Clingcon: The Next Generation

Mutsunori Banbara
Kobe University, Japan

Benjamin Kaufmann and Max Ostrowski
University of Potsdam, Germany

Torsten Schaub∗
University of Potsdam, Germany and INRIA Rennes, France

submitted 1 January 2003; revised 1 January 2003; accepted 1 January 2003

Abstract

We present the third generation of the constraint answer set system *clingcon*, combining Answer Set Programming (ASP) with finite domain constraint processing (CP). While its predecessors rely on a black-box approach to hybrid solving by integrating the CP solver *gecode*, the new *clingcon* system pursues a lazy approach using dedicated constraint propagators to extend propagation in the underlying ASP solver *clasp*. No extension is needed for parsing and grounding *clingcon*'s hybrid modeling language since both can be accommodated by the new generic theory handling capabilities of the ASP grounder *gringo*. As a whole, *clingcon* 3 is thus an extension of the ASP system *clingo* 5, which itself relies on the grounder *gringo* and the solver *clasp*. The new approach of *clingcon* offers a seamless integration of CP propagation into ASP solving that benefits from the whole spectrum of *clasp*'s reasoning modes, including for instance multi-shot solving and advanced optimization techniques. This is accomplished by a lazy approach that unfolds the representation of constraints and adds it to that of the logic program only when needed. Although the unfolding is usually dictated by the constraint propagators during solving, it can already be partially (or even totally) done during preprocessing. Moreover, *clingcon*'s constraint preprocessing and propagation incorporate several well established CP techniques that greatly improve its performance. We demonstrate this via an extensive empirical evaluation contrasting, first, the various techniques in the context of CSP solving and, second, the new *clingcon* system with other hybrid ASP systems.

1 Introduction

The shortcoming of Answer Set Programming (ASP; (Lifschitz 2008)) to succinctly represent variables over large numeric domains has led to the development of several systems enhancing ASP with capabilities for finite domain Constraint Processing (CP; (Rossi et al. 2006)). Starting from the seminal work in (Baselice et al. 2005) and the consecutive development of traditional DPLL[1]-style hybrid ASP solvers like *adsolver* (Mellarkod et al. 2008), modern hybrid ASP solvers take advantage of CDCL[2]-based solving technology (Marques-Silva and Sakallah 1999; Zhang et al. 2001; Gebser et al. 2007) in different ways. Let us illustrate this by describing the approach of three representative Constraint Answer Set Programming (CASP; (Balduccini and Lierler 2013)) systems.

∗ Affiliated with the Simon Fraser University, Canada, and Griffith University, Australia.
[1] Tracing back to the Davis-Putman-Logemann-Loveland procedure (Davis and Putnam 1960; Davis et al. 1962)
[2] Standing for: Conflict-Driven Constraint Learning

A black-box approach is pursued in the two previous *clingcon* series where the ASP solver *clasp* is combined with the CP solver *gecode* (Gecode Team 2006) by following the lazy approach to SMT[3]solving (Barrett et al. 2009). In the *clingcon* setting, this means that *clasp* only generates truth assignments for abstracted constraint expressions, while *gecode* checks whether the actual constraints can be made true or false accordingly. On the one hand, this black-box approach benefits from the vast spectrum of constraints available in *gecode* and seamlessly keeps up with advanced CP technology, among others regarding preprocessing and propagation. Moreover, this approach avoids an explicit representation of integer variables in ASP and thus can deal with very large domains. On the other hand, the usage of an external CP solver restricts information exchange which impedes the CDCL approach of *clasp*. First, neither conflict nor propagation information is provided by *gecode* and thus must be approximated within the interface to sustain conflict analysis in CDCL. Second, the granularity induced by constraint abstraction leads to weaker propagation than what is obtainable when encoding integer variables.

A translation-based approach is pursued by the *aspartame* system (Banbara et al. 2015) where a CSP[4] is fully translated into ASP and then solved by an ASP solver. This approach follows the one of the CP solver *sugar* (Tamura et al. 2009) translating CSPs to SAT[5] (Biere et al. 2009). This is done by representing each integer variable along with its domain according to the order encoding scheme (Crawford and Baker 1994). Such an approach is called eager in SMT solving. On the one hand, this approach benefits from the full power of CDCL-based search. Also, the granularity induced by an explicit representation of integer variables provides more accurate conflict and propagation information, and approximations for reasons and conflicts as used in the former *clingcon* system (Ostrowski and Schaub 2012) are made obsolete. On the other hand, such an explicit representation limits scalability: *aspartame* (just as *sugar*) can only deal with medium sized domains up to a few thousand integers. Also, when dealing with larger domains, CDCL search may suffer from congestion due to too much conflict information. Finally, *aspartame* cannot make use of readily available CP techniques for preprocessing and propagation; all this must be captured in the underlying ASP encoding.

A lazy approach is pursued by the *inca* system (Drescher and Walsh 2012) where the ASP solver *clasp* is augmented with dedicated propagators for linear and selected global constraints by following the approach of lazy clause generation (Ohrimenko et al. 2009). The idea is to make parts of the encoding explicit whenever they reflect a conflict or propagation signaled by a propagator. In this way, the explicit representation of constraints is only unfolded when needed and its extent is controlled by the deletion scheme of the ASP solver. This approach also benefits from the full power of CDCL-based search but outsources constraint oriented inferences. In this way, the overall size of the hybrid problem is under control of the ASP solver. As a consequence, *inca* can deal with large domains. But it has its limits because the vocabulary and basic inference schemes of the order encoding must be provided at the outset by introducing auxiliary variables and nogoods. The propagators rely on this for making parts of the constraint encoding explicit. Moreover, this lazy approach cannot harness implemented CP techniques for preprocessing and propagation; *inca* provides advanced means for propagation but uses no sophisticated preprocessing techniques.

The third generation of *clingcon* also follows a lazy approach to hybrid ASP solving but largely extends the lazy one of *inca* while drawing on experience with *aspartame* and the previous

[3] Standing for: Satisfiability Modulo Theories
[4] Standing for: Constraint Satisfaction Problem
[5] Standing for: Satisfiability Testing

clingcon series. The current version of *clingcon* 3 features propagators for linear constraints and can translate distinct constraints. The ultimate design goal was to conceive a hybrid solver architecture that integrates seamlessly with the infrastructure of the ASP system *clingo* in order to take advantage of its full spectrum of grounding and solving capabilities. For the latter, it is essential to give the solver access to the representation of constraint variables and their domains, otherwise hybrid forms of multi-objective optimization or operations on models like intersection or union cannot reuse existing capacities. The lazy approach lets us accomplish this while controlling space demands. However, we take the approach of *inca* one step further by permitting lazy variable generation (Thibaut and Stuckey 2009) to unfold the vocabulary and the basic inference schemes of the order encoding only when needed. This enables *clingcon* 3 to represent very large (and possibly non-contiguous) domains of integer variables. Furthermore, *clingcon* 3 features a variety of established CP preprocessing techniques to enhance its lazy approach. This also includes an initial eager translation that allows for unfolding up front parts or even the entire CSP.

What is more, *clingcon* is not restricted to single-shot solving but fully blends in with *clingo*'s multi-shot solving capabilities (Gebser et al. 2015). This does not only allow for incremental hybrid solving but moreover equips *clingcon* with powerful APIs. For instance, the latter allow for conceiving reactive procedures to loop on solving while acquiring changes in the problem specification. In fact, due to our design, most of *clingo*'s elaborate features carry over to *clingcon*. Among others, this includes multi-threaded solving as well as unsatisfiable core and model-driven multi-criteria optimization. Exceptions to this are signature-based forms of reasoning, like projective enumeration or heuristic modifications that must be dealt with indirectly by associating constraint atoms with auxiliary regular atoms with which such operations can be performed.

Our paper is structured as follows. The next section provides the formal foundations of Constraint Answer Set Programming (CASP) and presents the basics of CDCL-based ASP solving along with their extension to CASP solving. Section 3 details relevant features of *clingcon* 3. We start with an architectural overview in Section 3.1 and introduce the input language of *clingcon* 3 in Section 3.2. We then explain *clingcon*'s extended solving algorithms in Section 3.3 and detail distinguished features in Section 3.4. The final subsection of Section 3 is dedicated to multi-shot CASP solving. Section 4 provides a detailed empirical analysis of *clingcon*'s features and performance in contrast to competing CP and CASP systems. We summarize the salient features of the new *clingcon* series in Section 5 and discuss related work.

2 Formal Preliminaries

We begin in Section 2.1 with a gentle introduction to CASP along with some auxiliary concepts. We then provide the basics of CDCL-based ASP solving and show how they extend to CASP solving in Section 2.2.

2.1 Constraint Answer Set Programming

Constraint logic programs consist of a logic program P over disjoint sets $\mathcal{A}, \mathcal{C}$ of propositional variables, and an associated constraint satisfaction problem (CSP) $(\mathcal{V}, D, C)$. Elements of $\mathcal{A}$ and $\mathcal{C}$ are referred to as regular and constraint atoms, respectively. We consider linear CSPs, where $\mathcal{V}$ is a set of integer variables, D is a set of corresponding variable domains, and C is a set of linear constraints.

Logic programs. A logic program P consists of rules of the form[6]

$$a_0 \leftarrow a_1, \dots, a_m, \sim a_{m+1}, \dots, \sim a_n \tag{1}$$

where $0 \leq m \leq n$ and $a_0 \in \mathcal{A}$ and each $a_i \in \mathcal{A} \cup \mathcal{C}$ is an atom for $1 \leq i \leq n$.

As an example, consider the logic program P_1:

$$a \leftarrow \sim b \tag{2}$$
$$b \leftarrow \sim a \tag{3}$$
$$c \leftarrow a, x < 7 \tag{4}$$

This program contains regular atoms a, b, and c from $\mathcal{A}$ along with the constraint atom $x < 7$ from $\mathcal{C}$. Accordingly, x is an integer variable in $\mathcal{V}$.

We need the following auxiliary definitions. We define $\mathit{head}(r) = a_0$ as the head of rule r in (1), $\mathit{body}(r) = \{a_1, \dots, a_m, \sim a_{m+1}, \dots, \sim a_n\}$ as its body, and $\mathit{atom}(r) = \{a_0, a_1, \dots, a_m, a_{m+1}, \dots, a_n\}$. Moreover, we let $\mathit{head}(P) = \{\mathit{head}(r) \mid r \in P\}$, $\mathit{body}(P) = \{\mathit{body}(r) \mid r \in P\}$, $\mathit{body}_P(a) = \{\mathit{body}(r) \mid r \in P, \mathit{head}(r) = a\}$, and $\mathit{atom}(P) = \{\mathit{atom}(r) \mid r \in P\}$. If $\mathit{body}(r) = \emptyset$, r is called a fact. If $\mathit{head}(r)$ is missing, r is called an integrity constraint and r stands for $x \leftarrow \mathit{body}(r), \sim x$ where x is a new atom.[7]

In ASP, the semantics of a logic program is given by its *(constraint) stable models* (Gelfond and Lifschitz 1988; Gebser et al. 2009). However, in view of our focus on computational aspects, we rather deal with Boolean assignments and constraints and give a corresponding characterization of a program's stable models below.

Constraint Satisfaction Problems. A linear CSP $(\mathcal{V}, D, C)$ deals with linear constraints in C of the form

$$a_1 v_1 + \dots + a_n v_n \leq b \tag{5}$$

where a_i and b are integers and $v_i \in \mathcal{V}$ for $1 \leq i \leq n$. The domain of a variable $v \in \mathcal{V}$ is given by $D(v)$. The complement of a constraint $c \in C$ is denoted as $\overline{c}$. We require that C is closed under complements. Constraint atoms in $\mathcal{C}$ are identified with constraints in C via a function $\gamma : \mathcal{C} \rightarrow C$.

In our example, we have $x \in \mathcal{V}$ and let $D(x) = \{1, \dots, 10\}$. Moreover, we associate the constraint atom $x < 7$ with the linear constraint $x \leq 6$, or formally, $\gamma(x < 7) = x \leq 6$. Since we require C to be closed under complements, it contains both $x \leq 6$ and its complement $-x \leq -7$.

An assignment $\mathbf{C} : v \in \mathcal{V} \mapsto d \in D(v)$ *satisfies* a linear constraint, if (5) holds after replacing each v_i by $\mathbf{C}(v_i)$. We let $\mathit{sat}_{\mathbf{C}}(C)$ denote the set of all constraints in C satisfied by $\mathbf{C}$. Following (Drescher 2015), we call $(\mathbf{C}, \mathit{sat}_{\mathbf{C}}(C))$ a *configuration* of $(\mathcal{V}, D, C)$. For instance, the assignment $\mathbf{C} = \{x \mapsto 5\}$ satisfies the linear constraint $x \leq 6$. Accordingly, $(\{x \mapsto 5\}, \{x \leq 6\})$ is a configuration of $(\{x\}, \{D(x)\}, \{x \leq 6, -x \leq -7\})$.

Moreover, we rely on the CP concept of a *view*. Following (Schulte and Tack 2005), a *view* on a variable x is an expression $ax + b$ for integers a, b; its *image* is defined as $\mathit{img}(ax + b) = \{ax + b \mid x \in D(x)\}$.[8] Since a view $ax + b$ can always be replaced with a fresh variable y along with a constraint $y = ax + b$, we may use them nearly everywhere where we would otherwise

[6] We present our approach in the context of normal logic programs, though it readily applies to disjunctive logic programs — as does *clingcon* 3.
[7] As syntactic sugar, a rule $c \leftarrow a_1, \dots, a_m, \sim a_{m+1}, \dots, \sim a_n$ with a constraint atom $c \in \mathcal{C}$ in the head stands for $\leftarrow a_1, \dots, a_m, \sim a_{m+1}, \dots, \sim a_n, \sim c$.
[8] Any linear expression with only one variable can be converted to an expression of the form $ax + b$.

use variables. For a view v, we define $lb(v)$ and $ub(v)$ as the smallest/largest value in $img(v)$.[9] Then, $prev(d, v)$ ($next(d, v)$) is a function mapping a value d to the largest (smallest) element d' in $img(v)$ which is smaller (larger) than d if $d > lb(v)$ ($d < ub(v)$), otherwise it is $-\infty$ (∞). In our example, we have $lb(x) = 1$ and $ub(x) = 10$, and for instance $prev(17, 2x + 3) = 15$, $prev(5, x) = 4$, and $prev(0, x) = -\infty$, respectively.

2.2 Basics of ASP and CASP Solving

The basic idea of CDCL-based ASP solving is to map inferences from rules as in (1) to unit propagation on Boolean constraints. Our description of this approach follows the one given in (Gebser et al. 2012).

Accordingly, we represent Boolean assignments, $\mathbf{B}$, over a set of atoms $\mathcal{A} \cup \mathcal{C}$ by sets of *signed literals* $\mathbf{T}a$ or $\mathbf{F}a$ standing for $a \mapsto \mathbf{T}$ and $a \mapsto \mathbf{F}$, respectively, where $a \in \mathcal{A} \cup \mathcal{C}$. The complement of a signed literal σ is denoted by $\overline{\sigma}$. We define $\mathbf{B}^\mathbf{T} = \{a \in \mathcal{A} \cup \mathcal{C} \mid \mathbf{T}a \in \mathbf{B}\}$ and $\mathbf{B}^\mathbf{F} = \{a \in \mathcal{A} \cup \mathcal{C} \mid \mathbf{F}a \in \mathbf{B}\}$. Then, an assignment $\mathbf{B}$ is *complete*, if $\mathbf{B}^\mathbf{T} \cap \mathbf{B}^\mathbf{F} = \emptyset$ and $\mathbf{B}^\mathbf{T} \cup \mathbf{B}^\mathbf{F} = \mathcal{A} \cup \mathcal{C}$. For instance, the assignment $\{\mathbf{T}a, \mathbf{F}b, \mathbf{F}c, \mathbf{F}(x < 7)\}$ is complete wrt the atoms in our example.

Boolean constraints are represented as *nogoods*. A nogood is a set of signed literals representing an invalid partial assignment. A nogood δ is *violated* by a Boolean assignment $\mathbf{B}$ whenever $\delta \subseteq \mathbf{B}$. A complete Boolean assignment is a *solution* of a set of nogoods, if it violates none of them. Given a Boolean assignment $\mathbf{B}$ and a nogood δ such that $\delta \setminus \mathbf{B} = \{\sigma\}$ and $\overline{\sigma} \notin \mathbf{B}$, we say that δ is *unit* wrt $\mathbf{B}$ and *asserts* the *unit-resulting literal* $\overline{\sigma}$. For a set Δ of nogoods and an assignment $\mathbf{B}$, *unit propagation* is the iterated process of extending $\mathbf{B}$ with unit-resulting literals until no further literal is unit-resulting for any nogood in Δ.

With these concepts in hand, the Boolean constraints induced by a logic program P can be captured as follows:

$$\begin{aligned}
\Delta_P &= \bigcup_{\substack{B \in body(P),\\ B=\{a_1,\dots,a_m,\sim a_{m+1},\dots,\sim a_n\}}} \left\{ \begin{array}{l} \{\mathbf{F}B, \mathbf{T}a_1, \dots, \mathbf{T}a_m, \mathbf{F}a_{m+1}, \dots, \mathbf{F}a_n\}, \\ \{\mathbf{T}B, \mathbf{F}a_1\}, \dots, \{\mathbf{T}B, \mathbf{F}a_m\}, \\ \{\mathbf{T}B, \mathbf{T}a_{m+1}\}, \dots, \{\mathbf{T}B, \mathbf{T}a_n\} \end{array} \right\} \\
&\cup \bigcup_{\substack{a \in atom(P),\\ body_P(a)=\{B_1,\dots,B_k\}}} \left\{ \begin{array}{l} \{\mathbf{T}a, \mathbf{F}B_1, \dots, \mathbf{F}B_k\}, \\ \{\mathbf{F}a, \mathbf{T}B_1\}, \dots, \{\mathbf{F}a, \mathbf{T}B_k\} \end{array} \right\} && (6) \\
\Lambda_P &= \bigcup_{\substack{U \subseteq atom(P),\\ EB_P(U)=\{B_1,\dots,B_k\}}} \{\{\mathbf{T}a, \mathbf{F}B_1, \dots, \mathbf{F}B_k\} \mid a \in U\} && (7) \\
&\quad \text{where } EB_P(U) = \{body(r) \in P \mid head(r) \in U, body(r) \cap U = \emptyset\}.
\end{aligned}$$

Then, according to (Gebser et al. 2012), a set of atoms X is a stable model of a regular logic program P iff $X = \mathbf{B}^\mathbf{T} \cap atom(P)$ for a (unique) solution $\mathbf{B}$ of $\Delta_P \cup \Lambda_P$.

For example, the nogoods obtained in (6) for the atom a in our example are $\{\mathbf{T}a, \mathbf{F}\{\sim b\}\}$ and $\{\mathbf{F}a, \mathbf{T}\{\sim b\}\}$. Similarly, the body $\{\sim b\}$ of Rule (2) gives rise to nogoods $\{\mathbf{F}\{\sim b\}, \mathbf{F}b\}$ and $\{\mathbf{T}\{\sim b\}, \mathbf{T}b\}$. Hence, once an assignment contains $\mathbf{T}a$, we may derive $\mathbf{F}b$ via unit propagation (using both the first and last nogood).

To extend this characterization to programs with constraint atoms, it is important to realize that the truth value of such atoms is determined external to the program. In CASP, this is reflected

[9] Note that for a view of the form $1x + 0$ we have $D(x) = img(x)$.

by the requirement that constraint atoms must not occur in the head of rules.[10] Hence, treating constraint atoms as regular ones leaves them unfounded. For instance, in our example, we would get from both (6) and (7) the nogood $\{\mathbf{T}(x<7)\}$, which would set $(x<7)$ permanently to false. To address this issue, (Drescher and Walsh 2012) exempt constraint atoms from the respective sets of nogoods and define the variants $\Delta_P^{\mathcal{C}}$ and $\Lambda_P^{\mathcal{C}}$ by replacing $atom(P)$ in the qualification of (6) and (7) with $atom(P)\setminus\mathcal{C}$.

Then, in (Ostrowski 2017) it is shown that $(X,\mathbf{C})$ is a constraint stable model of a program P wrt $(\mathcal{V},D,C)$ as defined in (Gebser et al. 2009) iff and $X=\mathbf{B}^{\mathbf{T}}\cap atom(P)$ for a (unique) solution $\mathbf{B}$ of $\Delta_P^{\mathcal{C}}\cup\Lambda_P^{\mathcal{C}}\cup\{\{\mathbf{F}c\}\mid\gamma(c)\in sat_{\mathbf{C}}(C)\}\cup\{\{\mathbf{T}c\}\mid\overline{\gamma(c)}\in sat_{\mathbf{C}}(C)\}$.

Accordingly, our example yields the following constraint stable models

$$\begin{array}{ll}
X & \mathbf{C}\\
\{a\} & x\in\{7,\dots,10\}\\
\{b\} & x\in\{7,\dots,10\}\\
\{b,x<7\} & x\in\{1,\dots,6\}\\
\{a,c,x<7\} & x\in\{1,\dots,6\}
\end{array}\tag{8}$$

where $x\in\{m,\dots,n\}$ means that either $x\mapsto m$, or $x\mapsto m+1,\dots$ or $x\mapsto n$. For instance, the very first constraint stable model corresponds to the Boolean assignment $\{\mathbf{T}a,\mathbf{F}b,\mathbf{F}c,\mathbf{F}(x<7)\}$ paired with the constraint variable assignment $\{x\mapsto 7\}$.

