# Constraint Answer Set Solving

Martin Gebser, Max Ostrowski, and Torsten Schaub*

Universität Potsdam, Institut für Informatik, August-Bebel-Str. 89, D-14482 Potsdam

**Abstract.** We present a new approach to integrating Constraint Processing (CP) techniques into Answer Set Programming (ASP). Based on an alternative semantic approach, we develop an algorithmic framework for conflict-driven ASP solving that exploits CP solving capacities. A significant technical issue concerns the combination of conflict information from different solver types. We have implemented our approach, combining ASP solver *clingo* with the generic CP solver *gecode*, and we empirically investigate its computational impact.

## 1 Introduction

Answer Set Programming (ASP;[1]) is a declarative problem solving approach, combining a rich yet simple modeling language with high-performance solving capacities. This has already resulted in various applications, among them decision support systems for NASA shuttle controllers [2, 3] and various reasoning tools in systems biology [4–6]. However, certain aspects of such applications are more naturally modeled by additionally using non-Boolean constructs, accounting for resources, fine timings, or functions over finite domains. Moreover, a dedicated treatment of large domains avoids the grounding bottleneck inherent to all propositional solving approaches.

In Satisfiability checking (SAT;[7, 8]), this led to the subarea of Satisfiability Modulo Theories (SMT;[9]), extending SAT solvers by theory-specific solvers. This allows SMT problems to incorporate predicates from specialized theories into propositional formulas. Solving an SMT problem then consists of finding a (hybrid) assignment to all Boolean and theory-specific variables satisfying a given formula along with its theory-specific constituents. Apart from a close solver integration, the key to efficient SMT solving lies in elaborated conflict-driven learning techniques that are capable of combining conflict information from different solver types (cf. [9]).

Groundbreaking work on enhancing ASP with Constraint Processing (CP;[10, 11]) techniques was conducted in [12–14]. Based on firm semantic underpinnings, these approaches provide a family of ASP languages parametrized by different constraint classes. While [12] develops a high-level algorithm viewing both ASP and CP solvers as black boxes, [14] embeds a CP solver into a traditional DPLL-style backtracking algorithm, similar to the one underlying the ASP solver *smodels* [15]. Although [12–14] resulted in two consecutive extensions of *smodels* with CP capacities, they do not match the performance of state-of-the-art SMT solvers, simply because they do not support advanced backjumping and conflict-driven learning techniques.

* Affiliated with Simon Fraser University, Canada, and Griffith University, Australia.

We address this problem and propose an alternative way of combining ASP and CP solving. To begin with, we pursue an alternative semantic approach that is based on a propositional language rather than a multi-sorted, first-order language, as used in [12–14]. Our approach follows the so-called lazy approach of advanced SMT solvers by abstracting from the constraints in a specialized theory [9]. 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. Yet, to the best of our knowledge, this is not supported by off-the-shelf CP solvers.[1] As a consequence, we develop an algorithmic framework for conflict-driven ASP solving that integrates CP solving capacities while overcoming the aforementioned difficulty. We have implemented our approach in the new system *clingcon* [16], combining ASP solver *clingo* [17] with the generic CP solver *gecode* [18], and provide an empirical analysis demonstrating its computational impact.

## 2 Background

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

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

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 $r$ 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. [19]).

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 [10], 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 $A : V \rightarrow \bigcup_{v \in V} dom(v)$ and a constraint $(S, R)$ with $S = (v_1, \ldots, v_k)$, define $A(S) = (A(v_1), \ldots, A(v_k))$, and let $sat_C(A) = \{c \in C \mid A(S(c)) \in R(c)\}$.

[1] Advanced SMT solvers, like [9], address this through handcrafted theory solvers.

[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 $a' \leftarrow not\ a$ and similarly $\leftarrow a$ for $a' \leftarrow a, not\ a'$, for a new atom $a'$.

## 3 Constraint Logic Programs: Syntax and Semantics

For extending logic programs with constraint handling capacities, we consider an extended alphabet distinguishing regular and constraint atoms, denoted by $\mathcal{A}$ and $\mathcal{C}$, respectively. Then, *constraint logic programs* $P$ are defined as regular logic programs over an extended alphabet $\mathcal{A} \cup \mathcal{C}$ such that $head(r) \in \mathcal{A}$ for each $r \in P$.

We identify constraint atoms with constraints via a function $\gamma : \mathcal{C} \to C$; furthermore, $\gamma(Y) = \{\gamma(c) \mid c \in Y\}$ for any $Y \subseteq \mathcal{C}$. The set of constraints comprised in a constraint logic program $P$ is given by $C[P] = \gamma(atom(P) \cap \mathcal{C})$. While the associated variables $V[P]$ are obtained from the respective constraint scopes, we assume a default domain $D[P]$ for each variable (e.g., provided by a declaration within $P$).

For a constraint logic program $P$ over $\mathcal{A} \cup \mathcal{C}$ and an assignment $A : V[P] \to D[P]$, we define the *constraint reduct* as

$$P^A = \{head(r) \leftarrow body(r)|_{\mathcal{A}} \mid r \in P, \\ \gamma(body(r)|_{\mathcal{C}}{}^{+}) \subseteq sat_{C[P]}(A), \gamma(body(r)|_{\mathcal{C}}{}^{-}) \cap sat_{C[P]}(A) = \emptyset\} \ .$$

Then, a set $X \subseteq \mathcal{A}$ is a *constraint answer set* of $P$ wrt $A$, if $X$ is an answer set of $P^A$.

Unlike with (standard) atoms in $\mathcal{A}$, the unique names assumption cannot be applied to constraint atoms in $\mathcal{C}$, intentionally representing relations, in a meaningful way. For instance, the same relation between integer variables $x$ and $y$ is described via syntactically different expressions $x < y$ and $((-y - 1) \leq -(x + 1)) \wedge (x \neq y)$. To reflect this, the definitions of the constraint reduct and constraint answer sets treat constraint literals over $\mathcal{C}$ similar to negative body literals, and truth values are determined outside the actual logic program. Hence, we also do not directly consider constraint atoms as heads but view a rule $r$ with $head(r) \in \mathcal{C}$ as standing for $\leftarrow body(r), not\ head(r)$.

Although our semantics is propositional, the atoms in $\mathcal{A}$ and $\mathcal{C}$ are constructible from a multi-sorted, first-order signature given by:

- a set $\mathcal{P}_{\mathcal{A}} \cup \mathcal{P}_{\mathcal{C}}$ of predicate symbols such that $\mathcal{P}_{\mathcal{A}} \cap \mathcal{P}_{\mathcal{C}} = \emptyset$,
- a set $\mathcal{F}_{\mathcal{A}} \cup \mathcal{F}_{\mathcal{C}}$ of function symbols (including constant symbols),
- a set $\mathcal{V}_{\mathcal{A}}$ of regular variable symbols, and
- a set $\mathcal{V}_{\mathcal{C}} \subseteq \mathcal{T}(\mathcal{F}_{\mathcal{A}})$ of constraint variable symbols, where $\mathcal{T}(\mathcal{F}_{\mathcal{A}})$ denotes the set of all ground terms over $\mathcal{F}_{\mathcal{A}}$.

