ASP modulo CSP: The clingcon system

Max Ostrowski

Institut für Informatik, Universität Potsdam, August-Bebel-Str. 89, D–14482 Potsdam, Germany ostrowsk@cs.uni-potsdam.de

Abstract

Answer Set Programming (ASP; [1]) has become a prime paradigm for declarative problem solving due to its combination of an easy yet expressive modeling language with high-performance Boolean constraint solving technology. However, certain applications are more naturally modeled by mixing Boolean with non-Boolean constructs, for instance, accounting for resources, fine timings, or functions over finite domains. The challenge lies in combining the elaborated solving capacities of ASP, like backjumping and conflict-driven learning, with advanced techniques from the area of constraint programming (CP). I therefore developed the solver *clingcon*, which follows the approach of modern Satisfiability Modulo Theories (SMT; [2, Chapter 26]). My research shall contribute to bridging the gap between Boolean and Non-Boolean reasoning, in order to bring out the best of both worlds.

1998 ACM Subject Classification D.1.6. Logic Programming, I.2.3 Deduction and Theorem proving/Logic programming, D.3.2 Language Classifications/Constraint and logic languages

Keywords and phrases Answer Set Programming, Constraint Programming

Digital Object Identifier 10.4230/LIPIcs.ICLP.2012.458

1 Introduction and Motivation

clingcon is a hybrid solver for ASP, combining the simple modeling language and the high performance Boolean solving capacities of ASP with techniques for using non-Boolean constraints from the area of Constraint Programming (CP). Although *clingcon*'s solving components follow the approach of modern Satisfiability Modulo Theories (SMT; [2, Chapter 26]) solvers when combining the ASP solver *clasp* with the CP solver *gecode* [3], *clingcon* furthermore adheres to the tradition of ASP in supporting a corresponding modeling language by appeal to the ASP grounder *gringo*. Although in the current implementation the theory solver is instantiated with the CP solver *gecode*, the principal design of *clingcon* along with the corresponding interfaces are conceived in a generic way, aiming at arbitrary theory solvers.

I will first give a general background over the theory of ASP and CP and will afterwards describe the architecture of *clingcon* which follows the approach of SMT. Given this, the main contribution of my work is a comparison of simple methods to compute minimal inconsistencies and explanations for any black-box CP system. These minimal conflicts and reasons can then be used for driving the conflict-driven learning process of the system. These methods have been implemented in the system *clingcon* and yield a performance improvement of an order of magnitude on a broad range of benchmarks.

2 Background

A (*normal*) *logic program* over an alphabet $\mathcal{A}$ is a finite set of *rules* of the form

$$a_0 \leftarrow a_1, \ldots, a_m, \mathit{not}\ a_{m+1}, \ldots, \mathit{not}\ a_n\ , \tag{1}$$

Technical Communications of the 28th International Conference on Logic Programming (ICLP'12).
Editors: A. Dovier and V. Santos Costa; pp. 458–463

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

where $a_i \in \mathcal{A}$ is an *atom* for $0 \leq i \leq n$.[1] A *literal* is an atom a or its (default) negation *not* a. For a rule r as in (1), let $head(r) = a_0$ be the *head* of r and $body(r) = \{a_1, \ldots, a_m, not\ a_{m+1}, \ldots, not\ a_n\}$ be the *body* of r. Given a set B of literals, let $B^+ = \{a \in \mathcal{A} \mid a \in B\}$ and $B^- = \{a \in \mathcal{A} \mid not\ a \in B\}$. Furthermore, given some set $\mathcal{B}$ of atoms, define $B|_{\mathcal{B}} = (B^+ \cap \mathcal{B}) \cup \{not\ a \mid a \in B^- \cap \mathcal{B}\}$. The set of atoms occurring in a logic program P is denoted by $atom(P)$. A set $X \subseteq \mathcal{A}$ is an *answer set* of a program P over $\mathcal{A}$, if X is the $\subseteq$-smallest model of the *reduct* $P^X = \{head(r) \leftarrow body(r)^+ \mid r \in P, body(r)^- \cap X = \emptyset\}$. An answer set can also be seen as a Boolean assignment satisfying all conditions induced by program P (cf. [4]).