Similar to logic programs, linear constraints can be represented as sets of nogoods by means of an *order encoding* (Tamura et al. 2009). This amounts to representing the above unit nogoods $\{\{\mathbf{F}c\}\mid\gamma(c)\in sat_{\mathbf{C}}(C)\}\cup\{\{\mathbf{T}c\}\mid\overline{\gamma(c)}\in sat_{\mathbf{C}}(C)\}$ by more elaborate nogoods capturing the semantics provided by $sat_{\mathbf{C}}(C)$.

To this end, we let $\mathcal{O}_{\mathcal{V}}$ stand for the set of order atoms associated with variables in $\mathcal{V}$ and require it to be disjoint from $\mathcal{A}\cup\mathcal{C}$. Whenever the set $\mathcal{V}$ is clear from the context, we drop it and simply write $\mathcal{O}$. More precisely, we introduce an *order atom* $(v\le d)\in\mathcal{O}$ for each constraint variable $v\in\mathcal{V}$ and value $d\in D(v), d\neq ub(v)$. We refer to signed literals over $\mathcal{O}$ as *signed order literals*.

Now, we are ready to map a linear CSP $(\mathcal{V},D,C)$ into a set of nogoods.

First, we need to make sure that each variable in $\mathcal{V}$ has exactly one value from its domain in D. To this end, we define the following set of nogoods.

$$\begin{aligned}\Phi(\mathcal{V},D)=\{\{\mathbf{T}(v\le d),\mathbf{F}(v\le next(d,v))\}\mid\ & v\in\mathcal{V},d\in D(v),\\ & next(d,v)<ub(v)\}\end{aligned}\tag{9}$$

Intuitively, each such nogood stands for an implication "$(v\le d)\Rightarrow(v\le d+1)$". In our example, we get the following nogoods.

$$\Phi(\{x\},\{D(x)\})=\{\{\mathbf{T}(x\le 1),\mathbf{F}(x\le 2)\},\dots,\{\mathbf{T}(x\le 8),\mathbf{F}(x\le 9)\}\}.\tag{10}$$

Second, we need to establish the relation between constraint atoms $\mathcal{C}$ and their associated linear constraints in C. Following (Feydy et al. 2011), a *reified constraint* is an equivalence "$\mathbf{T}c\Leftrightarrow\gamma(c)$" where $c\in\mathcal{C}$; it is decomposable into two *half-reified* constraints "$\mathbf{T}c\Rightarrow\gamma(c)$" and "$\mathbf{F}c\Rightarrow\overline{\gamma(c)}$".

[10] In alternative semantic settings, theory atoms may also occur as rule heads (cf. (Gebser et al. 2016a)).

To proceed analogously, we extend γ to signed literals over $\mathcal{C}$ as follows:

$$\gamma(\sigma) = \begin{cases} \overline{\gamma(a)} & \text{if } \sigma = \mathbf{F}a, a \in \mathcal{C} \\ \gamma(a) & \text{if } \sigma = \mathbf{T}a, a \in \mathcal{C} \end{cases}$$

For instance, we have $\gamma(\mathbf{F}(x < 7)) = (-x \leq -7)$.

To translate constraints into nogoods, we need to translate expressions of the form $av + b \leq 0$ for $v \in \mathcal{V}$ and integers a, b into signed ordered literals.[11] Following (Tamura et al. 2009), we then define $(av + b \leq 0)^\ddagger$ as

$$(av + b \leq 0)^\ddagger = \begin{cases} (v \leq \lfloor \frac{-b}{a} \rfloor)^\dagger & \text{if } a > 0 \\ \overline{(v \leq \lceil \frac{-b}{a} \rceil - 1)^\dagger} & \text{if } a < 0 \end{cases}$$

where $(v \leq d)^\dagger$ is defined for $lb(v) \leq d < ub(v)$ as

$$(v \leq d)^\dagger = \begin{cases} \mathbf{T}(v \leq d) & \text{if } d \in D(v) \\ \mathbf{T}(v \leq \mathit{prev}(d, v)) & \text{if } d \notin D(v) \end{cases}$$

If $d \geq ub(v)$ then $(v \leq d)^\dagger = \mathbf{T}\emptyset$; if $d < lb(v)$ then $(v \leq d)^\dagger = \mathbf{F}\emptyset$, where $\emptyset$ stands for the empty body.[12] Expressing our example constraint $x \leq 6$ in terms of signed order literals results in $(1 \cdot x + (-6) \leq 0)^\ddagger = \mathbf{T}(x \leq 6)$. The signed literal $\mathbf{T}(x \leq 6)$ indicates that 6 is the largest integer satisfying the constraint. Also, we get the signed literals $(x \leq 0)^\dagger = \mathbf{F}\emptyset$ and $(x \leq 10)^\dagger = \mathbf{T}\emptyset$.

We sometimes use $<$,$>$, or $\geq$ as operators in these expressions and implicitly convert them to the normal form $av + b \leq 0$ to be used in this translation. Accordingly, the complementary constraint yields $(x > 6)^\ddagger = ((-1) \cdot x + 7 \leq 0)^\ddagger = \overline{(x \leq \lceil \frac{-7}{-1} \rceil - 1)^\dagger} = \mathbf{F}(x \leq 6)$.

The actual relation between the constraint atoms in $\mathcal{C}$ and their associated linear constraints in C is established via the following nogoods.

$$\Psi(\mathcal{C}) = \textstyle\bigcup_{c \in \mathcal{C}} \psi(\mathbf{T}c, \gamma(c)) \cup \psi(\mathbf{F}c, \overline{\gamma(c)}) \, . \tag{11}$$

For all constraint atoms $c \in \mathcal{C}$ associated with the linear constraint $\gamma(c) = \sum_{i=1}^n a_i v_i \leq b$ in C, we define for both of its half-reified constraints the set of nogoods

$$\psi(\mathbf{T}c, \textstyle\sum_{i=1}^n a_i v_i \leq b) = \{\{\mathbf{T}c\} \cup \delta \setminus \{\mathbf{T}\emptyset\} \mid \delta \in \phi(\sum_{i=1}^n a_i v_i \leq b), \mathbf{F}\emptyset \notin \delta\} \tag{12}$$

$$\psi(\mathbf{F}c, \overline{\textstyle\sum_{i=1}^n a_i v_i \leq b}) = \{\{\mathbf{F}c\} \cup \delta \setminus \{\mathbf{T}\emptyset\} \mid \delta \in \phi(\overline{\textstyle\sum_{i=1}^n a_i v_i \leq b}), \mathbf{F}\emptyset \notin \delta\} \tag{13}$$

where

$$\phi(\textstyle\sum_{i=1}^n a_i v_i \leq b) = \left\{ \begin{array}{ll} \{(a_1 v_1 > b)^\ddagger\} & \text{if } n = 1 \\ \{(a_1 v_1 \geq d)^\ddagger\} \cup \delta & \text{if } n > 1 \\ \qquad \delta \in \phi(\sum_{i=2}^n a_i v_i \leq b - d), d \in \mathit{img}(a_1 v_1) & \end{array} \right\}$$

Note that nogoods with $\mathbf{T}\emptyset$ and $\mathbf{F}\emptyset$ are simplified in (12) and (13). Also, observe that the definition of ϕ is recursive although this does not show with our simple examples.

[11] Any linear inequality using $<, >, \leq, \geq$ and one variable can be converted into this form.

[12] We use $\mathbf{T}\emptyset$ and $\mathbf{F}\emptyset$ as representatives for tautological and unsatisfiable signed literals; they are removed in (12) and (13) below.

In our example, we obtain

$$\psi(\mathbf{T}(x < 7), x \leq 6) = \{\{\mathbf{T}(x < 7), \mathbf{F}(x \leq 6)\}\} \tag{14}$$

$$\psi(\mathbf{F}(x < 7), -x \leq -7) = \{\{\mathbf{F}(x < 7), \mathbf{T}(x \leq 6)\}\} \tag{15}$$

Taken together, both nogoods realize the aforementioned equivalence between the constraint atom $(x < 7)$ and its associated constraint. Note that $(x < 7)$ is a constraint atom in $\mathcal{C}$, while $(x \leq 6)$ is an order atom in $\mathcal{O}$ and thus belongs to the encoding of the constraint associated with $(x < 7)$. For further illustration, reconsider the Boolean assignment $\{\mathbf{T}a, \mathbf{F}b, \mathbf{F}c, \mathbf{F}(x < 7)\}$ inducing the first constraint stable models in (8). Applying unit propagation, we get $\mathbf{F}(x \leq 6)$ via (15) and in turn $\mathbf{F}(x \leq 5)$ to $\mathbf{F}(x \leq 1)$ via the nogoods in $\Phi(\{x\}, \{D(x)\})$ in (10). Similarly, making $\mathbf{T}(x \leq 7)$ true yields $\mathbf{T}(x \leq 8)$ and $\mathbf{T}(x \leq 9)$ also via the nogoods in (10).

All in all, a CSP $(\mathcal{V}, D, C)$ is characterized by the nogoods in $\Phi(\mathcal{V}, D)$ and $\Psi(\mathcal{C})$.

While in (8) the corresponding constraint variable assignment $\mathbf{C}$ is determined externally, it can be directly extracted from a solution $\mathbf{B}$ for $\Phi(\mathcal{V}, D)$ by means of the following functions: The upper bound for a view v relative to a Boolean assignment $\mathbf{B}$ is given by $ub_{\mathbf{B}}(v) = \min(\{ub(v)\} \cup \{d \mid d \in img(v), (v \leq d)^{\ddagger} \in \mathbf{B}\})$ and its lower bound by $lb_{\mathbf{B}}(v) = \max(\{lb(v)\} \cup \{d \mid d \in img(v), (v \geq d)^{\ddagger} \in \mathbf{B}\})$. Then, $\mathbf{C}(v) = lb_{\mathbf{B}}(v) = ub_{\mathbf{B}}(v)$ for all $v \in \mathcal{V}$. Accordingly, the above Boolean assignment corresponds to $\mathbf{C} = \{x \mapsto 7\}$.

Combining the nogoods stemming from the logic program and its associated CSP, we obtain the following characterization of constraint logic programs.

Theorem 2.1
Let P be a constraint logic program over $\mathcal{A} \cup \mathcal{C}$ associated with the CSP $(\mathcal{V}, D, C)$ and let $X \subseteq \mathcal{A} \cup \mathcal{C}$ and $\mathbf{C}$ a total assignment over $\mathcal{V}$.

Then, $(X, \mathbf{C})$ is a constraint stable model of P wrt $(\mathcal{V}, D, C)$ as defined in (Gebser et al. 2009) iff $(\mathbf{C}, sat_{\mathbf{C}}(C))$ is a configuration for $(\mathcal{V}, D, C)$, $X = \mathbf{B}^{\mathbf{T}} \cap atom(P)$ for a (unique) solution $\mathbf{B}$ of $\Delta_P^{\mathcal{C}} \cup \Lambda_P^{\mathcal{C}} \cup \Psi(\mathcal{C}) \cup \Phi(\mathcal{V}, D)$, and $\mathbf{C} = \{v \mapsto lb_{\mathbf{B}}(v) \mid v \in \mathcal{V}\}$.

The proof of this theorem is obtained by combining existing characterizations of logic programs in terms of nogoods and similar ones for CSPs in terms of clauses in CNF (Ostrowski 2017).

Nogood propagators. The basic idea of lazy constraint propagation is to make the nogoods in $\Psi(\mathcal{C})$ and $\Phi(\mathcal{V}, D)$ only explicit when needed. This is done by propagators corresponding to the respective set of nogoods. A popular example of this is the unfounded-set algorithm in ASP solvers that only makes the nogoods in Λ_P in (7) explicit when needed.

Following (Drescher and Walsh 2012), a *propagator* for a set Θ of nogoods is a function Π_Θ mapping a Boolean assignment $\mathbf{B}$ to a subset of Θ such that for each total assignment $\mathbf{B}$: if $\delta \subseteq \mathbf{B}$ for some $\delta \in \Theta$, then $\delta' \subseteq \mathbf{B}$ for some $\delta' \in \Pi_\Theta(\mathbf{B})$. That is, whenever there is a nogood in Θ violated by an assignment $\mathbf{B}$, then $\Pi_\Theta(\mathbf{B})$ yields a violated nogood, too. A propagator Π_Θ is *conflict optimal*, if for all partial assignments $\mathbf{B}$, the violation of a nogood in Θ by $\mathbf{B}$ implies that some nogood in $\Pi_\Theta(\mathbf{B})$ is violated by $\mathbf{B}$. Π_Θ is *inference optimal*, if it is conflict optimal and $\Pi_\Theta(\mathbf{B})$ contains all unit nogoods of Θ wrt $\mathbf{B}$.

We obtain the following extension of Theorem 2.1.

Theorem 2.2
Let P be a constraint logic program over $\mathcal{A} \cup \mathcal{C}$ associated with the CSP $(\mathcal{V}, D, C)$ and let Π_Θ be a propagator for $\Theta = \Lambda_P^{\mathcal{C}}$, $\Psi(\mathcal{C})$, and $\Phi(\mathcal{V}, D)$, respectively.

Then, $\mathbf{B}$ is a solution of $\Delta_P^{\mathcal{C}} \cup \Lambda_P^{\mathcal{C}} \cup \Psi(\mathcal{C}) \cup \Phi(\mathcal{V}, D)$ iff $\mathbf{B}$ is a solution of

$$\Delta_P^{\mathcal{C}} \cup \Pi_{\Lambda_P^{\mathcal{C}}}(\mathbf{B}) \cup \Pi_{\Psi(\mathcal{C})}(\mathbf{B}) \cup \Pi_{\Phi(\mathcal{V},D)}(\mathbf{B}) .$$

This theorem tells us that the nogoods in $\Psi(\mathcal{C})$, $\Phi(\mathcal{V}, D)$, and $\Lambda_P^{\mathcal{C}}$ must not be explicitly represented but can be computed by corresponding propagators Π_Θ that add them lazily when needed.

To relax the restrictions imposed by this theorem, the idea is to compile out a subset of constraints and variables of the CSP while leaving the others subject to lazy constraint propagation. This is captured by the following corollary to Theorem 2.2.

Corollary 2.1
Let P be a constraint logic program over $\mathcal{A} \cup \mathcal{C}$ associated with the CSP $(\mathcal{V}, D, C)$ and let Π_Θ be a propagator for $\Theta = \Lambda_P^{\mathcal{C}}$, $\Psi(\mathcal{C} \setminus \mathcal{C}')$, and $\Phi(\mathcal{V} \setminus \mathcal{V}', D \setminus D')$, respectively, for subsets $\mathcal{C}' \subseteq \mathcal{C}$, $\mathcal{V}' \subseteq \mathcal{V}$, and $\mathcal{D}' \subseteq \mathcal{D}$.

Then, $\mathbf{B}$ is a solution of $\Delta_P^{\mathcal{C}} \cup \Lambda_P^{\mathcal{C}} \cup \Psi(\mathcal{C}) \cup \Phi(\mathcal{V}, D)$ iff $\mathbf{B}$ is a solution of

$$\Delta_P^{\mathcal{C}} \cup \Psi(\mathcal{C}') \cup \Phi(\mathcal{V}', D') \cup \Pi_{\Lambda_P^{\mathcal{C}}}(\mathbf{B}) \cup \Pi_{\Psi(\mathcal{C}\setminus\mathcal{C}')}(\mathbf{B}) \cup \Pi_{\Phi(\mathcal{V}\setminus\mathcal{V}',D\setminus D')}(\mathbf{B}) .$$

This correspondence nicely reflects upon the basic idea of our approach. While the entire set of loop nogoods $\Lambda_P^{\mathcal{C}}$ is handled by the unfounded set propagator $\Pi_{\Lambda_P^{\mathcal{C}}}(\mathbf{B})$ as usual, the ones capturing the CSP is divided among the explicated nogoods in $\Psi(\mathcal{C}') \cup \Phi(\mathcal{V}', D')$ and the implicit ones handled by the propagators $\Pi_{\Psi(\mathcal{C}\setminus\mathcal{C}')}(\mathbf{B})$ and $\Pi_{\Phi(\mathcal{V}\setminus\mathcal{V}',D\setminus D')}(\mathbf{B})$. Note that variables and domain elements are often only dealt with implicitly through their induced order atoms in $\mathcal{O}$.

3 The *clingcon* system

We now detail various aspects of the new *clingcon* 3 system. We begin with an overview of its architecture along with its salient components. The next sections detail its input language and major algorithms. The subsequent section is dedicated to distinguished *clingcon* features, which are experimentally evaluated in Section 4. Finally, we illustrate in the last section *clingcon*'s multi-shot solving capabilities by discussing several incremental solutions to the n-queens puzzle.

3.1 Architecture

clingcon 3 is an extension of the ASP system *clingo* 5, which itself relies on the grounder *gringo* and the solver *clasp*. The architecture of *clingcon* 3 is given in Figure 1. More precisely, *clingcon*

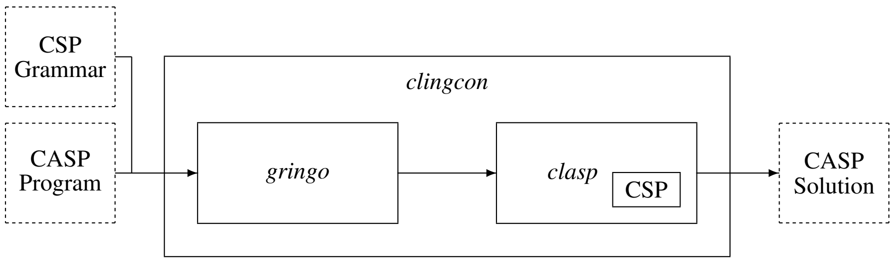

Fig. 1: Architecture of *clingcon* 3
.

uses *gringo*'s capabilities to specify and process customized theory languages. For this, it is sufficient to supply a grammar fixing the syntax of constraint-related expressions. As detailed in Section 3.2, this allows us to express linear constraints similar to standard ASP aggregates by using first-order variables. Unlike this, *clingcon* extends *clasp* in several ways to accommodate its lazy approach to constraint solving. First, *clasp*'s preprocessing capabilities are extended to integrate linear constraints. Second, dedicated propagators are added to account for lazy constraint propagation. Both extensions are detailed in Section 3.3 and 3.4. And finally, a special output module was created to integrate CSP solutions. Notably, *clingcon* pursues a lazy yet two-fold approach to constraint solving that allows for making a part of the nogoods in $\Psi(\mathcal{C})$ explicit during preprocessing, while leaving the remaining constraints implicit and the creation of corresponding nogoods subject to the constraint propagator. In this way, a part of the CSP can be put right up front under the influence of CDCL-based search. All other constraints are only turned into nogoods when needed. Accordingly, only a limited subset of order atoms from $\mathcal{O}$ must be introduced at the beginning; further ones are only created if they are needed upon the addition of new nogoods. This is also called lazy variable generation.

It is worth mentioning that both the grounding and the solving component of *clingcon* can also be used separately via *clingo*'s option '`--mode`'. That is, the same result as with *clingcon* is obtained by passing the output of '`clingcon --mode=gringo`' to '`clingcon --mode=clasp`'. The intermediate result of grounding a CASP program is expressed in the *aspif* format (Gebser et al. 2016b) that accommodates both the regular ASP part of the program as well as its constraint-based extension. This modular design allows others to take advantage of *clingcon*'s infrastructure for their own CASP solvers. Also, other front ends can be used for generating ground CASP programs; eg. the *flatzinc* translator used in Section 4.

Finally, extra effort was taken to transfer *clasp* specific features to *clingcon*'s solving component. This includes multi-threading (Gebser et al. 2012), unsatisfiable core techniques (Andres et al. 2012), multi-criteria optimization (Gebser et al. 2011), domain-specific heuristics (Gebser et al. 2013), multi-shot solving (Gebser et al. 2015), and *clasp*'s reasoning modes like enumeration, intersection and union of models. Vocabulary-sensitive reasoning modes like projective enumeration and domain-specific heuristics can be used via auxiliary atoms.

3.2 Language

As mentioned, the treatment of the extended input language of CASP programs can be mapped onto *gringo*'s theory language capabilities (Gebser et al. 2016a). For this, it is sufficient to supply a corresponding grammar fixing the syntax of the language extension. The one used for *clingcon* is given in Listing 1. The grammar is named `csp` and consists of two parts, one defining theory terms in lines 2-27 and another defining theory atoms in lines 29-33. All regular terms are implicitly included in the respective theory terms. Theory terms are then used to represent constraint-related expressions that are turned by grounding into linear constraint atoms using predicate `&sum`, domain restrictions using predicate `&dom`, directives `&show` and `&minimize`, and the predefined global constraint `&distinct`.

