TPLP: Page 1–8. © The Author(s), 2021. Published by Cambridge University Press 2021
doi:10.1017/xxxxx

EZSMT Version 3, Matured DRAFT *

KEERAN DHAKAL
University of Nebraska Omaha, USA

YULIYA LIERLER
University of Nebraska Omaha, USA

Abstract

Constraint Answer Set Programming (CASP) is a hybrid reasoning paradigm that combines Answer Set Programming (ASP) with Constraint Processing and Satisfiability Modulo Theories (SMT), enabling powerful declarative encodings of complex combinatorial search problems. This paper presents the design and implementation of EZSMTV3, an extensible SMT-based CASP framework that advances the translational approach to CASP solving. Building upon the foundation of the EZSMT+ system, EZSMTV3 introduces a more expressive input language, supports optimization via weak constraints, and offers foundations for streamlined integration of new constraint types. Rather than implementing custom search procedures, EZSMTV3 leverages state-of-the-art SMT solvers, such as CVC5, YICES, and Z3 to perform reasoning. The paper provides benchmarking results comparing EZSMTV3 with its CASP peers such as CLINGCON, CLINGO[DL], and CLINGO[LP], while showcasing its ability to handle mixed-domain constraints involving both integers and reals. The system provides a robust platform for future extensions and theoretical exploration within the CASP domain.

KEYWORDS: Constraint Answer Set Programming and Satisfiability Modulo Theory

1 Introduction

Constraint Answer Set Programming (CASP) is a hybrid methodology in automated reasoning that integrates advancements from several research domains, namely answer set programming (Niemelä; Marek and Truszczyński; Brewka et al., 1999; 1999; 2011), constraint processing (Rossi et al.; Jaffar and Maher, 2008; 1994), and satisfiability modulo theories (Nieuwenhuis et al.; Barrett et al.; Barrett and Tinelli, 2006; 2008; 2014). Works by Elkabani et al. (2004); Mellarkod et al. (2008); Lierler (2014) are among earlier references to CASP. It has shown significant potential, leading to the creation of numerous solvers such as ACSOLVER (Mellarkod et al., 2008), CLINGCON (Gebser et al., 2009a), EZCSP (Balduccini and Lierler, 2017), IDP (Wittocx et al., 2008), INCA (Drescher and Walsh, 2010), DINGO (Janhunen et al., 2011), MINGO (Liu et al., 2012), ASPMT (Bartholomew and Lee, 2014), CLINGO[L,DL] (Janhunen et al., 2017), and EZSMT+ (Susman and Lierler; Shen and Lierler, 2016a; 2018a). CASP opens up new possibilities for declarative programming, enabling it to tackle such complex tasks as train scheduling and product configurations. Solvers for CASP can be broadly categorized based on their construction strategy into integrational and translational approaches. This paper describes not just a

* Partially funded by UNO GRACA 2024

solver that practices translational approach but an extensible CASP framework that is geared to ease the implementation of new systems in this field.

This paper presents the design, development, and implementation of an extensible SMT-based constraint answer set programming framework EZSMT version 3 (EZSMTV3). We build upon the initial vision outlined in our earlier work on the EZSMT+ system (Susman and Lierler; Shen and Lierler, 2016a; 2018a). In particular, we continue championing the practice of so called translational approaches within the automated reasoning realm. The work on EZSMTV3 turns preliminary ideas behind the CASP EZSMT+ solver into mature extensible framework for CASP. With that not only EZSMTV3 is the solver itself, it is also designed to support extensions of this system to new kinds of constraints in a simple, streamlined manner. In a nutshell, the EZSMTV3 system computes answer sets to constraint answer set (CAS) programs providing support for various kinds of constraints. Yet, while doing so it does not implement native search procedures. Instead, it translates a given logic program with constraint atoms into a formula within some dialect of satisfiability modulo theory (SMT). This formula is then processed by one of the off-the-shelf SMT solvers. Historically, the EZSMT+ language adopted the conventions of the CASP language developed for the EZSCP system (Balduccini and Lierler, 2017). Thus, its constraint atoms (marked by the keyword `required`) were restricted to rule heads, making certain domains cumbersome to formalize. While sufficient for bootstrapping a proof-of-concept system, EZCSP's language features revealed the need for a more expressive and flexible alternative.

