doi:10.1017/S1471068425100264

ASP-FZN: A Translation-Based Constraint Answer Set Solver[*]

THOMAS EITER, TOBIAS GEIBINGER AND NYSRET MUSLIU
TU Wien, Vienna, Austria
(*e-mails:* thomas.eiter@tuwien.ac.at, tobias.geibinger@tuwien.ac.at, nysret.musliu@tuwien.ac.at)

JOHANNES OETSCH
Jönköping University, Jönköping, Sweden
(*e-mail:* johannes.oetsch@ju.se)

TOBIAS KAMINSKI
Bosch Center for AI, Renningen, Germany
(*e-mail:* tobias.kaminski@de.bosch.com)

submitted 28 July 2025; revised 28 July 2025; accepted 29 July 2025

Abstract

We present the solver asp-fzn for Constraint Answer Set Programming (CASP), which extends ASP with linear constraints. Our approach is based on translating CASP programs into the solver-independent FlatZinc language that supports several Constraint Programming and Integer Programming backend solvers. Our solver supports a rich language of linear constraints, including some common global constraints. As for evaluation, we show that asp-fzn is competitive with state-of-the-art ASP solvers on benchmarks taken from past ASP competitions. Furthermore, we evaluate it on several CASP problems from the literature and compare its performance with clingcon, which is a prominent CASP solver that supports most of the asp-fzn language. The performance of asp-fzn is very promising as it is already competitive on plain ASP and even outperforms clingcon on some CASP benchmarks.

Keywords: answer set programming, constraint programming, integer programming

1 Introduction

Answer Set Programming (ASP) is a popular rule-based formalism for various AI applications and combinatorial problem-solving, where a problem is represented by an ASP program whose answer sets (models) represent the solutions, potentially also under

[*] This work was supported by funding from the Bosch Center for AI at Renningen, Germany. Tobias Geibinger is a recipient of a DOC Fellowship of the Austrian Academy of Sciences at the Institute of Logic and Computation at the TU Wien.

certain optimization criteria. Especially for modeling industrial problems, Constraint Answer Set Programming (CASP), which adds reasoning over linear constraints to ASP, proved to be quite effective, for example, for scheduling problems (Balduccini 2011; Geibinger *et al.* 2021).

While efficient CASP solvers are available, cf. the recent survey by Lierler (2023), they still often lag behind state-of-the-art Constraint Programming (CP) or Mixed Integer Programming (MIP) solvers for certain problem domains. CASP solvers are either based on dedicated algorithms or translations into related formalisms such as Satisfiability Modulo Theory (SMT). The latter approach is inspired by similar works for solving plain ASP programs, but has the downside that SMT solvers generally lack optimization features and are thus not applicable for many problems appearing in practice. This begs the question why, instead of targeting SMT, the translation is not aimed at FlatZinc (Nethercote *et al*. 2007), which is a solver-independent intermediate language that offers those lacking optimization features and works with many modern CP and MIP solvers as backend engines. The lack of such an approach was also noted by Lierler (2023).

To fill this gap, we present the CASP solver asp-fzn, which translates CASP programs into FlatZinc, thereby leveraging decades of CP and MIP solver engineering for efficient (optimal) solution finding. To support modern CASP encodings featuring not only linear constraints but also specific scheduling constraints and ASP constructs like variables, aggregates, choice, and disjunction, we utilize gringo's theory interface (Gebser *et al.* 2019; Kaminski *et al*. 2023) to obtain a simplified program format. Our approach then combines and extends ideas from translation-based ASP solving (Alviano and Dodaro 2016; Janhunen 2023) to create a FlatZinc representation encompassing all mentioned constructs. By the richness of FlatZinc, incorporating complex global constraints and hybrid optimization of both ASP weak constraints and objectives over linear variables is easy. Notably, those features are not yet fully supported by other state-of-the-art CASP solvers like clingcon (Banbara *et al*. 2017; Cabalar *et al*. 2023).

Our main contributions are briefly summarized as follows:

- We present a translation $Tr(P)$ of head-cycle-free CASP programs P into a low level constraint language, which can be parsed by several state-of-the-art CP and MIP solvers.
- Our translation extends and combines existing concepts from the literature and supports not only linear constraints but also choice rules, weight rules, disjunction, and optimization.
- We show that $Tr(P)$ captures all answer sets of P, with a one-to-one or many-to-one mapping to its models, depending on the presence of correspondence constraints.
- We introduce our solver asp-fzn, which implements the described translation and utilizes external grounding and a parametric backend solver for answer-set optimization.
- We evaluate asp-fzn using different backend solvers against state-of-the-art (C)ASP solvers, finding that it is competitive on plain ASP and outperforms clingcon on some CASP benchmarks.

The solver asp-fzn thus enables solving expressive (C)ASP programs via CP and MIP solvers, leveraging their strengths. As with SAT-based ASP solvers, this approach benefits from the substantial engineering behind these solvers, future advancements, and the decoupling of (C)ASP solving from specialized, maintenance-heavy algorithms.

2 Preliminaries

We consider propositional ASP (Brewka *et al.* 2011) with *programs* P that are sets of rules r of the form

$$H \leftarrow B \tag{1}$$

where H is the *head* of the rule and B its *body*, also denoted by $H(r)$ and $B(r)$, respectively; by $\mathcal{A}_P$ we denote the set of all propositional atoms occurring in P. We distinguish two types of rules: 1) *disjunctive rules* and 2) *choice rules*, where H has the form

$$a_1 \mid \cdots \mid a_n \quad (\textit{disjunctive head}) \tag{2}$$

$$\text{respectively } \{a_1, \ldots, a_m\} \quad (\textit{choice head}) \tag{3}$$

where all a_i are atoms. Intuitively, "$|$" stands for logical disjunction, that is, at least one of the atoms must hold, while for choice, any number of a_i can be true if H is true. A disjunctive rule is a *constraint rule* if $H(r) = \emptyset$ and a *normal rule* if $|H(r)| = 1$.

Furthermore, we consider two types of rule bodies: 1) *normal* rule bodies of the form

$$b_1, \ldots, b_k, \neg b_{k+1}, \ldots, \neg b_n \tag{4}$$

where all b_i are atoms, $\neg$ is negation as failure, and "," is conjunction, and 2) *weighted* rule bodies

$$l \leq \{b_1 : w_1, \ldots, b_k : w_k, \neg b_{k+1} : w_{k+1}, \ldots, \neg b_n : w_n\} \tag{5}$$

where all b_i are atoms, all w_i are integer *weights*, and l is the integer *lower bound*; we let $B^+(r) = \{b_1, \ldots, b_k\}$ and $B^-(r) = \{b_{k+1}, \ldots, b_n\}$.

By slight abuse of notation, $a \in H(r)$ denotes that atom a occurs in $H(r)$ and $l \in B(r)$ that literal l, that is, an atom or its negation, occurs in $B(r)$. We further let w_b^r denote the weight of atom b in the body of rule r, let $\top$ denote an empty conjunction, and let $\bot$ denote an empty rule head.

Example 1.
Consider the program $P_1 = \{\ \{a, b\} \leftarrow c,\ \bot \leftarrow 3 \leq \{a : 1, b : 2\},\ c \leftarrow \neg d\ \}$. *The first rule of* P_1 *is a choice rule with normal body, the second rule is a constraint rule with a weighted body, and the last rule is a normal rule.*

Semantics. An *interpretation* of a program P is a set $I \subseteq \mathcal{A}_P$ of atoms, which satisfies a disjunctive head (2) if $a_i \in I$ for some $i \in [1, m]$, and satisfies every choice rule head (3).