A *constraint satisfaction problem* (CSP) is a triple (V, D, C), where V is a set of *variables* with respective *domains* D, and C is a set of *constraints*. Each variable $v \in V$ has an associated domain $dom(v) \in D$. Following [5], a constraint c is a pair (S, R) consisting of a k-ary *relation* R defined on a vector $S \subseteq V^k$ of variables, called the *scope* of R. That is, for $S = (v_1, \ldots, v_k)$, we have $R \subseteq dom(v_1) \times \cdots \times dom(v_k)$. We use $S(c) = S$ and $R(c) = R$ to access the scope and the relation of $c = (S, R)$. For an assignment $A : V \to \bigcup_{v \in V} dom(v)$ and a constraint (S, R) with $S = (v_1, \ldots, v_k)$, define $A(S) = (A(v_1), \ldots, A(v_k))$, and let $sat_C(A) = \{c \in C \mid A(S(c)) \in R(c)\}$.

The input language of *clingcon* extends the one of *gringo* (cf. [6]) by CP-specific operators marked with a preceding `$` symbol. After grounding, a propositional program is then composed of regular and constraint atoms, denoted by $\mathcal{A}$ and $\mathcal{C}$, respectively. The set of constraint atoms induces an ordinary constraint satisfaction problem (CSP) (V, D, C). This CSP is to be addressed by the corresponding CP solver, in our case *gecode*. As detailed in [7], the semantics of such constraint logic programs is defined by appeal to a two-step reduction. For this purpose, we consider a regular Boolean assignment over $\mathcal{A} \cup \mathcal{C}$ (in other words, an interpretation) and an assignment of V to D (for interpreting the variables V in the underlying CSP). In the first step, the constraint logic program is reduced to a regular logic program by evaluating its constraint atoms. To this end, the constraints in C associated with the program's constraint atoms $\mathcal{C}$ are evaluated w.r.t. the assignment of V to D. In the second step, the common Gelfond-Lifschitz reduct [8] is performed to determine whether the Boolean assignment is an answer set of the obtained regular logic program. If this is the case, the two assignments constitute a (hybrid) constraint answer set of the original constraint logic program.

In what follows, we rely upon the following terminology. We use signed literals of form **T**a and **F**a to express that an atom a is assigned **T** or **F**, respectively. That is, **T**a and **F**a stand for the Boolean assignments $a \mapsto \mathbf{T}$ and $a \mapsto \mathbf{F}$, respectively. We denote the complement of such a literal ℓ by $\overline{\ell}$. That is, $\overline{\mathbf{T}a} = \mathbf{F}a$ and $\overline{\mathbf{F}a} = \mathbf{T}a$. We represent a Boolean assignment simply by a set of signed literals. Sometimes we restrict such an assignment A to its regular or constraint atoms by writing $A|_{\mathcal{A}}$ or $A|_{\mathcal{C}}$, respectively. For instance, given the regular atom '`person(adam)`' and the constraint atom '`work(adam) $> 4`', we may form the Boolean assignment $\{\mathbf{T}\texttt{person(adam)}, \mathbf{F}work(adam)\ \$ > 4\}$.

We identify constraint atoms in $\mathcal{C}$ with constraints in (V, D, C) via a function $\gamma : \mathcal{C} \to C$. Provided that each constraint $c \in C$ has a complement $\overline{c} \in C$, like $\overline{`x = y'} = `x \neq y'$ or $\overline{`x < y'} = `x \geq y'$ and vice versa, we can extend γ to signed constraint atoms over $\mathcal{C}$ as

[1] The semantics of choice rules and integrity constraints is given through program transformations. For instance, $\{a\} \leftarrow$ is a shorthand for $a \leftarrow not\ a'$ plus $a' \leftarrow not\ a$ and similarly $\leftarrow a$ for $a' \leftarrow a, not\ a'$, for a new atom a'.

follows.

$$\gamma(\ell) = \begin{cases} c & \text{if } \ell = \mathbf{T}c \\ \overline{c} & \text{if } \ell = \mathbf{F}c \end{cases}$$

For instance, we get $\gamma(\mathbf{F}work(adam)\ \$ >\ 4) = work(adam) \leq 4$, where `work(adam)` $\in V$ is a constraint variable and $(work(adam) \leq 4) \in C$ is a constraint. An assignment satisfying the last constraint is $\{work(adam) \mapsto 3\}$.

