ASP modulo CSP: The *clingcon* system*

Max Ostrowski and Torsten Schaub

Institut für Informatik, Universität Potsdam

Abstract. We present the hybrid ASP solver *clingcon*, combining the simple modeling language and the high performance Boolean solving capacities of Answer Set Programming (ASP) with techniques for using non-Boolean constraints from the area of Constraint Programming (CP). The new *clingcon* system features an extended syntax supporting global constraints and optimize statements for constraint variables. The major technical innovation improves the interaction between ASP and CP solver through elaborated learning techniques based on so-called *irreducible inconsistent sets*. An empirical evaluation on a broad class of benchmarks shows that these techniques yield a performance improvement of an order of magnitude.

1 Introduction

clingcon is a hybrid solver for Answer Set Programming (ASP; [1]), 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*[1], *clingcon* 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.

The underlying formal framework, defining syntax and semantics of constraint logic programs, the principal algorithms, and first experimental analysis were presented in [3]. Apart from major re-factoring, *clingcon* now features an extended syntax supporting global constraints and optimize statements for constraint variables. Also, it allows for more fine-grained configurations of constraint-based lookahead, optimization, and propagation delays. However, the major technical innovation improves the interaction between ASP and CP solver through elaborated learning techniques. We introduce filtering methods for conflicts and reasons based on so-called *irreducible inconsistent sets*. An empirical evaluation on a broad class of benchmarks shows that these techniques yield a performance improvement of an order of magnitude.

* An extended version of this paper was submitted to ICLP'12.

[1] `http://www.gecode.org`

2 The *clingcon* approach

We will briefly explain ASP, a declarative problem solving approach, combining a rich yet simple modeling language with high-performance solving capacities. A (*normal*) *logic program* over an alphabet $\mathcal{A}$ is a finite set of *rules* of the form $a_0 \leftarrow a_1, \ldots, a_m, not\ a_{m+1}, \ldots, not\ a_n$, where $a_i \in \mathcal{A}$ is an *atom* for $0 \leq i \leq n$.[2] A *literal* is an atom a or its (default) negation $not\ a$. For a rule r, 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 $B^- = \{a \in \mathcal{A} \mid not\ a \in B\}$. 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.

The input language of *clingcon* extends the one of *gringo* (cf. [4]) by CP-specific operators marked with a preceding `$` symbol, as detailed in Section 4. 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), where V is a set of variables with common domain D, and C is a set of constraints. This CSP is to be addressed by the corresponding CP solver, in our case *gecode*. As detailed in [3], 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 $\mathcal{A} \cup \mathcal{C}$ (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 wrt the assignment of V to D. In the second step, the common Gelfond-Lifschitz reduct [5] 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 $\overline{\ell}$. Sometimes we restrict such an assignment A to its constraint atoms by writing $A|_{\mathcal{C}}$. For instance, given the regular atom '`person(adam)`' and the constraint atom '`work(adam) $> 4`', we may form the Boolean assignment $\{\mathbf{T}$`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 $\overline{\text{`}=\text{'}} = \text{`}\neq\text{'}$ or $\overline{\text{`}<\text{'}} = \text{`}\geq\text{'}$, we can extend γ to signed constraint atoms over $\mathcal{C}$:

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

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 $\{work(adam) \mapsto 3\}$.

[2] 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 $a' \leftarrow not\ a$.

Following [6], we represent Boolean constraints issuing from a logic program under ASP semantics in terms of *nogoods* [7]. 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. 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 [6] for details on how logic programs are translated into nogoods within ASP.

3 The *clingcon* Architecture

Although *clingcon*'s solving components follow the approach of modern SMT solvers when combining the ASP solver *clasp* with the CP solver *gecode*, *clingcon* furthermore adheres to the tradition of ASP in supporting a corresponding modeling language by appeal to the ASP grounder *gringo*. The resulting tripartite architecture is depicted in Figure 1. Although in the current implementation the theory solver is instantiated with

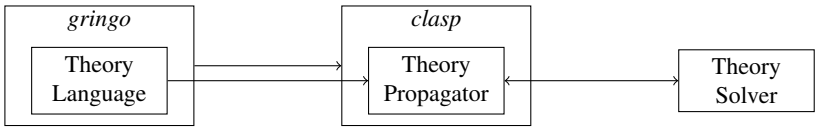


Fig. 1: Architecture of *clingcon*

the CP solver *gecode*, the principal design of *clingcon* along with its interfaces are conceived in a generic way, aiming at arbitrary theory solvers.