Given a rule r and an interpretation I, $I \models H(r)$ denotes that I satisfies the head of r. Satisfaction of the body $B(r)$ by I, denoted $I \models B(r)$, is as follows: 1) for a normal rule body (4), $b_i \in I$ for every $i \in [1, k]$ and $b_j \notin I$ for every $j \in (k, n]$ must hold; 2) for a weighted rule body (5), the following linear inequality must hold: $l \leq \sum_{i \in [1,k], b_i \in I} w_i + \sum_{j \in (j,n], b_j \notin I} w_j$.

An interpretation I satisfies a rule r, denoted $I \models r$, whenever $I \models B(r)$ implies $I \models H(r)$ and I is a model of program P, denoted $I \models P$, if $I \models r$ for all $r \in P$.

Answer sets. The *(FLP) reduct* P^I of program P w.r.t. interpretation I is the program containing, for each $r \in P$ s.t. $I \models B(r)$, the following rules: (1) if r is disjunctive,

$H(r) \leftarrow B$, and (2) if r is a choice rule, for each $a \in H(r)$ the rule $a \leftarrow B^+(r)$ if $B(r)$ is normal and $a \leftarrow l' \leq \{b_1 : w_1, \ldots, b_k : w_k\}$ if $B(r)$ is a weighted body (4), where $l' = max(0, l - \sum_{j \in (k,n], b_j \notin I} w_j)$.

Finally, an interpretation I is an *answer set* of program P if I is a $\subseteq$-minimal model of P^I. The set of all answer sets of P is denoted by $AS(P)$.

Example 2.
Program P_1 from Example 1 has $AS(P_1) = \{\{c\}, \{c, a\}, \{c, b\}\}$.

We allow programs P to contain also a single *minimization* statement (Priority levels can be added and compiled to this form using known techniques):

$$min\ a_1 : w_1, \ldots, a_k : w_k, \neg a_{k+1} : a_{k+1}, \ldots, \neg b_n : w_n \tag{6}$$

The *cost* of interpretation I is $c_P(I) = \sum_{i \in [1,k], a_i \in I} w_i + \sum_{j \in (k,n], a_j \notin I} w_j$ and 0 if P has no minimization. An answer set I of P is *optimal* if $c_P(I)$ is minimal over $AS(P)$.

2.1 Constraint answer set programming

We next introduce *linear constraints and variables* in our programs, thus turning to *CASP*. We consider a countable set $\mathcal{V}$ of linear variables. Each $v \in \mathcal{V}$ has a *domain* $D(v)$, that is, assumed to be an integer range, which defaults to $[-\infty, +\infty]$; it can be restricted by a *domain constraint* of the form

$$v \in [l, u] \tag{7}$$

where l and u, $l \leq u$, are integer lower and upper bounds. In general, bounding the linear variables is not required but the CASP solver might infer bounds or fallback to some default values.

A *linear constraint* is of the form

$$a \leftrightarrow v_1 \cdot w_1 + \cdots + v_n \cdot w_n \circ g \tag{8}$$

where a is an atom, each v_i is a linear variable, each w_i and g are integer constants, and $\circ \in \{<, >, =, \neq, \leq, \geq\}$ is a comparison operator. Intuitively, a is constrained to the truth value of the linear constraint. Syntactically, a can appear in the bodies of standard ASP rules (1). For any CASP program P, we denote by $\mathcal{V}_P$ and $\mathcal{A}_P{}^{lin}$ the sets of all linear variables and all propositional atoms occurring in linear constraints of P, respectively.

We additionally allow a CASP to contain global constraints. An *alldifferent* constraint is of the form

$$\&distinct\{v_1, \ldots, v_n\} \tag{9}$$

where each v_i is a linear variable and all are constrained to be pair-wise different. A *cumulative* constraint is of the form

$$\&cumulative\{(s_1, l_1, r_1), \ldots, (s_n, l_n, r_n)\} \leq g \tag{10}$$

where s_i is a linear variable representing the start of each interval, l_i is a linear variable representing the length, r_i is a linear variable denoting the resource usage, and g is an integer bound. The constraint then enforces that at each time point, the sum of the resource usages of the overlapping intervals does not exceed g. A global *disjoint* constraint is of form $\&disjoint\{(s_1, l_1), \dots, (s_n, l_n)\}$ and can be seen as a special case of a constraint (10) where r_i and g are assumed to be 1.

Semantics. An *extended (e-)* interpretation for a CASP program P is a tuple $\mathcal{I} = \langle I, \delta \rangle$ where I is a set of propositional atoms and $\delta : \mathcal{V}_P \to \mathbb{Z}$ is an *assignment* of integers to linear variables $\mathcal{V}_P$. Satisfaction $\mathcal{I} \models \phi$, where ϕ is a head, body, rule, program etc., is defined as above via I.

An e-interpretation $\mathcal{I} = \langle I, \delta \rangle$ is a *constraint answer set* of P if (1) I is an answer set of $P \cup \{\{a\} \leftarrow \mid a \in \mathcal{A}_P{}^{lin}\}$, (2) for each domain constraint (7) in P, $\delta(v) \in [l, u]$, and (3) for each linear constraint (8) in P, $a \in I$ if $\sum_{1 \le i \le n} \delta(v_i) \cdot w_i \circ g$. By slight abuse of notation we also use $AS(P)$ to refer to the constraint answer sets of a CASP program P.

Example 3
(Ex. 1 cont'd).Let $P_2 = P_1 \cup \{x \in [0, 2],\quad y \in [0, 1],\quad d \leftrightarrow x \cdot 1 + y \cdot 1 \neq 3\}$. *Clearly,* P_2 *is a CASP program with* $AS(P_2) = \{\, \langle\{c\}, \{(x, 2), (y, 1)\}\rangle,\ \langle\{b, c\}, \{(x, 2), (y, 1)\}\rangle, \langle\{a, c\}, \{(x, 2), (y, 1)\}\rangle, \langle\{d\}, \{(x, 0), (y, 0)\}\rangle, \langle\{d\}, \{(x, 1), (y, 0)\}\rangle, \langle\{d\}, \{(x, 2), (y, 0)\}\rangle, \langle\{d\}, \{(x, 1), (y, 1)\}\rangle, \langle\{d\}, \{(x, 0), (y, 1)\}\rangle \,\}$.

For CASP programs, we allow minimization over the linear variables with statements

$$min\ v_1 \cdot w_1 + \cdots + v_n \cdot w_n \tag{11}$$

where each v_i is a linear variable and each w_i is an integer constant. The cost $c_P(\mathcal{I})$ of an e-interpretation $\mathcal{I}$ of a CASP program P is the sum of the costs determined by statements (6) and (11), and optimal constraint answer sets are, *mutatis mutandis*, analogous to optimal answer sets.

3 Supported models and ranked interpretations

Prior to the translation, we introduce a few auxiliary concepts. The *positive dependency graph* of a (C)ASP program P is $DG^+_P = (V, E)$ with nodes $V = \mathcal{A}_P$ and edges $(a, b) \in E$. for all atoms a, b s.t. $a \in H(r)$ and $b \in B^+(r)$ for some rule $r \in P$. A program P is *tight* if DG^+_P is acyclic; a rule $r \in P$ is *locally tight* if $H(r) \cap B^+(r) = \emptyset$. We denote for $a \in \mathcal{A}_P$ by $SCC_P(a)$ its strongly connected component (SCC) in DG^+_P, which is *non-trivial* if $|SCC_P(a)| > 1$. A program P is *head-cycle free (HCF)* if every rule $r \in P$ and distinct $a \neq b \in H(r)$ fulfill $b \notin SCC_P(a)$.