As common in ASP, the atoms in $\mathcal{A} \cup \mathcal{C}$ are obtained by a grounding process, systematically substituting all occurrences of regular variables in $\mathcal{V}_{\mathcal{A}}$ by (ground) terms from $\mathcal{T}(\mathcal{F}_{\mathcal{A}})$. Atoms in $\mathcal{A}$ are formed from predicate symbols in $\mathcal{P}_{\mathcal{A}}$ and terms in $\mathcal{T}(\mathcal{F}_{\mathcal{A}})$, while the ones in $\mathcal{C}$ are formed from predicate symbols in $\mathcal{P}_{\mathcal{C}}$ and terms over $\mathcal{F}_{\mathcal{C}}$ and $\mathcal{V}_{\mathcal{C}}$. This definition tolerates occurrences of similar ground terms in atoms of both $\mathcal{A}$ and $\mathcal{C}$.

Our approach 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. Whenever a constraint atom $c \in \mathcal{C}$ is assigned to true ($\mathbf{T}$) or false ($\mathbf{F}$) by the ASP solver, the CP solver enforces the satisfaction or violation of the associated constraint $\gamma(c)$.

For illustration, let us consider a constraint logic program consisting of the rules in (2)–(12). This is an authentic program, processable by our solver; its syntax extends the input language of *gringo* [20] and thus allows for using integral ranges, as

in (2), and cardinality rules, as in (4). For simplicity, we omit domain atoms $bucket(B)$, $bucket(C)$, and $time(T)$, respectively, in rules (5)–(10):

$$time(0..t_{max}) \tag{2}$$
$$bucket(a) \quad bucket(b) \tag{3}$$
$$1\ \{pour(B,T) : bucket(B)\}\ 1 \leftarrow time(T), T < t_{max} \tag{4}$$
$$1 \leq^{\$} amt(B,T) \leftarrow pour(B,T), T < t_{max} \tag{5}$$
$$amt(B,T) \leq^{\$} 3 \leftarrow pour(B,T), T < t_{max} \tag{6}$$
$$amt(B,T) =^{\$} 0 \leftarrow not\ pour(B,T), T < t_{max} \tag{7}$$
$$vol(B,T{+}1) =^{\$} vol(B,T) + amt(B,T) \leftarrow T < t_{max} \tag{8}$$
$$down(B,T) \leftarrow vol(C,T) <^{\$} vol(B,T) \tag{9}$$
$$up(B,T) \leftarrow not\ down(B,T) \tag{10}$$
$$vol(a,0) =^{\$} 0 \quad vol(b,0) =^{\$} 1 \tag{11}$$
$$\leftarrow up(a,t_{max})\ . \tag{12}$$

This program describes a balance with two buckets, $a$ and $b$, at each end. According to (4), 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 1 and 3. 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 1 unit. The goal is to find sequences of *pour* actions making the side of bucket $a$ be down after $t_{max}$ time steps (cf. (12)).

The above program has the following signature:

$$\{B, C, T\} \subseteq \mathcal{V}_{\mathcal{A}}$$
$$\{0, \ldots, t_{max}, +, a, b, amt, vol\} \subseteq \mathcal{F}_{\mathcal{A}} \qquad \{0, 1, 3, +\} \subseteq \mathcal{F}_{\mathcal{C}}$$
$$\{<, time, bucket, pour, up, down\} \subseteq \mathcal{P}_{\mathcal{A}} \qquad \{=^{\$}, <^{\$}, \leq^{\$}\} \subseteq \mathcal{P}_{\mathcal{C}}\ .$$

The contents of $\mathcal{V}_{\mathcal{C}}$ as well as of $\mathcal{A}$ and $\mathcal{C}$ becomes clear when looking at the ground program obtained by instantiating all variables in $\mathcal{V}_{\mathcal{A}}$ with terms from $\mathcal{T}(\mathcal{F}_{\mathcal{A}})$. To see this, let us look at the ground instantiation of rule (7) and (8) obtained from substitution $\{B \mapsto b, T \mapsto 1\}$ along with constant mapping $t_{max} \mapsto 2$, and evaluating $1 < 2$ as well as $1{+}1$ (as done by grounders like *gringo*):

$$amt(b,1) =^{\$} 0 \leftarrow not\ pour(b,1)$$
$$vol(b,2) =^{\$} vol(b,1) + amt(b,1) \leftarrow\ .$$

These two ground rules encompass three constraint variables and three atoms:

$$\{amt(b,1), vol(b,1), vol(b,2)\} \subseteq \mathcal{V}_{\mathcal{C}}$$
$$\{pour(b,1)\} \subseteq \mathcal{A} \qquad \{amt(b,1) =^{\$} 0,\ vol(b,2) =^{\$} vol(b,1) + amt(b,1)\} \subseteq \mathcal{C}\ .$$

While the actual ASP solver assigns a Boolean value to the constraint atom $vol(b,2) =^{\$} vol(b,1) + amt(b,1)$, the CP solver deals with the associated constraint $\gamma(vol(b,2) =^{\$} vol(b,1) + amt(b,1))$, eventually assigning (integral) values to constraint variables $vol(b,2)$, $vol(b,1)$, and $amt(b,1)$.

For $t_{max} \mapsto 2$, the above program has eleven constraint answer sets, namely, four different Boolean assignments associated with varying constraint assignments, summarized by the following Boolean and constraint variable assignments:

| $up(a,0)$ | $pour(a,0)$ | $amt(a,0)$ | $up(a,1)$ | $pour(a,1)$ | $amt(a,1)$ | $up(a,2)$ |
|---|---|---|---|---|---|---|
| **T** | **T** | 1 | **T** | **T** | $1,2,3$ | **F** |
| **T** | **T** | $2,3$ | **F** | **T** | $1,2,3$ | **F** |
| **T** | **T** | 3 | **F** | **F** | 0 | **F** |
| **T** | **F** | 0 | **T** | **T** | 3 | **F** |

While the first two groups of answer sets "pour into bucket $a$" twice, the last two also "pour into bucket $b$", namely, one unit at either time point 0 or 1.