In our work on EZSMTV3 we found such an alternative. In particular, it builds upon the developments in the CLINGO 5 series (Gebser et al.; Kaminski et al., 2019; 2023) that promotes the extensibility philosophy. The CLINGO 5 system provides means to elaborate the specifications for new kinds of constructs to be incorporated for processing within its grounding tool GRINGO (Gebser et al.; Gebser et al.; Kaminski, 2009b; 2015; 2023). In addition, CLINGO 5 provides means to incorporate custom propagators to ensure proper processing of newly incorporated syntactic language features. In this work, we embrace the extensibility philosophy of CLINGO 5. Yet, we diverged from its provisions for the custom implementations of the search mechanisms. We advocate the utilization of already existing state-of-the-art automated reasoning tools, specifically, SMT solvers. Thus, this work relies on a body of theoretical findings relating CASP and SMT as well as a body of sophisticated algorithmic developments within SMT solving resulting in such exemplary systems as CVC4 (Barrett et al., 2011), CVC5 (Barrett et al.,), Z3 (De Moura and Bjørner, 2008), and YICES (Dutertre and De Moura, 2006). Instead, we focused on creating a streamlined interface to SMT technologies. Our approach relies on an easily extensible API to support translations from ASP to SMT—paving the way for future CASP dialects to be seamlessly integrated. In addition, EZSMTV3 implements support for optimization statements, namely, weak constraints. This feature is new to EZSMTV3 and was missing from the CASP EZSMT+ solver.

Section 2 of this paper provides a review of key concepts in constraint answer set programming. Section 3 starts by detailing the CASP dialects supported by EZSMTV3 by utilizing a formalization of a variant of the traveling salesman problem as our running example. It concludes with the presentation on the architecture of the EZSMTV3 system and the discussion of its implementation. Given that optimization statements are new to EZSMTV3 in relation to its older "sibling" EZSMT+, Section 4 introduces syntax and semantics of language constructs used to express optimization statements within programs supported by the system. This section concludes with the details on the implementation. In Section 5, we discuss results on benchmarking the performance of EZSMTV3 against its closest CASP relatives such as CLINGCON, CLINGO[LP]

and CLINGO[DL]. We note that the capabilities of EZSMTV3 extends beyond any of these peers as, for example, the system is capable to support reasoning with constraint atoms that contain both integer and real variables. At last we remark on future work.

2 Background

2.1 Logic Programs and Input Answer Sets

Many definitions presented in this section follow the lines by Lierler (2023a, Sections 3 and 4).

Logic programs A *vocabulary* is a set of propositional symbols, also called atoms. A *literal* is an atom a or its negation $\neg a$. A *(propositional) logic program* over vocabulary σ is a set of *rules* of the form

$$a \leftarrow b_1, \ldots, b_\ell,\ not\ b_{\ell+1}, \ldots,\ not\ b_m,\ \ not\ \ not\ b_{m+1}, \ldots,\ not\ \ not\ b_n, \tag{1}$$

where a is an atom in σ or $\bot$, and each b_i, where $1 \leq i \leq n$, is an atom in σ. We will use the abbreviated form of a rule (1), i.e.,

$$a \leftarrow B, \tag{2}$$

where B stands for the right hand side of the arrow in (1), and is also called a *body*. By B^+ we denote the *positive* part of body B, i.e., $b_1, \ldots, b_\ell$. We sometimes identify body B with the propositional formula

$$b_1 \wedge \ldots \wedge b_\ell \wedge \neg b_{\ell+1} \wedge \ldots \wedge \neg b_m \wedge \neg\neg b_{m+1} \wedge \ldots \wedge \neg\neg b_n. \tag{3}$$

and rule (1) with the propositional formula (implication) $B \rightarrow a$. The expression a is the *head* of the rule. A rule whose head is the symbol $\bot$ is called a *denial*. A rule (2) whose body is empty, i.e., $n = 0$ is called a *fact*; in which case it is frequently written as a. (while B is identified with $\top$ and $\top \rightarrow a$ is identified with a). For a logic program Π (a propositional formula F), by $At(\Pi)$ (by $At(F)$) we denote the set of atoms occurring in Π (in F).

