate`" to *gringo*'s input language.

fmc2iasp is used for computing finite models of first-order formulas. The input formulas are written in TPTP format. FM-Darwin is needed for clausification and flattening of the input. *iclingo* is used for finding answer sets of the logic program formed by *fmc2iasp*. An answer set represents a finite model of the input.

inca is a preprocessor compiling variables and constraints over finite domains into logic programs. It offers various options leading to (non-ground) enodings that can be grounded by *gringo*. Details can be found in [22].

lp2txt is a simple script that transforms ground *lparse* output format back into human-readable format.

plasp is an interpreter for a subset of the Planning Domain Definition Language (PDDL). Since it uses ASP for the actual search, it can also be seen as a PDDL to ASP compiler. For solving, a modified version of *iclingo* is used.
Details can be found in [74,43].

pyngo is a bottom-up ASP grounder written in Python with the goal to provide a well-documented grounder exploring bottom-up grounding and related techniques.

sbass detects and breaks syntactic symmetries in logic programs by adding respective constraints.
Details can be found in [21].

xorro exploits XOR constraints to calculate samples with near uniform distribution, inspired by a similar approach in the field of SAT [63]. Hence, it allows for calculating a few answer sets representative for all answer sets of a logic program. This is particularly useful if the computation of all answer sets is practically infeasible.

xpanda is a preprocessor compiling variables and constraints over finite domains into logic programs that can be grounded by *gringo*. Its compilation methods are less efficient yet simpler than the ones of *inca*.
Details can be found in [37].

misc is not a particular tool, but a collection of miscellaneous helper scripts and files.

And there is more to come in the future (see below).

13. Discussion

The goal of the *Potassco* initiative is to furnish an open access to tools for ASP. This is why *Potassco* is hosted at *Sourceforge*, a prime location for downloading and developing free open source software. In this sense, *Potassco* is meant as a community platform for users and developers of ASP software. In addition to the available sources and binaries for Linux, Macintosh[11], and Windows at `http://potassco.sourceforge.net`, most of the aforementioned systems are meanwhile also available as Debian and Ubuntu packages and can thus be easily integrated in existing Linux environments.[12]

Upcoming extensions to *Potassco* include a reactive ASP system, *oclingo*, that allows for incorporating online data streams (and requests) coming from external sources (see [34]), a Linux package configuration system, *aspcud*, a pre-processor, *metasp*, offering complex optimization capacities, supporting, for instance, inclusion-based minimization or Pareto efficiency, an extension of *gringo*'s input language with *dlv*-style aggregates and weak constraints, and last but not least the new construction series 2.0 of *clasp* is close to be released, featuring multi-threading and advanced optimization techniques.

Also, we would like to point the interested reader to the ASP benchmark repository at `http://asparagus.cs.uni-potsdam.de`. It's a great resource for learning about how to encode problems in ASP and thus of particular value for teaching ASP.

[11] Thanks to Gregory Gelfond, ASU!

[12] Thanks to Thomas Krennwallner, TU Vienna!

Acknowledgments

This work was supported by the German Science Foundation (DFG) under grants SCHA 550/8-1 and -2.

The authors are grateful to Arne König, Christian Drescher, Orkunt Sabuncu, Sven Thiele, and Torsten Grote for their support and contributions to the overall efforts of Potassco. Moreover, the authors are indebted to Marcello Balduccini, Mauricio Osorio, and Stefan Woltran for valuable comments on the manuscript of this paper.

References

[1] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison-Wesley, 1995.

[2] K. Apt, H. Blair, and A. Walker. Towards a theory of declarative knowledge. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*, chapter 2, pages 89–148. Morgan Kaufmann Publishers, 1987.

[3] Y. Babovich and V. Lifschitz. Computing answer sets using program completion. Unpublished draft; available at `http://www.cs.utexas.edu/users/tag/cmodels.html`, 2003.

[4] M. Balduccini, E. Pontelli, O. El-Khatib, and H. Le. Issues in parallel execution of non-monotonic reasoning systems. *Parallel Computing*, 31(6):608–647, 2005.

[5] C. Baral. *Knowledge Representation, Reasoning and Declarative Problem Solving*. Cambridge University Press, 2003.

[6] C. Baral, G. Brewka, and J. Schlipf, editors. *Proceedings of the Ninth International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'07)*, volume 4483 of *Lecture Notes in Artificial Intelligence*. Springer-Verlag, 2007.

