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 \(NP^{\text{NP}}\)) 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.
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(1,3), edge(1,4), ..., 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.

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:

\[
\{ \text{cycle}(1,2), \text{cycle}(1,3), \text{cycle}(1,4) \}.
\]

Note that grounding simplifies the rule by eliminating true components. The above constraint stipulates
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 ∈ {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,b,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
\[ \text{conf}(S, L, A, R) \leftarrow \text{start}(S), \text{tape}(L, A, R). \]
\[ \text{conf}(SN, 1(L, AN), AR, R) \leftarrow \text{conf}(S, L, A, r(AR, R)), \text{trans}(S, A, AN, SN, r). \]
\[ \text{conf}(SN, 1(L, AN), AR, n) \leftarrow \text{conf}(S, L, A, n), \text{blank}(AR), \text{trans}(S, A, AN, SN, r). \]
\[ \text{conf}(SN, n, AL, r(AN, R)) \leftarrow \text{conf}(S, l(L, AL), A, R), \text{trans}(S, A, AN, SN, l). \]
\[ \text{conf}(SN, n, AL, r(AN, R)) \leftarrow \text{conf}(S, n, A, R), \text{blank}(AL), \text{trans}(S, A, AN, SN, l). \]

Table 4
An ASP encoding of a universal Turing Machine

by facts over \text{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 \text{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 \text{gringo} with files containing the rules in Table 3 (\text{beaver}) and Table 4 (\text{turing}) yields:

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

In fact, the Turing Machine is completely evaluated by \text{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 \text{gringo}, starting with version 3.0. A comprehensive documentation is found in \text{gringo}'s manual [38]; previous versions are described in [54,45]. First of all, the 3.0 series only stipulates rules to be \text{safe} (cf. [1]) rather than to be domain-restricted through additional domain predicates, as in previous versions of \text{gringo}. As a consequence, programs are no longer subject to any restriction guaranteeing a finite grounding (like \( \lambda \)-restrictedness [54]). Rather, this responsibility is left with the user in order to provide her with the greatest flexibility. To see this, consider the \( \lambda \)-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>
<thead>
<tr>
<th>Table 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>A ( \lambda )-restricted logic program for the transitive closure of ( q/2 )</td>
</tr>
</tbody>
</table>

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 ground-
\[ q(1,2). \ q(2,3). \ q(3,1). \ q(X,Z) :- q(X,Y), q(Y,Z). \]

<table>
<thead>
<tr>
<th>Table 6</th>
</tr>
</thead>
<tbody>
<tr>
<td>A safe logic program for the transitive closure of ( q/2 )</td>
</tr>
</tbody>
</table>

\[ 1 \ #count \ { \ cycle(X,Y) : \ edge(X,Y) } \ 1. \]

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 is 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
      backjump  // undo assignments until conflict constraint is unit
  // analyze conflict and add a conflict constraint
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]). 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 models7 of a logic program (via command line option --supp-models). 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-relevant 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-

---

6Informally, tightness [31] indicates that a program is free of recursion through positive literals.
7The models of the Clark completion [12] of a program are called supported models [2]. On tight programs, supported models and answer sets coincide [31].
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 \texttt{--cautious} and \texttt{--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 \texttt{#minimize} or \texttt{#maximize} statements. For finding optimal solutions, \texttt{clasp} offers several options. First, \texttt{clasp} allows for computing one or all (\texttt{--opt-all}) optimal solutions. Second, the objective function can be initialized via \texttt{--opt-value}. The latter turns out to be useful when one is interested in computing consequences belonging to all optimal solutions (in combination with \texttt{--cautious}). To this end, one starts with searching for an (arbitrary) optimal answer set and then re-launches \texttt{clasp} by bounding its search with the obtained optimum. Doing the latter with \texttt{--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 \texttt{--restart-on-model} (making \texttt{clasp} restart after each (putatively) optimal solution) in order to ameliorate convergence to the optimum. Moreover, option \texttt{--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 \texttt{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 \texttt{--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 \texttt{clasp}'s manifold reasoning modes.

### 4.3. Propagation and Search

Propagation in \texttt{clasp} relies on an interface \textit{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 \textit{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 \texttt{--loops}. In the default setting, loop nogoods are created for individual unfounded atoms, as shown in [40].