Following the workflow in Figure 1, 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. The language extensions allow for expressing arithmetic constraints over integer variables. as well as global constraints and optimization statements. These constraints are treated as atoms and passed to *clasp* via the standard *gringo-clasp* interface, also used in *clingo*, the monolithic combination of *gringo* and *clasp*. Information about these constraints is furthermore directly shared with the theory propagator. In the new version of *clingcon*, the theory propagator is implemented as a post propagator, as furnished by *clasp*. Theory propagation is done by appeal to 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 *clasp* using a corresponding nogood. 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

Listing 1.1: Example of a constraint logic program

```
1  $domain(0..10).
2  person(adam;smith;lea;john).
3  1{team(A,B) : person(B) : B != A}1 :- person(A), A == adam.
4  {friday}.

6  work(A) $+ work(B) $> 6 :- team(A,B).
7  work(B) $- work(adam) $== 1 :- friday, team(adam,B).
8  :- team(adam,lea), not work(lea) $== work(adam).
9  work(B) $== 0 :- person(B), not team(adam,B), B != adam.

11 $count[work(A) $== 8 : person(A)] $== fulltime.

13 $maximize{work(A) : person(A)}.
```

determining a conflict nogood (which is usually not provided by theory solvers, as in the case of *gecode*). This is elaborated upon in Section 5. Similarly, the theory propagator is in charge of enumerating constraint variable assignments, whenever needed.

4 The *clingcon* Language

We explain the syntax of our constraint logic programs via the example program in Listing 1.1. Suppose Adam wants to do a house renovation with the help of three of his friends. We encode the problem as follows. In Line 1 we restrict all our constraint variables to the domain $[0, 10]$ as nobody wants to work more than 10 hours a day. In Line 3 we choose teams. They agreed that each team has to work more than six hours a day (Line 6). Within this line we show the syntax of linear constraints. They can be used in the head or body of a rule[3]. We use the `$` sign in front of every relation and function symbol referring to the underlying CSP. In this new *clingcon* version this also applies to arithmetic operators to better separate them from *gringo* operators. Many arithmetical operators are supported, like plus($+$), times($*$) and absolute(abs). We use the grounding capabilities of *gringo* to create the constraint variables. Grounding Line 6 yields:

```
work(adam) $+ work(smith) $> 6 :- team(adam,smith).
work(adam) $+ work(lea)   $> 6 :- team(adam,lea).
work(adam) $+ work(john)  $> 6 :- team(adam,john).
```

We created three ground rules containing three different constraints, using four different constraint variables. Note that the constraint variables have not been defined beforehand. All variables occurring in a constraint are automatically constraint variables. On Fridays, Adam has to pick up his daughter from sports and therefore works one hour less than his partner (Line 7). Furthermore Lea and Adam are a couple and decided to have an equal work load if they are in the same team (Line 8). Finally, to prevent persons from working if they are not in a team we use Line 9.

With this little constraint logic program, we want to show how for example quantities can be easily expressed. Constraints and constraint variables fit naturally into the logic program. Not representing quantities explicitly with propositional variables eases the modelling of problems and also decreases the size of the ground logic program.

[3] Constraint atoms in the head are shifted to the negative body.

Global Constraints are a new feature of *clingcon*. They capture relations between a non-fixed number of variables. As with aggregates in *gringo*, we can use conditional literals [8] to represent these sets of variables. In the example we can see a *count* constraint in Line 11. After grounding this yields:

```
$count[work(adam) $== 8, work(smith) $== 8,
       work(lea)  $== 8, work(john)  $== 8] $== fulltime.
```

which constrains the number of variables in {work(adam), work(smith), work(lea), work(john)} that are equal to 8, to be equal to fulltime. Constraint variable fulltime counts how many persons are working full time. Global constraints do have a similar syntax to propositional aggregates. Also their semantics is similar to *count* aggregates in ASP (cf. [9]). But global constraints only constrain the values of constraint variables, not propositional ones.

Clingcon supports the global constraint *distinct*, `$distinct{work(A) : person(A)}` means that all persons should have a different workload. That is, all values assigned to constraint variables in {work(adam), work(smith), work(lea), work(john)} have to be distinct from each other. This constraint could also be expressed using a quadratic number of inequalities. Using a single dedicated constraint is usually much more efficient in terms of memory consumption and runtime.

As global constraint are usually not supported in a negated form in a CP solver, we have the syntactic restriction that all global constraints must become facts during grounding and therefore may only occur in the head of rules. Further global constraints can easily be integrated into this generic framework.

A valid solution to our constraint logic program in Listing 1.1 contains the regular literal **T**team(adam,smith), but also constraint literals like **T**work(lea)$==0, **T**work(john)$==0 and **T**work(adam)$+work(smith)$>6. The solution also contains the assignment of the constraint variables, like work(adam)↦8, work(smith)↦3, work(lea)↦0, work(john)↦0 and fulltime↦1.