Before delving into further details, let us illustrate the resulting syntax by the CASP program for two dimensional strip packing given in Listing 2, originally due to (Soh et al. 2010). Given a set of rectangles, each represented by a fact `r(I,W,H)` where `I` identifies a rectangle with width `W` and height `H`, the task is to fit all into a container of width `w` and height `ub` while minimizing the needed height of the container. The first two lines of Listing 2 restrict the domain of the

```
#theory csp {
    dom_term {
    + : 5, unary;
    - : 5, unary;
    .. : 1, binary, left;
    * : 4, binary, left;
    + : 3, binary, left;
    - : 3, binary, left
    };
    linear_term {
    + : 5, unary;
    - : 5, unary;
    * : 4, binary, left;
    + : 3, binary, left;
    - : 3, binary, left
    };
    show_term {
    / : 1, binary, left
    };
    minimize_term {
    + : 5, unary;
    - : 5, unary;
    * : 4, binary, left;
    + : 3, binary, left;
    - : 3, binary, left;
    @ : 0, binary, left
    };

    &dom/0 : dom_term, {=}, linear_term, any;
    &sum/0 : linear_term, {<=,=,>=,<,>,!=}, linear_term, any;
    &distinct/0 : linear_term, any;
    &show/0 : show_term, directive;
    &minimize/0 : minimize_term, directive
}.
```

Listing 1: *Language Syntax*

left lower corner of each rectangle `I`. The respective instantiations of `x(I)` and `y(I)` yield constraint variables denoting the `x` and `y` coordinate of `I`, respectively. Note that in both lines the consecutive dots '`..`' construct a theory term '`0..w-W`' and '`0..ub-H`' once `w` and `ub` are replaced, respectively. The choice rule in Line 4-7 lets us choose among all combinations of two rectangles, that is, which one is left, right, below or above. At least one of these relations must hold so that no two rectangles overlap. Atoms of form `le(VI,C,VJ)` indicate that coordinate `VI+C` must be less than or equal to `VJ`. This property is enforced by the linear constraint in Line 9. Finally, to minimize the overall height of (stacked) rectangles, we introduce the variable `height`. This variable's value has to be greater than or equal to the `y` coordinate of any rectangle `I` plus the rectangle's height `H`. This ensures that `height` is greater or equal to the height of the highest rectangle. Finally, `height` is minimized in Line 13.

Now, if we take the three rectangles `r(a,5,2)`, `r(b,2,3)`, `r(c,2,2)` along with `ub=10` and `w=6`, we obtain the ground program in Listing 3. The domains of the constraint variables giving

```
 1  &dom{0..w-W}  = x(I) :- r(I,W,H).
 2  &dom{0..ub-H} = y(I) :- r(I,W,H).

 4  1 { le(x(I),WI,x(J));
 5      le(x(J),WJ,x(I));
 6      le(y(I),HI,y(J));
 7      le(y(J),HJ,y(I)) } :- r(I,WI,HI), r(J,WJ,HJ), I < J.

 9  &sum{VI; C} <= VJ :- le(VI,C,VJ).

11  &dom{0..ub} = height.
12  &sum{y(I); H} <= height :- r(I,W,H).
13  &minimize {height}.
14  &show {height}.
```

Listing 2: *Two Dimensional Strip Packing*

the x- and y-coordinates are delineated in Line 3 and 4. Note that in contrast to regular ASP the grounder leaves terms with the theory symbol `..` intact. The orientation of each pair of rectangles is chosen in Lines 6-11. If for example `le(x(c),2,x(b))` becomes true, that is, rectangle c is left of b, then the constraint $x(c) + 2 \leq x(b)$ is enforced in Line 22. After setting the domain for the `height` variable in Line 26, we restrict it to be greater or equal to the top y-coordinate of all rectangles in Lines 28-30. Line 32 enforces the minimization of this variable. A solution with minimal `height` consists of the regular atoms `le(y(b),3,y(a))`, `le(y(c),2,y(a))`, and `le(x(c),2,x(b))` and the constraint variable assignment $\{\texttt{height} \mapsto 5, \texttt{y(c)} \mapsto 1, \texttt{x(c)} \mapsto 2, \texttt{x(a)} \mapsto 1, \texttt{x(b)} \mapsto 4, \texttt{y(a)} \mapsto 3, \texttt{y(b)} \mapsto 0\}$. Of course other minimal configurations exist.

We have seen above how seamlessly theory atoms capturing constraint-related expressions can be used in logic programs. We detail below the five distinct atoms featured by *clingcon* and refer the interested reader for a general introduction to theory terms and atoms to (Gebser et al. 2016a).

Actual constraints are represented by the theory atoms `&dom`, `&sum`, and `&distinct`. All three can occur in the head and body of rules, as indicated by `any` in Line 29-31 in Listing 1. We discuss below their admissible format after grounding. In the following, a linear expression is a sum of integers, products of integers, or products of an integer and a constraint variable.

Domain constraints are of form $\texttt{\&dom}\{d_1; \dots; d_n\} = t$ where

- each d_i is a domain term of form
  - — u or
  - — $v..w$

  where u, v, w are constraint variable free linear expressions and
- t is a linear expression containing exactly one constraint variable.

Then, the previous expression represents the constraint $t \in \bigcup_{i=1}^{n} [\![d_i]\!]$, where $[\![d]\!] = \{u\}$ if $d = u$, $[\![d]\!] = \{v, \dots, w\}$ if $d = v..w$, and undefined otherwise.
This constraint can be used to set the domain of variables where even non-contiguous domains can be used by having $n > 1$. For example $\texttt{\&dom}\{1..3; 5\} = x$ represents the constraint $x \in \{1, \dots, 3\} \cup \{5\}$.

Linear constraints are of form $\texttt{\&sum}\{t_1; \dots; t_n\} \circ t_{n+1}$ where

```
r(a,5,2). r(b,2,3). r(c,2,2).

&dom{0..(6-5)}  = x(a). &dom{0..(6-2)}  = x(b). &dom{0..(6-2)}  = x(c).
&dom{0..(10-2)} = y(a). &dom{0..(10-3)} = y(b). &dom{0..(10-2)} = y(c).

1 <= { le(x(a),5,x(b)); le(x(b),2,x(a));
       le(y(a),2,y(b)); le(y(b),3,y(a)) }.
1 <= { le(x(a),5,x(c)); le(x(c),2,x(a));
       le(y(a),2,y(c)); le(y(c),2,y(a)) }.
1 <= { le(x(b),2,x(c)); le(x(c),2,x(b));
       le(y(b),3,y(c)); le(y(c),2,y(b)) }.

&sum{ x(a); 5 } <= x(b) :- le(x(a),5,x(b)).
&sum{ x(b); 2 } <= x(a) :- le(x(b),2,x(a)).
&sum{ y(a); 2 } <= y(b) :- le(y(a),2,y(b)).
&sum{ y(b); 3 } <= y(a) :- le(y(b),3,y(a)).
&sum{ x(a); 5 } <= x(c) :- le(x(a),5,x(c)).
&sum{ x(c); 2 } <= x(a) :- le(x(c),2,x(a)).
&sum{ y(a); 2 } <= y(c) :- le(y(a),2,y(c)).
&sum{ y(c); 2 } <= y(a) :- le(y(c),2,y(a)).
&sum{ x(b); 2 } <= x(c) :- le(x(b),2,x(c)).
&sum{ x(c); 2 } <= x(b) :- le(x(c),2,x(b)).
&sum{ y(b); 3 } <= y(c) :- le(y(b),3,y(c)).
&sum{ y(c); 2 } <= y(b) :- le(y(c),2,y(b)).

&dom{ 0..10 } = height.

&sum{ y(a); 2 } <= height.
&sum{ y(b); 3 } <= height.
&sum{ y(c); 2 } <= height.

&minimize{ height }.
&show{ height }.
```

Listing 3: *Two Dimensional Strip Packing Example*

- each t_i is a linear expression containing at most one constraint variable, and
- $\circ$ is one of the operators `<=`, `=`, `>=`, `<`, `>`, `!=`

This expression represents the linear constraint $(t_1 + \cdots + t_n) \circ t_{n+1}$, which can be translated into one or two linear constraints as in (5).

Distinct constraints are of form `&distinct`$\{t_1; \ldots; t_n\}$ where each t_i is a linear expression containing at most one constraint variable. Such an expression stands for the constraints $t_i \neq t_j$ for $0 \leq i < j \leq n$.

The distinct constraint is one of the most common global constraints in CP. We use it to show how global constraints can be incorporated into the language.

The two remaining theory atoms provide directives, similar to their regular counterparts.

Output directives are of form `&show`$\{s_1; \ldots; s_n\}$ where each s_i is a show term of form

- f/m where f is a function symbol and m a positive integer
- t, where t is a constraint variable.

While the latter adds variable t to the list of output variables, the first one adds all variables of the form $f(t_1, \ldots, t_m)$ (where t_i is a term) as output variables. For all constraint stable models, the value of the output variables is shown in a solution.

Minimize directives are of form `&minimize`$\{m_1; \ldots; m_n\}$ where each m_i is a minimize term of form $t_i@l_i$ and t_i being a linear expression with at most one constraint variable. Since we support multi-objective optimization, l_i is an integer stating the priority level. Whenever $@l_i$ is omitted, it is assumed to be zero. Priorities allow for representing lexicographically ordered minimization objectives. As in regular ASP, higher levels are more significant than lower ones. Let us make precise how minimize statements induce optimal constraint stable models. Let P be a constraint logic program associated with $(\mathcal{V}, D, C)$. For a variable assignment $\mathbf{C}$ and an integer l, define $\sum_l^{\mathbf{C}}$ as the sum of all values $a \cdot \mathbf{C}(v) + c$ for all occurrences of minimize terms $av + c@l$ in all minimize statements in P. A constraint stable model $(X, \mathbf{C})$ of P wrt $(\mathcal{V}, D, C)$ is dominated if there is a constraint stable model $(X', \mathbf{C}')$ such that $\sum_l^{\mathbf{C}'} < \sum_l^{\mathbf{C}}$ and $\sum_{l'}^{\mathbf{C}'} = \sum_{l'}^{\mathbf{C}}$ for all $l' > l$, and optimal otherwise. Maximization can be achieved by multiplying each minimize term by -1.

Note that the set of constraints supported by *clingcon* is only a subset of the constraints expressible with the syntax fixed in Listing 1. While for example expressions with more than one constraint variable are well-formed according to the syntax, they are not supported by *clingcon*.

3.3 Algorithms

As mentioned, *clingcon* pursues a lazy approach to constraint solving that distinguishes two phases. During preprocessing, any part of the nogoods representing a CSP can be made explicit and thus put right away under the influence of CDCL-based solving. Unlike this, the remaining constraints are at first kept implicit and their corresponding nogoods are only added via constraint propagators to CDCL solving when needed. This partitioning of constraints constitutes a trade-off. On the one hand, constraint propagators are usually slower than unit propagation, in particular, when dealing with sets of nogoods of moderate size because of modern SAT techniques such as the two-watched-literals scheme (Zhang et al. 2001). On the other hand, translating all constraints is often impracticable, in particular, when dealing with very large domains. Hence, a good trade-off is to restrict the translation to "small constraints" in order to benefit from the high performance of CDCL solving and to unfold "larger constraints" only by need.

In what follows, we make *clingcon*'s two-fold approach precise by presenting algorithms for translation and propagation of constraints before discussing implementation details in Section 3.4.

Partial Translation. Following Corollary 2.1, a subset $\mathcal{C}' \subseteq \mathcal{C}$ of the constraint atoms is used to create the set of nogoods $\Psi(\mathcal{C}')$. Therefore, Algorithm 1 creates a set of nogoods that is equivalent to $\psi(\sigma, a_1v_1 + \cdots + a_nv_n \leq b)$, as defined in (12) and (13); in turn, they are used to create $\Psi(\mathcal{C}')$ as shown in (11). To this end, it is initially engaged by $\textsc{Translate}(\{\sigma\}, a_1v_1 + \cdots + a_nv_n \leq b)$. We start the algorithm by having σ in our set of literals δ, and setting d to the smallest value greater than $b - \sum_{j=2}^{n} ub(a_jv_j)$ in the image of a_1v_1. This is the smallest value needed to violate the constraint. If d and the least sum $\sum_{j=2}^{n} lb(a_jv_j)$ added by all other views is still less than b in Line 4, we have to recursively translate the rest of the constraint, while subtracting d from the right-hand side in Line 5. Otherwise the constraint is already violated and we return all nogoods created so far in Line 7. We iteratively increase d in Line 8 and repeat this process (Line 3) for all values

```
Algorithm 1: TRANSLATE
  Input  : A set of signed literals $\delta$ and a linear constraint $a_1v_1 + \cdots + a_nv_n \leq b$
  Output : A set of nogoods
1 $\Sigma \leftarrow \emptyset$
2 $d \leftarrow next(b - \sum_{j=2}^{n} ub(a_jv_j), a_1v_1)$
3 while $d \leq ub(a_1v_1)$
4     if $d + \sum_{j=2}^{n} lb(a_jv_j) \leq b$ then
5         $\Sigma \leftarrow \Sigma \cup$ TRANSLATE($\delta \cup \{(a_1v_1 \geq d)^{\ddagger}\}$, $a_2v_2 + \cdots + a_nv_n \leq b - d$)
6     else
7         return $\Sigma \cup \{\delta \cup \{(a_1v_1 \geq d)^{\ddagger}\}\}$
8     $d \leftarrow next(d, a_1v_1)$
9 return $\Sigma$
```

in $img(a_1v_1)$. Note that this also involves adding all order atoms $\mathcal{O}_{\Psi(\mathcal{C}')} = \bigcup_{\delta \in \Psi(\mathcal{C}')} \delta^{\mathbf{T}} \cup \delta^{\mathbf{F}}$ included in the created nogoods $\Psi(\mathcal{C}')$ to the solver.

Which constraints to translate is subject to heuristics and command line options, as explained in Section 3.4.

Extended Conflict Driven Constraint Learning. After translating a part of the problem into a set of nogoods $\Psi(\mathcal{C}')$, using the order atoms $\mathcal{O}_{\Psi(\mathcal{C}')} \subseteq \mathcal{O}$, we explain how to solve the remaining constraint logic program P over $\mathcal{A}, \mathcal{C}$ associated with $(\mathcal{V}, D, C)$. Our algorithmic approach follows the one in (Drescher and Walsh 2012), where a modified CDCL algorithm supporting external propagators is presented. We extend this algorithm with lazy nogood and variable generation in Algorithm 2. The algorithm relies upon a growing set of Boolean variables $\mathcal{B}$, which is initiated

```
Algorithm 2: EXTENDED CDCL
   Input  : A constraint logic program $P$ over $\mathcal{A}, \mathcal{C}$ associated with $(\mathcal{V}, D, C)$, a set of
            constraints atoms $\mathcal{C}' \subseteq \mathcal{C}$, and a set of order atoms $\mathcal{O}_{\Psi(\mathcal{C}')} \subseteq \mathcal{O}$
   Output : A constraint stable model or unsatisfiable
1  $\mathcal{B} \leftarrow \mathcal{A} \cup \mathcal{C} \cup \mathcal{O}_{\Psi(\mathcal{C}')}$          // set of atoms
2  $\mathbf{B} \leftarrow \emptyset$                                  // assignment over $\mathcal{A} \cup \mathcal{C} \cup \mathcal{O}$
3  $\nabla \leftarrow \emptyset$                                  // set of (dynamic) nogoods
4  loop
5      $(\mathcal{B}, \mathbf{B}, \nabla) \leftarrow$ PROPAGATION($\mathcal{B}, \mathbf{B}, \mathcal{C}', \nabla$)
6      if CONFLICT($\mathbf{B}$) then
7          if ROOTCONFLICT($\mathbf{B}$) then return unsatisfiable
8          $(\mathbf{B}, \nabla) \leftarrow$ BACKTRACK$_P$($\mathbf{B}, \nabla$)
9      else if COMPLETE($\mathbf{B}$) then
10         if $lb_{\mathbf{B}}(v) = ub_{\mathbf{B}}(v)$ for all $v \in \mathcal{V}$ then
11             return $(\mathbf{B}^{\mathbf{T}} \cap atom(P), \{v \mapsto lb_{\mathbf{B}}(v) \mid v \in \mathcal{V}\})$
12         else $\mathcal{B} \leftarrow \mathcal{B} \cup \{$SPLIT$_{\mathcal{V},D}(\mathcal{B}, \mathbf{B})\}$
13     else $\mathbf{B} \leftarrow \mathbf{B} \cup \{$SELECT($\mathcal{B}$)$\}$
```

with all atoms (regular, constraint, and a subset of the order atoms in $\mathcal{O}_{\Psi(\mathcal{C}')}$), and subsequently expanded by further order atoms. Accordingly, the Boolean assignment $\mathbf{B}$ is restricted to atoms in $\mathcal{B}$, and recorded nogoods are accumulated in ∇. Starting with an empty assignment, the PROPAGATION method (Line 5), extends the assignment $\mathbf{B}$ with propagated literals, adds new nogoods to ∇ and extends the set of atoms $\mathcal{B}$. This method is detailed below in Algorithm 3. When encountering a conflicting assignment (Line 6), we either backtrack (Line 8) or, if we cannot recover from the conflict, return *unsatisfiable*. Whenever all atoms in $\mathcal{B}$ are assigned (Line 9), we check whether a complete assignment for the variables in $\mathcal{V}$ is obtained from $\mathbf{B}$ in Line 10. If this is the case, we return the constraint stable model $(\mathbf{B}^{\mathbf{T}} \cap \mathit{atom}(P), \{v \mapsto \mathit{lb}_{\mathbf{B}}(v) \mid v \in \mathcal{V}\})$. Otherwise, $\text{SPLIT}_{\mathcal{V},D}(\mathcal{B}, \mathbf{B})$ creates a new order atom for the constraint variable with the currently largest domain that splits the domain in half. If we face an incomplete assignment, we extend it using the SELECT function.

Algorithm 3: PROPAGATION

```
Global : A constraint logic program $P$ over $\mathcal{A}, \mathcal{C}$ associated with $(\mathcal{V}, D, C)$
Input  : A set of atoms $\mathcal{B}$, a Boolean assignment $\mathbf{B}$, a set of constraint atoms $\mathcal{C}'$, and a set of
         learned nogoods $\nabla$
Output : A set of atoms, a Boolean assignment, and a set of learned nogoods
1  loop
2      $\mathbf{B} \leftarrow \text{UNITPROPAGATION}_P(\mathbf{B}, \Psi(\mathcal{C}') \cup \nabla)$
3      if CONFLICT($\mathbf{B}$) then return $(\mathcal{B}, \mathbf{B}, \nabla)$
4      $\nabla' \leftarrow \text{UFSPROPAGATION}_P(\mathbf{B})$
5      if $\nabla' \neq \emptyset$ then
6          $\nabla \leftarrow \nabla \cup \nabla'$
7      else
8          $\nabla' \leftarrow \text{CSPPROPAGATION}(\mathcal{B}, \mathcal{C}', \mathbf{B})$
9          if $\nabla' \neq \emptyset$ then
10             for $\delta \in \nabla'$ do $\mathcal{B} \leftarrow \mathcal{B} \cup \delta^{\mathbf{T}} \cup \delta^{\mathbf{F}}$
11             $\nabla \leftarrow \nabla \cup \nabla'$
12         else return $(\mathcal{B}, \mathbf{B}, \nabla)$
13     end
```

Algorithm 3 reflects the proceeding of our propagators. At first, UNITPROPAGATION is run on the completion nogoods $\Delta_P^{\mathcal{C}}$, the nogoods from the partial translation $\Psi(\mathcal{C}')$, and finally the already learned nogoods ∇. Then, propagator $\Pi_{\Lambda_P^{\mathcal{C}}}$ is engaged via UFSPROPAGATION. If it does not add any new nogoods to ∇, CSPPROPAGATION is called. This method acts as a propagator, returning a set of nogoods ∇'. Since some of these nogoods may use new order atoms not introduced so far, we dynamically extend the set of atoms $\mathcal{B}$ by the atoms in $\delta^{\mathbf{T}} \cup \delta^{\mathbf{F}}$ stemming from the added nogoods $\delta \in \nabla'$.

New nogoods produced by any propagator are added to the set ∇ of recorded nogoods and propagation resumes afterwards (lines 6 and 11). Notably, CSPPROPAGATION is not run until a fixpoint is obtained. However, its set of returned nogoods remains non empty until a fixpoint is reached. In this way, unit propagation interleaves with constraint propagation while delaying more complex propagation. In all, since unit propagation is much faster, it always precedes unfounded set propagation, which again precedes constraint propagation. This order reflects the complexity of the respective propagators, so that the faster the propagation, the sooner it is engaged.