As a general remark, note that replacing 3 in rule (6) by a significantly larger number (e.g., 30000) does neither affect the size of the ground program nor the number of different Boolean assignments. In fact, the size of the ground instantiation of a program is completely independent of the domain size of its constraint variables. Given the simplicity of the above example, larger domains do also not deteriorate the runtime of a CP solver like *gecode* too much, while they would drastically increase runtime and space required by ASP grounders and solvers.

## 4 Conflict-Driven Nogood Learning with Constraint Processing

We now develop an algorithm for computing constraint answer sets that extends a previous algorithm to compute standard answer sets [19] 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 [21, 22, 7]. 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. In view of the fact that CP solver *gecode* used in our implementation does not provide any reasons (it only reports whether a conflict has occurred), the extended algorithm works under the assumption that its CP oracle cannot be queried for reasons. Nonetheless, conflict resolution requires some reason when resolving out a literal, and the major difficulty we address is to identify sufficient yet non-trivial reasons outside the CP oracle.

As mentioned before, a standard answer set can be seen as an assignment satisfying certain conditions induced by a program $P$. A (Boolean) *assignment* $\mathbf{A}$ over domain $atom(P) \cup \{body(r) \mid r \in P\}$ is a sequence $(\sigma_1, \ldots, \sigma_m)$ of (*signed*) *literals* $\sigma_i$ of the form $\mathbf{T}v_i$ or $\mathbf{F}v_i$, where $v_i \in atom(P) \cup \{body(r) \mid r \in P\}$ for $1 \leq i \leq m$; $\mathbf{T}v_i$ expresses that $v_i$ is *true* and $\mathbf{F}v_i$ that it is *false*. (We omit the attribute *signed* for literals whenever clear from the context.) The complement of a literal $\sigma$ is denoted by $\overline{\sigma}$, that is, $\overline{\mathbf{T}v} = \mathbf{F}v$ and $\overline{\mathbf{F}v} = \mathbf{T}v$, and we let $var(\sigma) = v$. For $\mathbf{A} = (\sigma_1, \ldots, \sigma_{i-1}, \sigma_i, \ldots)$, $\mathbf{A}[\sigma_i] = (\sigma_1, \ldots, \sigma_{i-1})$ is the prefix of $\mathbf{A}$ up to $\sigma_i$. We sometimes abuse notation and identify an assignment with the set of its contained literals. Given this, we access the true and false variables in $\mathbf{A}$ via $\mathbf{A}^{\mathbf{T}} = \{v \mid \mathbf{T}v \in \mathbf{A}\}$ and $\mathbf{A}^{\mathbf{F}} = \{v \mid \mathbf{F}v \in \mathbf{A}\}$. For a canonical representation of (Boolean) constraints, we make use of nogoods [10, 11].

In our setting, a *nogood* is a finite set $\{\sigma_1, \ldots, \sigma_k\}$ of literals, expressing a constraint violated by any assignment $\mathbf{A}$ containing $\sigma_1, \ldots, \sigma_k$. The nogoods derived from the completion of $P$ are denoted by $\Delta_P$, and $\Lambda_P$ contains the ones that are implicitly given by loop formulas (cf. [19]). An assignment $\mathbf{A}$ is a *solution* for $P$ if $\mathbf{A}^{\mathbf{T}} \cap \mathbf{A}^{\mathbf{F}} = \emptyset$, $\mathbf{A}^{\mathbf{T}} \cup \mathbf{A}^{\mathbf{F}} = atom(P) \cup \{body(r) \mid r \in P\}$, and $\delta \not\subseteq \mathbf{A}$ for all $\delta \in \Delta_P \cup \Lambda_P$. As shown in [19], $\mathbf{A}^{\mathbf{T}} \cap atom(P)$ is an answer set of $P$ iff $\mathbf{A}$ is a solution for $\Delta_P \cup \Lambda_P$. We skip further details on $\Delta_P$ and $\Lambda_P$, as they are not affected by adding a CP oracle.[3]

Switching back to constraint logic programs $P$ over $\mathcal{A} \cup \mathcal{C}$, by $\mathbf{A}|_{\mathcal{C}} = \{\mathbf{T}c \in \mathbf{A} \mid c \in \mathcal{C}\} \cup \{\mathbf{F}c \in \mathbf{A} \mid c \in \mathcal{C}\}$, we denote the projection of a Boolean assignment $\mathbf{A}$ to literals over constraint atoms. Furthermore, we associate $P$ with the CSP

$$CSP[P] = \big(V[P] \cup atom(P)|_{\mathcal{C}}, D, \big\{\big((S(\gamma(c)), c), c \equiv R(\gamma(c))\big) \mid c \in atom(P)|_{\mathcal{C}}\big\}\big)$$

where $D$ contains $dom(v) = D[P]$ for every $v \in V[P]$ and $dom(c) = \{\mathbf{T}, \mathbf{F}\}$ for every $c \in atom(P)|_{\mathcal{C}}$.[4] A constraint relation of the form $c \equiv R(\gamma(c))$ is called *reified*: it associates the truth value of $c \in atom(P)|_{\mathcal{C}}$ with the valuation of the corresponding constraint $\gamma(c)$. We below slightly abuse notation by identifying the scope $S(\gamma(c)) = (v_1, \ldots, v_k)$ of $\gamma(c)$ with the corresponding set $\{v_1, \ldots, v_k\}$.

### 4.1 Main Algorithm

Our main algorithm for computing a constraint answer set of $P$ is shown in Algorithm 1. It shares with the one in [19] the assignment $\mathbf{A}$, recorded nogoods $\nabla$, and decision level $dl$ but adds a flag $event$ (cf. Line 4 in Algorithm 1), whose admissible values are $assertion$ and $decision$. The purpose of this flag is to enable propagation, invoked in Line 6, to mark derived literals such that blocks can be distinguished: all literals in the same block are derived either by unit propagation on $\Delta_P \cup \Lambda_P \cup \nabla$ or by constraint propagation on $CSP[P]$. In order to retrieve such blocks in conflict analysis, invoked in Line 9 and 23, each literal $\sigma \in \mathbf{A}$ is associated with a reason flag $res(\sigma)$. A block of literals derived by unit propagation starts with a literal $\sigma_{dc}$ where $res(\sigma_{dc}) = dc$, followed by arbitrarily many literals $\sigma_{up}$ for which $res(\sigma_{up}) = up$. In turn, a block of literals derived by constraint propagation is given by consecutive literals $\sigma_{cp}$ such that $res(\sigma_{cp}) = cp$.