Optimization is also a new feature of *clingcon*. In Line 13 we give a maximize statement over constraint variables. We maximize the sum over a set of variables, in this case work(adam) $+ work(smith) $+ work(lea) $+ work(john). For optimization statements over constraint variables, we also rely on the syntax of *gringo's* propositional optimization statements. We support minimization/maximization and multi-level optimization. To distinguish propositional and constraint statements, we precede the latter with a $ sign. One optimal solution to the problem contains the propositional literal **T**team(adam,john), and the constraint literals **T**work(lea)$==0, **T**work(smith)$==0, and **T**work(adam)$+work(john)$>6. To maximize work load, the constraint variables are assigned work(adam)↦10, work(smith)↦0, work(lea)↦0, work(john)↦10 and fulltime↦0.

To find a constraint optimal solution, we have to combine the enumeration techniques of *clasp* with the ones from the CP solver. Therefore, when we first encounter a full propositional assignment, we search for an optimal(wrt to the optimize statement) assignment of the constraint variables using the search engine of the CP solver. We will explain this given the following constraint logic program.

```
$domain(1..100).
a :- x $* x $< 25.
$minimize{x}.
```

Assume *clasp* has computed the full assignment $\{\mathbf{F}\texttt{x \$* x \$< 5}, \mathbf{F}a\}$. Afterwards, we search for the constraint optimal solution to the constraint variable x which is $\{\texttt{x} \mapsto 5\}$. Given this optimal assignment, a constraint can be added to the CP solver that all further solutions shall be below/above this optimum (x$<5). This constraint now will restrict all further solutions to be "better". We enumerate further solutions, using the enumeration techniques of *clasp*. So the next assignment is $\{\mathbf{T}\texttt{x \$* x \$< 5}, \mathbf{T}a\}$ and the CP solver finds the optimal constraint variable assignment $\{\texttt{x} \mapsto 1\}$. Each new solution restricts the set of further solutions, so our constraint is changed to (x$<1) which will then allow no further solutions to be found.

5 Conflict Filtering in *clingcon*

The development of Conflict Driven Clause Learning (CDCL) algorithms was a major breakthrough in the area of SAT. Also, CDCL is crucial in SMT solving (cf. [10]). A prerequisite to combine a CDCL-based SAT solver with a theory solver is the possibility to generate good conflicts and reasons originating in the underlying theory. Therefore, modern SMT solvers use their own specialized theory propagators that can produce such witnesses. *Clingcon* instead uses a black-box approach as regards theory solving. In fact, off-the-shelf CP solvers, like *gecode*, do usually not provide any reason for their underlying inference. As a consequence, conflict and reason information was so far only crudely approximated in *clingcon*. We address this shortcoming by developing mechanisms for extracting minimal reasons and conflicts from any CP solver using monotone propagators. We assume that the reader has basic knowledge on CDCL-based ASP solving, and direct the interested reader to [6].

Whenever the CP solver finds out that the set of constraints is inconsistent under the current assignment A, a conflicting nogood N must be generated, which can then be used by the ASP solver in its conflict analysis. 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 the cause for $N = \{\ell \mid \ell \in A|_{\mathcal{C}}\}$. In this case, the corresponding list of inconsistent constraints is

$$I = [\ \gamma(\ell) \mid \ell \in A|_{\mathcal{C}}\ ]. \tag{1}$$

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 [11] 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, it is actually possible to drastically reduce such exhaustive sets of inconsistent constraints as in (1) and to create a much smaller conflict nogood. *Clingcon* now features several alternatives to reduce such conflicts. To this end, we build upon the approach of [12] who propose different algorithms for computing IISs, among them the so-called *Deletion Filtering* algorithm. In what follows, we first present the original idea of *Deletion Filtering* and afterwards propose several refinements that can then be used to reduce inconsistent lists of constraints in the context of ASP modulo CSP.

Algorithm 1: DELETION_FILTERING

Input : An inconsistent list of constraints $I = [c_1, \ldots, c_n]$.
Output : An irreducible inconsistent list of constraints.

```
1 i ← 1
2 while i ≤ |I| do
3     if I \ c_i is inconsistent then
4         I ← I \ c_i
5     else
6         i ← i + 1
7 return I
```

Deletion Filtering reduces an inconsistent list of constraints $I = [c_1, \ldots, c_n]$ as in (1) to an irreducible list. In Algorithm 1 we test for each c_i whether $I \setminus c_i$ is inconsistent or not. If it is inconsistent we restart the algorithm with the list $I \setminus c_i$.