Lazy Variable Generation. Realizing CSPPROPAGATION as a propagator for $\Pi_{\Psi(\mathcal{C})}$ and $\Pi_{\Phi(\mathcal{V},D)}$ allows for lazy nogood generation and for capturing inferences of the order encoding. However, to be effective, lazy variable generation requires a different set of constraints to be propagated. For illustration, suppose CSPPROPAGATION is a propagator for $\Psi(\mathcal{C}) \cup \Phi(\mathcal{V}, D)$. Considering our example program P_1 along with $\mathbf{T}(x < 7) \in \mathbf{B}$ results in CSPPROPAGATION$(\emptyset, \emptyset, \{\mathbf{T}(x < 7)\}) = \{\{\mathbf{T}(x < 7), \mathbf{F}(x \leq 6)\}\}$, which is a subset of $\Psi(\mathcal{C})$ according to (14). This nogood comprises the order atom $(x \leq 6)$ which is added to $\mathcal{B}$ in Line 10 of Algorithm 3. Having this nogood, unit propagation adds in turn $\mathbf{T}(x \leq 6)$ to the assignment in Line 2. Then, CSPPROPAGATION$(\{(x \leq 6)\}, \emptyset, \{\mathbf{T}(x \leq 6)\})$ yields the nogoods $\{\{\mathbf{T}(x \leq 6), \mathbf{F}(x \leq 7)\}, \dots, \{\mathbf{T}(x \leq 8), \mathbf{F}(x \leq 9)\}\}$ belonging to $\Phi(\mathcal{V}, D)$ and produces the corresponding order atoms $(x \leq 7), \dots, (x \leq 9)$. We see that once a certain upper bound $\mathbf{T}(v \leq x) \in \mathbf{B}$ is found, all order atoms in $\{(v \leq x') \mid x' > x, x' \in D(v), x' < ub(v)\}$ are added to $\mathcal{B}$. Similarly, if a lower bound $\mathbf{F}(v \leq x) \in \mathbf{B}$ is fixed, all order atoms $\{(v \leq x') \mid x' \leq x, x' \in D(v)\}$ are added to $\mathcal{B}$. To avoid adding superfluous order atoms, we let CSPPROPAGATION be a propagator for $\Psi(\mathcal{C}) \cup \Phi'(\mathcal{V}, D)$ where

$$\Phi'(\mathcal{V}, D) = \{\{\mathbf{T}(v \leq d), \mathbf{F}(v \leq e)\} \mid v \in \mathcal{V}, d \in D(v), e \in D(v), d < e < ub(v)\}.$$

Although $\Phi'(\mathcal{V}, D)$ is a superset of $\Phi(\mathcal{V}, D)$, CSPPROPAGATION only adds nogoods from $\Phi'(\mathcal{V}, D)$ whose order atoms have already been introduced, that is, $\{(v \leq d), (v \leq e)\} \subseteq \mathcal{B}$. While $\Phi(\mathcal{V}, D)$ contains for each variable v a linear number of nogoods of form $\{\mathbf{T}(v \leq d), \mathbf{F}(v \leq \mathit{next}(d, v))\}$, $\Phi'(\mathcal{V}, D)$ contains a quadratic number of nogoods for each variable. The nogoods in $\Phi(\mathcal{V}, D)$ allow for propagating the truth value of one order literal to its adjacent one. Unlike this, $\Phi'(\mathcal{V}, D)$ contains redundant nogoods that allow for propagating the truth value of one order literal to all greater ones by means of nogoods of form $\{\mathbf{T}(v \leq d), \mathbf{F}(v \leq e)\}$ for all values $e \in D(v)$ such that $d < e < ub(v)$. Instead of "chaining" all values together, the latter nogoods allow us to directly jump to any value. As we restrict our propagator for $\Phi'(\mathcal{V}, D)$ to only return nogoods where all order atoms are included in $\mathcal{B}$, no new order atoms are created. In our example, this optimized CSPPROPAGATION function does not return any nogoods, viz. CSPPROPAGATION$(\{(x \leq 6)\}, \emptyset, \{\mathbf{T}(x \leq 6)\}) = \emptyset$, as none of the order atoms $(x \leq 7), \dots, (x \leq 9)$ are included in $\mathcal{B}$ and no propagation needs to be done.

CSPPROPAGATION is depicted in Algorithm 4 and consists of two parts (lines 1-10 and 11-21). The first part starts with selecting the unit nogoods from $\Phi'(\mathcal{V}, D)$. For every variable $v \in \mathcal{V}$, we check if it already has an upper bound ub (lines 3-4) given by $\mathbf{T}(v \leq ub) \in \mathbf{B}$. If this is the case, we add the nogoods

$$\{\{\mathbf{T}(v \leq ub), \mathbf{F}(v \leq x)\} \mid x > ub, (v \leq x) \in \mathcal{B}, \mathbf{T}(v \leq x) \notin \mathbf{B}\}$$

to Σ to ensure consistency of all order atoms $(v \leq x) \in \mathcal{B}$ where $x > ub$ that are not already true. Lines 6-8 do the same for current lower bound of the variable. If any nogoods are found, they are immediately returned in Line 10. The PROPAGATION function continues with unit propagation on the new nogoods. The second part of the constraint propagation (lines 11-21), generating the nogoods in $\Psi(\mathcal{C} \setminus \mathcal{C}')$ lazily, is only done if all order atoms are properly propagated, i.e. no new nogoods have been generated in the first part (lines 1-10). This is detailed in the next paragraph.

Algorithm 4: CspPropagation

```
Global : A constraint logic program $P$ over $\mathcal{A},\mathcal{C}$ associated with $(\mathcal{V},D,C)$
Input  : A set of atoms $\mathcal{B}$, a set of constraint atoms $\mathcal{C}'$, and a Boolean assignment $\mathbf{B}$
Output : A set of nogoods
1  $\Sigma \leftarrow \emptyset$                    // an empty set of nogoods
2  for $v \in \mathcal{V}$ do
3      if $\mathbf{T}(v \le d) \in \mathbf{B}$ for some $d \in D(v)$ then
4          $ub \leftarrow \min\{d \mid d \in D(v), \mathbf{T}(v \le d) \in \mathbf{B}\}$
5          $\Sigma \leftarrow \Sigma \cup \{\{\mathbf{T}(v \le ub), \mathbf{F}(v \le x)\} \mid x > ub, (v \le x) \in \mathcal{B}, \mathbf{T}(v \le x) \notin \mathbf{B}\}$
6      if $\mathbf{F}(v \le d) \in \mathbf{B}$ for some $d \in D(v)$ then
7          $lb \leftarrow \max\{d \mid d \in D(v), \mathbf{F}(v \le d) \in \mathbf{B}\}$
8          $\Sigma \leftarrow \Sigma \cup \{\{\mathbf{T}(v \le x), \mathbf{F}(v \le lb)\} \mid x < lb, (v \le x) \in \mathcal{B}, \mathbf{F}(v \le x) \notin \mathbf{B}\}$
9  end
10 if $\Sigma \ne \emptyset$ then return $\Sigma$
11 for $c \in \mathcal{C} \setminus \mathcal{C}'$ do
12     if $\mathbf{T}c \in \mathbf{B}$ then
13         $\Sigma \leftarrow \Sigma \cup$ PropagateBounds$(\mathbf{B}, \mathbf{T}c \Rightarrow \gamma(c))$
14     else if $\mathbf{F}c \in \mathbf{B}$ then
15         $\Sigma \leftarrow \Sigma \cup$ PropagateBounds$(\mathbf{B}, \mathbf{F}c \Rightarrow \overline{\gamma(c)})$
16     else
17         $\Sigma \leftarrow \Sigma \cup$ PropagateReification$(\mathbf{B}, \mathbf{T}c \Rightarrow \gamma(c))$
18         $\Sigma \leftarrow \Sigma \cup$ PropagateReification$(\mathbf{B}, \mathbf{F}c \Rightarrow \overline{\gamma(c)})$
19     if $\Sigma \ne \emptyset$ then return $\Sigma$
20 end
21 return $\emptyset$
```

Constraint Propagation. To generate the nogoods in $\Psi(\mathcal{C} \setminus \mathcal{C}')$ lazily, Algorithm 4 uses functions PropagateBounds and PropagateReification for half-reified constraint $\mathbf{T}c \Rightarrow \gamma(c)$ and $\mathbf{F}c \Rightarrow \overline{\gamma(c)}$, respectively, for each $c \in \mathcal{C} \setminus \mathcal{C}'$. In the respective algorithms 5 and 6, we consider four different strengths of propagation, denoted by ps. A strength of 1 means that our propagator only produces conflicting nogoods. A strength of 2 means that it additionally checks if yet undecided constraints became true. Strength 3 furthermore adds unit nogoods that also propagate the bounds of the variables in a constraint if it is already decided to be true, whereas strength 4 also computes optimized nogoods for yet undecided constraints. The propagators are conflict optimal and for strength 4 even inference optimal. We divided our propagator into two algorithms, handling reified constraints of form $\sigma \Rightarrow a_1v_1 + \cdots + a_nv_n \le b$. Algorithm 5 is only called if $\sigma \in \mathbf{B}$. Whenever σ is true, we check whether the constraint $a_1v_1 + \cdots + a_nv_n \le b$ can be falsified. If it can never be falsified, e.g. the sum of the current upper bounds already satisfies the constraint in Line 1, we are done. If we only have propagation strength 1 or 2, we check in Line 3 whether the sum of the current lower bounds is already above the bound b. In this case, we simply return the current lower bounds of the views as a nogood, since the constraint is already violated. For example, take the constraint $\sigma \Rightarrow x + y \le 9$ with $D(x) = D(y) = \{1, \ldots, 15\}$ and the current lower and upper bounds $lb_{\mathbf{B}}(x) = 7$, $ub_{\mathbf{B}}(x) = 10$, $lb_{\mathbf{B}}(y) = 5$, and $ub_{\mathbf{B}}(y) = 12$. The sum of the lower bounds $7 + 5$ is greater than 9, and so the constraint is violated. Therefore, we add the nogood

Algorithm 5: PROPAGATEBOUNDS

```
Global : An integer $ps$
Input  : A Boolean assignment $\mathbf{B}$ and a half-reified constraint $\sigma \Rightarrow a_1v_1 + \cdots + a_nv_n \leq b$
Output : A set of nogoods
1  $\Sigma \leftarrow \emptyset$                                   // An empty set of nogoods
2  if $\sum_{j=1}^{n} ub_{\mathbf{B}}(a_jv_j) \leq b$ then return $\emptyset$
3  if $ps \leq 2$ then
4      if $\sum_{j=1}^{n} lb_{\mathbf{B}}(a_jv_j) > b$ then
5          $\Sigma \leftarrow \{\{\sigma\} \cup \{(a_jv_j \geq lb_{\mathbf{B}}(a_jv_j))^{\ddagger} \mid 1 \leq j \leq n\}\}$
6      return $\Sigma$
7  for $i = 1..n$ do
8      $cur \leftarrow b - \sum_{j=1,j\neq i}^{n} lb_{\mathbf{B}}(a_jv_j)$
9      if $cur < ub_{\mathbf{B}}(a_iv_i)$ then
10         $\Sigma \leftarrow \Sigma \cup \{\{\sigma, (a_iv_i > cur)^{\ddagger}\} \cup \{(a_jv_j \geq lb_{\mathbf{B}}(a_jv_j))^{\ddagger} \mid 1 \leq j \leq n, j \neq i\}\}$
11         if $cur < lb_{\mathbf{B}}(a_iv_i)$ then return $\Sigma$
12 end
13 return $\Sigma$
```

$\{\sigma, (x \geq 7)^{\ddagger}, (y \geq 5)^{\ddagger}\}$. If the propagation strength is greater than 2 (Lines 6-10), we try to find new upper bounds for the views of the constraint. For this purpose, cur represents the maximal value that a_iv_i can take without violating the constraint. All other views a_jv_j ($j \neq i$) contribute at least their current lower bound to the sum. In our example, this means that $cur = 9 - 5 = 4$. If this value is less than the current upper bound of a_iv_i (Line 8), we create a nogood that allows us to propagate the new upper bound. In the example, this is $\{\sigma, (x > 4)^{\ddagger}, (y \geq 5)^{\ddagger}\}$. Compared to the nogood that was created in Line 4, this nogood is stronger as the required minimum for x is lower. If cur is even below the current lower bound of a_iv_i, we have a conflict and stop eagerly (Line 11). Since $cur = 4$ and $lb_{\mathbf{B}}(x) = 7$, this is the case in our example. This algorithm has linear complexity $O(n)$, but since we consider domains/images with holes, finding the literal $(a_iv_i > cur)^{\ddagger}$ is actually $O(log(|D(v_i)|))$ which raises the overall complexity for propagation strength greater than 2.

Algorithm 6 is only called if neither $\sigma \in \mathbf{B}$ nor $\overline{\sigma} \in \mathbf{B}$, e.g. whenever σ is unknown, and propagation strength is at least 2 (Line 1). If the sum of all current lower bounds of the left hand side is greater than b (Lines 2 and 3), the constraint can never become satisfied. Given a propagation strength below 4, we simply create a nogood based on the current lower bounds. In our example, this is the same nogood $\{\sigma, (x \geq 7)^{\ddagger}, (y \geq 5)^{\ddagger}\}$ generated in Algorithm 5. If the propagation strength is 4 (Lines 8-13), we try to find a sum of the views that is minimally greater than b. In our example, we start with a lower bound $low = 12$. By subtracting $lb_{\mathbf{B}}(x)$, we get $low' = 5$. This leaves us with $cur = next(9 - 5, x) = 5$ adding $(x \geq 5)^{\ddagger}$ to our nogood. In the second iteration, we now have to find a sufficient lower bound for y that violates the constraint. We see that this value is 5, adding $(y \geq 5)^{\ddagger}$ to δ in Line 11 resulting in the nogood $\{\sigma, (x \geq 5)^{\ddagger}, (y \geq 5)^{\ddagger}\}$. Again, the complexity of the refined search is higher but also the produced nogoods are stronger. Note that as an optimization, PROPAGATEBOUND and PROPAGATEREIFICATION are only called if the bounds of the variables of the constraints have changed. The propagation strength is set using the option `--prop-strength`.

Algorithm 6: PROPAGATEREIFICATION

```
   Global : An integer $ps$
   Input  : A Boolean assignment $\mathbf{B}$ and a half-reified constraint $\sigma \Rightarrow a_1v_1 + \cdots + a_nv_n \leq b$
   Output : A set of nogoods
1  if $ps = 1$ then return $\emptyset$
2  $low \leftarrow \sum_{j=1}^{n} lb_{\mathbf{B}}(a_jv_j)$
3  if $low > b$ then
4      $\delta \leftarrow \{\sigma\}$
5      if $ps < 4$ then
6          $\delta \leftarrow \delta \cup \{(a_jv_j \geq lb_{\mathbf{B}}(a_jv_j))^{\ddagger} \mid 1 \leq j \leq n\}$
7      else
8          for $j \in 1..n$ do
9              $low' \leftarrow low - lb_{\mathbf{B}}(a_jv_j)$
10             $cur \leftarrow next(b - low', a_jv_j)$
11             $\delta \leftarrow \delta \cup \{(a_jv_j \geq cur)^{\ddagger}\}$
12             $low \leftarrow low' + cur$
13         end
14     return $\{\delta\}$
```

3.4 Distinguished Features

After presenting the algorithmic framework of *clingcon* 3, we now describe some of its specific features. Many of them aim at reducing the size of domains and the number of variables, while others address special functionalities, like global constraints or multi-objective optimization over integer variables, respectively.

When we refer in the following to the truth value of atoms, we consider a partial assignment obtained by propagation and/or preprocessing.

Views. A view $av+b$ can be represented with the same set of order atoms as its variable v (Thibaut and Stuckey 2009). Consider the view $-5v + 7$ together with the domain $D(v) = \{1, 2, 3, 4, 5\}$. We show how the order atoms of v are used to encode constraints over the view in *clingcon*. The view $-5v+7$ has the following values in its image: $img(-5v+7) = \{-18, -13, -8, -3, 2\}$. The order literals for $\{(v \leq x)^{\ddagger} \mid x \in D(v)\}$ and $\{(-5v+7 \leq x)^{\ddagger} \mid x \in img(-5v+7)\}$ are given in Table 1. We see that the set of order atoms used for these literals is the same. By allowing views in-

Expression	Image	Order Literals				
v	$\{1, 2, 3, 4, 5\}$	$\mathbf{T}v \leq 1$	$\mathbf{T}v \leq 2$	$\mathbf{T}v \leq 3$	$\mathbf{T}v \leq 4$	$\mathbf{T}\emptyset$
$-5v + 7$	$\{-18, -13, -8, -3, 2\}$	$\mathbf{F}v \leq 4$	$\mathbf{F}v \leq 3$	$\mathbf{F}v \leq 2$	$\mathbf{F}v \leq 1$	$\mathbf{T}\emptyset$

Table 1: *Order literals of different views of one variable.*

stead of variables, we avoid introducing new variables (for views). In fact, neither the XCSP (Roussel and Lecoutre 2009) nor the *flatzinc*[13] format allow for using views in global constraints. For instance, a distinct constraint over the set of views $\{1000v_1, 1000v_2, 1000v_3, 1000v_4, 1000v_5\}$

[13] http://www.minizinc.org/downloads/doc-1.3/flatzinc-spec.pdf

translates into the same nogoods as a distinct constraint over $\{v_1, v_2, v_3, v_4, v_5\}$. Due to the restriction to use variables, according solvers like *sugar* (Tamura et al. 2009) introduce auxiliary variables $v'_i = 1000v_i$ for $1 \leq i \leq 5$. If $D(v_i) = \{1, \ldots, 10\}$, bound propagation yields the domains $D(v'_i) = \{1000{\cdot}1, \ldots, 1000{\cdot}10\} = \{1000, \ldots, 10000\}$.[14] Furthermore, around 220000 nogoods for the equality constraints are created. By handling views directly, we avoid introducing these auxiliary variables and constraints in *clingcon*.

The same holds for minimize statements. Views on variables such as $3 * v_2$ or $-v_3$ allow for weighting variables during minimization as well as maximization, without the need of introducing auxiliary variables and additional constraints.

Non-Contiguous Integer Domains. We represent domains of variables (and images of views) as sorted lists of ranges like $[1..3, 7..12, 39..42]$ instead of single ranges like $[1..42]$. This has the advantage that we can represent domains with holes directly, without any additional constraints. Introducing order atoms for such a non-contiguous domain produces fewer atoms (12 in this example) than for a domain only represented with two bounds (41). A drawback of this representation is that the lookup for a certain value d in the domain becomes logarithmic, as we rely upon binary search in the list of ranges. This is frequently done in Algorithms 1, 5 and 6 whenever a calculated value d leads to searching for a literal $(v \leq d)^{\ddagger}$.

Equality Processing. To minimize the number of atoms and nogoods that have to be created during a translation or solving process, we need to reduce the number of integer variables. To accomplish this, we consider the equalities in a CSP that include only two integer variables, and replace all occurrences of the first variable with a view on the second variable in all other constraints. Consider a constraint logic program P over $\mathcal{A}$, $\mathcal{C}$ associated with $(\mathcal{V}, D, C)$. For each element $\gamma(\sigma) \in C$ of the form $ax + c_1 = by + c_2$ (or $ax + c_1 \neq by + c_2$) where σ is true (false), a, b, c_1, c_2 are integers, and $x, y \in \mathcal{V}$, we successively replace constraints in C. For this, we normalize the constraint $\gamma(\sigma)$ to $ax = by + c$ where x is lexicographically smaller than y and multiply all constraints in C containing variable y with b and replace $by + c$ by ax in them. The domain of x is made domain consistent such that $ad \in img(by + c)$ holds for all $d \in D(x)$. Afterwards, we remove $\gamma(\sigma)$ from C and y from $\mathcal{V}$. Note that by replacing variables, new equalities may arise, which we process until a fixpoint is reached.

For illustration, consider the following set C of constraints.

$$a = 2b \tag{16}$$
$$b = 2c \tag{17}$$
$$c = 2d \tag{18}$$
$$d = 2e \tag{19}$$
$$e = 2f \tag{20}$$
$$a + 14d - 3f + b \leq -g \tag{21}$$

And assume that the constraint literals associated with the first 5 constraints are true. Furthermore, let $D = \{D(x) = \{-2^{12}, \ldots, 2^{12} \mid x \in \{a, b, c, d, e, f, g\}\}$. Without any simplification, we have 7 variables, all with a domain size of roughly 8000. By simply translating these constraints, we

[14] As done in the *sugar* system.

would create around 120000 order atoms and 118 million nogoods. Let us show how equality processing allows us to significantly reduce these numbers in our example. To begin with, we multiply the constraint in (21), viz. $a + 14d - 3f + b \leq -g$, with 2 and replace $-6f$ with $-3e$ using the constraint in (20). This yields $2a + 28d - 3e + 2b \leq -2g$. Also, (20) allows us to restrict the domain of e to $D(e) = \{-2^{11}, \dots, 2^{11}\}$. We then remove $e = 2f$ from the set of constraints and f from the set of variables. We repeat this procedure for all other equalities. To replace e, we again multiply the obtained constraint by 2, yielding $4a + 56d - 6e + 4b \leq -4g$, and replace $6e$ with $3d$ using (19). This results in $4a + 53d + 4b \leq -4g$. Again, we remove $d = 2e$ and variable e, and obtain $D(d) = \{-2^{10}, \dots, 2^{10}\}$. Using (18), we multiply by 2 and replace $106d$ with $53c$ which leads to the constraint $8a + 53c + 8b \leq -8g$. To remove c, the constraint in (17) is used to replace $106c$ with $53b$ resulting in $16a + 69b \leq -16g$. In the last step, we apply (16) to get $32a + 69a \leq -32g$ which simplifies to $101a \leq -32g$. As a result, the overall set of constraints is thus reduced to a single constraint $101a \leq -32g$. This constraint uses only two variables with domains $D(a) = \{-2^7, \dots, 2^7\}$ and $D(g) = \{-2^{12}, \dots, 2^{12}\}$. All other constraints and variables have been removed. To translate this constraint, we need 9265 order atoms and 268 nogoods.