After propagation in Line 6, we distinguish the cases of a *conflict* (Line 7–12), a *total assignment* (Line 13–27), or a *partial assignment* (Line 29–33). In the latter case, a heuristic decision needs to be made, and an undecided literal $\sigma_d$, whose reason is by decision, is selected (Line 29–30). Furthermore, the decision level is incremented, and $\sigma_d$ is appended to $\mathbf{A}$ (Line 31–32). Finally, setting $event$ to $decision$ in Line 33 signals to the following propagation step that the last literal in $\mathbf{A}$ is a decision literal. The case of a *conflict* is signaled via a $status$ flag returned by propagation, if its value is either $cUP$ (conflict in unit propagation) or $cCP$ (conflict in constraint propagation). A conflict above decision level 0, i.e., at least one decision literal is involved in the conflict,

[3] The only difference is that atoms of $\mathcal{C}$ are not subject to completion in $\Delta_P$ and loop nogoods in $\Lambda_P$. That is, they can be assigned to $\mathbf{T}$ without requiring any justification from $P$.

[4] We assume that $\{\mathbf{T}, \mathbf{F}\} \cap D[P] = \emptyset$. Moreover, we write literal $\mathbf{T}c$ or $\mathbf{F}c$ for $c \in atom(P)|_{\mathcal{C}}$ assigned to either $\mathbf{T}$ or $\mathbf{F}$, respectively.

**Algorithm 1**: CDNL-ASPmCSP

**Input** : A constraint logic program $P$.
**Output** : A constraint answer set of $P$.

```
1  A ← ∅                                                    // (Boolean) assignment
2  ∇ ← ∅                                                    // set of (dynamic) nogoods
3  dl ← 0                                                   // decision level
4  event ← assertion                                        // propagation mode
5  loop
6      (A, ∇, status) ← PROPAGATION(P, ∇, A, event)
7      if status ∈ {cUP, cCP} then
8          if dl = 0 then exit
9          (δ, dl) ← CONFLICTANALYSIS(P, ∇, A, status)
10         ∇ ← ∇ ∪ {δ}
11         A ← A \ {σ ∈ A | dl(σ) > dl}
12         event ← assertion
13     else if A^T ∪ A^F = atom(P) ∪ {body(r) | r ∈ P} then
14         (A, status) ← LABELING(CSP[P], A|_C)
15         if status = conflict then
16             dl ← dl + 1
17             repeat
18                 dl ← max{dl(σ) | σ ∈ A|_C, dl(σ) < dl}
19                 if dl = 0 then exit
20                 (A, status) ← LABELING(CSP[P], {σ ∈ A|_C | dl(σ) < dl})
21             until status ≠ conflict
22             A ← A \ {σ ∈ A | dl(σ) > dl}
23             (δ, dl) ← CONFLICTANALYSIS(P, ∇, A, cAS)
24             ∇ ← ∇ ∪ {δ}
25             A ← A \ {σ ∈ A | dl(σ) > dl}
26             event ← assertion
27         else return (A^T ∩ A, {v ↦ A(v) | v ∈ V[P]})
28     else
29         σ_d ← SELECT(P, ∇, A)
30         res(σ_d) ← dc
31         dl ← dl + 1
32         A ← A ∘ σ_d
33         event ← decision
```

is analyzed in Line 9. It results in a new nogood $\delta$, recorded in Line 10, that implies the complement of a UIP by unit propagation at a decision level to which backjumping returns to in Line 11. (Note that $dl(\sigma)$ provides the decision level at which $\sigma$ has been assigned.) Finally, by setting $event$ to $assertion$ in Line 12, we signal the following propagation step that $\delta$ will be asserting.

The treatment of a *total assignment* is the main difference to the algorithm in [19]. Before also solving $CSP[P]$, we cannot be sure whether Boolean assignment $\mathbf{A}$ belongs to a constraint answer set of $P$. Thus, the CP oracle is queried whether there is a solution $A$ for $CSP[P]$ (Line 14), given the truth values assigned to atoms of $\mathcal{C}$ in $\mathbf{A}$. If such a solution $A$ exists, we have found a constraint answer set that is returned in

Line 27. Note that the additional check is necessary because the CP oracle is not permitted to make choices during constraint propagation, which in general will not be able to assign all variables in $V[P]$ or to detect unsatisfiability, given only the truth values of Boolean variables shared with $atom(P)$. This is also the reason why, in case of unsatisfiability detected now, we do not know from which decision level on $CSP[P]$ had actually been unsatisfiable wrt the literals over $\mathcal{C}$ in $\mathbf{A}$. Hence, the loop in Line 17–21 successively backtracks through $\mathbf{A}$ until hitting a decision level $dl$ such that $CSP[P]$ can be satisfied, given only the literals over $\mathcal{C}$ in $\mathbf{A}$ from decision levels smaller than $dl$. Then, conflict analysis is invoked in Line 23 with $cAS$ signaling a conflict on a putative constraint answer set. Conflict-driven learning, backjumping, and the following assertion (Line 24–26) are similar to a conflict encountered by propagation.

### 4.2 Propagation Algorithm

The main idea of our propagation procedure, shown in Algorithm 2, is to iterate unit propagation on $\Delta_P \cup \nabla$, unfounded set detection accompanied by selective recording of nogoods from $\Lambda_P$, and finally constraint propagation on $CSP[P]$. Before this process starts, we set a flag $cp$ to $true$ if constraint propagation should be performed initially (Line 1) or in reaction to a decision literal over $\mathcal{C}$ (Line 2). Otherwise, $cp$ is made $false$ (Line 3), given that $\mathbf{A}$ has not been extended by literals over constraint atoms since the last constraint propagation step.

Conflicts during unit propagation are in Line 6 detected via some nogood from $\Delta_P \cup \nabla$ violated by $\mathbf{A}$, and they are signaled via return value $cUP$. If there is no conflict, we in Line 7 check whether there are nogoods $\delta$ that contain a single unassigned literal, while all other literals belong to $\mathbf{A}$. Then, unit propagation infers the complement $\overline{\sigma}$ of such a last unassigned literal $\sigma$ in order to avoid the inclusion of $\delta$ in $\mathbf{A}$. As mentioned above, we use a flag $res(\overline{\sigma})$ to later on identify a block of literals derived by unit propagation. The value of $event$ now determines whether a new block begins (Line 10–11), which is marked by setting $res(\overline{\sigma})$ to $dc$, or an existing one is extended (Line 12). Finally, flag $cp$ is set in Line 13 if $\overline{\sigma}$ is over an atom of $\mathcal{C}$. After reaching a fixpoint of unit propagation without any conflict, unfounded set handling (cf. [19]) is performed for non-tight [23] programs in Line 17–19. Note that an already identified nonempty unfounded set needs first to be falsified completely before a new (nonempty) unfounded set $U \subseteq atom(P)|_{\mathcal{A}} \setminus \mathbf{A^F}$ is determined in Line 18 (if no such $U$ exists, UNFOUNDEDSET returns $\emptyset$). Finally, atoms in a nonempty unfounded set $U$ will be falsified by unit propagation after adding a loop nogood from $\Lambda_P$ to $\nabla$ in Line 19.