Clearly, a tight program has no non-trivial SCCs and are HCF, while a non-tight program may or may not be HCF. In the sequel, we assume that all programs are HCF; while this excludes some programs, it still allows us with minimization to embrace the class of NP-optimization problems[1] as follows from (Eiter *et al.*, 2007), and thus most problems appearing in practice.

Recall that for an ASP program P, an interpretation I is a *supported model* of P if (1) $I \models P$ and (2) for each $a \in I$ some rule $r \in P$ exists such that $I \models B(r)$, $a \in H(r)$, and $H(r) \cap I = \{a\}$ if r is disjunctive. For tight ASP programs, supported models and answer sets coincide (Erdem and Lifschitz 2003). For non-tight HCF programs, we consider *ranked supported models* as follows.

[1] see https://complexityzoo.net/Complexity˙Zoo

We assume that $\mathcal{V}_P$ includes for each atom $a \in \mathcal{A}_P$ a variable ℓ_a not occurring in P; intuitively, it denotes the *rank* (or *level*) of a. An e-interpretation $\mathcal{I} = \langle I, \delta \rangle$ is *ranked*, if for each $a \in \mathcal{A}_P$, $\delta(\ell_a) = \infty$ if $a \notin I$ and $\delta(\ell_a) < \infty$ otherwise. A rule r *supports* atom $a \in I$, if $a \in H(r)$, $H(r) \cap I = \{a\}$ if r is disjunctive, and $B(r)$ fulfills: 1) if $B(r)$ is normal (form (4)), (i) $\delta(\ell_{b_i}) < \delta(\ell_a)$ for each $i \in [1, k]$ and (ii) $b_j \notin I$ for each $j \in (k, n]$ and 2) if $B(r)$ is a weighted rule body,

$$l \leq \sum_{b \in B^+(r), \delta(\ell_b) < \delta(\ell_a)} w_b^r + \sum_{b \in B^-(r), b \notin I} w_b^r. \tag{12}$$

Definition 1.
A ranked supported model of program P is a ranked interpretation $\mathcal{I} = \langle I, \delta \rangle$ of P such that $\mathcal{I} \models P$ and each $a \in I$ is supported by some rule $r \in P$.

We then obtain:

Proposition 1.
For every HCF program P, $I \in AS(P)$ if $\langle I, \delta \rangle$ is a ranked supported model of P for some δ.

We can refine this characterization by considering the modular structure of answer sets along the SCCs. A ranked interpretation $\langle I, \delta \rangle$ of program P is *modular*, if each $a \in I$ fulfills $\delta(\ell_a) \leq |SCC_P(a)|$; hence true atoms in trivial components must have rank 1. We say a rule r *scc-supports* $a \in I$ by changing in "r supports a" above for $B(r)$ condition (i) in case 1) to "$b_i \in I$ for each $i \in [1, k]$ where $\delta(\ell_{b_i}) < \delta(\ell_a)$ if $b_i \in SCC_P(a)$," and condition (12) in case 2) to

$$l \leq \sum_{b \in B^+(r) \setminus SCC_P(a)} w_b^r + \sum_{b \in B^+(r) \cap SCC_P(a), \delta(\ell_{b_i}) < \delta(\ell_a)} w_b^r + \sum_{b \in B^-(r) \setminus I} w_b^r,$$

and define scc-supported models analogous to supported models. We then can show:

Proposition 2.
For every HCF program P, $I \in AS(P)$ if $\langle I, \delta \rangle$ is a modular ranked scc-supported model of P for some level assignment δ.

4 Translation

In this section, we describe our translation of a (C)ASP program P into a constraint program. We assume that the considered program adheres to the following property.

Definition 2.
A HCF program P is called partially shifted if every rule $r \in P$ with a weighted body $B(r)$ fulfills either $|H(r)| \leq 1$ or $H(r) \cap SCC_P(a) = \emptyset$ for every $a \in B^+(r)$.

The property is named so because any HCF program can be transformed into partially shifted form by applying the well-known *shifting* operation (Ben-Eliyahu and Dechter 1994) to the violating rules, resulting in two rules that satisfy the property.

For CASP programs, the translation simply includes the theory atoms as reified constraints and the domain constraints are used as bounds of the introduced variables. If there are no bounds, we simply declare the variables as integer and delegate the handling of unbounded variables to the underlying FlatZinc solver. Minimization statements must be combined into a single objective, which is trivial in absence of priority levels. For priority level minimization, we rely on well-known methods to compile them away.

4.1 Translation constraints

The translation, $\mathit{Tr}(P)$ consists of several groups of constraints, which encode different aspects of an answer set of a (C)ASP program P:

- ranking constraints $\mathit{TrRk}(P)$, which encode the level ranking constraint;
- rule body constraints $\mathit{TrBd}(r)$, which encode the satisfaction of rules bodies;
- rule head constraints $\mathit{TrHd}(r)$, which must be satisfied when rule bodies fire; and
- supportedness constraints $\mathit{TrSupp}(P)$, ensuring that true atoms are supported.

The complete translation for a program $\mathit{Tr}(P)$ is then given by

$$\mathit{Tr}(P) = \mathit{TrRk}(P) \cup \textstyle\bigcup_{r\in P} \mathit{TrRule}(r) \cup \mathit{TrSupp}(P)\,,$$

where $\mathit{TrRule}(r) = \mathit{TrBd}(r) \cup \mathit{TrHd}(r)$ is the combined body and head translation of r.

Ranking constraints $\boldsymbol{\mathit{TrRk}(P)}$. First, we introduce some auxiliary atoms to handle the level ranking constraints, which follows the formulation given by Janhunen (2023). Note that we assume that there are no tautological rules, that is, DG_P^+ has no self-loops.

For each atom a such that $|SCC(a)| > 1$, we introduce an integer variable ℓ_a with domain $[1, |SCC(a)| + 1]$ and add the following reified constraint to the translation:

$$\ell_a \leq |SCC(a)| \leftrightarrow a \tag{13}$$

The constraint enforces that atom a has rank $|SCC(a)| + 1$ if a is set to false. Now, for all $b \in SCC(a)$ such that DG_P^+ has an edge (a, b), we add a boolean auxiliary variable $\mathit{dep}_{a,b}$ and

$$\ell_a - \ell_b \geq 1 \;\leftrightarrow \mathit{dep}_{a,b} \tag{14}$$

which ensures that $\mathit{dep}_{a,b}$ is true if a has higher rank than b. The rank defined by these constraints is not *strict*, i.e., an answer set may have multiple rankings. To enforce strictness, we add

$$\ell_a - \ell_b \geq 2 \;\leftrightarrow y_{a,b} \tag{15}$$

$$a \wedge b \wedge y_{a,b} \;\leftrightarrow \mathit{gap}_{a,b} \tag{16}$$

where $\mathit{gap}_{a,b}$ is a Boolean variable indicating a gap in the ranks of true atoms a and b.

We denote the ranking constraints (13)–(16) by $\mathit{TrRk}(P)$; if P is tight, $\mathit{TrRk}(P) = \emptyset$.

Body translation $\boldsymbol{\mathit{TrBd}(r)}$. Next, for each $r \in P$, we perform a body translation $\mathit{TrBd}(r)$. Suppose first that r is a constraint. If $B(r)$ is normal, that is, of form (4), then we add the clause

$$\textstyle\bigvee_{b\in B^+(r)} \neg b \vee \bigvee_{b\in B^-(r)} b\,, \tag{17}$$

whereas if $B(r)$ is weighted (5), we add the constraint

$$\textstyle\sum_{b \in B^+(r)} b \cdot w^r_b + \sum_{b \in B^-(r)} \neg b \cdot w^r_b \ \leq \ l - 1\,. \tag{18}$$

Note that this is a pseudo-Boolean constraint, which our intended formalism does not support, and likewise Boolean variables in linear constraints. To circumvent this, we introduce new 0-1 integer variables for each literal and link their values; for better readability, we will leave this implicit. We similarly use auxiliary variables for negated atoms in conjunctions and leave this also implicit.