Our approach to equivalence processing is inspired by Boolean *Equi-propagation* (Metodi et al. 2013), which directly replaces the order atoms of one variable with the other. Directly using integer variables, without considering the order literal representation, allows us to use this technique also in the context of lazy variable generation. Here, it reduces the number of variables, which leads to shorter constraints, which ultimately reduces the number of nogoods in the translation process.

Equality preprocessing is done once in *clingcon*, before the actual solving starts and can be controlled using the command line option `--equality-processing`.

Distinct Translation. *clingcon* features two alternatives for translating global distinct constraints. Assume that constraint atom c represents a distinct constraint over a set $\{v_1, \dots, v_n\}$. Since we represent distinct constraints in terms of rules and other linear constraints, this constraint atom becomes a regular atom and is used in the head of rules.

The first method to handle this constraint uses a quadratic number of new, regular atoms $neq(v_i, v_j)$ for all $1 \leq i < j \leq n$ together with the rules

$$\begin{aligned} neq(v_i, v_j) &\leftarrow (v_i - v_j \leq 1) \\ neq(v_i, v_j) &\leftarrow (v_j - v_i \leq 1) \end{aligned}$$

to represent that two variables are unequal. By adding the following rule to the program

$$\begin{aligned} c \leftarrow neq(v_1, v_2), neq(v_1, v_3), \dots, neq(v_1, v_n),& \\ neq(v_2, v_3), \dots, neq(v_2, v_n),& \\ \ddots \quad \vdots& \\ neq(v_{n-1}, v_n)& \end{aligned}$$

clingcon ensures that c is only true if all variables are distinct from each other.

The second alternative uses a so-called *direct encoding* (Walsh 2000). For each value $d \in \bigcup_{i=1}^{n} img(v_i)$, we ensure that at most one variable from $\{v_1, \dots, v_n\}$ takes this value. Therefore, we introduce regular atoms of form $eq(v_i, d)$ for all these variables together with the rule

$$eq(v_i, d) \leftarrow (v_i \leq d), (-v_i \leq -d) \tag{22}$$

representing that $v_i = d$. Furthermore, we add a cardinality constraint (Simons et al. 2002) for each value d to the effect that no two or more variables may have the same value, viz.

$$c' \leftarrow 2 \; \{eq(v_1, d), \dots, eq(v_n, d)\}$$

The new regular atom c' is true if two or more variables have the same value d. If this is not the case, the distinct constraint atom holds via the rule:

$$c \leftarrow {\sim} c'$$

We reuse the direct encoding atoms $eq(v_i, d)$ for other distinct constraints. Note that introducing all direct encoding atoms also involves the creation of corresponding order atoms before the solving process. So no variable from a distinct constraint can be created lazily. This is also the reason why this option is not enabled in *clingcon* by default and distinct constraints are translated using inequalities. The use of the direct encoding along with cardinality constraints is enabled with the option `--distinct-to-card`.

Pigeon Hole Constraints. To enhance the propagation strength when translating distinct constraints in *clingcon*, we add rules for the lower and upper bounds. Consider the constraint atom c for a distinct constraint over $\{v_1, \dots, v_n\}$ and let $U = \bigcup_{i=0}^{n} img(v_i)$, l be the nth smallest element in U, and u be the nth greatest element in U. We add the rules:

$$\begin{aligned} &\leftarrow c, (v_1 > u), \dots, (v_n > u) \\ &\leftarrow c, (v_1 < l), \dots, (v_n < l) \end{aligned}$$

where as before, c is treated as regular atom.

So given a distinct constraint over $\{v_1, v_2, v_3\}$ with $D(v_i) = \{1, \dots, 10\}$ for $1 \le i \le 3$ we add the rules

$$\begin{aligned} &\leftarrow c, (v_1 > 8), (v_2 > 8), (v_3 > 8) \\ &\leftarrow c, (v_1 < 3), (v_2 < 3), (v_3 < 3) \end{aligned}$$

This forbids all variables to have a value greater than eight or to have a value less than three. This feature only causes a constant overhead in the number of rules. It can be controlled using the option `--distinct-pigeon`.

Permutation Constraints. A distinct constraint over $\{v_1, \dots, v_n\}$ where $U = \bigcup_{i=1}^{n} img(v_i)$ and $|U| = n$ induces a permutation on the variables. Let c be the constraint atom representing this global constraint. In this special case, we can add the rules

$$\leftarrow c, {\sim} eq(v_1, d), \dots, {\sim} eq(v_n, d) \qquad \text{for all } d \in U.$$

These rules enforce that each value is taken at least once.

For example, given a distinct constraint over $\{v_1, v_2, v_3\}$ with $D(v_i) = \{1, \dots, 3\}$ for $1 \le i \le 3$ we add the rules

$$\begin{aligned} &\leftarrow c, {\sim} eq(v_1, 1), {\sim} eq(v_2, 1), {\sim} eq(v_3, 1) \\ &\leftarrow c, {\sim} eq(v_1, 2), {\sim} eq(v_2, 2), {\sim} eq(v_3, 2) \\ &\leftarrow c, {\sim} eq(v_1, 3), {\sim} eq(v_2, 3), {\sim} eq(v_3, 3) \end{aligned}$$

This feature introduces direct encoding atoms along with the respective rules and order atoms in (22). Since these atoms cannot be treated lazily, this feature is disabled by default but can be controlled using the option `--distinct-permutation`.

Sorting. Sorting constraints by descending coefficients is known to avoid redundant nogoods in the translation process (Tamura et al. 2013). Also, systems like *sugar* sort constraints by smallest domain first, and when tied, with largest coefficient. *clingcon* can either sort by coefficient or domain size first, in decreasing or increasing order. The option `--sort-coefficient` controls the sorting of the constraints.

Splitting Constraints. Considering that directly translating a linear constraint $a_1v_1+\dots+a_nv_n \leq b$ with the order encoding leads to an exponential number of nogoods, we split long constraints into shorter ones by introducing new variables. Thereby we adapt the heuristics of *sugar*. We only split a constraint if the number of variables is greater than α and if its translation produces more than β nogoods. If both conditions hold, we recursively split a constraint into α parts. The new constraints have the form $a_kv_k + \dots + a_lv_l = v_l^k$ where $1 \leq k \leq l \leq n$. α and β are freely configurable. By default, splitting is disabled in *clingcon*, but α and β can be changed with options `--split-size` and `--max-nogoods-size`.

Symmetry Breaking. When splitting a constraint like $a_1v_1 + a_2v_2 + a_3v_3 \leq b$, we get the constraints $a_1v_1 + a_1v_2 = v_2^1$ and $v_2^1 + a_3v_3 \leq b$. Equations like $a_1v_1 + a_1v_2 = v_2^1$ are represented as conjunctions of $a_1v_1 + a_1v_2 \leq v_2^1$ and $a_1v_1 + a_1v_2 \geq v_2^1$. By dropping the latter inequality, we obtain an equi-satisfiable set of constraints being smaller than before but admitting more (symmetric) solutions, as v_2^1 freely varies. Symmetry breaking should therefore be enabled if one wants to enumerate all solutions without duplicates. This form of symmetry breaking is usually skipped in SAT-based CSP solvers like *sugar*. This option is set via `--break-symmetries`.

Domain Propagation. To create the domain of variables like v_n^1 in the aforementioned constraints of form $a_1v_1 + \dots + a_nv_n = v_n^1$, we may use bound propagation. For example, the constraint $42x + 1337z = y$ where $D(x) = D(z) = \{0, 1\}$ results in the domain $D(y) = \{42 \cdot lb(x) + 1137 \cdot lb(z), \dots, 42 \cdot ub(x) + 1137 \cdot ub(z)\} = \{0, \dots, 1379\}$. Using domain propagation instead leads to the much smaller domain $D(y) = \{42d_x + 1137d_z \mid d_x \in D(x), d_z \in D(z)\} = \{0, 42, 1337, 1379\}$. However, we restrict domain propagation to preprocessing by default, as it has an exponential runtime. *clingcon* allows for controlling domain propagation by setting a threshold on the domain size; this is set by option `--domain-size`.

Translate Constraints. Following a two-fold approach, *clingcon* is able to translate some constraints while leaving others to constraint propagators as shown in Section 3.3. *clingcon* provides the option `--translate-constraints=m` to decide which constraints to translate or not. The translation depends on the estimated number of nogoods $\prod_{i=1}^{n-1} |D(v_i)|$ that Algorithm 1 produces for a constraint $a_1v_1 + \dots + a_nv_n \leq b$. If this number is below the threshold `m`, *clingcon* translates the constraint. Also all order atoms used in these nogoods are created.

Redundant Nogood Check. A nogood δ is said to be stronger than a nogood δ', iff for all literals $(v > d)^\ddagger \in \delta$, there exists a literal $(v > d')^\ddagger \in \delta'$ such that $d \leq d'$ and v is a view. Whenever a nogood is created in Line 7 in Algorithm 1, we compare it to the previously created one. If

logic program P	constraint stable models of P
$\{a\}$	$\{(\{a,(x>7)\}, \quad \{x \mapsto d\}) \mid d \in \{8,\dots,10\}\}\cup$
$\leftarrow a, \sim(x>7)$	$\{(\{(x>7)\}, \quad \{x \mapsto d\}) \mid d \in \{8,\dots,10\}\}\cup$
	$\{(\emptyset, \quad \{x \mapsto d\}) \mid d \in \{1,\dots,\ 7\}\}$
$\{a\}$	$\{(\{a,(x>7)'\}, \quad \{x \mapsto d\}) \mid d \in \{8,\dots,10\}\}\cup$
$\leftarrow a, \sim(x>7)'$	$\{(\emptyset, \quad \{x \mapsto d\}) \mid d \in \{1,\dots,10\}\}$
$\leftarrow \sim a, (x>7)'$	

Table 2: *Constraint logic programs using reified* $\mathbf{T}(x > 7) \Leftrightarrow x > 7$ *and half-reified* $\mathbf{T}(x > 7)' \Rightarrow x > 7$ *constraints.*

one of them is stronger, we only keep the stronger one, otherwise, we keep both. This feature allows *clingcon* to remove some redundant nogoods during the translation process. It is especially useful if the constraints are not sorted by descending coefficients. The check just adds constant overhead to the translation process but avoids creating a significant amount of nogoods. For instance, translating the famous *send more money* problem results in 628 nogoods among which 327 are redundant, when using `--split-size=3`. This feature can be triggered using option `--redundant-nogood-check`.

Don't Care Propagation. Suppose we want to express that $(x > 7)$ should hold whenever a holds; otherwise we do not care whether $(x > 7)$ holds or not. A corresponding constraint logic program is given in the first row of Table 2 together with its constraint stable models. In the standard case for CASP, the constraint atom is reified with its constraint via $\mathbf{T}(x > 7) \Leftrightarrow x > 7$. In the case that a is true, the constraint atom $(x > 7)$ has to be true. The reification ensures that x is greater than 7, leading to three different assignments $\{\{x \mapsto d\} \mid d \in \{8,\dots,10\}\}$ for variable x. In the case that a is false, the constraint atom $(x > 7)$ can either be true or false. The first case results in the same three assignments, while the latter corresponds to seven others, viz. $\{\{x \mapsto d\} \mid d \in \{1,\dots,7\}\}$, as the reification imposes that the constraint $x > 7$ does not hold, basically enforcing $x \leq 7$. We note that in case a is false, the constraint imposed on x is either $x > 7$ or $x \leq 7$. This means that there is actually no restriction on the assignment of x. We exploit this observation by replacing $(x > 7)$ with a new constraint atom $(x > 7)'$ and adding the rule $\leftarrow \sim a, (x > 7)'$. The idea is that atom $(x > 7)'$ imposes $(x > 7)$ as a half-reified constraint, meaning that x is enforced to be greater than 7 only if the constraint atom $(x > 7)'$ is true, i.e. $\mathbf{T}(x > 7)' \Rightarrow x > 7$. We obtain exactly the same stable models in terms of the regular atoms and integer variable assignments, as depicted in the second row of Table 2. The difference between these two programs lies in the assignment of the constraint atoms. The additional rule $\leftarrow \sim a, (x > 7)'$ ensures that the constraint atom $(x > 7)'$ is false, whenever a is false. Since we connect the constraint atom with its constraint using a half-reified constraint, this constraint has no effect on the assignment of x, resulting in $\{\{x \mapsto d\} \mid d \in \{1,\dots,10\}\}$. Although the number of constraint stable models stays the same, the number of different Boolean assignments is reduced.

This technique is called *Don't Care Propagation* (Thiffault et al. 2004). All constraint atoms that only occur in integrity constraints and only positively (negatively) in the whole program are don't care atoms. *clingcon* fixes the truth value of don't care atoms to false (true), if all integrity constraints containing the atom have at least one literal being false under the current assignment. Don't care propagation can be useful in SAT, but it has even more potential to be helpful in CASP/SMT, since we not only reduce the search space but also the theory propagator has to handle only one half-reified constraint per don't care atom. This means only half of the inferences have to be checked. This technique is not specifically designed for CSP but it can also be used for other theories. Don't care propagation is controlled using the option `--dont-care-propagation`.

Order Atom Generation. When translating a constraint, all order atoms for all its integer variables must be available. By not translating all constraints, we also do not need to create all order atoms. Some of them can be created on the fly during propagation. With this in mind, it might still be useful to create a certain number of order atoms per variable in a preprocessing step. *clingcon* can create n atoms evenly spread among the domain values of a variable v. So if we have a domain $D(v) = \{1, \dots, 10, 90, \dots, 100\}$ and create four order atoms we use $(v \leq 3), (v \leq 8), (v \leq 92)$ and $(v \leq 97)$. These order atoms allow the solver to split the domain during the search. Option `--min-lits-per-var=n` adds at least $min(n, |D(v)| - 1)$ order atoms for each variable v.

Explicit Binary Order Nogoods. Some order atoms are created before solving. Therefore, it can also be beneficial to create a subset of the order nogoods $\Phi'(\mathcal{V}, D)$ in advance, as shown in Corollary 2.1. Given that we created the set of order atoms $\{(v \leq x_1), \dots, (v \leq x_n)\}$ for a variable $v \in \mathcal{V}$ where $x_i < x_{i+1}$ for $1 \leq i \leq n$, the explicit order nogoods

$$\{\{\mathbf{T}v \leq x_1, \mathbf{F}v \leq x_2\}, \dots, \{\mathbf{T}v \leq x_{n-1}, \mathbf{F}v \leq x_n\}\}$$

can also be created. To introduce these binary order nogoods for all order atoms that have been created before the solving process, the option `--explicit-binary-order` can be used.

Objective Functions. We support multi-objective optimization on sets of views. For all views $av + c$ subject to minimization, we use the signed order literals $(av + c \geq d)^{\ddagger}$ with weight

$$\begin{cases} d - prev(d, av + c) & \text{if } d > lb(av + c) \\ d & \text{if } d = lb(av + c) \end{cases}$$

for all values $d \in img(av + c)$ in an ASP minimize statement. This minimizes the total sum of the set of views. By using native ASP minimize statements, *clingcon* reuses *clasp*'s branch and bound and unsatisfiable core based techniques (Andres et al. 2012). For instance, for minimizing $3x$ where $D(x) = \{1, 3, 7\}$, we have the following weighted literals in the (internal) ASP minimize statement $(3x \geq 3)^{\ddagger} = 3$, $(3x \geq 9)^{\ddagger} = 6$, and $(3x \geq 21)^{\ddagger} = 12$. In terms of ASP-pseudo-code this amounts to a minimize statement of form $\#minimize\{6 : {\sim} x \leq 1; 12 : {\sim} x \leq 3\}$ although order literals are not part of the input language. $(3x \geq 3)^{\ddagger}$ evaluates to true, while $(3x \geq 9)^{\ddagger}$ and $(3x \geq 21)^{\ddagger}$ can be expressed via order literals as ${\sim}(x \leq 1)$ and ${\sim}(x \leq 3)$, respectively.

Flattening Objective Functions. Minimizing the value of an integer variable y that is included in a constraint $\gamma(\sigma) = a_1v_1 + \dots + a_nv_n = y$ where σ is true, is equivalent to minimizing the value of $a_1v_1 + \dots + a_nv_n$. Directly using the views a_iv_i strengthens the nogoods used to represent the minimize statement. The constraint $a_1v_1 + \dots + a_nv_n = y$ can be removed if y is not used anywhere else.[15] In fact, this pattern occurs quite often in our *minizinc* benchmark set. Replacing variable y with its constituents $a_1v_1 + \dots + a_nv_n$ can be controlled with the option `--flatten-optimization`.

[15] We keep the constraint to be able to correctly print y in a solution.

Reduced Nogood Learning. Whenever CSPPROPAGATION in Algorithm 3 and 4 derives a nogood, it is possible to not add it to the store of learned nogoods ∇ but rather keep it implicit and only add it if it is really needed for conflict analysis. The internal interface of *clasp* supports such a behavior. While the learned nogoods ∇ improve the strength of unit propagation, too many nogoods decrease its performance. Therefore, lazily adding these nogoods when they are actually needed can improve unit propagation. To disable the storage of nogoods and handle them implicitly, *clingcon* provides option `--learn-nogoods`.

3.5 Multi-Shot CASP Solving

As mentioned, a major design objective of *clingcon* 3 is to transfer *clingo*'s functionalities to CASP solving. A central role in this is played by multi-shot solving (Gebser et al. 2014; Gebser et al. 2015) because it allows for casting manifold reasoning modes. More precisely, multi-shot solving is about solving continuously changing logic programs in an operative way. This can be controlled via reactive procedures that loop on solving while reacting, for instance, to outside changes or previous solving results. These reactions may entail the addition or retraction of rules that the operative approach can accommodate by leaving the unaffected program parts intact within the solver. This avoids re-grounding and benefits from heuristic scores and nogoods learned over time.

To extend multi-shot solving to CASP, our propagators allow for adding and deleting constraints in order to capture evolving CSPs. Evolving constraint logic programs can be extremely useful in dynamic applications, for example, to:

- add new resources in a planning domain,
- set the value of an observed variable measured using sensors,
- add restrictions to reduce the capacity of containers, or
- increase their capacity depending on other systems like weather forecast etc.

The presented propagators provide means for all these issues. New resources can be added using additional constraint variables and domains. Values can be limited by adding constraints and rules to the constraint logic program. Due to our monotone treatment of CSPs in CASP, it is always possible to add new constraint atoms. Since they are not allowed to occur in rule heads they to not interfere with the completion of the logic program. Hence, we can combine (and therefore extend) two constraint logic programs under exactly the same restrictions that apply to normal logic programs (cf. (Gebser et al. 2014)).

While confining variables is easy, accomplished by adding constraints on those variables, increasing their capacity is addressed via lazy variable generation. That is, we start with a virtually maximum domain that is restrained by retractable constraints. The domain is then increased by relaxing these constraints. Importantly, the order atoms representing the active domain are only generated when needed. This avoids introducing a large amount of atoms, especially in the non-active area of the domain. As an example, consider the variable x and its domain $D(x) = \{1, \dots, 10^9\}$ having one billion elements. By adding the constraint $x \leq 10$, only the first 10 values are valid assignments. After retracting $x \leq 10$ and adding $x \leq 20$, only the first 20 values constitute the search space. Since order atoms are only introduced in the actual search space, no atoms are introduced for the huge amount ($10^9 - 20$) of other values. Using this technique, CASP can deal with increasing domains within reasonable space.