The result of this simple approach is a minimal inconsistent list. Suppose we branch on $\mathbf{T}$`team(adam,lea)`. Unit propagation implies the literals $\mathbf{T}$`work(lea)$==work(adam)`, $\mathbf{T}$`work(john)$==0`, $\mathbf{T}$`work(smith)$==0`, and $\mathbf{T}$`work(adam)$+work(lea)$>6`. At this point we cannot do any constraint propagation[4] and make another choice, $\mathbf{T}$`friday`, and some unit propagation, resulting in $\mathbf{T}$`work(lea)$-work(adam)$==1`. As unit propagation is at fixpoint, the CP solver checks the constraints in the partial assignment $A|_{\mathcal{C}}$ for consistency. As it is inconsistent, a *simple* conflicting nogood would be $N = \{\ell \mid \ell \in A|_{\mathcal{C}}\}$. To minimize this nogood, we now apply *Deletion Filtering* to I as defined in (1):

$$\begin{aligned} I =& [\gamma(\ell) \mid \ell \in A|_{\mathcal{C}}] \\ =& [work(lea) = work(adam), work(john) = 0, work(smith) = 0] \\ & \circ [work(adam) + work(lea) > 6, work(lea) - work(adam) = 1] \end{aligned}$$

For $i = 1$, we test $[work(john) = 0, work(smith) = 0, work(adam) + work(lea) > 6, work(lea) - work(adam) = 1]$, but it does not lead to inconsistency (Line 3). Next list to test is $[work(lea) = work(adam), work(smith) = 0, work(adam) + work(lea) > 6, work(lea) - work(adam) = 1]$ which restricts the domains of `work(adam)` and `work(lea)` to $\emptyset$. As this is inconsistent, we remove $work(john) = 0$ from I and go on, also removing $work(smith) = 0$ and $work(adam) + work(lea) > 6$. We finally end up with the irreducible list $I = [work(lea) = work(adam), work(lea) - work(adam) = 1]$, and can now build a much smaller conflicting nogood $N = \{\gamma^{-1}(c) \mid c \in I\} = \{\mathbf{T}$`work(lea)$==work(adam)`$, \mathbf{T}$`work(lea)$-work(adam)$==1`$\}$ as this really describes the cause of the inconsistency.

The new feature of *clingcon* is the possibility to apply an IIS algorithm to every conflicting set of constraints in order to provide *clasp* with smaller nogoods. This en-

[4] w.l.o.g. we assume arc consistency [13]