Following [4], we represent Boolean constraints issuing from a logic program under ASP semantics in terms of *nogoods* [5]. This allows us to view inferences in ASP as unit propagation on nogoods. A *nogood* is a set $\{\sigma_1, \ldots, \sigma_m\}$ of signed literals, expressing that any assignment containing $\sigma_1, \ldots, \sigma_m$ is unintended. Accordingly, a total assignment A is a *solution* for a set Δ of nogoods if $\delta \not\subseteq A$ for all $\delta \in \Delta$. Whenever $\delta \subseteq A$, the nogood δ is said to be *conflicting* with A. For instance, given atoms a, b, the total assignment $\{\mathbf{T}a, \mathbf{F}b\}$ is a solution for the set of nogoods containing $\{\mathbf{T}a, \mathbf{T}b\}$ and $\{\mathbf{F}a, \mathbf{F}b\}$. Likewise, $\{\mathbf{F}a, \mathbf{T}b\}$ is another solution. Importantly, nogoods provide us with reasons explaining why entries must (not) belong to a solution, and lookback techniques can be used to analyze and recombine inherent reasons for conflicts. We refer the interested reader to [4] for details on how logic programs are translated into nogoods within ASP.

3 Research Program

My research work started from the question how to combine the advantages of ASP with Non-Boolean constraint processing techniques. In the process of writing my diploma thesis I developed the hybrid solver *clingcon*. I discovered that the major difficulties lay within the conflict-driven learning techniques of ASP. A black-box CSP solver like *gecode* is not able to provide any useful evidence for its propagation. Such an evidence is needed in an advanced learning setting to provide useful reasons and conflicts for the ASP solver. I addressed this shortcoming by developing mechanisms for extracting minimal reasons and conflicts from any CP solver. The method of minimizing sets of constraints that we present are similar to the ones depicted in [9], Our method furthermore take the incremental nature of the CP solver into account. In contrast to [10], we do not incorporate the learning mechanism into the CP solver but rather use it for the interaction between the ASP and the CP solver. Furthermore, we cope with the difficulty having a black-box system as a CP solver.

Clingcon is based on an algorithm for computing constraint answer sets that extends a previous algorithm to compute standard answer sets [4] by a CP "oracle." The basic algorithm for finding standard answer sets is called Conflict-Driven Nogood Learning (CDNL); it includes conflict-driven learning and backjumping according to the First-UIP scheme [11, 12, 13]. That is, whenever a conflict happens, a conflict nogood containing a Unique Implication Point (UIP) is identified by iteratively resolving a violated nogood against a second nogood that is a reason for some literal in it. A basic CDNL algorithm is depicted in Algorithm 1.

The principal design of *clingcon* along with the corresponding interfaces are conceived in a generic way, aiming at arbitrary theory solvers. The first extension concerns the input language of *gringo* with theory-specific language constructs. Just as with regular atoms, the grounding capabilities of *gringo* can be used for dealing with constraint atoms containing first-order variables. As regards the current *clingcon* system, the language extensions allow for expressing constraints over integer variables. This involves arithmetic constraints as well as global constraints and optimization statements. These constraints are treated as atoms

Algorithm 1: CDNL-ASPmCSP

```
   input  : A program Π.
   output : A constraint answer set of Π.
 1 loop
 2     Propagation
 3     if hasConflict then
 4         if decisionLevel = 0 then return no Answer Set
 5         ConflictAnalysis
 6         Backjump
 7     else if complete Assignment then
 8         Labeling
 9         if hasConflict then
10             Backjump
11         else
12             return Constraint Answer Set
13     else
14         Select
```

and passed to the ASP solver. Information about these constraints is furthermore directly shared with the theory propagator and in turn the theory solver, viz. *gecode*. The theory propagator is implemented as a post propagator. Theory propagation is done by the theory solver until a fixpoint is reached. In doing so, decided constraint atoms are transferred to the theory solver, and conversely constraints whose truth values are determined by the theory solver are sent back to the ASP solver using a corresponding nogood. Note that theory propagation is not only invoked when propagating partial assignments but also whenever a total Boolean assignment is found. Whenever the theory solver detects a conflict, the theory propagator is in charge of conflict analysis. Apart from reverting the state of the theory solver upon backjumping, this involves the crucial task of determining a conflict nogood (which is usually not provided by theory solvers, as in the case of *gecode*). Similarly, the theory propagator is in charge of enumerating constraint variable assignments, whenever needed. Determining a good conflict nogood is the main part of my research that I want to present.