For illustration, let us consider the well-known n-queens puzzle for demonstrating how to incrementally add new constraints and constraint variables to a constraint logic program and how to remove constraints from it. To illustrate how seamlessly *clingcon* integrates CASP and multi-shot solving, we apply *clingo*'s exemplary Python script for incremental solving to model different incremental versions of the n-queens puzzle in CASP. Multi-shot solving in *clingo* relies on two directives (Gebser et al. 2014), the `#program` directive for regrouping rules and the `#external` directive for declaring atoms as being external to the program at hand. The truth value of such external atoms is set via *clingo*'s API. *Clingo*'s incremental solving procedure is provided in Python and loops over increasing integers until a stop criterion is met. It presupposes three groups of rules declared via `#program` directives. At step 0 the programs named `base` and `check(n)` are ground and solved for `n` $=$ 0. Then, in turn programs `check(n)` and `step(n)` are added for `n` $>$ 0 and the obtained program is grounded and solved. Other names and components are definable by appropriate changes to the Python program. Stop criteria can be the satisfiability or unsatisfiability of the respective program at each iteration. In addition, at each step `n` an external atom `query(n)` is introduced; it is set to true for the current iteration `n` and false for all previous instances with smaller integers than `n`. Although we reproduce the exemplary Python program from *clingo*'s example pool in Listing 4, we must refer the reader to (Gebser et al. 2014) for further details.

```
#script (python)

import clingo

def get(val, default):
    return val if val != None else default

def main(prg):
    imin   = get(prg.get_const("imin"), clingo.Number(0))
    imax   = prg.get_const("imax")
    istop  = get(prg.get_const("istop"), clingo.String("SAT"))

    step, ret = 0, None
    while ((imax is None or step < imax.number) and
           (step == 0 or step < imin.number or (
              (istop.string == "SAT"     and not ret.satisfiable) or
              (istop.string == "UNSAT"   and not ret.unsatisfiable) or
              (istop.string == "UNKNOWN" and not ret.unknown)))):
        parts = []
        parts.append(("check", [step]))
        if step > 0:
            prg.release_external(clingo.Function("query", [step-1]))
            parts.append(("step", [step]))
            prg.cleanup()
        else:
            parts.append(("base", []))
        prg.ground(parts)
        prg.assign_external(clingo.Function("query", [step]), True)
        ret, step = prg.solve(), step+1
#end.

#program check(t).
#external query(t).
```

Listing 4: *Incremental mode of* Clingo

The CASP encoding of the incremental n-queens puzzle in Listing 5 demonstrates the addition and removal of constraints and also shows how variable domains are dynamically increased. As usual, the goal is to put n queens on an $n \times n$ board such that no two queens threaten each other. Here, however, this is done for an increasing sequence of integers n such that the queens puzzle for

```
 1  #include "incmode.lp".
 2  #include "csp.lp".
 3  #show.

 5  #program step(n).

 7  pos(n).

 9  &sum{ q(n) } > 0.
10  &sum{ q(X) } <= n :- pos(X), query(n).

12  &distinct{ q(X)      : pos(X) }.

14  &distinct{ q(X)+X-1  : pos(X) }.
15  &distinct{ q(X)-X+1  : pos(X) }.

17  &show{ q(n) }.
```

Listing 5: *Incremental n-queens encoding Q_1 (`incqueens.lp`)*

n is obtained by extending the one for $n-1$. While the first line of Listing 5 includes the Python program in Listing 4, the next one includes the grammar from Listing 1. Line 3 suppresses the output of regular atoms. The remaining encoding makes use of two features of *clingo*'s exemplary incremental solving procedure, viz. subsequently grounding and solving rules regrouped under program `step(n)` and the external atom `query(n)`.[16] In Listing 5, all rules in lines 7-17 are regrouped under subprogram `step(n)`. The Python program in Listing 4 makes *clingcon* in turn solve the empty program, then program `step(1)`, then program `step(1)` and `step(2)` together, then both former programs and `step(3)`, etc. This is done by keeping the previous programs in the solver and by replacing parameter `n` in lines 7-17 with the respective integer when grounding the added subprogram. Thus, at each step `n` a fact '`pos(n).`' is added to the solver (cf. Line 7). The heads of Line 9 and 10 represent the linear constraints

$$q(n) > 0 \qquad \text{and} \qquad q(x) \leq n \text{ for } x \in \{1, \ldots, n\} .$$

At each step `n`, the integer variable `q(n)` is introduced and required to be a positive integer. Moreover, all integer variables `q(1)` to `q(n)` are required to take values less or equal than n. However, while the former constraint is unconditional, the latter are subject to the external atom `query(n)`. The functioning of Listing 4 ensures that only `query(n)` is true while `query(s)` is false for all `s` $<$ `n`. In this way, the domain of all constraint variables `q(1)` to `q(n)` is increased by one at each step. Lines 12-15 in Listing 4 add distinct constraints to the effect that no two queens can be placed on the same row or diagonal of the board. Line 17 simply instructs *clingcon* to add `q(n)` to the output constraint variables.

In the following, we detail the grounding process for this example. The base program simply consists of the first 3 lines of the original encoding. Afterwards, program `step(1)` is grounded, adding the first constraints of the problem. The result is shown in Listing 6. The first variable `q(1)` is introduced and its lower bound is fixed to 1 in Line 3. Its upper bound is also restricted

[16] Strictly speaking, Line 1-3 belong to the program `base` that is treated once at the beginning (cf. Listing 4 and (Gebser et al. 2014) for details).

```
 1 pos(1).

 3 &sum{ q(1) } > 0.
 4 &sum{ q(1) } <= 1 :- query(1).

 6 &distinct{ q(1) }.

 8 &distinct{ q(1) }.
 9 &distinct{ q(1) }.

11 &show{ q(1) }.
```

Listing 6: *Grounded incremental n-queens program* `step(1)`.

```
 1 pos(2).

 3 &sum{ q(2) } > 0.
 4 &sum{ q(1) } <= 2 :- query(2).
 5 &sum{ q(2) } <= 2 :- query(2).

 7 &distinct{ q(1), q(2) }.

 9 &distinct{ q(1), q(2)+1 }.
10 &distinct{ q(1), q(2)-1 }.

12 &show{ q(2) }.
```

Listing 7: *Grounded incremental n-queens program* `step(2)`.

to 1 but here only if `query(1)` holds. This is only the case of `n=1` when solving program `step(1)` (Line 4). In all subsequent cases, `query(1)` is false, and hence $q(1) \leq 1$ is not imposed anymore. Accordingly, the atom `&sum`{`q(1)`} `<= 1` can vary freely (since it is an external constraint atom). Don't care propagation, described in Section 3.4, addresses such atoms and removes them from the system.

As solving the 1-queen problem is uninteresting, the second solving step adds program `step(2)` shown in Listing 7. We are now solving the second step and `query(1)` is no longer true, which amounts to removing the rule from Line 4 in Listing 6. The new step adds two rules for this instead (lines 4-5) and restricts all variables to be less than or equal 2. Also, additional distinct constraints are added involving `q(2)`. The next step again removes the rules in lines 4-5 by making `query(2)` false and adds a new restriction (lines 4-6 in Listing 8). In this way, we not only add new variables at each step, but also increase the upper bounds of existing ones. For solving the third step, the grounded rules of all three steps are taken together, only `query(3)` is set to true, and all previously added instances of `query/1` are false.

Listing 9 shows a run of Listing 5 up to 10 steps. Setting the stop criterion to `UNKNOWN` makes sure that the process neither terminates upon satisfiable nor unsatisfiable result.

A closer look at the distinct constraints in lines 12 to 15 of Listing 5 reveals quite some redundancy. This is because the constraints added at each step supersede the ones added previously, and they all coexist in the system. For example, at Step 3 the system

```
pos(3).

&sum{ q(3) } > 0.
&sum{ q(1) } <= 3 :- query(3).
&sum{ q(2) } <= 3 :- query(3).
&sum{ q(3) } <= 3 :- query(3).

&distinct{ q(1),q(2),q(3) }.

&distinct{ q(1), q(2)+1, q(3)+2 }.
&distinct{ q(1), q(2)-1, q(3)-2 }.

&show{ q(3) }.
```

Listing 8: *Grounded incremental n-queens program* `step(3)`.

```
$ clingcon incqueens.lp -c imax=10 -c istop=\"UNKNOWN\"
clingcon version 3.2.0
Reading from incqueens.lp
Solving...
Answer: 1

Solving...
Answer: 1
q(1)=1
Solving...
Solving...
Solving...
Answer: 1
q(4)=2 q(3)=4 q(2)=1 q(1)=3
Solving...
Answer: 1
q(5)=3 q(1)=1 q(2)=4 q(3)=2 q(4)=5
Solving...
Answer: 1
q(6)=5 q(5)=3 q(1)=2 q(2)=4 q(3)=6 q(4)=1
Solving...
Answer: 1
q(7)=6 q(6)=3 q(5)=5 q(1)=2 q(2)=4 q(3)=1 q(4)=7
Solving...
Answer: 1
q(8)=7 q(7)=3 q(6)=1 q(5)=6 q(1)=4 q(2)=2 q(3)=5 q(4)=8
Solving...
Answer: 1
q(9)=3 q(8)=6 q(7)=8 q(6)=5 q(5)=2 q(1)=1 q(2)=4 q(3)=7 q(4)=9
SATISFIABLE

Models       : 8+
Calls        : 10
Time         : 0.075s (Solving: 0.02s 1st Model: 0.02s Unsat: 0.00s)
CPU Time     : 0.070s
```

Listing 9: *Running Listing 5 (Q_1;* `incqueens.lp`*)*

contains 3 instances of Line 12, namely `&distinct{q(1)}`, `&distinct{q(1),q(2)}`, and `&distinct{q(1),q(2),q(3)}`. Clearly, the first two constraints are redundant in view of the third but remain in the system. To avoid this redundancy, we can make use of the external atom `query(n)` to remove the redundant distinct constraints at each step in the same way we tighten the upper bound of variable domains. This

Measure	Q_1	Q_2	Q_3
Time	138s	10s	16s
Variables	55k	55k	32k
Static Nogoods	24k	5k	2k
Dynamic Nogoods	1181k	320k	301k

Table 3: *Comparison of different incremental n-queens programs.*

amounts to replacing lines 12-15 in Listing 5 with the ones given in Listing 10 below.

```
12  &distinct{ q(X)     : pos(X)} :- query(n).

14  &distinct{ q(X)+X-1 : pos(X)} :- query(n).
15  &distinct{ q(X)-X+1 : pos(X)} :- query(n).
```

Listing 10: *Retracting Constraints, encoding Q_2*

Although the last modification guarantees that the system bears no redundant distinct constraints,[17] it leads to adding and removing the same restrictions over and over again. For example, the constraint that `q(1)` and `q(2)` must have different values is included in every distinct constraint after step 1. And this information is retracted and re-added at each step. This is avoided by the constraints in Listing 11. This formulation only adds constraints for the new variable `q(n)` at each step `n` and stays clear from retracting any constraints.

```
12  &sum{ q(X)     } != q(n)     :- X=1..n-1.

14  &sum{ q(X)+X-1 } != q(n)+n-1 :- X=1..n-1.
15  &sum{ q(X)-X+1 } != q(n)-n+1 :- X=1..n-1.
```

Listing 11: *Partial Constraints, encoding Q_3*

Table 3 gives a comparison of the three different encodings for the incremental n-queens problem for 30 steps. The first row gives the respective total running time. The second one reports the total number of introduced atoms. The third one gives the sum of static nogoods generated at each step, and the last one the sum of dynamic nogoods generated by lazy constraint propagation. We observe that the initial encoding Q_1 performs worst in all aspects. The inherent redundancy of Q_1 is reflected by the high number of dynamic nogoods generated by the constraint propagator. This is the source of its inferior overall performance. Unlike this, the two alternative approaches bear less redundancy, as reflected by their much lower number of dynamic nogoods. In Q_2, this is achieved by eliminating duplicate inferences from redundant constraints. Although Q_3 even further reduces the number of atoms as well as static and dynamic nogoods, its runtime is slightly inferior. This is arguably due to the usage of elementary linear constraints rather then global distinct constraints (and the pigeon hole constraints which are enabled by default).

4 Experiments

In this section, we evaluate the afore-presented features and compare *clingcon* with other systems. We performed all our benchmarks on an Intel Xeon E5520 2.27GHz processor with Debian GNU/Linux 7.9 (wheezy). We used a timeout of 1800 seconds and restricted main memory to

[17] Given that don't care propagation is enabled by default.

6GB. In all tests, we count a memory out as a timeout. The experiments are split into three sections. First, we evaluate the presented features and discuss corresponding configurations of *clingcon*. Second, we compare *clingcon* with state of the art CP solvers using the benchmark classes of the *minizinc* competition 2015. And finally, we contrast *clingcon* with other CASP systems using different CASP problems.

Option	Value	Explanation
`--equality-processing`	`true`	Enable equality processing
`--distinct-to-card`	`false`	Translate distinct constraints using inequalities
`--distinct-pigeon`	`true`	Use pigeon hole constraints
`--distinct-permutation`	`false`	Not using permutation constraints
`--sort-coefficient`	`false`	Sort by domain size first
`--sort-descend-coefficient`	`true`	Sort using decreasing coefficients
`--sort-descend-domain`	`false`	Sort using increasing domain sizes
`--split-size`	`-1`	Not splitting constraints
`--max-nogoods-size`	`1024`	Not splitting constraints with less than 1024 nogoods
`--translate-constraints`	`10000`	Translate constraints with less than 10000 nogoods
`--break-symmetries`	`true`	Break symmetries when splitting
`--domain-size`	`10000`	Use 10000 as a threshold for domain propagation
`--redundant-nogood-check`	`true`	Enable redundant nogood check when translating
`--dont-care-propagation`	`true`	Enable don't care propagation
`--min-lits-per-var`	`1000`	Introduce at least 1000 order atoms per variable
`--flatten-optimization`	`true`	Flatten the objective function
`--prop-strength`	`4`	Use highest propagation strength 4
`--explicit-binary-order`	`false`	Not explicitly creating nogoods from $\Phi'(\mathcal{V}, D)$
`--learn-nogoods`	`true`	Add all learned nogoods to ∇ immediately

Table 4: *Default configuration D of clingcon 3.2.0.*

To evaluate the presented techniques, we give a comprehensive comparison in Table 5. To concentrate on the CP techniques of *clingcon* 3.2.0, we use the CP benchmarks of the *minizinc* competition 2015.[18] We removed the benchmark classes *large scheduling* and *project planning* as they cannot be translated into the *flatzinc* format without the use of special global constraints. For all other classes, we used the *mzn2fzn*[19] toolchain to convert all instances to *flatzinc* while removing all non-linear and global constraints except for distinct. This functionality is provided by *mzn2fzn*, which translates non-supported constraints away. We use the standard translation provided by *mzn2fzn* to be able to handle all benchmark classes. In this way, even problems using constraints on sets, non-linear equations, or complex global constraints can be handled by solvers restricted to basic linear constraints. For making this benchmark suite available to the CASP community, we build a converter from *flatzinc* to the *aspif* format (Gebser et al. 2016b) used by *clingcon*; it is called *fz2aspif*.[20] To evaluate the different features, we modified the scoring system of the *minizinc* competition, which is based on the Borda count evaluation technique. On a per instance basis, a configuration gets one point for every other configuration being worse. A configuration is considered worse, if either the found optimization value is at least 1% lower, or if it has the same optimization value but is slower. A configuration is considered slower if it is at least 5 seconds slower. Classes marked with * are decision problems (all others are optimization

[18] `http://www.minizinc.org/challenge2015/challenge.html`
[19] `http://www.minizinc.org/software.html`
[20] `https://potassco.org/labs/2016/12/02/fz2aspif.html`

problems); classes containing the global distinct constraint are marked with †. We have exactly five instances per class.

The following discussion refers to the results shown in Table 5. The columns used for comparison are named in the paragraph heading. Column D presents the default configuration of *clingcon* given in Table 4. All other listed configurations differ only in one or two options from this default in order to test specific techniques. For instance, for evaluating equality processing, we compare default configuration D, using equality processing, with configuration NE, disabling equality processing. Thus, except for `--equality-processing`, all other options remain unaltered.

Equality Processing (D, NE) To evaluate the influence of equality processing, we compare default configuration D (with equality processing) with configuration NE (without equality processing). This feature improves performance on nearly all benchmark classes significantly. By simply removing constraints and variables the underlying CSP gets easier to solve (no matter if it is solved by translation or propagators).

Distinct Translation (D, DT) Translating global distinct constraints into cardinality rules prevents order atoms from being created lazily. The default configuration D translates them into a set of inequalities. The translation using cardinality constraints in column DT performs better on *cvrp*, *open-stacks*, and *p1f*, while it performs worse on the benchmark class *costas*. As long as the domain size is small, this feature can be useful for problems using distinct constraints. The configuration DT performs best of all tested configurations.

Pigeon Hole Constraints (D, NP) Since pigeon hole constraints add only constant overhead in the number of nogoods, they are enabled in configuration D. Disabling their addition, slightly increases performance on benchmark classes containing distinct constraints (marked with †), as witnessed in column NP. Although these constraints have no positive effect on the benchmarks at hand, we keep this feature enabled by default since it increases propagation strength.

Permutation Constraints (D, PO) Unlike pigeon hole constraints, permutation constraints introduce direct encoding atoms which prevents lazy variable generation for some constraint variables. This is the reason why this feature is disabled by default in configuration D. We enabled it in column PO. Again, this feature only influences benchmark classes containing distinct constraints. It improves performance for the *cvrp* class but decreases it on the other classes. The impact of this feature depends upon the respective problem.

Sorting (D, SC) As we cannot account for all combinations of sorting mechanisms, we evaluate this feature only on the cases discussed in (Tamura et al. 2013). Default configuration D implements the one in *sugar*; it sorts by smallest domain first and prefers on ties larger coefficient. The alternative sorting recommended in (Tamura et al. 2013) first sorts on larger coefficients and afterwards uses the smaller domain. This behavior is enforced by setting `--sort-coefficient=true` and reflected in column SC. We see that both sorting methods yield a similar performance when applied to our lazy nogood generating approach.

Splitting Constraints (D, SP, T_4, ST) Splitting constraints into smaller ones is mandatory for any translation-based approach using the order encoding to avoid an exponential number of nogoods. We restricted our evaluation to a splitting size of 3, as done in *sugar*. The default configuration D of *clingcon* does not split any constraints. The effect of splitting constraints into ternary ones (`--split-size=3`) is reflected by column SP; it performs poorly in our lazy nogood generating setting because it introduces many new constraints and variables. On the other hand, when translating all constraints (`--translate-constraints=-1`) as shown in column T_4, the split into constraints of up to three variables (`--translate-constraints=-1`

instances	D	NE	DT	NP	PO	SC	SP	ST	NS	D_1	D_2	D_3	T_1	T_2	T_3	T_4	NR	ND	M_1	M_2	M_3	NF	P_1	P_2	P_3	EO	RL
*costas**†	38	47	15	39	41	38	27	27	24	27	27	26	12	33	0	0	16	38	36	38	40	38	12	12	38	32	**60**
cvrp†	49	38	90	61	65	49	51	44	52	51	51	51	31	67	41	0	34	55	93	95	97	49	9	9	49	**100**	66
freepizza	**121**	111	**121**	**121**	**121**	**121**	47	26	49	51	47	47	79	**121**	**121**	26	26	100	99	107	102	115	109	113	120	120	109
gfd-schedule	105	69	103	102	102	103	52	11	47	54	54	54	58	102	102	0	36	50	84	**114**	112	103	81	81	105	102	93
grid-colour	**130**	**130**	**130**	**130**	**130**	**130**	**130**	**130**	**130**	**130**	**130**	**130**	**130**	**130**	**130**	**130**	**130**	**130**	**130**	**130**	**130**	**130**	**130**	**130**	**130**	**130**	**130**
is†	118	111	116	119	111	118	45	0	45	44	45	44	96	118	118	0	0	93	**128**	53	56	118	0	0	117	90	80
mapping	106	87	106	106	106	106	33	0	18	30	33	33	104	112	110	0	0	**124**	84	109	111	106	0	0	106	97	112
knapsack	66	55	64	66	66	64	7	2	7	3	7	7	65	65	66	0	2	66	44	73	76	57	**99**	**99**	64	40	34
*nmseq**	100	100	100	100	98	99	16	27	25	25	28	28	100	67	100	0	28	100	88	79	100	96	0	0	92	39	**130**
opd	89	82	88	89	89	89	**125**	**125**	121	**125**	**125**	**125**	52	89	88	0	**125**	63	44	89	90	88	79	79	88	78	83
open-stacks†	96	73	116	107	77	95	57	54	71	65	58	59	33	96	77	21	55	95	79	95	103	92	61	61	94	**117**	54
p1f†	66	66	**100**	70	71	66	35	65	35	36	35	35	84	66	66	0	65	75	57	74	76	71	64	64	66	78	68
radiation	125	118	125	125	124	115	62	43	40	52	62	62	108	107	103	0	50	125	112	119	110	101	68	68	125	**128**	40
roster	**130**	**130**	**130**	**130**	**130**	**130**	**130**	**130**	**130**	**130**	**130**	**130**	**130**	**130**	**130**	0	**130**	**130**	**130**	**130**	**130**	**130**	**130**	**130**	**130**	**130**	**130**
spot5	86	86	86	86	86	86	105	75	105	100	105	105	86	86	86	0	78	86	68	111	68	86	102	102	86	**120**	106
tdtsp	77	69	77	77	77	77	94	0	35	94	94	94	47	76	30	0	0	3	70	59	**119**	77	44	12	84	46	75
triangular	15	15	15	15	15	15	38	11	38	38	38	15	56	15	15	0	11	15	15	15	15	8	**77**	**77**	15	33	27
zephyrus	72	47	72	72	72	72	62	0	62	62	62	62	64	72	72	0	0	72	71	**103**	0	72	0	0	46	72	7
total	1589	1434	**1654**	1615	1581	1573	1116	770	1034	1117	1131	1107	1335	1552	1455	177	786	1420	1432	1593	1535	1537	1065	1037	1555	1552	1404

Table 5: *Comparison of different features of clingcon 3.2.0 on the benchmark set of the minizinc competition 2015. Shown are scores of how often a configuration is better than another one. Bold numbers indicate the best configuration for the benchmark class.*

and `--split-size=3`) increases performance significantly, as witnessed by column ST. We conclude that splitting constraints is not necessary for lazy nogood generating solvers but essential for translational approaches that use the order encoding.