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'
```

hances the information content of the learnt nogood and hopefully speeds up the search process by better pruning the search space.

In most CP solvers, propagation is done in a constraint space. This space contains the constraints and the variables of the problem. After doing propagation, the domains of the variables are restricted. As long as we add further constraints to the constraint space this effect cannot be undone, as another constraint restricts the domain of the variables even more. If we want to remove a constraint from a constraint space we have to create a new space containing only the constraints we want and redo all the propagation. This is why we identified Line 4 in Algorithm 1 as the efficiency bottleneck. To address this problem, we propose some derivatives of the algorithm.

Forward Filtering is shown in Algorithm 2; 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 our 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 them. 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. So we add $work(lea) = work(adam)$ to I', resulting in $I' = [work(lea) - work(adam) = 1, work(lea) = work(adam)]$. This is then our reduced list of constraints and the same IIS as we got with the *Deletion Filtering* method (as it is the only IIS of the example). But this time we only needed one reset of the constraint space (Line 3) instead of five.

Backward Filtering is similiar to Algorithm 2. But this time, we reverse the order of the inconsistent constraint list. Therefore, we first test the last assigned constraint and iterate to the first. In this way we want to accommodate the fact, that one of the literals that was decided on the current decision level has to be included in the conflicting nogood. Otherwise we would have recognized the conflict before.

Range Filtering does not aim at computing an irreducible list of constraints, but tries to approximate a smaller one to find a nice tradeoff between reduction of size and runtime of the algorithm. Therefore, we move through the reversed list of constraints I and add constraints to the result until it becomes inconsistent. In our example we cannot reduce the inconsistent list anymore, as the first and the last constraint is needed in the IIS.

Connected Components Filtering tries to make use of the structure of the constraints. Therefore, it does not go forward or backward through the list of constraints but follows their used constraint variables. Given that ω is the set of constraint variables from the last added constraint, we iterate over our list of constraint and only test a constraint if it contains some of the variables. If this is the case we also extend ω with the variables of this constraint. In this way we will first test constraints that are somehow related via their structure. We end up with the same IIS as with *Forward Filtering* without checking $work(john) = 0$ and $work(smith) = 0$, as they do not have common variables with the constraints from the IIS.

Connected Components Range Filtering is a combination of the *Connected Components Filtering* and the *Range Filtering* algorithms. That is why it does not compute an irreducible list of constraints. We move through the list I like in *Connected Components Filtering* and once our test list T becomes inconsistent we simply return it. This shall combine the advantages of using the structure of the constraints in the *Connected Components Filtering* and the simplicity of the *Range Filtering*. We ignore $work(john) = 0$ and $work(smith) = 0$ and end up with $I' = \{work(lea) - work(adam) = 1, work(adam) + work(lea) > 6, work(lea) = work(adam)\}$.

6 Reason Filtering in *clingcon*

Up to now we only considered reducing an inconsistent list of constraints to reduce the size of a conflicting nogood. But we can do even more. If the CP solver propagates the literal l, a *simple* reason nogood is $N = \{\ell \mid \ell \in A|_C\} \cup \{\bar{l}\}$. If we have

for example $A|_{\mathcal{C}} = \{\mathbf{T}\texttt{work(john)\$==0}, \mathbf{T}\texttt{work(lea)-work(adam)\$==1}\}$, the CP solver propagates the literal $\mathbf{F}\texttt{work(lea)\$==work(adam)}$. 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 $\{\mathbf{T}\texttt{work(lea)-work(adam)\$==1}, \mathbf{T}\texttt{work(lea)\$==work(adam)}\}$. Smaller reasons reduce the size of conflicts, as they are constructed using unit resolution.

These two new features of *clingcon* are available via the command line parameters: `--csp-reduce-conflict=X` and `--csp-reduce-reason=X` where $\texttt{X} = \{\texttt{simple}, \texttt{forward}, \texttt{backward}, \texttt{range}, \texttt{cc}, \texttt{ccrange}\}$.

7 The *clingcon* System

The new filtering methods enhance the learning capabilities of *clingcon*. However, the new version also features *Initial Lookahead*, *Optimization* and *Propagation Delay*. We will now present some of them in more detail.

Initial Lookahead on constraints can be very helpful in the context of SMT [14]. It makes implicit knowledge (stored in the propagators of the theory solver) explicitly available to the propositional solver. Our *Initial Lookahead* is a preprocessing step, restricted to constraint literals. All of them are separately set to true and constraint propagation is done. So, binary relations between constraints become explicitly available to the ASP solver. For example, $\mathbf{T}\texttt{work(smith)\$==0}$ implies $\mathbf{F}\texttt{work(smith)-work(adam)\$==1}$ whereas $\mathbf{T}\texttt{work(lea)\$==work(adam)}$ implies $\mathbf{F}\texttt{work(lea)-work(adam)\$==1}$. These are then directly translated into a nogood. Or more formal: all constraints c implied by a constraint literal $\mathbf{T}\ell$ wrt the theory are added to *clasp* as the respective binary nogood $\{\mathbf{T}\ell, \overline{\gamma^{-1}(c)}\}$.