After doing theory propagation either a conflict occurs or some constraints (boolean literals in our case) could be evaluated to true or false. In both cases an explanation is needed, either in form of a conflict or a reason nogood. The *simple* version of generating the conflicting nogood N, is just to take the entire assignment of constraint literals. In this way, all yet decided constraint atoms constitute $N = \{\ell \mid \ell \in A|_{\mathcal{C}}\}$. The corresponding list of inconsistent constraints is

$$I = [\ \gamma(\ell) \mid \ell \in A|_{\mathcal{C}}\ ]. \tag{2}$$

In order to reduce this list of inconsistent constraints and to find the real cause of the conflict, we apply an *Irreducible Inconsistent Set* (IIS) algorithm. The term IIS was coined in [14] for describing inconsistent sets of constraints having consistent subsets only. We use the concept of an IIS to find the minimal cause of a conflict. With this technique, [9] showed that it is actually possible to drastically reduce such exhaustive sets of inconsistent constraints as in (2) and to create a much smaller conflict nogood. Similar to the algorithm

Algorithm 2: FORWARD_FILTERING

input : An inconsistent list of constraints $I = [c_1, \ldots, c_n]$.
output : An irreducible inconsistent list of constraints I'.

```
1 I' ← []
2 while I' is consistent do
3 |   T ← I'
4 |   i ← 1
5 |   while T is consistent do
6 |   |   T ← T ∘ c_i
7 |   |   i ← i + 1
8 |   I' ← I' ∘ c_i
9 return I'
```

in [9], we developed a set of algorithms that exploits the features of an incremental CSP solver even more. I will shortly explain one of these algorithms. Algorithm 2 is called Forward Filtering; it is designed to avoid resetting the search space of the CP solver. It incrementally adds constraints to a testing list T, starting from the first assigned constraint to the last one (lines 5 and 6). Remember that incrementally adding constraints is easy for a CP solver as it can only further restrict the domains. If our test list T becomes inconsistent we add the currently tested constraint to the result I' (lines 5 and 8). If this result is inconsistent (Line 2), we have found a minimal list of inconsistent constraints. Otherwise, we start again, this time adding all yet found constraints I' to our testing list T (Line 1). Now we have to create a new constraint space. But by incrementally increasing the testing list, we already reduced the number of potential candidates that contribute to the IIS, as we never have to check a constraint behind the last added constraint. We illustrate this again on a little example. We start Algorithm 2 with $T = I' = []$ and

$$\begin{aligned} I =& [work(lea) = work(adam), work(john) = 0, work(smith) = 0] \\ & \circ [work(adam) + work(lea) > 6, work(lea) - work(adam) = 1] \end{aligned}$$

in Line 3. We add $work(lea) = work(adam)$ to T, as this constraint alone is consistent, we loop and add constraints until $T = I$. As this list is inconsistent, we add the last constraint $work(lea) - work(adam) = 1$ to I' in Line 8. We can do so, as we know that the last constraint is indispensable for the inconsistency. As I' is consistent we restart the whole procedure, but this time setting $T = I' = [work(lea) - work(adam) = 1]$ in Line 3. Please note that, even if I would contain further constraints, we would never have to check a constraint behind $work(lea) - work(adam) = 1$. Our testing list already contained an inconsistent set of constraints, consequently we can restrict ourself to this subset. Now we start the loop again, adding $work(lea) = work(adam)$ to T. On their own, those two constraints are inconsistent, as there exists no valid pair of values for the variables. So we add $work(lea) = work(adam)$ to I', resulting in $I' = [work(lea) - work(adam) = 1, work(lea) = work(adam)]$. With this much smaller conflict we hope to speed up the search process.