Symmetry Breaking (SP, NS) Splitting constraints introduces auxiliary variables that may lead to redundant solutions. Symmetry breaking eliminates such redundancies and has only an effect when splitting constraints. This is why it is interesting to compare column SP (`--split-size=3`) where symmetry breaking is enabled with column NS (`--split-size=3` and `--break-symmetries=false`) where it is disabled. In both cases, all constraints are split into ternary ones. The additional constraints remove symmetric solutions from the search space and therefore seem to be beneficial, especially on classes *tdtsp*, *radiation*, and *mapping*.

Domain Propagation (D_1, D_2, SP, D_3) To investigate the impact of domain propagation during preprocessing, we tested four different configurations that all split constraints into ternary ones (`--split-size=3`). They only differ in using the options `--domain-size=0` (no domain propagation) in column D_1, `--domain-size=1000` in column D_2, `--domain-size=10000` in column SP, and `--domain-size=-1` (unlimited domain propagation) in column D_3. We observe that unlimited domain propagation reduces performance in benchmark class *triangular* but has no significant influence otherwise. The other tested configurations have no influence on the runtime of the benchmarks. We assume that domain propagation prunes the domains not enough to make a considerable difference. For the default configuration of *clingcon*, we decided to restrict it to a reasonable number (10000) which leaves it enabled for mid-sized domains.

Translate Constraints (T_1, T_2, D, T_3, T_4) We have already seen that translating all constraints as shown in column T_4 is not very beneficial. Now, we evaluate whether the translation of “small” constraints improves performance through a mixture of “translating small constraints” and “handling larger ones lazily”. Therefore, we compare the results obtained with option `--translate-constraints=0` (no constraints are translated) in column T_1, with T_2 where `--translate-constraints=1000` (translate constraints that produce up to 1000 nogoods) is used, with D using `--translate-constraints=10000` (up to 10000 nogoods), with T_3 using `--translate-constraints=50000` (up to 50000 nogoods), and T_4 using `--translate-constraints=-1` (all constraints are translated). There is a trade-off on the size of constraints to translate. While translating small constraints (constraints that produce up to 1000 nogoods) improves performance, the translation of larger constraints decreases it again. On some benchmarks, like *triangular* and *p1f*, translating no constraints is beneficial. Also, translating all constraints in T_4 performs worst of all tested configurations.

Redundant Nogood Check (ST, NR) To evaluate this feature, we decided to translate all constraints (`--translate-constraints=-1`). Since this configuration is not producing good results for a comparison (most of the time the translation is simply too large to be finished), we additionally split the constraints into ternary ones with option `--split-size=3`. With this, we compare the configuration with redundancy check in column ST with NR where redundancy checking is disabled (`--redundant-nogood-check=false`). The redundant nogood check is fast and simply removes redundant nogoods from the order encoding. Benchmark classes like *costas* and *cvrp* perform better with the reduced set of nogoods, while redundant nogoods are beneficial for *gfd-schedule* and *radiation*.

Don’t Care Propagation (D, ND) is enabled by default and removes unnecessary implications from the problem. Disabling this feature (`--dont-care-propagation=false`) in column *ND* decreases performance.

Order Atom Generation (M_1, D, M_2, M_3) Adding order atoms lazily is mandatory to handle large domains. We now evaluate the effect of adding a small amount of order atoms eagerly for every constraint variable, evenly spread among its domain values. Therefore, we compare column M_1 using `--min-lits-per-var=0` (adding no atoms), with D using `--min-lits-per-var=1000` (adding 1000 order atoms per variable), with M_2 using `--min-lits-per-var=10000` (adding 10000), and M_3 using `--min-lits-per-var=-1` (adding all order atoms). Adding no order atoms in advance drastically reduces performance of the system while adding 1000 to 10000 order atoms achieves best performance. When adding too many or even all order atoms before solving, performance is again decreased, especially on classes with large domains like *zephyrus*. Also, note that the tested benchmark classes are very sensitive to this option as adding atoms beforehand may influence the heuristic of the search.

Flattening Objective Functions (D, NF) is a feature well received by this benchmark set. All *flatzinc* encodings contain only one variable subject to minimization. On most benchmark classes this variable simply represents the sum of a set of variables. Adding this set directly to the objective function avoids adding an unnecessary and probably large constraint and also improves propagation strength of the learned nogoods. Unlike D, configuration NF disables this feature via `--flatten-optimization=false`. We observe that flattening the optimization statement increases the performance on many benchmark classes.

Lazy Nogood Generation (P_1, P_2, P_3, D) We now evaluate the four afore-described propagation strengths where `--prop-strength=1` is reflected by the results in column P_1, `--prop-strength=2` by the ones in column P_2, `--prop-strength=3` in column P_3, and `--prop-strength=4` in the default configuration D. We see that a high propagation strength is important. Especially propagating changed bounds with `--prop-strength=3` is necessary for many benchmark classes. Interestingly, less propagation performs best for the classes *knapsack* and *triangular* where constraint propagation is not dominating the search but still takes time. On these classes, configurations with propagation strength 1 or 2 spend less time on CSPPROPAGATION() and more on pure CDCL search, as attested by a much higher number of choices.

Explicit Binary Order Nogoods (D, EO) Default configuration D does not introduce explicit binary order nogoods $\Phi(\mathcal{V}, D)$ but uses a propagator for capturing the corresponding inferences lazily. The option `--explicit-binary-order=true` (reflected by column EO) creates these nogoods explicitly for all order atoms created during preprocessing, leaving the others subject to lazy nogood propagation. Although, overall performance of the implicit binary order nogoods is better, for some benchmark classes like *cvrp* and *spot5* using binary order nogoods explicitly is the best choice. This is one of the options for which it is hard to find a clear cut default setting and that needs consideration for each benchmark class.

Reduced Nogood Learning (D, RL) *clingcon*'s default configuration D adds all nogoods returned by CSPPROPAGATION to the set of learned nogoods (viz. ∇ in Algorithm 3). Lazily adding these nogoods when they are actually needed for conflict analysis is achieved with `--learn-nogoods`; the results are shown in column RL. The average performance of adding nogoods lazily is inferior to the one obtained by learning all nogoods. Nevertheless, the latter setting performs best on *costas* and *nmseq*, the two decision problems in our benchmark set.

instances	*clingcon*	*g12fd*	*gecode*	*minisatid*	*chuffed*	*chuffed'*	*picatsat*	*picatsat'*
costas*†	6	0	**19**	9	8	8	11	11
cvrp†	24	3	6	5	**30**	22	4	6
freepizza	**35**	15	0	31	26	26	3	3
gfd-schedule	10	9	12	27	**28**	**28**	20	14
grid-colour	**35**	1	8	34	23	23	31	31
is†	17	15	2	17	**35**	32	19	11
mapping	17	14	0	17	**29**	**29**	23	16
knapsack	19	11	14	8	5	5	**26**	**26**
nmseq*	24	15	**25**	3	21	21	5	5
opd	25	6	4	15	21	17	31	**32**
open-stacks†	20	0	9	8	**35**	**35**	21	16
p1f†	21	0	22	3	**34**	21	5	8
radiation	28	7	7	12	**33**	**33**	18	14
roster	**35**	**35**	0	**35**	**35**	**35**	29	28
spot5	20	10	0	19	16	16	**35**	31
tdtsp	8	**35**	10	1	31	24	10	9
triangular	15	**29**	5	26	23	24	12	12
zephyrus	3	20	7	10	26	26	**30**	**30**
total	362	225	150	280	**459**	425	333	303

Table 6: *Comparing clingcon 3.2.0 DT with different state of the art CP solvers on the minizinc competition 2015 benchmark set.*

Future work has to investigate which of the nogoods have to be learned and which of them can be added lazily.

Configuration DT is the configuration with the highest overall score. Nevertheless, *clingcon*'s default configuration is more conservative since it allows for using lazy variable generation in all cases. For instance, with configuration DT it is impossible to run the multi-shot n-queens example presented in Section 3.5, because 2^{30} order atoms had to be created per queen in order to use cardinality constraints for the distinct constraint.

Next, we compare *clingcon* to state of the art CP solvers on the same set of benchmarks with the same scoring system. The second column of Table 6 shows configuration DT of *clingcon* 3. This is the best configuration of the internal comparison in Table 5, which is obtained using the command line option `--distinct-to-cardinality=true`. We compare it to *g12fd* (Mercury FD Solver), which is the G12 FlatZinc interpreter's default solver, taken from the *minizinc* 2.0.11 package.[21] Furthermore, we have taken *gecode* 4.4.0,[22] a well-known classical CP solver. Also, the lazy clause generating solvers *minisatid* 3.11.0 (De Cat et al. 2013)[23] as well as *chuffed*,[24] the best solver of the *minizinc* competition 2015.[25] Finally, we compare to *picatsat* 2.0,[26] a CP solver that won the second place at the *minizinc* competition 2016 by translating constraints into SAT using a logarithmic encoding. We ran *g12fd* and *gecode* with `--ignore-user-search` to disable any special heuristic given in the problem encodings for all solvers. In the competition,

[21] `http://www.minizinc.org/software.html`
[22] `http://www.gecode.org`
[23] With some bugfixes. Special thanks to Bart Bogaerts for his great support on this work.
[24] `https://github.com/geoffchu/chuffed` — SHA 5b379ed9942ee59e8684149eae3fec1af426f6ee
[25] It did not participate in the ranking as it is was entered by the organizers. It ran outside of competition and was faster than the winning system.
[26] `http://picat-lang.org`

this is called “free search”. To measure the core performance of the systems, it is most instructive to consider *chuffed′* and *picatsat′* which use the two solvers on exactly the same set of constraints as *clingcon*. Hence, all non-linear and global constraints (except distinct) are translated using *mzn2fzn* in the same way for all systems. [27]

The results in Table 6 show that *clingcon*[28] outperforms established systems such as *g12fd*, *gecode*, *minisatid*, and even *picatsat*. There are also different benchmark classes where solvers dominate each other and vice versa. We point out that *gecode* has special propagators for many non-linear and global constraints that have been used in the benchmarks. Also *chuffed*, as a lazy clause generating solver, has propagators for many other constraints and can therefore handle some of the benchmark classes much better. As we are building a CASP system, we refrain from supporting a broad variety of global constraints, as some of them can be modeled in ASP. So for a better comparison on the features of *clingcon*, we translated all non-linear and global constraints except for distinct in the columns *chuffed′* and *picatsat′* into linear ones. Here, we see that these systems profit from the dedicated treatment of global constraints but that the base performance of *clingcon* is comparable. In general, *clingcon* does not match the performance of the best solver of the *minizinc* competition 2015 but on benchmark classes like *freepizza*, *grid-colour*, *opd*, *knapsack*, and *spot5*, it even outperformed *chuffed*. We conclude that *clingcon*, despite being a CASP system, is at eye level with state of the art CP solvers but cannot top the best lazy clause generating systems.

Finally, we compare *clingcon* against six other CASP systems.

- *inca* (Drescher and Walsh 2010) with the option `--linear-bc`,[29] a lazy nogood generating system not supporting lazy variable generation.
- *clingcon* 2 (Ostrowski and Schaub 2012), using *gecode* 3.7.3 as a black-box CP solver.
- *ezcsp* 1.6.24 (Balduccini and Lierler 2013), also pursuing a black-box approach but using CP solver B-Prolog 7.4 with ASP solver *clasp*.
- *aspartame* (Banbara et al. 2015), a system using an eager translation of the constraint part by means of an ASP encoding.
- *ezsmt* 1.0.0 (Lierler and Susman 2016), translating CASP programs to SMT, solved by SMT solver *z3* 4.2.2.
- *clingo* 5.1.0, a pure ASP solver to measure the influence of the CP part on solving.

The first benchmark class is the two dimensional strip packing problem (Soh et al. 2010); its encoding is shown in Listing 2. In Table 7, column *clingo* 5 reflects the results obtained with a highly optimized ASP encoding, using a handcrafted order encoding. Time is given in seconds, letting - denote a timeout of 1800 seconds. The best objective value computed so far is given in the columns headed with opt. For *aspartame*, we have taken an encoding provided in (Banbara et al. 2015). For the other systems such as *clingcon* 2, *clingcon* 3, and *inca*, we adjusted the syntax for the linear constraints. We refrained from comparing with *ezcsp* or *ezsmt* as both systems are not supporting optimization of integer variables. The bottom row counts the number of times a system performed best. We clearly see that *clingcon* 2 is outperformed even by the manual ASP

[27] Unfortunately, we were unable to compare to the lazy clause generating system *g12lazy*, as it produced wrong results on some of the benchmarks and is no longer maintained. We were also unable to convert the competition benchmarks to a format readable by *sugar*, as existing converters are outdated and not compatible anymore.
[28] Note that the Borda Count scores are relative to the compared systems, and therefore are different for the same configuration of *clingcon* in Table 5 and 6.
[29] This option was recommended by the authors of the system for these kind of benchmarks.

	clingo 5		*aspartame*		*clingcon* 2		*inca*		*clingcon* 3	
instances	time	opt	time	opt	time	opt	time	opt	time	opt
BENG01	9	30	20	30	-		916	30	**2**	**30**
BENG02	-	58	**1336**	**57**	-		-	58	-	58
BENG03	-	87	-	85	-		-	85	**775**	**84**
BENG04	-	111	-	**108**	-		-	**108**	-	**108**
BENG05	-	141	-	**136**	-		-	**136**	-	**136**
BENG06	1226	36	32	36	-		**5**	**36**	23	36
BENG07	-	69	-	**68**	-		-	69	-	**68**
BENG08	-		-		-		-	104	-	**103**
BENG09	-		-		-		-		-	**128**
BENG10	-		-		-		-		-	**158**
CGCUT01	**1**	**23**	**1**	**23**	-	26	-	25	**0**	**23**
CGCUT02	-	66	-	**65**	-		-	67	-	**65**
CGCUT03	-		-		-		-		-	
GCUT01	-	1016	5	1016	**0**	**1016**	**0**	**1016**	**0**	**1016**
GCUT02	-	1242	-	1195	-		-	**1190**	-	**1190**
GCUT03	-		134	1803	-		**1**	**1803**	12	1803
GCUT04	-		-		-		-		-	
HT01	**1**	**20**	**1**	**20**	-	22	346	20	**0**	**20**
HT02	8	20	5	20	-	25	77	20	**1**	**20**
HT03	**1**	**20**	**1**	**20**	-		10	20	**0**	**20**
HT04	840	15	33	15	-		-	16	**8**	**15**
HT05	12	15	**9**	**15**	-		**8**	**15**	13	15
HT06	14	15	8	15	-		359	15	**1**	**15**
HT07	-	31	**175**	**30**	-		-	31	-	31
HT08	**1284**	**30**	-	31	-		-	31	-	31
HT09	-	31	272	37	-		-	31	**41**	**30**
NGCUT01	**0**	**23**	**0**	**23**	**1**	**23**	**0**	**23**	**0**	**23**
NGCUT02	**2**	**30**	**1**	**30**	-	33	80	30	**0**	**30**
NGCUT03	**2**	**28**	**2**	**28**	-		**1**	**28**	**0**	**28**
NGCUT04	**0**	**20**	**0**	**20**	**0**	**20**	**0**	**20**	**0**	**20**
NGCUT05	**0**	**36**	**0**	**36**	-		**0**	**36**	**0**	**36**
NGCUT06	8	31	**1**	**31**	-		**0**	**31**	**0**	**31**
NGCUT07	**0**	**20**	**0**	**20**	**0**	**20**	**0**	**20**	**0**	**20**
NGCUT08	**1**	**33**	**1**	**33**	-	36	38	**33**	**0**	**33**
NGCUT09	**87**	**50**	-	50	-	57	-	50	1549	50
NGCUT10	6	80	**1**	**80**	-	81	**0**	**80**	**0**	**80**
NGCUT11	**4**	**52**	**1**	**52**	-	55	**0**	**52**	**0**	**52**
NGCUT12	-	87	**3**	87	-		**0**	**87**	**0**	**87**
#best	13		21		4		16		**28**	

Table 7: *Comparison of different CASP systems on the two dimensional strip packing problem.*

encoding. The new *clingcon* 3 system performs best. The translational approach of *aspartame* is close to the *inca* system, and both perform better than the manual ASP approach. According to (Soh et al. 2010), these results are in accord with dedicated, state of the art systems.

The next benchmark classes are incremental scheduling, weighted sequence, and reverse folding, all stemming from the ASP competition.[30] Encodings for *clingo*, *ezcsp*,[31] *ezsmt* and *clingcon* 2

[30] `http://aspcomp2015.dibris.unige.it/LPNMR-comp-report.pdf`
[31] To be comparable, we used the encoding without *cumulative* constraint.

have been taken from (Lierler and Susman 2016) in combination with instances from the ASP competition.[32] We changed the pure ASP encoding for *clingo* slightly for a better grounding performance. For these classes, we could not provide an encoding for *aspartame*, as its prototypical CASP support does not allow for modeling parametrized n-ary constraints.

For incremental scheduling, *inca* produces wrong results due to its usage of an intermediate version of *gringo*, viz. 3.0.92. The runtime in seconds for incremental scheduling is shown in

instances	*clingo* 5	*clingcon* 2	*clingcon* 3	*ezcsp*	*ezsmt*
020-inc	302	**1**	**0**	**0**	**0**
028-inc	-	16	**4**	-	**5**
044-inc	-	518	149	-	**116**
063-inc	335	**0**	**1**	-	**0**
083-inc	268	-	**1**	-	**0**
096-inc	719	298	**1**	-	**0**
106-inc	470	-	**2**	-	**1**
158-inc	355	**4**	**1**	**0**	**1**
175-inc	-	83	**6**	-	**4**
181-inc	-	**5**	**1**	-	**2**
184-inc	425	-	**1**	-	**1**
211-inc	-	-	**194**	-	799
214-inc	-	-	**7**	-	76
230-inc	-	77	**9**	-	24
256-inc	-	-	-	-	-
257-inc	-	-	-	-	-
266-inc	-	-	**767**	-	-
334-inc	-	-	-	-	-
338-inc	-	-	-	-	-
362-inc	-	-	-	-	-
#best	0	4	**14**	2	11

Table 8: *Comparison of different CASP systems on the incremental scheduling problem.*

Table 8. We see that *clingcon* 2 improves on the dedicated ASP encoding. In fact, incremental scheduling is a true CASP problem where the pure ASP encoding can be improved by using CP. While the black-box approach of *ezcsp* performs worst, *ezsmt* and *clingcon* 3 clearly dominate this comparison.[33] The enhanced preprocessing techniques and the lazy variable generation of *clingcon* even outperforms the industrial SMT solver *z3* (as used in *ezsmt*).

For the weighted sequence problem, we see in Table 9 that *inca*, *clingo*, *ezsmt*, and *clingcon* 3 perform well on this benchmark set, while *clingcon* 2 could not compete with the timings of the other systems and *ezcsp* did not solve any of them. Again, time is shown in seconds and - denotes a timeout of 1800 seconds. We also see that the performance of the pure ASP encoding is in the same range as that of the winning CASP systems. Hence, the ASP solving part clearly dominates the CSP part. This also explains the slightly worse performance of *clingcon* 3 due to its heavy preprocessing of the CSP part.

For the reverse folding problem, we compare the same systems as before. Table 10 gives the running time in seconds. While all CASP systems improve upon the pure ASP encoding,

[32] We refrained from using the other three benchmark classes from this source as the available instances were too easy to solve to produce informative results.
[33] The time to run the completion and translation processes for *ezcsp* and *ezsmt* is not included in the tables.

instances	*clingo* 5	*inca*	*clingcon* 2	*clingcon* 3	*ezcsp*	*ezsmt*
01-tree	**6**	**5**	**6**	**1**	-	**1**
02-tree	**8**	**1**	**8**	**3**	-	**5**
05-tree	**5**	**1**	**4**	**1**	-	**3**
06-tree	**3**	**2**	19	**1**	-	**3**
07-tree	**3**	**4**	**1**	**1**	-	**4**
11-tree	**2**	**0**	6	**1**	-	**0**
15-tree	**2**	**3**	**1**	**1**	-	**1**
16-tree	**2**	**4**	15	**5**	-	**1**
22-tree	**4**	**0**	**4**	**1**	-	**0**
23-tree	**3**	**0**	**4**	**0**	-	**3**
26-tree	**2**	**3**	51	**4**	-	**1**
29-tree	7	**1**	26	**2**	-	**5**
33-tree	**9**	**7**	95	15	-	**9**
35-tree	24	**10**	231	30	-	26
38-tree	**7**	17	30	**10**	-	**11**
39-tree	**6**	**7**	330	46	-	23
40-tree	**3**	9	398	47	-	14
41-tree	36	**14**	**18**	**13**	-	34
49-tree	**7**	**3**	220	22	-	14
53-tree	**5**	8	297	33	-	**2**
#best	**17**	**17**	8	14	0	15

Table 9: *Comparison of different CASP systems on the weighted sequence problem.*

clingcon 2 and *clingcon* 3 perform best on this benchmark class. The preprocessing overhead of *clingcon* 3 does not pay off in terms of runtime on this benchmark class, making it perform slightly worse than *clingcon* 2. Of the two lazy nogood generating solvers *inca* and *clingcon* 3, the latter performs better due to lazy variable generation, as not all order atoms have to be generated before solving. While the black-box approach of *ezcsp* can solve the problem, the translation to SMT by *ezsmt* performs even better. We conclude that this is also due to the fact that no auxiliary atoms for an encoding of the constraints are used in *ezsmt*. A closer inspection revealed that the number of choices for *inca* and *clingcon* 2 is below 100 on average. For this problem, the ASP part is dominated by the CSP part. This is also the reason why the pure ASP encoding produces a memory out on all instances (it was not able to ground all constraints).