If r is not a constraint, we divide $H(r)$ into $T = \{a \in H(r) \mid SCC_P(a) \cap B^+(r) = \emptyset\}$ and $H(r) \setminus T$, where T are the head atoms that are locally tight. If $T \neq \emptyset$, we perform the standard Clark's completion (Clark 1977) to r, that is, if $B(r)$ is normal, we add

$$\textstyle\bigwedge_{b \in B^+(r)} b \wedge \bigwedge_{b \in B^-(r)} \neg b \ \leftrightarrow \ bd_r\,; \tag{19}$$

and if $B(r)$ is weighted, we add

$$\textstyle\sum_{b \in B^+(r)} b \cdot w^r_b + \sum_{b \in B^-(r)} \neg b \cdot w^r_b \geq l \leftrightarrow \ bd_r\,. \tag{20}$$

Furthermore, for each $a \in T$ such that $|SCC_P(a)| > 1$, we add the following constraint, which enforces that a has rank 1 if both a and bd_r are true, where $s_a = |SCC_P(a)| + 1$:

$$s_a \cdot bd_r \ + s_a \cdot a + \ 1 \cdot \ell_a \ \leq \ 2 \cdot s_a + 1\,, \tag{21}$$

and for each $a \in H(r) \setminus T$, we add constraints as follows: for a normal $B(r)$ of form (4),

$$\bigwedge_{b \in B^+(r) \setminus SCC_P(a)} b \ \wedge \bigwedge_{b \in B^+(r) \cap SCC_P(a)} dep_{a,b} \ \wedge \bigwedge_{b \in B^-(r)} b \ \leftrightarrow \ bd^a_r \tag{22}$$

$$\neg bd^a_r \ \vee \bigvee_{b \in B^+(r) \cap SCC_P(a)} \neg gap_{a,b}, \tag{23}$$

whereas for a weighted $B(r)$ of form (5), we add

$$\sum_{b \in B^+(r) \setminus SCC_P(a)} b \cdot w^r_b \ + \sum_{b \in B^-(r)} \neg b \cdot w^r_b \ \geq \ l \ \leftrightarrow \ ext^a_r \tag{24}$$

$$\sum_{b \in B^+(r) \setminus SCC_P(a)} b \cdot w^r_b \ + \sum_{b \in B^+(r) \cap SCC_P(a)} dep_{a,b} \cdot w^r_b + \sum_{b \in B^-(r)} \neg b \cdot w^r_b \ \geq \ l \ \leftrightarrow \ int^a_r \tag{25}$$

$$\sum_{b \in B^+(r) \setminus SCC_P(a)} b \cdot w^r_b \ + \sum_{b \in B^+(r) \cap SCC_P(a)} gap_{a,b} \cdot w^r_b + \sum_{b \in B^-(r)} \neg b \cdot w^r_b \ \leq \ l - 1 \ \leftrightarrow \ aux^a_r \tag{26}$$

$$ext^a_r \vee aux^a_r \vee \neg int^a_r \tag{27}$$

$$s_a \cdot ext^a_r \ + s_a \cdot a + \ 1 \cdot \ell_a \ \leq \ 2 \cdot s_a + 1 \tag{28}$$

$$ext^a_r \vee int^a_r \ \leftrightarrow \ bd^a_r. \tag{29}$$

Overall, the rule body translation follows the intuition of the original completion by Clark (1977). Namely, we introduce an auxiliary variable for each rule and constrain it to be true if the rule body is true. For each head atom a from the SCC of some body atom, we follow the approach by Janhunen (2023) and introduce an auxiliary atom bd_a^r for the pair of a and the rule body of r. The atom bd_a^r is set true exactly when the rule body "fires" without need of cyclic support, which is achieved by considering the dependency variables instead of the atoms, cf. (22). For weighted rule bodies, we follow Janhunen (2023) and introduce auxiliary variables for external (24) and internal (25) support of a rule body and a head atom. The former can be seen as the fact that the rule body fires regardless of any atoms in the SCC of the head atom, while the latter expresses rule firing despite some potentially cyclic dependencies. Constraint (29) defines an auxiliary variable denoting that the rule supports the head atoms, which is true whenever internal or external support exists. The constraints (21), (23), (27), and (28) ensure a strict ranking, that is, no gaps in the level mapping.

Head translation $\boldsymbol{TrHd}(r)$. To capture the semantics of a rule r, that is, if $B(r)$ holds then $H(r)$ hold as well, we need further constraints in the translation $TrHd(r)$.

For each $a \in H(r)$, we use a new Boolean variable sp_r^a to denote that r supports a. Suppose first r is a disjunctive rule and $|H(r)| > 1$. Recall that by our assumption, every $a \in H(r)$ is locally tight, so we only need to consider the single body variable bd_r.

Inspired by Alviano and Dodaro's (2016) disjunctive completion, we add for each $a_i \in H(r)$:

$$bd_r \wedge \bigwedge\nolimits_{a_j \in H(r), i \neq j} \neg a_j \quad \leftrightarrow \quad sp_r^a \tag{30}$$

Furthermore, we add the following clause ensuring that the rule is satisfied:

$$\bigvee\nolimits_{a \in H(r)} a \ \vee \neg bd_r \tag{31}$$

Otherwise, r is a choice rule or $|H(r)| = 1$. For each $a \in H(r)$ we add the constraint

$$sp_r^a \leftrightarrow bd_r \text{ if } SCC_P(a) \cap B^+(r) = \emptyset \tag{32}$$

$$\text{and} \qquad sp_r^a \leftrightarrow bd_r^a \text{ otherwise.} \tag{33}$$

Note that these constraints define the support variables as the respective rule bodies, and thus would make them redundant. However, we keep them to ease readability and for formulating further constraints. Furthermore, if r is not a choice rule, we add:

$$sp_r^a \rightarrow a \tag{34}$$

Supportedness constraints $\boldsymbol{TrSupp(P)}$. It remains to encode the supportedness condition of a model. This is achieved by adding for each $a \in \mathcal{A}_P \setminus {\mathcal{A}_P}^{lin}$ the following clause to $TrSupp(P)$:

$$\bigvee\nolimits_{r \in P, a \in H(r)} sp_r^a \vee \neg a \tag{35}$$

4.2 Correctness

That $Tr(P)$ captures the answer sets of a CASP program P faithfully in a 1-1 correspondence is shown in several steps. We view e-interpretations as models of $Tr(P)$ with the usual semantics. The following lemma is useful (cf. Def. 2 for partially shifted programs).

Lemma 1.
For every partially shifted HCF program P, if $\langle I, \delta \rangle \models Tr(P)$ then $\langle I \cap \mathcal{A}_P, \delta' \rangle$ is a modular ranked scc-supported model of P, where $\delta'(\ell_a) = 1$ for $a \in I$ s.t. $|SCC_P(a)| = 1$ and $\delta'(\ell_a) = \infty$ for $a \in \mathcal{A}_P \setminus I$.

Based on this lemma and Proposition 2, we obtain that the translation is sound.