It is customary for a given vocabulary σ, to identify a set X of atoms over σ with (i) a complete and consistent set of literals over σ constructed as $X \cup \{\neg a \mid a \in \sigma \setminus X\}$, and respectively with (ii) an assignment function or interpretation that assigns truth value $true$ to every atom in X and $false$ to every atom in $\sigma \setminus X$. Within the scope of this paper we are interested in sets of atoms in relation to respective programs so that the signature of that program will be considered for reference. We say a set X of atoms *satisfies* rule (2), if X (understood as an assignment function) satisfies the propositional formula $B \rightarrow a$. Naturally, we can speak about set X satisfying the body or the negative part of the body of the rule as we identify these with respective propositional formulas. We say X satisfies a program Π, if X satisfies every rule in Π. In this case, we also say that X is a model of Π. We may abbreviate the satisfaction relation with symbol $\models$ (to denote that a set of atoms satisfies a rule, a body, a program, or a formula).

The *reduct* Π^X of a program Π relative to a set X of atoms is obtained by first removing all rules (1) such that X does not satisfy the negative part of the body

$$\neg b_{\ell+1} \wedge \ldots \wedge \neg b_m \wedge \neg\neg b_{m+1} \wedge \ldots \wedge \neg\neg b_n,$$

and replacing all remaining rules with $a \leftarrow b_1, \ldots, b_\ell$.

Definition 1 (Answer set)**.** *A set X of atoms is an* answer set*, if it is the minimal set that satisfies all rules of Π^X (Lifschitz et al., 1999).*

Example 1. *Consider a program*

$$\begin{array}{l} b \leftarrow a. \\ c \leftarrow not\ a. \end{array} \tag{4}$$

This program has a single answer set, namely, $\{c\}$*. Let us construct a new program from program* (4) *by appending a single fact to it.*

$$\begin{array}{l} a. \\ b \leftarrow a. \\ c \leftarrow not\ a. \end{array} \tag{5}$$

This program has a single answer set, namely, $\{a, b\}$*.*

Consider now another program

$$\begin{array}{l} a \leftarrow not\ not\ a. \\ b \leftarrow a. \\ c \leftarrow not\ a. \end{array} \tag{6}$$

The first rule of this program is typically written as

$$\{a\}.$$

Rules of this form are called *choice rules*. We can intuitively read the rule above as "*atom* a *may be the case*". This program has two answer sets:

$$\{a, b\} \text{ and } \{c\}. \tag{7}$$

We now state the definition of an input answer set, as it is instrumental in defining constraint answer set programs.

Definition 2 (Input Answer Set)**.** *For a logic program* Π *over vocabulary* σ *and vocabulary* $\iota \subseteq \sigma$ *such that none of* ι*'s elements occur in the heads of rules in* Π*, a set* X *of atoms over* σ *is an* input answer set *of* Π *relative to* ι*, when* X *is an answer set of the program* $\Pi \cup (X \cap \iota)$*.*

Recall program (4), it has two input answer sets relative to signature $\{a\}$. These answer sets are listed in (7).

The reader may obtain new insights about the definition of an input answer set in light of the following formal result.

Proposition 1. *For a logic program* Π *over vocabulary* σ *and vocabulary* $\iota \subseteq \sigma$ *such that none of* ι*'s elements occur in the heads of rules in* Π*, the answer sets of a program*

$$\Pi \cup \{\ \{a\}.\ \mid a \in \iota\}$$

coincide with the input answer sets of program Π *relative to* ι*.*

Thus it is not by chance that answer sets of program (6) and input answer sets of program (4) relative to signature $\{a\}$ coincide.

The *dependency graph* of Π is the directed graph G such that

- the vertices of G are the atoms occurring in Π, and
- for every rule (1) in Π whose head is not $\bot$, G has an edge from atom a to each atom in positive part $b_1, \ldots, b_\ell$ of its body.

A program is called *tight* if its dependency graph is acyclic. It is easy to see that any sample program discussed so far is tight. The simplest nontight program follows $a \leftarrow a$.