[7] D. Basak, S. Pal, and D. Patranabis. Support vector regression. *Neural Information Processing — Letters and Reviews*, 11(10), 2007.

[8] S. Baselice, P. Bonatti, and M. Gelfond. Towards an integration of answer set and constraint solving. In M. Gabbrielli and G. Gupta, editors, *Proceedings of the Twenty-first International Conference on Logic Programming (ICLP'05)*, volume 3668 of *Lecture Notes in Computer Science*, pages 52–66. Springer-Verlag, 2005.

[9] A. Biere, M. Heule, H. van Maaren, and T. Walsh. *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*. IOS Press, 2009.

[10] G. Boenn, M. Brain, M. de Vos, and J. Fitch. Automatic composition of melodic and harmonic music by answer set programming. In Garcia de la Banda and Pontelli [33], pages 160–174.

[11] T. Bylander. The computational complexity of propositional STRIPS planning. *Artificial Intelligence*, 69(1-2):165–204, 1994.

[12] K. Clark. Negation as failure. In H. Gallaire and J. Minker, editors, *Logic and Data Bases*, pages 293–322. Plenum Press, 1978.

[13] O. Cliffe, M. de Vos, M. Brain, and J. Padget. ASPVIZ: Declarative visualisation and animation using answer set programming. In Garcia de la Banda and Pontelli [33], pages 724–728.

[14] E. Dantsin, T. Eiter, G. Gottlob, and A. Voronkov. Complexity and expressive power of logic programming. *ACM Computing Surveys*, 33(3):374–425, 2001.

[15] A. Darwiche and K. Pipatsrisawat. Complete algorithms. In *Handbook of Satisfiability* [9], chapter 3, pages 99–130.

[16] M. de Vos and T. Schaub, editors. *Proceedings of the Workshop on Software Engineering for Answer Set Programming (SEA'07)*, Department of Computer Science, University of Bath, Technical Report Series, 2007.

[17] M. de Vos and T. Schaub, editors. *Proceedings of the Second Workshop on Software Engineering for Answer Set Programming (SEA'09)*, Department of Computer Science, University of Bath, Technical Report Series, 2009.

[18] J. Delgrande and W. Faber, editors. *Proceedings of the Eleventh International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'11)*, Lecture Notes in Artificial Intelligence. Springer-Verlag, 2011. To appear.

[19] R. Dechter. *Constraint Processing*. Morgan Kaufmann Publishers, 2003.

[20] C. Drescher, M. Gebser, T. Grote, B. Kaufmann, A. König, M. Ostrowski, and T. Schaub. Conflict-driven disjunctive answer set solving. In G. Brewka and J. Lang, editors, *Proceedings of the Eleventh International Conference on Principles of Knowledge Representation and Reasoning (KR'08)*, pages 422–432. AAAI Press, 2008.

[21] C. Drescher, O. Tifrea, and T. Walsh. Symmetry-breaking answer set solving. In M. Balduccini and S. Woltran, editors, *Proceedings of ICLP'10 Workshop on Answer Set Programming and Other Computing Paradigm*, 2010.

[22] C. Drescher and T. Walsh. A translational approach to constraint answer set solving. In *Theory and Practice of Logic Programming. Twenty-sixth International Conference on Logic Programming (ICLP'10) Special Issue*, volume 10(4-6), pages 465–480. Cambridge University Press, 2010.

[23] S. Dworschak, S. Grell, V. Nikiforova, T. Schaub, and J. Selbig. Modeling biological networks by action languages via answer set programming. *Constraints*, 13(1-2):21–65, 2008.

[24] S. Dworschak, T. Grote, A. König, T. Schaub, and P. Veber. The system BioC for reasoning about biological models in action language $\mathcal{C}$. In *Proceedings of the Twentieth International Conference on Tools with Artificial Intelligence (ICTAI'08)*, volume 1, pages 11–18. IEEE Computer Society Press, 2008.

[25] N. Eén and A. Biere. Effective preprocessing in SAT through variable and clause elimination. In F. Bacchus and T. Walsh, editors, *Proceedings of the Eigth International Conference on Theory and Applications of Satisfiability Testing (SAT'05)*, volume 3569 of *Lecture Notes in Computer Science*, pages 61–75. Springer-Verlag, 2005.