Finally, constraint propagation (Line 21–32) takes place only if unit propagation cannot infer any further literal, checked via $U = \emptyset$ in Line 20. Furthermore, we are sure that no new literals over $atom(P)|_{\mathcal{C}}$ will be derived if none was recently added to $\mathbf{A}$ (if $cp = false$), in which case the whole propagation terminates in Line 21. Otherwise, constraint propagation in Line 22 may result either in a conflict (Line 23), signaled via return value $cCP$, or in an assignment $A$ over $V[P] \cup atom(P)|_{\mathcal{C}}$, whose possible additions to $\mathbf{A}$ on the common constraint atoms are provided by $B$ (Line 24). If additions to $\mathbf{A}$ are possible ($B = \emptyset$ does not hold in Line 25), we do them in Line 26–30, and the reason flags of the derived literals are set to $cp$ (Line 28). Afterwards, flag $cp$ is reset to $false$ in Line 31, so that another constraint propagation step will be performed only

**Algorithm 2**: PROPAGATION

**Input** : A constraint logic program $P$, a set $\nabla$ of nogoods, a (Boolean) assignment $\mathbf{A}$, and an $event \in \{decision, assertion\}$.

**Output** : A (Boolean) assignment, set of nogoods, and a $status \in \{cUP, cCP, fix\}$.

```
1  if A = ∅ then cp ← true                                   // do initial constraint propagation
2  else if event = decision and var(σ) ∈ C where A = A' ∘ σ then cp ← true
3  else cp ← false
4  U ← ∅                                                     // unfounded set
5  loop
6      if δ ⊆ A for some δ ∈ Δ_P ∪ ∇ then return (A, ∇, cUP)
7      Σ ← {δ ∈ Δ_P ∪ ∇ | δ \ A = {σ}, σ̄ ∉ A}
8      if Σ ≠ ∅ then let δ \ A = {σ} for some δ ∈ Σ in
9          if event = assertion then
10             res(σ̄) ← dc
11             event ← decision
12         else res(σ̄) ← up
13         if var(σ̄) ∈ C then cp ← true                     // redo constraint propagation
14         A ← A ∘ σ̄
15     else
16         if P is non-tight then
17             U ← U \ A^F
18             if U = ∅ then U ← UNFOUNDEDSET(P, A)
19             if U ≠ ∅ then let a ∈ U in ∇ ← ∇ ∪ {λ(a, U)}
20         if U = ∅ then
21             if cp = false then return (A, ∇, fix)
22             (A, status) ← CONSTRAINTPROPAGATION(CSP[P], A|_C)
23             if status = conflict then return (A, ∇, cCP)
24             B ← {Tc ∈ A | Tc ∉ A} ∪ {Fc ∈ A | Fc ∉ A}
25             if B = ∅ then return (A, ∇, fix)
26             repeat
27                 B ← B \ {σ} for some σ ∈ B
28                 res(σ) ← cp
29                 A ← A ∘ σ
30             until B = ∅
31             cp ← false
32             event ← assertion
```

after inferring further literals over $atom(P)|_{\mathcal{C}}$ by unit propagation. Finally, flag $event$ is set to $assertion$, which has the effect that the next literal $\overline{\sigma}$ inferred by unit propagation (if any is inferred) will be marked as the first one of a new block via $dc$ for $res(\overline{\sigma})$. In view of the fact that constraint propagation may extend $\mathbf{A}$ with further literals, we note that our propagation technique matches "theory propagation" [9] in SMT solvers.

**Algorithm 3**: CONFLICTANALYSIS

**Input** : A constraint logic program $P$, a set $\nabla$ of nogoods, a (Boolean) assignment $\mathbf{A}$, and a $status \in \{cUP, cCP, cAS\}$.
**Output** : A derived nogood and a decision level.

**if** $status = cAS$ **then**
  $\delta \leftarrow \{\sigma \in \mathbf{A}|_{\mathcal{C}} \mid dl(\sigma) = \max\{dl(\sigma') \mid \sigma' \in \mathbf{A}\}\}$
  **repeat**
    $touched \leftarrow \delta$
    $\delta \leftarrow \{\sigma \in \mathbf{A}|_{\mathcal{C}} \mid S(\gamma(var(\sigma))) \cap \bigcup_{\sigma' \in \delta} S(\gamma(var(\sigma'))) \neq \emptyset\}$
  **until** $\delta = touched$
**else if** $status = cCP$ **then**
  **let** $\sigma \in \mathbf{A}$ such that $res(\sigma) = dc$ and $\forall \sigma' \in \mathbf{A} \setminus (\mathbf{A}[\sigma] \cup \{\sigma\}) : res(\sigma') = up$ **in**
    $\delta \leftarrow \mathbf{A}|_{\mathcal{C}} \setminus \mathbf{A}[\sigma]$
  **repeat**
    $touched \leftarrow \delta$
    $\delta \leftarrow \{\sigma \in \mathbf{A}|_{\mathcal{C}} \mid S(\gamma(var(\sigma))) \cap \bigcup_{\sigma' \in \delta} S(\gamma(var(\sigma'))) \neq \emptyset\}$
  **until** $\delta = touched$
**else**
  $\delta \leftarrow \varepsilon$ for some $\varepsilon \in \Delta_P \cup \nabla$ such that $\varepsilon \subseteq \mathbf{A}$
  $touched \leftarrow \emptyset$
**loop**
  **let** $\sigma \in \delta$ such that $\delta \setminus \mathbf{A}[\sigma] = \{\sigma\}$ **in**
    $k \leftarrow \max\{dl(\sigma') \mid \sigma' \in \delta \setminus \{\sigma\}\}$
    **if** $k = dl(\sigma)$ **then**
      **if** $res(\sigma) = cp$ **then**
        $\mathbf{A} \leftarrow \mathbf{A} \setminus \{\sigma' \in \mathbf{A} \setminus \mathbf{A}[\sigma] \mid \exists \sigma'' \in (\mathbf{A}[\sigma'] \setminus \mathbf{A}[\sigma]) \cup \{\sigma'\} : res(\sigma'') \neq cp\}$
        $\varepsilon \leftarrow \{\sigma' \in \delta \cap \mathbf{A} \mid \forall \sigma'' \in \mathbf{A}[\sigma] \setminus \mathbf{A}[\sigma'] : res(\sigma'') = cp\} \setminus touched$
        **while** $\varepsilon \not\subseteq touched$ **do**
          $touched \leftarrow touched \cup \varepsilon$
          $\varepsilon \leftarrow \{\sigma'' \in \mathbf{A}|_{\mathcal{C}} \mid S(\gamma(var(\sigma''))) \cap \bigcup_{\sigma' \in \varepsilon} S(\gamma(var(\sigma'))) \neq \emptyset\}$
        $\mathbf{A} \leftarrow \mathbf{A} \setminus \{\sigma' \in \mathbf{A} \mid \forall \sigma'' \in \mathbf{A} \setminus \mathbf{A}[\sigma'] : res(\sigma'') = cp\}$
        $\delta \leftarrow (\delta \cup \varepsilon) \cap \mathbf{A}$
      **else let** $\varepsilon \in \Delta_P \cup \nabla$ such that $\varepsilon \setminus \mathbf{A}[\sigma] = \{\overline{\sigma}\}$ **in** $\delta \leftarrow (\delta \setminus \{\sigma\}) \cup (\varepsilon \setminus \{\overline{\sigma}\})$
    **else return** $(\delta, k)$