2.2 Constraints, CSP, SMT

Lierler and Susman (2017) illustrated that the notion of a constraint syntactically coincides with ground literals of Satisfiability Modulo Theory (SMT). Furthermore, a constraint satisfaction problem (CSP) — posed as a set of constraints — can be identified with a conjunction of ground literals, which is evaluated by means of first-order logic interpretations/structures representative of a particular "uniform" SMT-logic (Lierler and Susman, 2017). Thus, in a way we can understand Satisfiability Modulo Theory via the lens of Satisfiability Modulo Constraints.

Intuitively, uniform SMT-logics are defined via interpretations/structures whose domain, interpretation of "theory/constraint/interpreted" predicate symbols, and "interpreted" function symbols are fixed. In practice, special forms of constraints are commonly used. *Integer linear constraints* are examples of these special cases. Let us recall their syntactic shape and provide some intuitions for their interpretations prior to diving into formal settings. An *(integer) linear expression* has the form

$$a_1 x_1 + \cdots + a_n x_n, \tag{8}$$

where $a_1, \ldots, a_n$ are (integer) numbers and $x_1, \ldots, x_n$ are (constraint) variables whose domain ranges over (integer) numbers. Note how this definition encapsulates both integer linear expressions and linear expressions over real numbers. For the latter we drop the word integer as a requirement on coefficients and variables. It is customary to omit coefficients when their value is 1 and also replace $+$ by $-$ when the coefficient is negative number, while that negative number is replaced by its absolute value. The SMT technology utilizes the standard SMT-LIB language (Barrett et al., 2010). In that language prefix notation is used so that expression (8) is written as

$$+(\times(a_1, x_1), +(\times(a_2, x_2), \cdots + (\times(a_{n-1}, x_{n-1}), \times(a_n, x_n)) \ldots).$$

We call a constraint *(integer) linear* when it has the form

$$e \bowtie k \tag{9}$$

where e is (an integer) linear expression, k is (an integer) number, and $\bowtie$ belongs to

$$\{<, >, \leq, \geq, =, \neq\}. \tag{10}$$

We can write (9) as an expression $\bowtie\ (e, k)$ in prefix notation; when e is also written in prefix notation it is easy to see how this constraint takes the shape of a ground atom. Let us agree to call these kinds of atoms *constraint (ground) atoms*. In the sequel, we use the terms *constraint* and *constraint atom* interchangeably.

For instance, consider an integer linear constraint

$$2x + 3y > 0. \tag{11}$$

When written in prefix notation it takes the shape of constraint ground atom

$$> (+(\times(2, x), \times(3, y)), 0),$$

where

- $>$ is a binary "interpreted" predicate symbol;
- $+$ and $\times$ are binary "interpreted" function symbols;
- 0, 2, and 3 are 0-arity "interpreted" function symbols; and
- x and y are 0-arity "un-interpreted" function symbols.

In the logic literature, 0-arity un-interpreted function symbols are frequently referred to as object constants, whereas in the constraint processing literature they are referred to as (constraint) variables. Here, we use the term *constraint variables*. Prior to some formal definitions let us talk about this constraint atom as a formula within Satisfiability Modulo Linear Integer Arithmetic Logic, intuitively. This logic is defined by interpretations, such that

- $>$ is interpreted as an arithmetic greater than relation;
- $+$ and $\times$ are interpreted as usual in arithmetic;
- 0, 2, and 3 are mapped into respective integer domain elements identified with the same symbol, thus we may also refer to these function symbols as integers; and
- x and y are mapped into integers; in general, the domain for 0-arity function symbols occurring in constraint atoms is the set of integers.

In naming, the constraints we use conventions adopted by SMT-LIB[1] so that

- IA stands for the theory Ints (Integer Arithmetic);
- RA stands for the theory Reals (Real Arithmetic);
- IRA stands for the theory Reals and Ints (mixed Integer Real Arithmetic);
- IDL stands for Integer Difference Logic
- L before IA, RA, or IRA stands for the linear fragment of those arithmetics.

Let us call

- integer linear constraints — *LIA constraint atoms (LIA constraints)*;
- linear constraints — *LRA constraint atoms (LRA constraints)*;
- constraints that syntactically have the form of expression (9), while the coefficients and constraint variables of this expression can be both integer and real numbers — *mixed integer real constraint atoms (LIRA constraints)*;
- constraints that have the form

$$x - y \bowtie c \qquad \text{or} \qquad x \bowtie y, \tag{12}$$

  where $\bowtie$ is one of the arithmetic relations in (10), x and y are constraint variables over integers, and c is an integer — *IDL constraint atoms (IDL constraints)*.