Theorem 1
(Soundness of $Tr(P)$). For every partially shifted HCF program P, if $\langle I, \delta \rangle \models Tr(P)$ then $\langle I', \delta' \rangle \in AS(P)$, where $I' = I \cap \mathcal{A}_P$ and $\delta'(v) = \delta(v)$ for each $v \in \mathcal{V}_P$.

Conversely, we show also completeness.

Theorem 2
(Completeness of $Tr(P)$). For every partially shifted HCF program P and answer set $\langle I, \delta \rangle$ of P, there exists some e-interpretation $\mathcal{I}' = \langle I', \delta' \rangle$ s.t. $I' \cap \mathcal{A}_P = I \cap \mathcal{A}_P$, $\delta'(v) = \delta(v)$ for $v \in \mathcal{V}_P$, and $\mathcal{I}' \models Tr(P)$.

Theorems 1 and 2 establish a many-to-one mapping between the models of the translation and the answer sets of the program. That the mapping is in fact 1-1 is achieved through *correspondence constraints* given by (15), (16), (21), (23), (26), (27), (28), and the gap variables, which – as for Janhunen (2023) – ensure that the level mapping is *strict*, that is, has no gaps and starts at 1.

Lemma 2.
Suppose P is a partially shifted HCF program and $\mathcal{I} = \langle I, \delta \rangle$, $\mathcal{I}' = \langle I', \delta' \rangle$ are models of $Tr(P)$. Then $I \cap \mathcal{A}_P = I' \cap \mathcal{A}_P$ implies $\delta(\ell_a) = \delta'(\ell_a)$ for every $a \in \mathcal{A}_P$.

Theorem 3
(1-1 model correspondence between P and $Tr(P)$). For a partially shifted HCF program P, $AS(P)$ corresponds 1-1 to the models of $Tr(P)$.

For the implementation and the experiments, we also consider a non-strict version of the translation without the mentioned constraints, where Theorem 3 does not hold.

5 Implementation

The translation $Tr(P)$ is available via the tool asp-fzn, which is implemented in Rust[2] the source code is online accessible.[3] As mentioned above, $Tr(P)$, as described, is not in the Integer Programming *standard form* (Wolsey 2021). However, using well-known transformations and 0-1 variables instead of Booleans, it can be easily cast into this form.

The asp-fzn tool translates a given CASP program P into a FlatZinc (Nethercote *et al.* 2007) theory that has corresponding models. Program P can be either in ASPIF format (Kaminski *et al.* 2023) as produced by gringo or as a non-ground ASP program, which is then passed on to gringo for grounding. The FlatZinc theory can then be processed externally or relayed by asp-fzn via an interface to MiniZinc with a back-end solver as a parameter. Note that we do no preprocessing of the given ASPIF input,

[2] https://www.rust-lang.org/
[3] https://www.kr.tuwien.ac.at/systems/asp-fzn/

```
> cat example.lp
{a;b} :- c.
:- 3 <= #sum{1: a; 2: b}.
c :- not d.
&dom{ 0..2 } = x.
&dom{ 0..1 } = y.
d :- &sum{ x ; y } != 3.
val(x,V) :- &sum{ x } = V, V = 1..2.
val(y,V) :- &sum{ y } = V, V = 1..1.
```

```
> asp-fzn -s cp-sat -a example.lp

d val(y,1)                      c b val(y,1) val(x,2)
----------                      ----------
d val(y,1) val(x,1)             d val(x,2)
----------                      ----------
c val(y,1) val(x,2)             d
----------                      ----------
c a val(y,1) val(x,2)           d val(x,1)
----------                      ----------
```

Fig 1. Running example (left) solved with asp-fzn (dashed lines separate answer sets).

as we generally expect the grounder (for us, gringo), to handle this step and investigating further preprocessing is a topic of future work.

The tool supports linear constraints similar to the gringo-based CASP solver clingcon (Banbara *et al.* 2017), but expects them to occur in rule bodies, and further several global constraints, viz. *alldifferent*, *disjunctive*, and *cumulative* constraints. As for clingcon, these constraints are specified via gringo's theory interface (Kaminski *et al.* 2023); see Appendix A for theory definitions. Minimization objectives over the linear variables are akin to those in clingcon, yet asp-fzn allows to freely mix such objectives with plain weak constraints, resp. minimization objectives, in ASP.

The asp-fzn tool can be run via command line:

```
> asp-fzn [OPTIONS] [INPUT_FILES]...
```

A complete description of the arguments can be found in the appendix or online. Essentially, asp-fzn can be used either as a pure translation tool to convert ASPIF read from stdin into FlatZinc (optionally including an output specification which can be given to MiniZinc), or as a solver by specifying a backend solver for MiniZinc, which must be installed on the system. If a MIP solver is used, the translation output is in standard form and no further linearization is needed. By default, asp-fzn interprets input ASP files as non-ground programs and uses gringo to first ground them.

Example 4.
Listing 1 shows the CASP program P_2 from Ex. 3 in the language of gringo with the asp-fzn theory definition and the output set to enumerate all answer sets.

6 Experiments

We now demonstrate the effectiveness of asp-fzn on benchmark problems. All experiments were run on a cluster with 10 nodes, each having 2 Intel Xeon Silver 4314 (16 cores @ 2.40 GHz, 24 MB cache, no hyperthreading, 2 cores reserved for system, each core can use 1 MB L3 cache max.), running Ubuntu 22.04 (Kernel 5.15.0-131-generic), with memory limit 30 GB and 20 min timeout. All encodings, instances, and logs are available at https://doi.org/10.5281/zenodo.16267414.

6.1 ASP benchmarks

We compare asp-fzn 0.1.0 with ASP solvers clingo 5.7.1 (Gebser *et al.* 2019) and DLV 2.1.0 (Alviano *et al.* 2017) on benchmarks from ASP competitions (Alviano *et al.* 2013;

Calimeri *et al.* 2014; Calimeri *et al.* 2016). As backend solvers for asp-fzn, we used the MIP solver Gurobi 12.0.1 (Gurobi Optimization, LLC, 2025) and CP solvers CP-SAT 9.12.4544 from Google OR-Tools (Perron *et al.* 2023) and Chuffed 0.13.2 (Chu 2011).

Both CP-SAT and Chuffed are *lazy-clause generation* based, which is a method taken from SMT and has been highly effective for CP solving. In particular, CP-SAT has won the gold medal in the MiniZinc Challenge[4] for the last years. Gurobi on the other hand is a state-of-the-art, proprietary MIP solver, which has a MiniZinc interface. We ran all solvers using default settings, except for CP-SAT (interleaved search enabled). For asp-fzn, we used gringo 5.7.1 for grounding and MiniZinc 2.9.2 (Nethercote *et al.* 2007) to interface Gurobi and for output formatting, and we considered two settings: the strict translation $Tr(P)$ with a 1-1 mapping between the models of $Tr(P)$ and $AS(P)$, and the non-strict many-to-one variant.

We included both decision and optimization problems in the benchmark, listed in Table 1, with 31 problems and 772 instances in total. We used the encodings from the competition, but replaced in few some parts with modern constructs like choice rules. Note that the decision variants of all problems, except *StableMarriage*, are NP-hard and several encodings are non-tight.

Table 2 presents the comparison of asp-fzn with clingo and DLV, and cactus plots can be found in Appendix A. Here $\mathit{Score1} = \sum_{i=1}^{31} c_i/n_i * 100$ where c_i is the number of closed instances of domain D_i, that is, shown to be (un)satisfiable for type d resp. optimal for type o; the maximum score is 3100. *Score2* measures the best performers, by $\mathit{Score2} = \sum_{i=1}^{31} b_i/n_i * 100$, where b_i is the number of instances from D_i where the solver either closed the instance or found a solution of best value among all solvers.