### 4.3 Conflict Analysis Algorithm

On every conflict beyond decision level 0, our conflict analysis procedure in Algorithm 3 identifies a new nogood according to the First-UIP scheme. While a literal derived by unit propagation has at least one reason in $\Delta_P \cup \nabla$, no such reasons exist for literals derived by constraint propagation. Since we assume that the CP oracle does also not provide us with reasons, we can merely try to reconstruct a non-trivial reason (one that does not include all previously assigned literals over $\mathcal{C}$) from structural properties of $CSP[P]$. Our approach for this is inspired by graph-based backjumping/learning [10] where, for a variable in question, other variables it shares constraints with are con-

sidered as potential reasons. In fact, we identify a sufficient reason by considering all literals over $atom(P)|_{\mathcal{C}}$ assigned prior to a constraint atom $c$ and connected to $c$ via their scopes, starting from literals $\sigma$ with $S(\gamma(var(\sigma))) \cap S(\gamma(c)) \neq \emptyset$.

The sketched strategy is applied when a conflict is due to a putative answer set (Line 1–6), where $CSP[P]$ is unsatisfiable under the current assignment $\mathbf{A}$. Furthermore, since the backtracking scheme in Algorithm 1 guarantees satisfiability when taking only the literals in $\mathbf{A}$ at smaller decision levels than the current one, we also know that literals over $\mathcal{C}$ at the maximum decision level are involved in the conflict. Hence, we take them as initial reason $\delta$ for the conflict (Line 2), and iteratively add all literals in $\mathbf{A}$ over $\mathcal{C}$ connected to some literal in $\delta$ via non-disjoint scopes (Line 3–6). The so obtained $\delta$ provides a sufficient reason for the unsatisfiability of $CSP[P]$; it is processed further using the standard First-UIP scheme (described below). If the conflict at hand has been encountered during constraint propagation (Line 7–13), we know that the literals over $\mathcal{C}$ in the last block derived by unit propagation (determined in Line 8–9) are involved. In Line 10–13, we then use the same technique as above to identify a sufficient reason $\delta$ for the conflict. Finally, if the conflict has been detected during unit propagation (Line 15–16), we can simply determine some violated nogood $\delta$ in $\Delta_P \cup \nabla$.

The loop in Line 17–30 eventually exploits the First-UIP scheme, eliminating literals from $\delta$ until it contains exactly one literal $\sigma$ from the current decision level. If this is not yet the case (tested in Line 20), some $\sigma$ in $\delta$ needs to be replaced with a reason why it was included in $\mathbf{A}$. Here, we distinguish the cases that $\sigma$ has been derived by constraint propagation (Line 21–28) or by unit propagation (Line 29). In the latter case, we can simply resolve $\delta$ against a known nogood $\varepsilon$ in $\Delta_P \cup \nabla$, as done in [19]. Otherwise, determining an appropriate reason is more sophisticated. In fact, in Line 22, we eliminate all successors of $\sigma$ in $\mathbf{A}$ that do not belong to the same block as $\sigma$ of literals derived by constraint propagation. This reflects that the removed literals cannot have contributed to the CP oracle deriving $\sigma$. In Line 23, we then determine in $\varepsilon$ all literals (over $\mathcal{C}$) of $\delta$ that belong to the same block as $\sigma$ in order to explore their constraint interdependencies, where an optimization consists of ignoring literals in $touched$, given that they have been explored already. In Line 24–26, we further extend $\varepsilon$ with connected literals over $\mathcal{C}$, like in Line 3–6 and Line 10–13. Finally, we remove the whole block of $\sigma$ from $\mathbf{A}$ and $\delta$ (Line 27–28), as possible contributions to the conflict have been explored exhaustively, while the remaining literals of $\varepsilon$ are added to $\delta$.

In comparison to [19], it is apparent that the accommodation of a CP oracle makes the required computations more sophisticated, as extra information is needed to distinguish literals derived by constraint propagation from those inferred by unit propagation. The identification of appropriate reasons is a major bottleneck in a conflict-driven learning ASP solver, in particular, if the CP oracle does not support it. In such a case, workarounds are needed to approximate sufficient reasons. Their impact regarding computational cost is empirically investigated in the next section.

## 5 Experiments

We implemented our approach to constraint answer set solving within the new solver *clingcon* (0.1.0;[16]), extending ASP system *clingo* (2.0.2;[17]) with the generic CP

| | *clingo* | *adsolver* | | | | *clingcon* | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| *Benchmark* | 5 | 5 | 7 | 11 | 13 | 5 | 7 | 11 | 13 | 20 |
| 3-0/025 | 162.84 | 14.74 | 51.42 | 460.57 | 365.37 | 1.19 | 1.97 | 4.21 | 5.99 | 17.84 |
| 3-0/050 | 173.28 | 31.39 | 108.21 | 471.41 | — | 1.26 | 2.32 | 6.80 | 11.85 | 27.36 |
| 3-0/100 | 175.94 | 448.90 | 188.33 | — | — | 1.32 | 2.35 | 10.11 | 12.04 | 38.78 |
| 3-0/125 | 165.64 | 19.78 | 60.07 | 224.60 | — | 1.18 | 1.94 | 4.05 | 10.00 | 133.99 |
| 5-0/025 | 174.12 | 28.78 | 107.41 | — | — | 1.28 | 2.90 | 5.87 | 14.27 | 66.55 |
| 5-0/050 | 163.25 | 13.57 | 42.00 | 204.34 | 497.64 | 1.18 | 1.97 | 4.71 | 10.04 | 241.59 |
| 5-0/100 | 168.16 | 21.50 | 66.10 | 282.36 | 514.08 | 1.20 | 1.98 | 4.13 | 6.45 | 25.32 |
| 5-0/125 | 174.38 | 32.02 | 104.32 | 429.72 | — | 1.34 | 2.95 | 6.39 | 9.70 | 81.17 |
| 8-0/025 | 177.82 | 41.57 | 140.93 | — | — | 1.30 | 2.73 | 11.00 | 12.69 | 222.49 |
| 8-0/050 | 167.72 | 18.83 | 54.76 | 215.43 | — | 1.18 | 1.93 | 4.02 | 7.76 | 457.86 |
| 8-0/100 | 165.55 | 13.72 | 41.03 | 208.74 | — | 1.21 | 2.00 | 5.05 | 6.10 | 26.17 |
| 8-0/125 | 162.29 | 16.81 | 53.40 | 246.64 | 519.59 | 1.20 | 1.99 | 4.15 | 6.69 | 17.82 |
| ∅ | 169.25 | 58.47 | 84.83 | 378.65 | 558.06 | 1.24 | 2.25 | 5.87 | 9.47 | 113.08 |