[26] N. Eén and N. Sörensson. Temporal induction by incremental SAT solving. *Electronic Notes in Theoretical Computer Science*, 89(4), 2003.

[27] N. Eén and N. Sörensson. An extensible SAT-solver. In E. Giunchiglia and A. Tacchella, editors, *Proceedings of the Sixth International Conference on Theory and Applications of Satisfiability Testing (SAT'03)*, volume 2919 of *Lecture Notes in Computer Science*, pages 502–518. Springer-Verlag, 2004.

[28] E. Ellguth, M. Gebser, M. Gusowski, R. Kaminski, B. Kaufmann, S. Liske, T. Schaub, L. Schneidenbach, and B. Schnor. A simple distributed conflict-driven answer set solver. In Erdem et al. [29], pages 490–495.

[29] E. Erdem, F. Lin, and T. Schaub, editors. *Proceedings of the Tenth International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'09)*, volume 5753 of *Lecture Notes in Artificial Intelligence*. Springer-Verlag, 2009.

[30] E. Erdem and F. Türe. Efficient haplotype inference with answer set programming. In D. Fox and C. Gomes, editors, *Proceedings of the Twenty-third National Conference on Artificial Intelligence (AAAI'08)*, pages 436–441. AAAI Press, 2008.

[31] F. Fages. Consistency of Clark's completion and the existence of stable models. *Journal of Methods of Logic in Computer Science*, 1:51–60, 1994.

[32] J. Freeman. *Improvements to propositional satisfiability search algorithms*. PhD thesis, University of Pennsylvania, 1995.

[33] M. Garcia de la Banda and E. Pontelli, editors. *Proceedings of the Twenty-fourth International Conference on Logic Programming (ICLP'08)*, volume 5366 of *Lecture Notes in Computer Science*. Springer-Verlag, 2008.

[34] M. Gebser, T. Grote, R. Kaminski, and T. Schaub. Reactive answer set programming. In Delgrande and Faber [18]. To appear.

[35] M. Gebser, T. Grote, and T. Schaub. Coala: A compiler from action languages to ASP. In Janhunen and Niemelä [73], pages 360–364.

[36] M. Gebser, C. Guziolowski, M. Ivanchev, T. Schaub, A. Siegel, S. Thiele, and P. Veber. Repair and prediction (under inconsistency) in large biological networks with answer set programming. In F. Lin and U. Sattler, editors, *Proceedings of the Twelfth International Conference on Principles of Knowledge Representation and Reasoning (KR'10)*, pages 497–507. AAAI Press, 2010.

[37] M. Gebser, H. Hinrichs, T. Schaub, and S. Thiele. xpanda: A (simple) preprocessor for adding multi-valued propositions to ASP. In U. Geske and A. Wolf, editors, *Proceedings of the Twenty-third Workshop on (Constraint) Logic Programming (WLP'09)*, 2009.

[38] M. Gebser, R. Kaminski, B. Kaufmann, M. Ostrowski, T. Schaub, and S. Thiele. A user's guide to gringo, clasp, clingo, and iclingo. Available at http://potassco.sourceforge.net.

[39] M. Gebser, R. Kaminski, B. Kaufmann, M. Ostrowski, T. Schaub, and S. Thiele. Engineering an incremental ASP solver. In Garcia de la Banda and Pontelli [33], pages 190–205.

[40] M. Gebser, R. Kaminski, B. Kaufmann, and T. Schaub. On the implementation of weight constraint rules in conflict-driven ASP solvers. In Hill and Warren [69], pages 250–264.

[41] M. Gebser, R. Kaminski, B. Kaufmann, T. Schaub, M. Schneider, and S. Ziller. A portfolio solver for answer set programming: Preliminary report. In Delgrande and Faber [18]. To appear.

[42] M. Gebser, R. Kaminski, B. Kaufmann, T. Schaub, and B. Schnor. Cluster-based asp solving with claspar. In Delgrande and Faber [18]. To appear.

[43] M. Gebser, R. Kaminski, M. Knecht, and T. Schaub. plasp: A compiler from PDDL to ASP. In Delgrande and Faber [18]. To appear.

[44] M. Gebser, R. Kaminski, A. König, and T. Schaub. Advances in gringo series 3. In Delgrande and Faber [18]. To appear.