Lastly, $\mathit{Score3} = \sum_{i=1}^{31} t_i/n_i$ is the *PAR10* score, where t_i is the time the solver took to complete instance i respectively 10×1200 if the solver did not complete the instance. Hence, here a lower number is better.

In single-threaded mode, clingo performs best on *Score1*, but asp-fzn with CP-SAT as backend is trailing closely behind, beating DLV. Under the non-strict translation, asp-fzn performs slightly better on *Score1* and significantly better on *Score2* . Furthermore, clingo also has the best *Score3*, indicating it is also closing most instances quicker than the rest; however, asp-fzn with CP-SAT under the non-strict translation is only 1.37% worse than clingo. Gurobi and Chuffed as backends perform worse than CP-SAT, but the non-strict variant is also better here. This difference between strict and non-strict variants is similar to previous observations for translation-based ASP solving (Janhunen *et al.* 2009). It seems non-strictness does not interfere with search-tree pruning.

For space reasons, we cannot give a detailed breakdown of the results over the particular problem domains, but unsurprisingly asp-fzn performs worse than clingo mostly on domains which are non-tight or feature heavy usage of disjunctions. An exception here is the Traveling Salesperson Problem where asp-fzn using CP-SAT or Gurobi outperforms clingo. Except for a few further non-tight domains, like Bayesian Network Learning and Systems Synthesis, Gurobi achieves worse results than CP-SAT as a backend solver.

Since clingo, Gurobi, and CP-SAT support parallel solving, we ran the benchmark on them using 8 threads. Again, clingo was best, cf. Table 2; while asp-fzn performed better

[4] https://www.minizinc.org/challenge/

Table 1. *ASP problems, n instances, type $\boldsymbol{T}$ = (o)ptimization | (d)ecision, (∗) non-tight*

Problem Domain	n	T	Problem Domain	n	T	Problem Domain	n	T
BayesianNL∗	60	*o*	Knight Tour With Holes∗	20	*d*	Sokoban	20	*d*
Bottle Filling Problem	20	*d*	Labyrinth∗	20	*d*	Solitaire	20	*d*
Combined Configuration∗	20	*d*	MarkovNL∗	60	*o*	*o* Stable Marriage	20	*d*
Connected Maximum-	20	*o*	MaxSAT	20	*o*	Steiner Tree∗	20	*o*
Density Still Life∗			Maximal Clique Problem	20	*o*	Supertree	60	*o*
Crew Allocation	52	*d*	Nomistery	20	*d*	System Synthesis∗	20	*o*
Crossing Minimization	20	*o*	Partner Units	20	*d*	Traveling Sales Person∗	20	*o*
Graceful Graphs	20	*d*	Permutation Pattern-	20	*d*	Valves Location Problem∗	20	*o*
Graph Colouring	20	*d*	Permutation Pattern- Matching	20	*d*	Video Streaming	20	*o*
Hanoi Tower	20	*d*	Qualitative Spatial-Reasoning	20	*d*	Visit-all	20	*d*
Incremental Scheduling	20	*d*	Ricochet Robots	20	*d*	Weighted Sequence Problem	20	*d*

https://doi.org/10.1017/S1471068425100264 Published online by Cambridge University Press

Table 2. *Comparison of asp-fzn with ASP solvers on plain ASP benchmarks. The symbols next to the score indicate whether a higher value (↑) or lower value (↓) is better*

	single thread		
	Score1 ↑	*Score2* ↑	*Score3* ↓
asp-fzn (CP-SAT)/(CP-SAT, non-strict)	1840.0/1871.7	1888.3/1978.3	153738.5/149807.9
asp-fzn (Chuffed)/(Chuffed, non-strict)	782.4/812.4	782.4/812.4	279592.2/275942.3
asp-fzn (Gurobi)/(Gurobi, non-strict)	1185.0/1265.0	1196.7/1290.0	231543.2/222057.8
clingo	**1890.4**	**1992.1**	**147786.8**
DLV	1524.4	1604.4	191445.8
	8 threads		
	Score1 ↑	*Score2* ↑	*Score3* ↓
asp-fzn (CP-SAT)/(CP-SAT, non-strict)	2025.0/2051.7	2051.7/2101.7	131003.7/128072.7
asp-fzn (Gurobi)/(Gurobi, non-strict)	1441.7/1478.3	1445.0/1486.7	201661.7/196921.8
clingo	**2351.2**	**2511.2**	**92028.5**

with Gurobi and CP-SAT, the gap to clingo widened. Nonetheless, the benchmarks show that asp-fzn with the right backend solver is competitive with known ASP solvers.

6.2 CASP benchmarks

We now turn our attention to CASP. We look at three problem domains with ASP benchmark instances from the literature that can be modeled with CASP. We compare asp-fzn against clingcon 5.2.1 (Banbara *et al*. 2017) as it supports a similar language.

Parallel Machine Scheduling Problem (PMSP) was first studied with ASP by Eiter *et al*. (2023), who provided a benchmark set of 500 instances. The task is assigning jobs with release dates and sequence-dependent setup times to capable machines. The objective is minimizing the total makespan, that is, the maximal completion time of any job.

Table 3 shows the results for PMSP on the 500 instances using the (non-tight) CASP encoding which for space reasons is given in the appendix. In single-threaded solving, asp-fzn with CP-SAT and the non-strict translation is again superior, closing 40 instances and achieving the best result for 166; the strict translation is slightly worse but closes the same number of instances. The solver clingcon closed 36 instances, which is more than asp-fzn with any of the other backend solvers.

Looking at the *PAR10* score, cf. Section 6.1, we see that asp-fzn with CP-SAT achieves the best score, indicating that it can close the instances faster than clingcon. Interestingly, the strict translation does better here but the difference is marginal.

The picture changes for multi-threaded solving: here clingcon achieved the top value for *best* with 298 instances *versus* 167 by asp-fzn with CP-SAT for the non-strict translation. The latter setting closed the second most instances (54); changing to the strict translation closed one instance but decreased best results. The large number of best results found by clingcon can be explained by its strength in finding feasible solutions for PMSP in parallel mode, while asp-fzn struggles. However, when a solution is found, asp-fzn and

Table 3. *asp-fzn vs. clingcon on PMSP (strict / non-strict)*

	single thread			8 threads		
	closed	*best*	*PAR10*	*closed*	*best*	*PAR10*
asp-fzn (CP-SAT)	**40**/ **40**	140/ **166**	**11050.6**/11051.4	**55**/54	155/167	**10699.0**/10719.5
asp-fzn (Chuffed)	18/20	18/20	11570.5/11525.0	–	–	–
asp-fzn (Gurobi)	26/27	26/29	11379.1/11355.8	28/28	36/41	11330.7/11330.4
clingcon	36	36	11147.8	31	**298**	11264.0

```
&dom{R..D} = start(J) :- job(J), release(J, R), deadline(J, D).
&dom{R..D} = end(J) :- job(J), release(J, R), deadline(J, D).
&dom{L..H} = duration(J) :- job(J), L = #min{ T : durationInMode(J, _, T) },
                            H = #max{ T : durationInMode(J, _, T) }.
1 {modeAssign(J, M) : modeAvailable(J, M)} 1 :- job(J).
:- job(J), modeAssign(J, M), durationInMode(J, M, T), &sum{ duration(J) } != T.
:- job(J), &sum{end(J); -start(J); -duration(J)} != 0.
:- precedence(J,K), &sum{start(J); -end(K)} < 0 .
   ...
&disjoint{ start(J)@duration(J) : workbenchAssign(J,W) } :- workbench(W).
&disjoint{ start(J)@duration(J) : empAssign(J,W) } :- employee(W).
&disjoint{ start(J)@duration(J) : equipAssign(J,W) } :- equipment(W).

#minimize{1,E,J,s2 : job(J), empAssign(J, E), not employeePreferred(J, E)  }.
#minimize{1,E,P,s3 : project(P), empAssign(J, E), projectAssignment(J, P)}.
&dom{0..H} = delay(J)  :- job(J), horizon(H).
:- job(J), due(J, T), &sum{end(J)} > T, &sum{-1*delay(J); end(J)} != T.
:- job(J), due(J, T), &sum{end(J)} <= T, &sum{delay(J)} != 0.
&minimize{delay(J) : job(J)}.
   ...
```

Fig 2. Partial TLSPS encoding used by asp-fzn.