**Table 1.** Comparing *clingo*, *adsolver*, and *clingcon*.

solver *gecode* (2.2.0;[18]). Our experiments consider *clingcon* using four different approaches to incorporate constraint propagation: (a) lazy reason calculation during conflict analysis exploiting constraint interdependencies, as shown in Algorithm 3; (b) immediate reason recording for literals derived by constraint propagation (discussed in the context of SMT in [9, Section 5.1]) exploiting constraint interdependencies; (c) lazy reason calculation during conflict analysis without using constraint interdependencies, rather, taking all assigned literals over $\mathcal{C}$ as trivial reason; (d) immediate reason recording for literals derived by constraint propagation without using constraint interdependencies. We also include *adsolver* (1.55;[14]), combining an (extended) ASP solver with a CP solver for difference constraints. Our experiments consider a benchmark suite stemming from decision support systems for NASA shuttle controllers [2, 3][5], which involve mapping logical time steps on real-time. All experiments were run on a 3.4GHz PC under Linux. We report results in seconds, taking the average of three runs, each restricted to 600s time and 3GB RAM.[6]

Table 1 compares *clingo*, *adsolver*, and variant (c) of *clingcon*, which turned out to be the best choice on the considered benchmarks (see below), on 12 randomly picked sample instances and varying number of logical time steps. The instances stem from the *instances-monica* folder, "3-0/025" means subfolder *ins-3-0* instance *instance_025*. Average runtimes over all instances are provided in the last row of Table 1, taking timeouts as maximum time, viz., 600s. Using *clingo* (on direct ASP encodings), we can solve the instances for 5 time steps, where the major effort is made in grounding; in fact, we observed memory exhaustion on all instances for 7 time steps. With *adsolver*, such space problems do not occur, but it runs into timeouts (indicated by —) for 11 and 13 time steps. Up to these time steps, *clingcon* still scales well and is an order of magnitude faster than *adsolver*. The last column shows results for the greatest step number, 20, for which *clingcon* solved all instances within 600s.

[5] `http://www.krlab.cs.ttu.edu/Software/Download/rcs`

[6] Much main memory is needed solely for grounding in *clingo*.

| | 11 | | | | 13 | | | | 20 | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| *Benchmark* | (a) | (b) | (c) | (d) | (a) | (b) | (c) | (d) | (a) | (b) | (c) | (d) |
| 3-0/025 | 12.83 | 4.90 | 4.21 | 4.21 | 36.71 | 8.09 | 5.99 | 5.90 | — | 64.89 | 17.84 | 18.78 |
| 3-0/050 | 49.21 | 13.25 | 6.80 | 8.42 | 219.01 | 35.38 | 11.85 | 12.41 | — | 370.71 | 27.36 | 30.46 |
| 3-0/100 | 36.44 | 9.54 | 10.11 | 5.30 | 34.82 | 53.96 | 12.04 | 7.16 | — | — | 38.78 | 33.67 |
| 3-0/125 | 6.31 | 4.25 | 4.05 | 4.07 | 163.47 | 21.09 | 10.00 | 8.52 | — | 377.06 | 133.99 | 31.46 |
| 5-0/025 | 69.36 | 15.38 | 5.87 | 10.52 | 176.81 | 23.39 | 14.27 | 15.06 | — | — | 66.55 | 118.65 |
| 5-0/050 | 24.40 | 6.58 | 4.71 | 4.67 | 311.11 | 22.93 | 10.04 | 7.19 | — | 542.27 | 241.59 | — |
| 5-0/100 | 6.39 | 4.30 | 4.13 | 4.11 | 72.95 | 6.36 | 6.45 | 5.70 | — | 83.85 | 25.32 | 49.03 |
| 5-0/125 | 61.72 | 17.66 | 6.39 | 6.96 | 111.92 | 41.84 | 9.70 | 10.87 | — | — | 81.17 | 73.68 |
| 8-0/025 | 76.73 | 7.69 | 11.00 | 9.25 | 248.29 | 37.33 | 12.69 | 7.81 | — | — | 222.49 | — |
| 8-0/050 | 6.33 | 4.19 | 4.02 | 4.05 | 189.04 | 15.75 | 7.76 | 8.05 | — | 52.14 | 457.86 | 32.29 |
| 8-0/100 | 7.71 | 4.49 | 5.05 | 4.16 | 220.12 | 11.71 | 6.10 | 5.94 | — | 159.37 | 26.17 | 23.72 |
| 8-0/125 | 5.11 | 4.31 | 4.15 | 4.14 | 38.91 | 15.31 | 6.69 | 7.06 | — | 69.64 | 17.82 | 18.80 |
| ∅ | 30.21 | 8.05 | 5.87 | 5.82 | 151.93 | 24.43 | 9.47 | 8.47 | — | 343.33 | 113.08 | 135.88 |

**Table 2.** Comparing different strategies within *clingcon*.

Table 2 compares the four different settings of *clingcon* with each other. We observed that exploiting constraint interdependencies in variants (a) and (b) may decrease the number of heuristic decisions made by *clingcon*. As regards runtime, it however turns out that the additional effort does not pay off. For one, this is because the calculation of constraint interdependencies is not yet fully optimized in *clingcon*, and the overhead could possibly be reduced. This also explains why variant (b), more eagerly recording reasons than (a), is faster: storing more reasons permits more inferences by unit propagation, and thus, it reduces calculations of constraint interdependencies. However, variants (c) and (d) using simple-to-compute trivial reasons still seem to be superior. Interestingly, the lazy approach of (c) to calculate reasons during conflict analysis performs better than (d) recording reasons during propagation, which is converse to the relationship between (a) and (b). This shift of behaviors can be explained by the overhead of reason calculation: while it is expensive with (a) so that recording more reasons with (b) helps, it is cheap with (c), and exhaustive reason recording in (d) slows down unit propagation more than additional inferences pay off.