[45] M. Gebser, R. Kaminski, M. Ostrowski, T. Schaub, and S. Thiele. On the input language of ASP grounder gringo. In Erdem et al. [29], pages 502–508.

[46] M. Gebser, B. Kaufmann, A. Neumann, and T. Schaub. clasp: A conflict-driven answer set solver. In Baral et al. [6], pages 260–265.

[47] M. Gebser, B. Kaufmann, A. Neumann, and T. Schaub. Conflict-driven answer set enumeration. In Baral et al. [6], pages 136–148.

[48] M. Gebser, B. Kaufmann, A. Neumann, and T. Schaub. Conflict-driven answer set solving. In M. Veloso, editor, *Proceedings of the Twentieth International Joint Conference on Artificial Intelligence (IJCAI'07)*, pages 386–392. AAAI Press/The MIT Press, 2007.

[49] M. Gebser, B. Kaufmann, A. Neumann, and T. Schaub. Advanced preprocessing for answer set solving. In M. Ghallab, C. Spyropoulos, N. Fakotakis, and N. Avouris, editors, *Proceedings of the Eighteenth European Conference on Artificial Intelligence (ECAI'08)*, pages 15–19. IOS Press, 2008.

[50] M. Gebser, B. Kaufmann, and T. Schaub. The conflict-driven answer set solver clasp: Progress report. In Erdem et al. [29], pages 509–514.

[51] M. Gebser, B. Kaufmann, and T. Schaub. Solution enumeration for projected Boolean search problems. In W. van Hoeve and J. Hooker, editors, *Proceedings of the Sixth International Conference on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems (CPAIOR'09)*, volume 5547 of *Lecture Notes in Computer Science*, pages 71–86. Springer-Verlag, 2009.

[52] M. Gebser, M. Ostrowski, and T. Schaub. Constraint answer set solving. In Hill and Warren [69], pages 235–249.

[53] M. Gebser, O. Sabuncu, and T. Schaub. An incremental answer set programming based system for finite model computation. In Janhunen and Niemelä [73], pages 169–181.

[54] M. Gebser, T. Schaub, and S. Thiele. Gringo: A new grounder for answer set programming. In Baral et al. [6], pages 266–271.

[55] M. Gebser, T. Schaub, S. Thiele, and P. Veber. Detecting inconsistencies in large biological networks with answer set programming. *Theory and Practice of Logic Programming*, 11(2):1–38, 2011.

[56] Gecode: Generic constraint development environment, 2006. Available from http://www.gecode.org.

[57] M. Gelfond. Answer sets. In Lifschitz et al. [79], chapter 7, pages 285–316.

[58] M. Gelfond and V. Lifschitz. The stable model semantics for logic programming. In R. Kowalski and K. Bowen, editors, *Proceedings of the Fifth International Conference and Symposium of Logic Programming (ICLP'88)*, pages 1070–1080. The MIT Press, 1988.

[59] M. Gelfond and V. Lifschitz. Action languages. *Electronic Transactions on Artificial Intelligence*, 3(6):193–210, 1998.

[60] E. Giunchiglia, J. Lee, V. Lifschitz, N. McCain, and H. Turner. Nonmonotonic causal theories. *Artificial Intelligence*, 153(1-2):49–104, 2004.

[61] E. Giunchiglia, Y. Lierler, and M. Maratea. Answer set programming based on propositional satisfiability. *Journal of Automated Reasoning*, 36(4):345–377, 2006.

[62] E. Goldberg and Y. Novikov. BerkMin: A fast and robust SAT solver. In *Proceedings of the Fifth Conference on Design, Automation and Test in Europe (DATE'02)*, pages 142–149. IEEE Press, 2002.

[63] C. Gomes, A. Sabharwal, and B. Selman. Near-uniform sampling of combinatorial spaces using XOR constraints. In B. Schölkopf, J. Platt, and T. Hoffman, editors, *Proceedings of the Twentieth Annual Conference on Neural Information Processing Systems (NIPS'06)*, pages 481–488. MIT Press, 2006.

[64] G. Grasso, S. Iiritano, N. Leone, V. Lio, F. Ricca, and F. Scalise. An ASP-based system for team-building in the Gioia-Tauro seaport. In M. Carro and R. Peña, editors, *Proceedings of the Twelfth International Symposium on Practical Aspects of Declarative Languages (PADL'10)*, volume 5937 of *Lecture Notes in Computer Science*, pages 40–42. Springer-Verlag, 2010.