Propagation Delay is a new experimental feature that balances the interplay between the ASP and the CSP part. Constraint propagation can be expensive, especially in combination with the filtering techniques from Section 5. It might be beneficial to give more attention to the ASP solver. This can be done by skipping constraint propagations. Whenever we can propagate a constraint atom or encounter a conflict with the CP solver, filtering methods can be applied. By skipping constraint propagation and doing it only every n'th time, *clasp* has the chance to find more conflicts. If we learn less from the CSP side, we learn more from the ASP side. Whenever we do constraint propagation we have to catch up on the missed propagation.

8 Experiments

We collected various benchmarks from different categories to evaluate the effects of our new features on a broad range of problems. All of them can be expressed using a mixed representation of Boolean and non-Boolean variables. We restrict ourself to

classes where the ASP and the CSP part interact tightly to solve the problem, as we focus on the learning capabilities between the two systems. On examples where we do not have an ASP or a CSP part of the problem, we will not see any effect of our new features. We focus on five different benchmark classes, where most of them are inspired by the 2011 ASP Competition[5]. This includes *Packing*, *Incremental Scheduling*, *Weighted Tree*, *Quasi Group*, and the *Unfounded Set Check (USC)*. All encodings and instances can be found online[6].

Settings We run our benchmarks single-threaded on a cluster with 24×8 cores with 2.27GHz each and restricted us to use 4GB RAM. In all benchmarks we used the standard configuration of *clingcon*. The new version of *clingcon* 2.0.0 is based on *clingo* 3.0.4 and *gecode* 3.7.1. We now evaluate the new features of *clingcon*.

Global Constraints We want to check whether the use of global constraints can speed up the computation. Therefore we have chosen the *Quasi Group* problem, as it can be easily expressed using the global constraint: *distinct*. We compare two different encodings for *Quasi Group*. The first one uses only one *distinct* constraint for every row and every column. The second one uses a cubic number of inequality constraints. We tested 78 randomly generated instances of size 20×20. While the first encoding using the global constraints results in an average runtime of 220 seconds and 18 timeouts over all instances, the decomposed version was much slower. It used 391 seconds on average and had 27 timeouts. We can confirm also as regards *clingcon*, that global constraints are handled more efficiently than their explicit decomposition.

Initial Lookahead In Section 7, we presented *Initial Lookahead* over constraints as a new feature of *clingcon*. We now want to study in which cases this technique can be useful in terms of runtime. We run all our benchmarks once with and without *Initial Lookahead*. In Table 1, the first column shows the problem class and its number of instances. The second and the third column show the average runtime in seconds that is used with and without *Initial Lookahead (I.L.)*. Timeouts are shown in parenthesis. The last two columns show the average runtime of the lookahead algorithm and the number of nogoods that have been produced on average per instance. As we can see for the problems *Packing*, *Quasi Group*, and *Weighted Tree*, direct relations between constraints can be detected and the overall runtime can therefore be reduced. But this technique does not work on all benchmark classes. For *Incremental Scheduling* relations between constraints are found but the additional nogoods seem to deteriorate the performance of the solver. In the case of the *Unfounded Set Check*, nearly no relations have been found, so no difference in runtime can be detected.

Conflict and Reason Filtering We want to analyze how much the different conflict and reason filtering methods presented in Section 5 differ in size of conflicts and average runtime. As conflicts and reasons are strongly interacting in the CDCL framework, we test the combination of all our proposed algorithms. We denote the filtering algorithms

[5] `https://www.mat.unical.it/aspcomp2011/`
[6] `http://www.cs.uni-potsdam.de/clingcon`

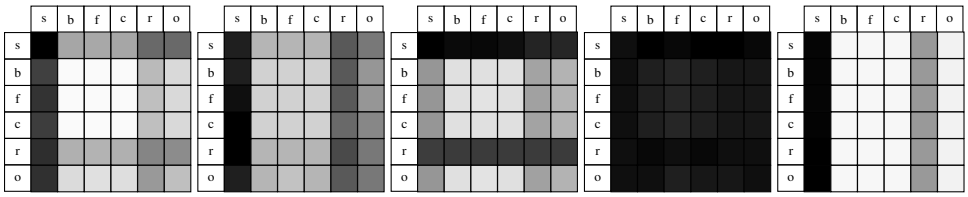
instances (#number)	time	time with *I.L*	time *I.L.*	nogoods from *I.L*
Packing(50)	888(49)	882(49)	5	7970
Inc. Sched.(50)	30(01)	40(02)	0	73
Quasi Group(78)	390(28)	355(24)	9	105367
Weighted Tree(30)	484(07)	312(04)	0	1520
USC(132)	721(104)	719(103)	3	1

Table 1: Initial Lookahead (*I.L.*)

time s/s	time o/b	acs s/s	acs o/b
888(49)	63(0)	293	40
30(01)	3(0)	15	5
390(28)	12(0)	480	56
484(07)	574(18)	31	31
721(104)	92(1)	454	13

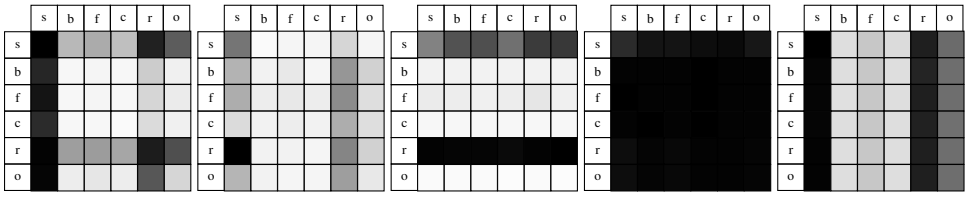
Table 2: Average time in s(timeouts)