## 6 Discussion

We introduced a novel approach to integrating CP capacities into modern ASP solvers based on conflict-driven learning and backjumping. Our semantic approach relies on a propositional language rather than a multi-sorted, first-order language, as used in [12–14]. Also, we follow the lazy approach of advanced SMT solvers by abstracting from the constraints in a specialized theory [9]. A major difficulty in this endeavor was the current lack of CP solvers providing an interface supporting conflict-driven learning. We addressed this problem by developing a new algorithmic framework for incorporating a CP "oracle" into the approach to conflict-driven ASP solving introduced in [19]. Apart from extending unit propagation through constraint propagation, the major extension dealt with conflict analysis and the elaboration of reasons for atoms derived by constraint propagation.

Our approach differs in several ways from the ones developed in [12–14]. As mentioned above, our semantic approach is propositional and abstracts from the constraints in a specialized theory. Unlike this, [12–14] start with a multi-sorted, first-order language leading to a propositional program through grounding. As well, they rely upon so-called mixed predicates for linking constants with constraint variables. Also, [12–14] use traditional ASP solving algorithms, based on DPLL-style backtracking. In fact, *adsolver*'s implementation relies on *lparse* and *smodels*. The implementation described in [13] allows the usage of difference constraints of the form $X - Y > k$ for variables $X, Y$ and constant $k$; at most one such constraint is allowed within an integrity constraint. The underlying CP solver is handcrafted and thus supports incremental solving and backtracking. Unlike this, we use with *gecode* an off-the-shelf CP system. Although it is incremental, backtracking and reason generation must be dealt with externally. In [24], "functional oracles" allow for computing instantiations of so-called external predicates during grounding. Constraint atoms in our sense can also be viewed as being external to some extent, given that the associated constraints are evaluated by a CP engine. Importantly, the non-Boolean domains of variables in such constraint are still present in the solving phase, while a functional oracle would have to make the domains explicit for constructing a propositional program under standard answer set semantics.

We have empirically evaluated *adsolver* and *clingcon* on the benchmark suite used to appraise *adsolver*'s performance in [13, 14]. First of all, we note that both systems escape the grounding bottleneck faced by traditional ASP systems like *clingo*. All in all, however, we observed that *clingcon* outperforms *adsolver* by up to two orders of magnitude. Also, we investigated the effect of different variants of reason generation on the performance of *clingcon*. As regards the current prototype, it turned out that additional efforts into the elaboration of constraint interdependencies do not pay off. However, this issue deserves further attention and is subject to future research.

*Acknowledgments.* We are grateful to Michael Gelfond and Yuanlin Zhang for useful discussions on the subject of this paper. This work was partially funded by DFG under Grant SCHA 550/8-1 and by the GoFORSYS[7] project under Grant 0313924.

## References

1. Baral, C.: Knowledge Representation, Reasoning and Declarative Problem Solving. Cambridge University Press (2003)
2. Nogueira, M., Balduccini, M., Gelfond, M., Watson, R., Barry, M.: An A-prolog decision support system for the space shuttle. In Ramakrishnan, I., ed.: Proceedings of the Third International Symposium on Practical Aspects of Declarative Languages (PADL'01). Springer (2001) 169–183
3. Balduccini, M., Gelfond, M., Nogueira, M.: Answer set based design of knowledge systems. Annals of Mathematics and Artificial Intelligence **47**(1-2) (2006) 183–219
4. Baral, C., Chancellor, K., Tran, N., Tran, N., Joy, A., Berens, M.: A knowledge based approach for representing and reasoning about signaling networks. In: Proceedings of the Twelfth International Conference on Intelligent Systems for Molecular Biology/Third European Conference on Computational Biology (ISMB'04/ECCB'04). (2004) 15–22

[7] `http://www.goforsys.org`

5. Dworschak, S., Grote, T., König, A., Schaub, T., Veber, P.: Tools for representing and reasoning about biological models in action language $\mathcal{C}$. In Pagnucco, M., Thielscher, M., eds.: Proceedings of the Twelfth International Workshop on Nonmonotonic Reasoning (NMR'08). The University of New South Wales, Technical Report Series (2008) 94–102
6. Gebser, M., Schaub, T., Thiele, S., Usadel, B., Veber, P.: Detecting inconsistencies in large biological networks with answer set programming. In Garcia de la Banda, M., Pontelli, E., eds.: Proceedings of the Twenty-fourth International Conference on Logic Programming (ICLP'08). Springer (2008) 130–144
7. Mitchell, D.: A SAT solver primer. Bulletin of the European Association for Theoretical Computer Science **85** (2005) 112–133
8. Biere, A., Heule, M., van Maaren, H., Walsh, T., eds.: Handbook of Satisfiability. IOS Press (2009)
9. 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
10. Dechter, R.: Constraint Processing. Morgan Kaufmann Publishers (2003)
11. Rossi, F., van Beek, P., Walsh, T., eds.: Handbook of Constraint Programming. Elsevier (2006)
12. Baselice, S., Bonatti, P., Gelfond, M.: Towards an integration of answer set and constraint solving. In Gabbrielli, M., Gupta, G., eds.: Proceedings of the Twenty-first International Conference on Logic Programming (ICLP'05). Springer (2005) 52–66
13. Mellarkod, V., Gelfond, M.: Integrating answer set reasoning with constraint solving techniques. In Garrigue, J., Hermenegildo, M., eds.: Proceedings of the Ninth International Symposium on Functional and Logic Programming (FLOPS'08). Springer (2008) 15–31
14. Mellarkod, V., Gelfond, M., Zhang, Y.: Integrating answer set programming and constraint logic programming. Annals of Mathematics and Artificial Intelligence (2008) To appear
15. Simons, P., Niemelä, I., Soininen, T.: Extending and implementing the stable model semantics. Artificial Intelligence **138**(1-2) (2002) 181–234
16. `http://www.cs.uni-potsdam.de/clingcon`
17. `http://potassco.sourceforge.net`
18. `http://www.gecode.org`
19. 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/MIT Press (2007) 386–392
20. Gebser, M., Kaminski, R., Kaufmann, B., Ostrowski, M., Schaub, T., Thiele, S.: A user's guide to `gringo`, `clasp`, `clingo`, and `iclingo`. [17]
21. Marques-Silva, J., Sakallah, K.: GRASP: A search algorithm for propositional satisfiability. IEEE Transactions on Computers **48**(5) (1999) 506–521
22. 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
23. Fages, F.: Consistency of Clark's completion and the existence of stable models. Journal of Methods of Logic in Computer Science **1** (1994) 51–60
24. Calimeri, F., Cozza, S., Ianni, G.: External sources of knowledge and value invention in logic programming. Annals of Mathematics and Artificial Intelligence **50**(3-4) (2007) 333–361