\texttt{clasp}'s primary decision heuristics, selectable via option \texttt{--heuristic}, use \textit{look-back} strategies derived from corresponding CDCL-based approaches in SAT, viz., \texttt{vsids} [84], \texttt{berkmin} [62], and \texttt{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 \texttt{berkmin} and \texttt{vsids} lies in the scope of variables considered during decision making. While \texttt{vsids} selects the free variable that is globally most active, \texttt{berkmin} restricts the selection to variables belonging to the most recently recorded but yet unsatisfied conflict nogood. Although the look-back heuristics implemented in \texttt{clasp} are modeled after the corresponding CDCL-based approaches, one important difference is that \texttt{clasp} optionally also scores variables contained in loop nogoods. In case of \texttt{berkmin}, it may also select a free variable belonging to a recently recorded loop nogood. Finally, we note that \texttt{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 \texttt{--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 \texttt{--restarts}. Usually, restart strategies are based on the global number of conflicts. Beyond that, clasp features local restarts [91], which can be activated with \texttt{--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 \texttt{--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 \textit{NP}^{\textit{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. 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.

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 (v1,⋯,vi−1,ui), can be divided into a prefix (v1,⋯,vi−1) of non-splittable variables and a postfix (ui,⋯,vn) 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 (u1,⋯,ui−1,ui). 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 πi,...,πj indicate that the search spaces for answer sets containing (v1,⋯,vi−1) and at least one of u1,⋯,uj have already been explored. In order to avoid repetitions, it is thus important to pass guiding path (v1,⋯,vi−1,πi,...,πj,vi) 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

---

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

```bash
$ 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:

```bash
$ 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 \( \neg p \) and \( \neg q \), respectively. The initial situation fulfills \( p \), and the goal is to satisfy \( r \).

This planning problem can be solved by the ASP encoding in Table 8. First of all, observe that the length 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 \( \text{not ocdel}(F,T) \) 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 \( B \cup \bigcup_{i \leq j \leq k} [P[k/j] \cup Q[k/i]] \) 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 \( \text{holds}/2 \) 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 \( \text{time}/1 \) 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.

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 \( --i\text{learnt} \) and \( --i\text{heuristic} \) 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.

### Table 9

An incremental ASP encoding of STRIPS Planning

<table>
<thead>
<tr>
<th>#base.</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{holds}(P,0) :- \text{init}(P). )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>#cumulative ( t )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( 1 { \text{occ}(A,t) : \text{action}(A) } 1. )</td>
</tr>
<tr>
<td>( :- \text{occ}(A,t), \text{pre}(A,F), \text{not holds}(F,t-1). )</td>
</tr>
<tr>
<td>( \text{holds}(F,t) :- \text{holds}(F,t-1), \text{not ocdel}(F,t). )</td>
</tr>
<tr>
<td>( \text{holds}(F,t) :- \text{occ}(A,t), \text{add}(A,F). )</td>
</tr>
<tr>
<td>( \text{ocdel}(F,t) :- \text{occ}(A,t), \text{del}(A,F). )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>#volatile ( t )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( :- \text{query}(F), \text{not holds}(F,t). )</td>
</tr>
</tbody>
</table>

---

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]). To this end, clingcon adopts techniques from

---

10Groundbreaking 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

\[ \text{volume}(b,2) \equiv \text{volume}(b,1) + \text{amount}(b,1). \]

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, \( \text{volume}(b,2), \text{volume}(b,1), \) and \( \text{amount}(b,1) \), such that the equation satisfies the assigned truth value.

10. 

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](image)

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 $\leq$ amount(B,T) :- pour(B,T), T < t.

amount(B,T) $\leq$ 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.

coola originates from al2asp, constituting the heart of the BioC system [24] used for reasoning about biological models in action language C_{TAID} [23]: al2asp compiles C_{TAID} to 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 C+ [60] and provides full support of B [97] (and 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, P, Q, and 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 classification 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) encodings 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. Details can be found in [21].

**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, Macintosh¹¹, 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, *metaslp*, offering complex optimization capacities, supporting, for instance, inclusion-based minimization or Pareto efficiency, an extension of *gringo*’s input language with div-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.

### 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


¹¹Thanks to Gregory Gelfond, ASU!

¹²Thanks to Thomas Krennwallner, TU Vienna!