But we can do even more. Up to now we only considered reducing an inconsistent list of constraints to reduce the size of a conflicting nogood. If the CP solver propagates the literal l, a *simple* reason nogood is $N = \{\ell \mid \ell \in A|_{\mathcal{C}}\} \cup \{\bar{l}\}$. If we have for example $A|_{\mathcal{C}} = \{\mathbf{T}work(john)\$ == 0, \mathbf{T}work(lea) - work(adam)\$ == 1\}$, the CP solver propagates the literal $\mathbf{F}work(lea)\$ == work(adam)$. To use the proposed algorithms

to reduce a reason nogood we first have to create an inconsistent list of constraints. As $J = [\gamma(\ell) \mid \ell \in A|_{\mathcal{C}}]$ implies $\gamma(l)$, this inconsistent list is $I = J \circ [\overline{\gamma(l)}] = [work(john) = 0, work(lea) - work(adam) = 1, work(lea) = work(adam)]$. So we can now use these various filtering methods also to reduce reasons generated by the CP solver. In this case the reduced reason is
$\{\mathbf{T}work(lea) - work(adam)\$ == 1, \mathbf{T}work(lea)\$ == work(adam)\}$. Smaller reasons reduce the size of conflicts even more, as they are constructed using unit resolution.

Evaluation on various benchmarks showed that, also filtering conflicts and reasons is a very time consuming process, it can speed up search by order of magnitudes.

4 Future Work

In my future work I want to focus on these reasons and also on a combination of the so called lazy approach which is implemented in *clingcon* with a translational approach. Currently only the ASP solver profits from the additional knowledge of the CP solver. I want to strengthen the CP solving capabilities with features from ASP such as dedicated heuristics like VSIDS and BerkMin and learning.

References

1 Baral, C.: Knowledge Representation, Reasoning and Declarative Problem Solving. Cambridge University Press (2003)
2 Biere, A., Heule, M., van Maaren, H., Walsh, T., eds.: Handbook of Satisfiability. Volume 185 of Frontiers in Artificial Intelligence and Applications. IOS Press (2009)
3 `http://www.gecode.org`
4 Gebser, M., Kaufmann, B., Neumann, A., Schaub, T.: Conflict-driven answer set solving. In Veloso, M., ed.: Proceedings of the Twentieth International Joint Conference on Artificial Intelligence (IJCAI'07), AAAI Press/The MIT Press (2007) 386–392
5 Dechter, R.: Constraint Processing. Morgan Kaufmann Publishers (2003)
6 Gebser, M., Kaminski, R., Kaufmann, B., Ostrowski, M., Schaub, T., Thiele, S.: A user's guide to `gringo`, `clasp`, `clingo`, and `iclingo`. Available at `http://potassco.sourceforge.net`
7 Gebser, M., Ostrowski, M., Schaub, T.: Constraint answer set solving. In Hill, P., Warren, D., eds.: Proc. of the 25th International Conference on Logic Programming (ICLP'09). Volume 5649 of Lecture Notes in Computer Science., Springer-Verlag (2009) 235–249
8 Gelfond, M., Lifschitz, V.: Classical negation in logic programs and disjunctive databases. New Generation Computing **9** (1991) 365–385
9 Junker, U.: QuickXPlain: Conflict detection for arbitrary constraint propagation algorithms. IJCAI'01 Workshop on Modelling and Solving problems with constraints (2001)
10 Moore, N.: Improving the Efficiency of Learning CSP Solvers. University of St Andrews thesis. University of St Andrews (2011)
11 Marques-Silva, J., Sakallah, K.: GRASP: A search algorithm for propositional satisfiability. IEEE Transactions on Computers **48**(5) (1999) 506–521
12 Zhang, L., Madigan, C., Moskewicz, M., Malik, S.: Efficient conflict driven learning in a Boolean satisfiability solver. In: Proceedings of the International Conference on Computer-Aided Design (ICCAD'01). (2001) 279–285
13 Mitchell, D.: A SAT solver primer. Bulletin of the European Association for Theoretical Computer Science **85** (2005) 112–133
14 van Loon, J.: Irreducible inconsistent systems of linear inequalities. In: European Journal of Operational Research. Volume 8., Elsevier Science (1981) 283–288