[65] J. Gressmann, T. Janhunen, R. Mercer, T. Schaub, S. Thiele, and R. Tichy. Platypus: A platform for distributed answer set solving. In C. Baral, G. Greco, N. Leone, and G. Terracina, editors, *Proceedings of the Eighth International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'05)*, volume 3662 of *Lecture Notes in Artificial Intelligence*, pages 227–239. Springer-Verlag, 2005.

[66] J. Gressmann, T. Janhunen, R. Mercer, T. Schaub, S. Thiele, and R. Tichy. On probing and multi-threading in platypus. In G. Brewka, S. Coradeschi, A. Perini, and P. Traverso, editors, *Proceedings of the Seventeenth European Conference on Artificial Intelligence (ECAI'06)*, pages 392–396. IOS Press, 2006.

[67] W. Gropp, E. Lusk, and R. Thakur. *Using MPI-2: Advanced Features of the Message-Passing Interface*. The MIT Press, 1999.

[68] K. Heljanko and I. Niemelä. Bounded LTL model checking with stable models. *Theory and Practice of Logic Programming*, 3(4-5):519–550, 2003.

[69] P. Hill and D. Warren, editors. *Proceedings of the Twenty-fifth International Conference on Logic Programming (ICLP'09)*, volume 5649 of *Lecture Notes in Computer Science*. Springer-Verlag, 2009.

[70] R. Ierusalimschy. *Programming in Lua*. lua.org, 2006.

[71] H. Ishebabi, P. Mahr, C. Bobda, M. Gebser, and T. Schaub. Answer set vs integer linear programming for automatic synthesis of multiprocessor systems from real-time parallel programs. *Journal of Reconfigurable Computing*, 2009.

[72] T. Janhunen. Intermediate languages of ASP systems and tools. In de Vos and Schaub [16], pages 12–25. ISSN 1740-9497.

[73] T. Janhunen and I. Niemelä, editors. *Proceedings of the Twelfth European Conference on Logics in Artificial Intelligence (JELIA'10)*, volume 6341 of *Lecture Notes in Artificial Intelligence*. Springer-Verlag, 2010.

[74] M. Knecht. Efficient domain-independent planning using declarative programming. M.Sc. thesis, Institute for Informatics, University of Potsdam, 2009.

[75] N. Leone, G. Pfeifer, W. Faber, T. Eiter, G. Gottlob, S. Perri, and F. Scarcello. The DLV system for knowledge representation and reasoning. *ACM Transactions on Computational Logic*, 7(3):499–562, 2006.

[76] V. Lifschitz. Answer set programming and plan generation. *Artificial Intelligence*, 138(1-2):39–54, 2002.

[77] V. Lifschitz and A. Razborov. Why are there so many loop formulas? *ACM Transactions on Computational Logic*, 7(2):261–268, 2006.

[78] V. Lifschitz and H. Turner. Representing transition systems by logic programs. In M. Gelfond, N. Leone, and G. Pfeifer, editors, *Proceedings of the Fifth International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'99)*, volume 1730 of *Lecture Notes in Artificial Intelligence*, pages 92–106. Springer-Verlag, 1999.

[79] V. Lifschitz, F. van Hermelen, and B. Porter, editors. *Handbook of Knowledge Representation*. Elsevier, 2008.

[80] F. Lin and Y. Zhao. ASSAT: computing answer sets of a logic program by SAT solvers. *Artificial Intelligence*, 157(1-2):115–137, 2004.

[81] J. Marques-Silva, I. Lynce, and S. Malik. Conflict-driven clause learning SAT solvers. In *Handbook of Satisfiability* [9], chapter 4, pages 131–153.

[82] V. Mellarkod and M. Gelfond. Integrating answer set reasoning with constraint solving techniques. In J. Garrigue and M. Hermenegildo, editors, *Proceedings of the Ninth International Symposium on Functional and Logic Programming (FLOPS'08)*, volume 4989 of *Lecture Notes in Computer Science*, pages 15–31. Springer-Verlag, 2008.

[83] V. Mellarkod, M. Gelfond, and Y. Zhang. Integrating answer set programming and constraint logic programming. *Annals of Mathematics and Artificial Intelligence*, 53(1-4):251–287, 2008.