We conclude that *clingcon* 3 improves significantly upon its predecessor *clingcon* 2, is comparable to state of the art CP systems, and the currently fastest CASP system available. All benchmarks, encodings, instances and results are available online.[34]

5 Discussion

CASP combines ASP with CP, and thus brings together various techniques from both areas. Groundbreaking work has been done with the systems *ac*- and *adsolver* (Mellarkod et al. 2008; Mellarkod and Gelfond 2008) by using an off-the-shelf CP solver. This is called a black-box approach. It features a very high abstraction level and allows for great flexibility, for instance, for changing solvers or theories. Unfortunately, this high abstraction hinders tight integration techniques that are necessary to achieve a performance suitable for real world problems. Still using

[34] `https://potassco.org/clingcon`

instances	*clingo* 5	*inca*	*clingcon* 2	*clingcon* 3	*ezcsp*	*ezsmt*
07-reverse	-	**1**	**0**	**1**	11	**1**
11-reverse	-	**1**	**0**	**1**	9	**1**
15-reverse	-	**1**	**1**	**1**	**6**	**1**
18-reverse	-	**2**	**1**	**2**	27	**1**
20-reverse	-	**6**	**3**	**4**	60	10
24-reverse	-	12	**5**	**8**	272	49
28-reverse	-	11	**5**	**7**	107	**8**
31-reverse	-	20	**8**	36	128	73
34-reverse	-	25	**11**	20	625	112
35-reverse	-	35	**15**	23	353	96
39-reverse	-	40	**18**	**23**	682	212
44-reverse	-	-	**33**	**38**	-	339
47-reverse	-	**6**	**4**	**4**	86	**4**
49-reverse	-	**7**	**4**	**4**	67	**4**
50-reverse	-	**2**	**1**	**2**	12	**4**
#best	0	8	**15**	12	1	8

Table 10: *Comparison of different CASP systems on the reverse folding problem.*

a black-box CP solver but having a tighter integration into modern CDCL algorithms is common to systems like *ezcsp* and its extensions (Balduccini 2009; Balduccini and Lierler 2013), *dlvhex* (Eiter et al. 2012), and *clingcon* 2 (Gebser et al. 2009). These systems use a CP solver for propagation and consistency checking. No auxiliary variables are used to represent non-Boolean variables. This prevents these systems from producing strong reasons and conflicts, needed for effective CDCL-based search. The system *clingcon* 2 tries to circumvent this problem. It strengthens propagation and integration (Ostrowski and Schaub 2012) by using filtering techniques and special knowledge about the theory. A different way to tackle the problem is the eager approach. The theory part of the problem is translated to ASP, SAT, or SMT in a preprocessing step. *dingo* (Janhunen et al. 2011) translates ASP enriched with difference constraints to SMT, *ezsmt* translates CASP to SMT, and *aspartame* (Banbara et al. 2015) provides an ASP encoding to translate CP (and CASP) into ASP. The eager approach has the strongest integration because only one solver without dedicated propagators is used to solve the problem. The features of modern CDCL algorithms such as conflict driven heuristics and learning are supported natively without any change to the ASP solver. Nevertheless, translational methods into ASP or SAT have the drawback of being very memory intensive since the whole theory has to be represented using propositional variables. Encodings that use for instance binary representations of integer variables lack propagation strength. To overcome these problems, *inca* (Drescher and Walsh 2012) translates constraints on the fly, that is, it relies upon lazy nogood generation, which is strongly inspired by lazy clause generation (Ohrimenko et al. 2009). It features a tight integration, profits from the learning capabilities of CDCL, and avoids the grounding bottleneck of eager techniques since only the currently interesting part of the theory is generated. *inca* concentrates on the support of various encodings and implements a propagator for linear and distinct constraints. Unfortunately, the basic vocabulary of these encodings has to be provided beforehand, not removing the grounding bottleneck for variables with large domains. Lazy variable generation (Thibaut and Stuckey 2009) overcomes this problem and is a state of the art technique in CP. Close work in the neighboring area of model expansion was done in the *idp* system (De Cat et al. 2014) using *minisatid* (De Cat et al. 2013) for combining lazy clause generation and lazy variable generation for handling linear constraints. Also, the constraint solver *chuffed*, the leading CP solver in the *minizinc* competition 2015, supports lazy clause and variable generation and has propagators for a set of global constraints.

We take this up to extend ASP with CP for tackling CASP problems with modern CP techniques. By extending the input language of *gringo* in a modular way, we enhance the modeling capabilities of ASP with linear constraints over integers and handle them with advanced hybrid search and propagation techniques. Our design goal is to have a tight integration, overcoming grounding and memory bottlenecks of translation-based approaches, while using the learning capabilities of CDCL algorithms. We integrated these techniques in *clasp* and *clingo* while preserving features like multi-threading, unsatisfiable core optimization, multi-objective optimization etc. We developed a propagator for linear constraints and are able to translate parts of the constraints beforehand. Furthermore, variables with huge domains are managed by introducing order atoms on the fly. Several dedicated preprocessing techniques improve our lazy nogood generation approach. Our empirical evaluation leads to the result that some techniques that are known to be crucial for translational approaches using the order encoding cannot be adopted easily. This concerns especially sorting and splitting of constraints, which has either no or even a negative effect on the performance of lazy nogood propagation. Other, more general techniques like equality processing, don't care propagation, and flattening of the objective function improve the performance in general. Another interesting result is that translating a subset of small constraints is beneficial over translating none or all. These techniques have allowed us to develop the modern CASP solver *clingcon*. It combines the first-order modeling language of ASP with the performance of state of the art CP solvers for handling constraints over integers. Also, making *clingcon* incremental, such that multi-shot solving can be used with *clingo*'s API, enables us to use CASP in reactive environments and thus opens up new application areas. Our software is open source and freely available as part of the potassco project.[35]

Future work. CASP is a useful paradigm to solve problems with resources, capacities, and fine-grained timing information. Its semantics has been extended in various ways, as for instance in bound founded ASP (Aziz et al. 2013) or default reasoning with constraints (Cabalar et al. 2016). The latter approach already presents a translator relying upon *clingcon* 3. This indicates that related approaches can take advantage of the development of CASP and its systems.

We plan to develop a translation option for converting a CASP problem into an (C)ASP problem by (partially) translating the constraints. The output can then be handled by other solvers than *clasp*. We also preserved special functionalities of the ASP solver *clasp* in order to use unsatisfiable core techniques (Andres et al. 2012) and multi-criteria optimization (Gebser et al. 2011) for integer variables. Also, domain-specific heuristics (Gebser et al. 2013) can be used in the encodings of CASP problems. However, all these features still need to be evaluated in the context of CASP. Furthermore, we want to use the ability to handle constraints over large domains to tackle complex planning problems (Balduccini et al. 2016). These often involve a fine grained handling of resources and timings and are a perfect area of application for CASP.

Acknowledgments. This work was partially funded by JSPS (KAKENHI 15K00099) and DFG (SCHA 550/9). We are grateful to Bart Bogaerts for his help with *minisatid*. A special thanks goes to Philipp Wanko for his comments, and of course, to Roland Kaminski for all his support!

[35] https://potassco.org

References

ANDRES, B., KAUFMANN, B., MATHEIS, O., AND SCHAUB, T. 2012. Unsatisfiability-based optimization in clasp. See Dovier and Santos Costa (2012), 212–221.

AZIZ, R., CHU, G., AND STUCKEY, P. 2013. Stable model semantics for founded bounds. *Theory and Practice of Logic Programming 13,* 4-5, 517–532.

BALDUCCINI, M. 2009. Representing constraint satisfaction problems in answer set programming. In *Proceedings of the Second Workshop on Answer Set Programming and Other Computing Paradigms (ASPOCP'09)*, W. Faber and J. Lee, Eds. 16–30.

BALDUCCINI, M. AND LIERLER, Y. 2013. Integration schemas for constraint answer set programming: a case study. *Theory and Practice of Logic Programming 13.*

BALDUCCINI, M., MAGAZZENI, D., AND MARATEA, M. 2016. PDDL+ planning via constraint answer set programming. *CoRR abs/1609.00030.*

BANBARA, M., GEBSER, M., INOUE, K., OSTROWSKI, M., PEANO, A., SCHAUB, T., SOH, T., TAMURA, N., AND WEISE, M. 2015. aspartame: Solving constraint satisfaction problems with answer set programming. In *Proceedings of the Thirteenth International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'15)*, F. Calimeri, G. Ianni, and M. Truszczyński, Eds. Lecture Notes in Artificial Intelligence, vol. 9345. Springer-Verlag, 112–126.

BARRETT, C., SEBASTIANI, R., SESHIA, S., AND TINELLI, C. 2009. Satisfiability modulo theories. See Biere et al. (2009), Chapter 26, 825–885.

BASELICE, S., BONATTI, P., AND GELFOND, M. 2005. Towards an integration of answer set and constraint solving. In *Proceedings of the Twenty-first International Conference on Logic Programming (ICLP'05)*, M. Gabbrielli and G. Gupta, Eds. Lecture Notes in Computer Science, vol. 3668. Springer-Verlag, 52–66.

BIERE, A., HEULE, M., VAN MAAREN, H., AND WALSH, T., Eds. 2009. *Handbook of Satisfiability.* Frontiers in Artificial Intelligence and Applications, vol. 185. IOS Press.

BRODSKY, A., Ed. 2013. *Proceedings of the Twenty-fifth IEEE International Conference on Tools with Artificial Intelligence (ICTAI'13).* IEEE Computer Society.

CABALAR, P., KAMINSKI, R., OSTROWSKI, M., AND SCHAUB, T. 2016. An ASP semantics for default reasoning with constraints. In *Proceedings of the Twenty-fifth International Joint Conference on Artificial Intelligence (IJCAI'16)*, R. Kambhampati, Ed. IJCAI/AAAI Press, 1015–1021.

CARRO, M. AND KING, A., Eds. 2016. *Technical Communications of the Thirty-second International Conference on Logic Programming (ICLP'16).* Vol. 52. Open Access Series in Informatics (OASIcs).

CRAWFORD, J. AND BAKER, A. 1994. Experimental results on the application of satisfiability algorithms to scheduling problems. In *Proceedings of the Twelfth National Conference on Artificial Intelligence (AAAI'94)*, B. Hayes-Roth and R. Korf, Eds. AAAI Press, 1092–1097.

DAVIS, M., LOGEMANN, G., AND LOVELAND, D. 1962. A machine program for theorem-proving. *Communications of the ACM 5*, 394–397.

DAVIS, M. AND PUTNAM, H. 1960. A computing procedure for quantification theory. *Journal of the ACM 7*, 201–215.

DE CAT, B., BOGAERTS, B., BRUYNOOGHE, M., AND DENECKER, M. 2014. Predicate logic as a modelling language: The IDP system. *CoRR abs/1401.6312.*

DE CAT, B., BOGAERTS, B., DEVRIENDT, J., AND DENECKER, M. 2013. Model expansion in the presence of function symbols using constraint programming. See Brodsky (2013), 1068–1075.

DOVIER, A. AND SANTOS COSTA, V., Eds. 2012. *Technical Communications of the Twenty-eighth International Conference on Logic Programming (ICLP'12).* Vol. 17. Leibniz International Proceedings in Informatics (LIPIcs).

DRESCHER, C. 2015. Conflict-driven constraint answer set solving. Ph.D. thesis, Computer Science and Engineering, Faculty of Engineering, UNSW.

DRESCHER, C. AND WALSH, T. 2010. A translational approach to constraint answer set solving. *Theory and Practice of Logic Programming 10,* 4-6, 465–480.

DRESCHER, C. AND WALSH, T. 2012. Answer set solving with lazy nogood generation. See Dovier and Santos Costa (2012), 188–200.

EITER, T., FINK, M., KRENNWALLNER, T., AND REDL, C. 2012. Conflict-driven ASP solving with external sources. *Theory and Practice of Logic Programming 12,* 4-5, 659–679.

FEYDY, T., SOMOGYI, Z., AND STUCKEY, P. 2011. Half reification and flattening. In *Proceedings of the Seventeenth International Conference on Principles and Practice of Constraint Programming (CP'11)*, J. Lee, Ed. Lecture Notes in Computer Science, vol. 6876. Springer-Verlag, 286–301.

GEBSER, M., KAMINSKI, R., KAUFMANN, B., OSTROWSKI, M., SCHAUB, T., AND WANKO, P. 2016a. Theory solving made easy with clingo 5. See Carro and King (2016), 2:1–2:15.

GEBSER, M., KAMINSKI, R., KAUFMANN, B., OSTROWSKI, M., SCHAUB, T., AND WANKO, P. 2016b. Theory solving made easy with clingo 5 (extended version). Available at `http://www.cs.uni-potsdam.de/wv/publications/`. Extended version of (Gebser et al. 2016a).

GEBSER, M., KAMINSKI, R., KAUFMANN, B., AND SCHAUB, T. 2011. Multi-criteria optimization in answer set programming. In *Technical Communications of the Twenty-seventh International Conference on Logic Programming (ICLP'11)*, J. Gallagher and M. Gelfond, Eds. Vol. 11. Leibniz International Proceedings in Informatics (LIPIcs), 1–10.

GEBSER, M., KAMINSKI, R., KAUFMANN, B., AND SCHAUB, T. 2012. *Answer Set Solving in Practice*. Synthesis Lectures on Artificial Intelligence and Machine Learning. Morgan and Claypool Publishers.

GEBSER, M., KAMINSKI, R., KAUFMANN, B., AND SCHAUB, T. 2014. *Clingo* = ASP + control: Preliminary report. In *Technical Communications of the Thirtieth International Conference on Logic Programming (ICLP'14)*, M. Leuschel and T. Schrijvers, Eds. Theory and Practice of Logic Programming, Online Supplement, vol. arXiv:1405.3694v1. Available at `http://arxiv.org/abs/1405.3694v1`.

GEBSER, M., KAMINSKI, R., OBERMEIER, P., AND SCHAUB, T. 2015. Ricochet robots reloaded: A case-study in multi-shot ASP solving. In *Advances in Knowledge Representation, Logic Programming, and Abstract Argumentation: Essays Dedicated to Gerhard Brewka on the Occasion of His 60th Birthday*, T. Eiter, H. Strass, M. Truszczyński, and S. Woltran, Eds. Lecture Notes in Artificial Intelligence, vol. 9060. Springer-Verlag, 17–32.

GEBSER, M., KAUFMANN, B., NEUMANN, A., AND SCHAUB, T. 2007. Conflict-driven answer set solving. In *Proceedings of the Twentieth International Joint Conference on Artificial Intelligence (IJCAI'07)*, M. Veloso, Ed. AAAI/MIT Press, 386–392.

GEBSER, M., KAUFMANN, B., OTERO, R., ROMERO, J., SCHAUB, T., AND WANKO, P. 2013. Domain-specific heuristics in answer set programming. In *Proceedings of the Twenty-Seventh National Conference on Artificial Intelligence (AAAI'13)*, M. desJardins and M. Littman, Eds. AAAI Press, 350–356.

GEBSER, M., KAUFMANN, B., AND SCHAUB, T. 2012. Multi-threaded ASP solving with clasp. *Theory and Practice of Logic Programming 12,* 4-5, 525–545.

GEBSER, M., OSTROWSKI, M., AND SCHAUB, T. 2009. Constraint answer set solving. In *Proceedings of the Twenty-fifth International Conference on Logic Programming (ICLP'09)*, P. Hill and D. Warren, Eds. Lecture Notes in Computer Science, vol. 5649. Springer-Verlag, 235–249.

GECODE TEAM. 2006. Gecode: Generic constraint development environment. Available from `http://www.gecode.org`.

GELFOND, M. AND LIFSCHITZ, V. 1988. The stable model semantics for logic programming. In *Proceedings of the Fifth International Conference and Symposium of Logic Programming (ICLP'88)*, R. Kowalski and K. Bowen, Eds. MIT Press, 1070–1080.

JANHUNEN, T., LIU, G., AND NIEMELÄ, I. 2011. Tight integration of non-ground answer set programming and satisfiability modulo theories. In *Proceedings of the First Workshop on Grounding and Transformation for Theories with Variables (GTTV'11)*, P. Cabalar, D. Mitchell, D. Pearce, and E. Ternovska, Eds. 1–13.

LIERLER, Y. AND SUSMAN, B. 2016. SMT-based constraint answer set solver EZSMT (system description). See Carro and King (2016), 1:1–1:15.

LIFSCHITZ, V. 2008. What is answer set programming? In *Proceedings of the Twenty-third National Conference on Artificial Intelligence (AAAI'08)*, D. Fox and C. Gomes, Eds. AAAI Press, 1594–1597.

MARQUES-SILVA, J. AND SAKALLAH, K. 1999. GRASP: A search algorithm for propositional satisfiability. *IEEE Transactions on Computers 48,* 5, 506–521.

MELLARKOD, V. AND GELFOND, M. 2008. Integrating answer set reasoning with constraint solving techniques. In *Proceedings of the Ninth International Symposium on Functional and Logic Programming (FLOPS'08)*, J. Garrigue and M. Hermenegildo, Eds. Lecture Notes in Computer Science, vol. 4989. Springer-Verlag, 15–31.

MELLARKOD, V., GELFOND, M., AND ZHANG, Y. 2008. Integrating answer set programming and constraint logic programming. *Annals of Mathematics and Artificial Intelligence 53,* 1-4, 251–287.

METODI, A., CODISH, M., AND STUCKEY, P. 2013. Boolean equi-propagation for concise and efficient SAT encodings of combinatorial problems. *Journal of Artificial Intelligence Research 46*, 303–341.

OHRIMENKO, O., STUCKEY, P., AND CODISH, M. 2009. Propagation via lazy clause generation. *Constraints 14,* 3, 357–391.

OSTROWSKI, M. 2017. Modern constraint answer set solving. Ph.D. thesis, University of Potsdam.

OSTROWSKI, M. AND SCHAUB, T. 2012. ASP modulo CSP: The clingcon system. *Theory and Practice of Logic Programming 12,* 4-5, 485–503.

ROSSI, F., VAN BEEK, P., AND WALSH, T., Eds. 2006. *Handbook of Constraint Programming*. Elsevier Science.

ROUSSEL, O. AND LECOUTRE, C. 2009. XML representation of constraint networks: Format XCSP 2.1. *CoRR abs/0902.2362*.

SCHULTE, C. AND TACK, G. 2005. Views and iterators for generic constraint implementations. In *Proceedings of the Eleventh International Conference on Principles and Practice of Constraint Programming (CP'05)*, P. van Beek, Ed. Lecture Notes in Computer Science, vol. 3709. Springer-Verlag, 118–132.

SIMONS, P., NIEMELÄ, I., AND SOININEN, T. 2002. Extending and implementing the stable model semantics. *Artificial Intelligence 138,* 1-2, 181–234.

SOH, T., INOUE, K., TAMURA, N., BANBARA, M., AND NABESHIMA, H. 2010. A SAT-based method for solving the two-dimensional strip packing problem. *Fundamenta Informaticae 102,* 3-4, 467–487.

TAMURA, N., BANBARA, M., AND SOH, T. 2013. Compiling pseudo-boolean constraints to SAT with order encoding. See Brodsky (2013), 1020–1027.

TAMURA, N., TAGA, A., KITAGAWA, S., AND BANBARA, M. 2009. Compiling finite linear CSP into SAT. *Constraints 14,* 2, 254–272.

THIBAUT, F. AND STUCKEY, P. 2009. Lazy clause generation reengineered. In *Proceedings of the Fifteenth International Conference on Principles and Practice of Constraint Programming (CP'09)*, I. Gent, Ed. Lecture Notes in Computer Science, vol. 5732. Springer-Verlag, 352–366.

THIFFAULT, C., BACCHUS, F., AND WALSH, T. 2004. Solving non-clausal formulas with DPLL search. In *Proceedings of the Tenth International Conference on Principles and Practice of Constraint Programming (CP'04)*, M. Wallace, Ed. Lecture Notes in Computer Science, vol. 3258. Springer-Verlag, 663–678.

WALSH, T. 2000. SAT versus CSP. In *Proceedings of the Sixth International Conference on Principles and Practice of Constraint Programming (CP'00)*, R. Dechter, Ed. Lecture Notes in Computer Science, vol. 1894. Springer-Verlag, 441–456.

ZHANG, L., MADIGAN, C., MOSKEWICZ, M., AND MALIK, S. 2001. Efficient conflict driven learning in a Boolean satisfiability solver. In *Proceedings of the International Conference on Computer-Aided Design (ICCAD'01)*. ACM Press, 279–285.