with the following shortcuts: s(*Simple*), b(*Backward*), f(*Forward*), c(*Connected Component*), r(*Range*) and o(*Connected Component Range*). We name the filtering algorithm for reasons first, separated by a slash from the algorithm used to filter conflicts. To denote *Range Filtering* for reasons and *Forward Filtering* for conflicts, we simply write r/f. The original configuration, which can be seen as the "old" *clingcon* is therefore denoted s/s. We start by showing the impact on average conflict size of all configurations using a heat map in Figure 2. It shows the reduction of the conflict size in percentage relative to the worst configuration. The rows represent the used algorithms for reason filtering, the columns represent the algorithms for filtering conflicts. So the worst configuration is represented by a totally black square and a configuration that reduces the average conflict size by half is gray. A completely white field would mean that the conflict size has been reduced to zero. As we can see in Figure 2, the average conflict size is reduced by all combinations of filtering algorithms. Furthermore, we see that the first row and column, respectively, is usually darker than the others, which indicates



(a) *Packing* (b) *Inc. Shed.* (c) *Quasi Group* (d) *Weighted Tree* (e) *USC*

Fig. 2: Average conflict size



(a) *Packing* (b) *Inc. Sched.* (c) *Quasi Group* (d) *Weighted Tree* (e) *USC*

Fig. 3: Average time in seconds

that filtering either only conflicts or only reasons is not enough. Also we see that for the *Unfounded Set Check (USC)* the filtering of reasons does not have any effect. This is due the encoding of the problem. As nearly no propagation takes place, no reasons are computed at all. The shades on the *Range Filtering* rows/columns (r) clearly show that it produces larger conflicts. But this is improved by incorporating structure to the filtering algorithm using *Connected Component Range Filtering*. Next, we want to see if the reduction of the average conflict size also pays off in terms of runtime. Therefore Figure 3 shows the heat map for average runtime. A black square denotes the slowest configuration, while a gray one is twice as fast. As we can clearly see, the reduction of runtime coincides with the reduction of conflict size in most cases. Furthermore, we can see a clear speedup for all benchmark classes except *Weighted Tree* using the filtering algorithms. Table 2 compares the *Simple* version s/s, with the configuration o/b (reducing reasons using *Connected Component Range* and conflicts using *Backward Filtering*), as it has the lowest number of timeouts. We can see a speedup of around one order of magnitude on all benchmarks except *Weighted Tree*. The same picture is given for the reduction of average conflict size (acs). So whenever it is possible to reduce the average conflict size, this also pays off in terms of runtime.

Propagation Delay As the filtering of conflicts and reasons takes a lot of time (e.g configuration o/b uses 43% of the overall runtime for filtering on average), we want to reduce the calls to the filtering algorithms. Therefore, with *Propagation Delay*, we can do less propagation with the CP solver and it will produce less conflicts/reasons, hopefully reducing the number of calls to the filtering algorithm. We therefore take the yet best configuration o/b and compare different propagation delays, namely $n = 1, 10$ and 0 (normal, every ten steps, only on model). Table 3 shows the average number of calls

n	1	10	0
Packing	31534	14897	8463
Inc. Sched.	3505	3240	6660
Quasi Group	4245	1535	1726
Weighted Tree	6868k	1168k	1042k
USC	2007	2118	1768

Table 3: Calls to filtering algorithms o/b

n	1	10	0
Packing	63	75	571
Inc. Sched.	3	6	11
Quasi Group	12	9	19
Weighted Tree	574	559	546
USC	92	91	82

Table 4: Times of configuration o/b

to a filtering algorithm for configuration o/b with different delays. We see a reduction of the number of calls on all benchmarks except on *Incremental Scheduling* it doubled. This is clearly due to a loss of information that is necessary for the search. If the CP solver has less influence on the search, the ASP part gets more control. But the missing knowledge from the CSP part has to be compensated by pure search in the ASP part. Therefore, Table 4 shows that some benchmarks like *Weighted Tree* and *Unfounded Set Check* can relinquish some propagation power gaining additional speedup. On others like *Packing*, propagation is needed to drive the search and cannot be compensated. We conclude that this feature has to be investigated further to benefit in practical usage.

9 Related Work

In the quite young field of ASP modulo CSP a lot of research has been done in the last years. The approaches can be separated roughly in two classes: integration and translation. The integrated approaches like *CASP* [15] and AD/ACSolver [16] are similar to the *clingcon* system. However, no learning is used in the approaches, as the constraint solver just checks the assignment of constraints. Later, [17] showed how to use ASP as a specification language, where each answer set represents a CSP. In this approach no coupling between the systems was possible and learning facilities were not used. Afterwards GASP [18] presented a bottom up approach where the logic program was grounded on the fly. With Dingo [19] ASP was translated to difference logic using level mapping for the unfounded set check. An SMT solver is used to solve the translated problem. Nowadays, more and more translational approaches arise in the area of SMT. *Sugar* [20] is a very successful solver which translates the various supported