CP-SAT typically provide the best final result and as the *PAR10* score shows, it also takes the least CPU time to prove optimality.

Test Laboratory Scheduling Problem (TLSPS) is a variant of a scheduling problem due to Mischek and Musliu (2018), that is, efficiently solvable using a CASP encoding (Geibinger *et al.* 2021; Eiter *et al.* 2024). As the encoding is tight, the strict and the non-strict translation are the same.

TLSPS concerns scheduling jobs in a test lab by assigning them an execution mode, a starting time in its time window, and required resources from a set of qualified resources. The overall objective has several components, like assigning preferred employees for certain jobs, minimizing the number of employees on a project, reducing tardiness, and minimizing the project duration.

For clingcon, we essentially use Eiter *et al.*'s (2024) encoding employing ASP minimization. The asp-fzn encoding, shown partially in Listing 2 (full version in Appendix A), mixes minimization of plain ASP and linear variables; clingcon does not support the latter, but allows for a more natural encoding of the objective. Also, the asp-fzn encoding uses global disjunctive constraints to enforce unary resource usage; this is not possible in clingcon but proved to be quite effective.

Table 4. *asp-fzn vs. clingcon on TLSPS*

	single thread			8 threads		
	closed	*best*	*PAR10*	*closed*	*best*	*PAR10*
asp-fzn (CP-SAT)	**55**	**76**	**6741.5**	64	76	5850.1
asp-fzn (Chuffed)	11	11	10940.0	–	–	–
clingcon	7	22	11329.1	**77**	**90**	**4553.3**

Table 5. *asp-fzn vs. clingcon on MAPF*

	single thread		8 threads	
	closed	*PAR10*	*closed*	*PAR10*
asp-fzn (CP-SAT)	**224**	**7116.6**	**233**	**6913.4**
asp-fzn (Chuffed)	159	8553.7	–	–
asp-fzn (Gurobi)	194	7766.5	194	7762.9
clingcon	177	8138.1	209	7428.2

Our benchmark consisted of 123 instances from Mischek and Musliu (2018) of which 3 are real-world; the instances were converted to ASP facts (see supplementary data).

The results, collected in Table 4, show that asp-fzn performed very well. Column *closed* lists how many instances were solved and proven optimal, and *best* lists the number of solutions that were best among all solvers; instances for which no solver found any solution were discarded. Our tool asp-fzn with backend CP-SAT performed best for TLSPS in single-threaded mode as it solved 55 instances to optimality and produced for 76 instances the best result. Furthermore, it also achieved the lowest, and thus best, *PAR10* score. With backend Chuffed, asp-fzn performed significantly worse but produced always best results; also clingcon lagged significantly behind. Gurobi was not used as it does not support disjunctive global constraints. With multi-threaded solving, clingcon outperformed asp-fzn and CP-SAT, closing more instances and more often yielding the best result, while also taking less time to prove optimality on average.

Multi agent path finding (MAPF) was recently studied by Kaminski *et al.* (2024), who provided an instances and a generator. The task is planning the routes of several agents to reach their goals without colliding. Our tight CASP encoding (see Appendix A) is similar to Kaminski *et al.*'s (2024) but uses linear constraints for the event ordering.

For our comparison, we selected 547 MAPF instances from one of the sets by Kaminski *et al.* The results are shown in Table 5, listing the number of instances for which a plan was found (MAPF has no optimization objective). With Gurobi and CP-SAT as backends, asp-fzn closed more instances than clingcon, but it closed fewer with Chuffed. The best result is achieved by asp-fzn and CP-SAT with 224 instances solved; it also achieves the best *PAR10* score. For parallel solving (8 threads), asp-fzn with CP-SAT closed the most instances (233, 9 more than single-threaded). Gurobi did not benefit from parallelism while it improved the clingcon results. However, the latter still lagged behind CP-SAT.

6.3 Summary

Overall, asp-fzn with CP-SAT as backend achieved decent results, being competitive as a plain ASP solver and performing better than clingcon for TLSPS and MAPF. However, we note that CP-SAT has a rather high memory footprint. The average total memory usage of clingo on the plain ASP benchmark was five times lower than the one of asp-fzn with CP-SAT and the latter hit the memory limit for several instances. This is not only due to the translation itself, but a high memory usage of CP-SAT in general.

Regarding strict *versus* non-strict translation, it appears beneficial to use the non-strict translation by default, except when solution enumeration is requested. The time it takes to translate the gringo output to FlatZinc, this never took longer than a couple of seconds and was dwarfed by the grounding time.

7 Related work and conclusion

For a thorough survey of CASP solvers, we refer to Lierler's (2023) survey. Closest related to asp-fzn is clingcon (Banbara *et al*. 2017) as it features a similar language and is based on clingo (Gebser *et al*., 2019). Notably, while clingcon supports some global constraints, their usage is often limited. For example, variables occur in disjunctive constraints unconditionally, that is, whether a linear variable is active depends only on the truth of atoms determined at grounding time. This excludes disjunctive constraints as used for TLSPS in asp-fzn. Further, clingcon lacks cumulative constraints and disallows mixing ASP minimization and minimization over linear variables. Closely related to clingcon is clingo-dl (Janhunen *et al*. 2017), which is not a full CASP solver as it only supports *difference constraints*, a special type of linear constraint. As we consider unrestricted linear constraints, we did not feature clingo-dl in the evaluation.

EZSMT+ (Shen and Lierler 2019) is also a translation-based CASP solver but targets SMT. As it does not support optimization, we did not feature it in the comparison. As a further impediment to a direct comparison, EZSMT+ uses the language of EZCSP (Balduccini, 2011), which is quite different from asp-fzn and clingcon's theory language. In difference to clingcon and EZSMT+, EZCSP has slightly different semantics, as the linear constraints are evaluated for each answer set that may be pruned on violation. Another translation-based CASP solver is mingo (Liu *et al*., 2012), which translates a CASP program into MIP. While mingo does feature optimization, it also differs in language from asp-fzn and was not compatible with Gurobi.

As for translation-based plain ASP, our approach borrows heavily from Alviano and Dodaro (2016) and Janhunen (2023). Janhunen extended the level mapping formulation to programs with weight rules but provided no implementation, while Alviano and Dodaro introduced completion for disjunctive rules not as a translation-based approach per se but for DLV (Alviano *et al*. 2017). Finally, Rankooh and Janhunen's (2024) translation of ASP into MIP relies on prior normalization and an acyclicity transformation that explicitly represents dependencies among atoms by auxiliary variables and encodes supported models; answer sets are obtained by adding acyclicity constraints.