[84] M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient SAT solver. In *Proceedings of the Thirty-eighth Conference on Design Automation (DAC'01)*, pages 530–535. ACM Press, 2001.

[85] R. Nieuwenhuis, A. Oliveras, and C. Tinelli. Solving SAT and SAT modulo theories: From an abstract Davis-Putnam-Logemann-Loveland procedure to DPLL(T). *Journal of the ACM*, 53(6):937–977, 2006.

[86] M. Nogueira, M. Balduccini, M. Gelfond, R. Watson, and M. Barry. An A-prolog decision support system for the space shuttle. In I. Ramakrishnan, editor, *Proceedings of the Third International Symposium on Practical Aspects of Declarative Languages (PADL'01)*, volume 1990 of *Lecture Notes in Computer Science*, pages 169–183. Springer-Verlag, 2001.

[87] K. Pipatsrisawat and A. Darwiche. A lightweight component caching scheme for satisfiability solvers. In J. Marques-Silva and K. Sakallah, editors, *Proceedings of the Tenth International Conference on Theory and Applications of Satisfiability Testing (SAT'07)*, volume 4501 of *Lecture Notes in Computer Science*, pages 294–299. Springer-Verlag, 2007.

[88] E. Pontelli, M. Balduccini, and F. Bermudez. Non-monotonic reasoning on Beowulf platforms. In V. Dahl and P. Wadler, editors, *Proceedings of the Fifth International Symposium on Practical Aspects of Declarative Languages (PADL'03)*, volume 2562 of *Lecture Notes in Artificial Intelligence*, pages 37–57. Springer-Verlag, 2003.

[89] F. Rossi, P. van Beek, and T. Walsh, editors. *Handbook of Constraint Programming*. Elsevier, 2006.

[90] L. Ryan. Efficient algorithms for clause-learning SAT solvers. Master's thesis, Simon Fraser University, 2004.

[91] V. Ryvchin and O. Strichman. Local restarts. In H. Kleine Büning and X. Zhao, editors, *Proceedings of the Eleventh International Conference on Theory and Applications of Satisfiability Testing (SAT'08)*, volume 4996 of *Lecture Notes in Computer Science*, pages 271–276. Springer-Verlag, 2008.

[92] M. Schaefer and C. Umans. Completeness in the polynomial-time hierarchy: A compendium. *ACM SIGACT News*, 33(3):32–49, 2002. Updated version available at `http://ovid.cs.depaul.edu/documents/phcom.ps`.

[93] J. Schlipf. The expressive powers of the logic programming semantics. *Journal of Computer and System Sciences*, 51:64–86, 1995.

[94] L. Schneidenbach, B. Schnor, M. Gebser, R. Kaminski, B. Kaufmann, and T. Schaub. Experiences running a parallel answer set solver on Blue Gene. In M. Ropo, J. Westerholm, and J. Dongarra, editors, *Proceedings of the Sixteenth European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface (PVM/MPI'09)*, volume 5759 of *Lecture Notes in Computer Science*, pages 64–72. Springer-Verlag, 2009.

[95] P. Simons, I. Niemelä, and T. Soininen. Extending and implementing the stable model semantics. *Artificial Intelligence*, 138(1-2):181–234, 2002.

[96] T. Soininen and I. Niemelä. Developing a declarative rule language for applications in product configuration. In G. Gupta, editor, *Proceedings of the First International Workshop on Practical Aspects of Declarative Languages (PADL'99)*, volume 1551 of *Lecture Notes in Computer Science*, pages 305–319. Springer-Verlag, 1999.

[97] T. Son, C. Baral, T. Nam, and S. McIlraith. Domain-dependent knowledge in answer set planning. *ACM Transactions on Computational Logic*, 7(4):613–657, 2006.

[98] T. Syrjänen. Lparse 1.0 user's manual. `http://www.tcs.hut.fi/Software/smodels/lparse.ps.gz`.

[99] L. Xu, F. Hutter, H. Hoos, and K. Leyton-Brown. SATzilla: Portfolio-based algorithm selection for SAT. *Journal of Artificial Intelligence Research*, 32:565–606, 2008.

[100] H. Zhang, M. Bonacina, and J. Hsiang. PSATO: a distributed propositional prover and its application to quasigroup problems. *Journal of Symbolic Computation*, 21(4):543–560, 1996.