theories to SAT. Also, in [21] it was shown how to translate constraints into ASP during solving. These translational approaches have the strongest coupling and therefore the highest learning capabilities. On the other hand, they do have problems to find a compact representation of the constraints without losing propagation strength. *Clingcon* therefore tries to catch up, improving the learning facilities and still preserving the advantages of integrated approaches like compact representation of constraint propagators. Furthermore, *clingcon* is a ASP modulo Theory solver that aims at taking advantage of arbitrary theories in the long run, eg description logics. Such a variety can only be supported by a black box approach. Similar results regarding the filtering methods have been introduced in [22] but have not been applied to an SMT framework.

10 Discussion

We extended and improve *clingcon* in various ways. At first, the input language was expanded to support *Global Constraints* and *Optimization Statements* over constraint variables. As the input language is a big advantage over pure SMT systems, complex hybrid problems can now easily be expressed as constraint logic programs. Furthermore, we extended the learning capacities of *clingcon*. We have shown that *Initial Lookahead* can give advantages in terms of speedup on some problems. We developed *Filtering* methods for conflicts and reasons that can be applied to any theory solver. This enables the ASP solver to learn about the structure of the theory, even if the theory solver is a black box system and does not give any information about it. We furthermore show that while applying these filtering methods, that knowledge is discovered that is valuable for the overall search process and can therefore speed up the search by orders of magnitude. With *Propagation Delay* we developed a method to control the impact of the interaction among both systems to the search. To balance the *Propagation Delay* dynamically during search will be topic of additional research. In the future we still want to focus on learning facilities and increase the coupling of the two systems even more, such that the CP solver also benefits from the elaborated learning techniques.

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.: Handbook of Satisfiability. IOS Press (2009)
3. Gebser, M., Ostrowski, M., Schaub, T.: Constraint answer set solving. [23] 235–249
4. 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`
5. Gelfond, M., Lifschitz, V.: Classical negation in logic programs and disjunctive databases. New Generation Computing **9** (1991) 365–385
6. Gebser, M., Kaufmann, B., Neumann, A., Schaub, T.: Conflict-driven answer set solving. Proceedings of the Twentieth International Joint Conference on Artificial Intelligence (IJCAI'07), AAAI Press/The MIT Press (2007) 386–392
7. Dechter, R.: Constraint Processing. Morgan Kaufmann Publishers (2003)
8. Syrjänen, T.: Lparse 1.0 user's manual. `http://www.tcs.hut.fi/Software/smodels/lparse.ps.gz`
9. Schulte, C., Tack, G., Lagerkvist, M.: Modeling. Modeling and Programming with Gecode. (2012) Corresponds to Gecode 3.7.2.
10. Nieuwenhuis, R., Oliveras, A., Tinelli, C.: Solving SAT and SAT modulo theories: From an abstract Davis-Putnam-Logemann-Loveland procedure to DPLL(T). Journal of the ACM **53**(6) (2006) 937–977
11. van Loon, J.: Irreducibly inconsistent systems of linear inequalities. European Journal of Operational Research. Volume 3., Elsevier Science (1981) 283–288
12. Chinneck, J., Dravinieks, E.: Locating minimal infeasible constraints sets in linear programs. ORSA Journal On Computing. Volume 3., (1991) 157–168
13. Mohr, R., Henderson, T.: Arc and path consistency revisited. Artificial Intelligence **28**(2) (1986) 225–233
14. Yu, Y., Malik, S.: Lemma learning in SMT on linear constraints. Theory and Applications of Satisfiability Testing Springer (2006) 142–155
15. Baselice, S., Bonatti, P., Gelfond, M.: Towards an integration of answer set and constraint solving. Proceedings of the Twenty-first International Conference on Logic Programming (ICLP'05). Springer (2005) 52–66
16. Mellarkod, V., Gelfond, M., Zhang, Y.: Integrating ASP and constraint logic programming. Annals of Mathematics and Artificial Intelligence **53**(1-4) (2008) 251–287
17. Balduccini, M.: Representing constraint satisfaction problems in answer set programming. Workshop on ASP and Other Computing Paradigms (ASPOCP'09). (2009)
18. Dal Palù, A., Dovier, A., Pontelli, E., Rossi, G.: Answer set programming with constraints using lazy grounding. [23] 115–129
19. Janhunen, T., Liu, G., Niemelä, I.: Tight integration of non-ground answer set programming and satisfiability modulo theories. Proceedings of the First Workshop on Grounding and Transformation for Theories with Variables (GTTV'11). (2011) 1–13
20. Tamura, N., Tanjo, T., Banbara, M.: System description of a SAT-based CSP solver Sugar. Third International CSP Solver Competition. (2008) 71–75
21. Drescher, C., Walsh, T.: A translational approach to constraint answer set solving. Theory and Practice of Logic Programming. 10(4-6)., Cambridge University Press (2010) 465–480
22. Junker, U.: Quickxplain: Conflict detection for arbitrary constraint propagation algorithms. IJCAI'01 Workshop on Modelling and Solving problems with constraints (2001)
23. Hill, P., Warren, D., eds.: Proceedings of the Twenty-fifth International Conference on Logic Programming (ICLP'09). Springer (2009)