Outlook. A promising avenue for future work is the investigation of vertex elimination, as used by Rankooh and Janhunen in their translation. While it does not guarantee a

1-1 correspondence, it has shown potential for improving performance on standard ASP optimization benchmarks. Additional directions for future research include incorporating more global constraints or exploring novel language constraints that can be modeled in FlatZinc. Another possibility is evaluating metaheuristic FlatZinc solvers, such as using CP-SAT as a purely local-search-based solver. Finally, CASP semantics was aligned more with stable reasoning, moving away from interpreting linear constraints classically, in Cabalar *et al*. (2016, 2020); Eiter and Kiesel (2020). A modified translation modeling those semantics would be another highly interesting avenue for future work.

Supplementary material

The supplementary material for this article can be found at https://doi.org/10.1017/S1471068425100264.

References

Alviano, M., Calimeri, F., Charwat, G., *et al.* 2013. The fourth answer set programming competition: Preliminary report. In *Proc. of LPNMR 2013*, Vol. 8148 of LNCS, Springer, 42–53. https://doi.org/10.1007/978-3-642-40564-8_5.

Alviano, M., Calimeri, F., Dodaro, C., Fuscà, D., Leone, N., Perri, S., Ricca, F., Veltri, P. and Zangari, J. 2017. The ASP system DLV2. In *Proc. of LPNMR 2017*, Vol. 10377 of LNCS, Springer, 215–221.

Alviano, M. and Dodaro, C. 2016. Completion of disjunctive logic programs. In *Proc. of IJCAI 2016*, IJCAI/AAAI Press, 886–892.

Balduccini, M. 2011. Industrial-size scheduling with ASP+CP. In *Proc. of LPNMR 2011*, Vol. 6645 of LNCS, Springer, 284–296.

Banbara, M., Kaufmann, B., Ostrowski, M. and Schaub, T. 2017. Clingcon: The next generation. *Theory and Practice of Logic Programming* 17, 4, 408–461.

Ben-eliyahu, R. and Dechter, R. 1994. Propositional semantics for disjunctive logic programs. *Annals of Mathematics and Artificial Intelligence* 12, 1-2, 53–87.

Brewka, G., Eiter, T. and Truszczyński, M. 2011. Answer set programming at a glance. *Communications of the ACM* 54, 12, 92–103.

Cabalar, P., Fandinno, J., Schaub, T. and Wanko, P. 2020. An ASP semantics for constraints involving conditional aggregates. In *Proc. of ECAI 2020*, G. De Giacomo *et al.*, Eds. IOS Press, 664–671.

Cabalar, P., Fandinno, J., Schaub, T. and Wanko, P. 2023. On the semantics of hybrid ASP systems based on clingo. *Algorithms* 16, 4, 185.

Cabalar, P., Kaminski, R., Ostrowski, M. and Schaub, T. 2016. An ASP semantics for default reasoning with constraints. In *Proc. of IJCAI 2016*, IJCAI/AAAI Press, 1015–1021.

Calimeri, F., Gebser, M., Maratea, M. and Ricca, F. 2016. Design and results of the fifth answer set programming competition. *Artificial Intelligence* 231, 151–181.

Calimeri, F., Ianni, G. and Ricca, F. 2014. The third open answer set programming competition. *Theory and Practice of Logic Programming* 14, 1, 117–135.

Chu, G. 2011. Improving combinatorial optimization. Ph.D. thesis, University of Melbourne, Australia.

Clark, K. L. 1977. Negation as failure. In *Logic and Data Bases*, Plemum Press, New York, 293–322.

Eiter, T., Faber, W., Fink, M. and Woltran, S. 2007. Complexity results for answer set programming with bounded predicate arities and implications. *Annals of Mathematics and Artificial Intelligence* 51, 2-4, 123–165. https://doi.org/10.1007/s10472-008-9086-5.

Eiter, T., Geibinger, T., Musliu, N., Oetsch, J., Skocovský, P. and Stepanova, D. 2023. Answer-set programming for lexicographical makespan optimisation in parallel machine scheduling. *Theory and Practice of Logic Programming* 23, 6, 1281–1306.

Eiter, T., Geibinger, T., Ruiz, N. H., Musliu, N., Oetsch, J., Pfliegler, D. and Stepanova, D. 2024. Adaptive large-neighbourhood search for optimisation in answer-set programming. *Artificial Intelligence* 337, 104230.

Eiter, T. and Kiesel, R. 2020. ASP($\mathcal{AC}$): Answer set programming with algebraic constraints. *Theory and Practice of Logic Programming* 20, 6, 895–910.

Erdem, E. and Lifschitz, V. 2003. Tight logic programs. *Theory and Practice of Logic Programming* 3, 4-5, 499–518.

Gebser, M., Kaminski, R., Kaufmann, B. and Schaub, T. 2019. Multi-shot ASP solving with clingo. *Theory and Practice of Logic Programming* 19, 1, 27–82.

Geibinger, T., Mischek, F. and Musliu, N. 2021. Constraint logic programming for real-world test laboratory scheduling. In *Proc. of AAAI 2021*, AAAI Press, 6358–6366.

Gurobi Optimization, LLC. 2025. Gurobi Optimizer Reference Manual.

Janhunen, T. 2023. Generalizing level ranking constraints for monotone and convex aggregates. In *ICLP 2023 Tech. Comm.*, Vol. 385 of EPTCS, 101–115. https://doi.org/10.4204/EPTCS.385.12.

Janhunen, T., Kaminski, R., Ostrowski, M., Schellhorn, S., Wanko, P. and Schaub, T. 2017. Clingo goes linear constraints over reals and integers. *Theory and Practice of Logic Programming* 17, 5-6, 872–888.

Janhunen, T., Niemelä, I. and Sevalnev, M. 2009. Computing stable models via reductions to difference logic. In *Proc. of LPNMR 2009*, Vol. 5753 of LNCS, Springer, 142–154.

Kaminski, R., Romero, J., Schaub, T. and Wanko, P. 2023. How to build your own ASP-based system?!. *Theory and Practice of Logic Programming* 23, 1, 299–361.

Kaminski, R., Schaub, T., Son, T. C., Svancara, J. and Wanko, P. 2024. Routing and scheduling in answer set programming applied to multi-agent path finding: Preliminary report. CoRR: abs/2403.12153. https://arxiv.org/abs/2403.12153.

Lierler, Y. 2023. Constraint answer set programming: Integrational and translational (or SMT-based) approaches. *Theory and Practice of Logic Programming* 23, 1, 195–225.

Liu, G., Janhunen, T. and Niemelä, I. 2012. Answer set programming via mixed integer programming. In *Proc. of KR 2012*, AAAI Press.

Mischek, F. and Musliu, N. 2018. The test laboratory scheduling problem. Technical Report CD-TR 2018/1, Christian Doppler Lab for AI and Optimization for Planning and Scheduling, TU Wien, Austria.

Nethercote, N., Stuckey, P. J., Becket, R., Brand, S., Duck, G. J. and Tack, G. 2007. MiniZinc: Towards a standard CP modelling language. In *Proc of CP 2007*, Vol. 4741 of LNCS, Springer, 529–543.

Perron, L., Didier, F. and Gay, S. 2023. The CP-SAT-LP solver. In *Proc. of CP 2023*, Vol. 280 of LIPIcs, Schloss Dagstuhl – LZI, 3:1- 3:2.

Rankooh, M. F. and Janhunen, T. 2024. Improved encodings of acyclicity for translating answer set programming into integer programming. In *Proc. of IJCAI 2024*, ijcai.org, 3369–3376.

Shen, D. and Lierler, Y. 2019. SMT-based constraint answer set solver EZSMT+. CoRR: abs/1905.03334. https://arxiv.org/abs/1905.03334.

Wolsey, L. 2021. *Integer Programming*. John Wiley & Sons.