We are now ready to provide formal definitions for four Logics within SMT framework utilized in this work.

Definition 3 (Satisfiability Modulo Theory Formula or SMT Formula)**.** Formula in Satisfiability Modulo Linear Integer Arithmetic Logic or SMT(LIA) Formula — *is a variable-free first order logic formula that consists of propositional or LIA constraint atoms. Its* interpretations *can be captured by* valuations *– functions – that map all propositional atoms to truth values and constraint variables to integers; while arithmetic predicate and function symbols are interpreted as customary in arithmetic.*

[1] https://smt-lib.org/logics.shtml

SMT(LRA) Formulas or SMT Formulas in LRA Logic, SMT(LIRA) Formulas or SMT Formulas in LIRA Logic, SMT(IDL) Formulas or SMT Formulas in IDL Logic *are defined similarly considering LRA, LIRA, and IDL constraint atoms, respectively, in place of LIA constraint atoms. Interpretations for these formulas are captured by valuations that respect domains of the constraint atoms occrding to their types.*

Models *of SMT formulas are interpretations that satisfy SMT formulas, where the satisfaction relation is understood classically as in first order logic.*

Example 2. *For instance, the SMT(LIA) formula*

$$p \to \big((x \geq 1 \wedge x \leq 3) \vee x = 5\big). \tag{13}$$

consists of propositional atom p and LIA constraint atoms

$$x \geq 1 \qquad x \leq 3 \qquad x = 5.$$

This formula has four models when p is interpreted as $true$ captured by the following valuations

$$\begin{array}{ll} x \mapsto 1 & p \mapsto \mathit{true} \\ x \mapsto 2 & p \mapsto \mathit{true} \\ x \mapsto 3 & p \mapsto \mathit{true} \\ x \mapsto 5 & p \mapsto \mathit{true} \end{array}$$

There are an infinite number of models for this formula when p is interpreted as $false$ including, for example, one captured by the valuation

$$x \mapsto 4 \qquad p \mapsto \mathit{false}.$$

We are now ready to state definitions for constraint satisfaction problems.

Definition 4 (Constraint satisfaction problem or CSP)**.** *We call a finite set of constraints a* constraint satisfaction problem (CSP)*. Within this work we will consider CSPs of particular kind. Namely,*

- Linear Integer Arithmetic CSP (LIA CSP) *formed as a set of LIA constraints;*
- Linear Real Arithmetic CSP (LRA CSP) *formed as a set of LRA constraints;*
- LIRA CSP *formed as a set of LIRA constraints;*
- Integer Difference Logic CSP (IDL CSP) *formed as a finite set of IDL constraints.*

As we identify constraints with ground atoms, we also identify a CSP with a conjunction of constraints/ground atoms occurring in its set. Thus, any LIA CSP, LRA CSP, LIRA CSP, and IDL CSP can be viewed as a special form of SMT(LIA), SMT(LRA), SMT(LIRA), SMT(IDL) formula, respectively. We call models of these formulas solutions *of respective CSPs.*

Example 3. *One of the solutions to the LIA CSP composed of a single constraint (11) is a valuation that maps x to 0 and y to 1. If we are to form another LIA CSP – a set composed of constraint (11) and constraint $y \neq 1$, then the valuation that maps x to 0 and y to 1 is not a solution, while, for instance, a valuation that maps x to 0 and y to 2 is.*

Both LRA and IDL CSPs are interesting from the perspective that there are tractable algorithms to decide whether these problems have solutions. This is not the case for LIA and LIRA CSPs.

2.3 Constraint answer set programs and their relation to SMT

Let σ_r and σ_i be two disjoint vocabularies. We refer to their elements as *regular* and *irregular* atoms, respectively.

Definition 5 (Constraint Answer Set Program or CAS Program)**.** *Let $\sigma = \sigma_r \cup \sigma_i$ be a vocabulary so that σ_r and σ_i are disjoint; $\mathcal{B}$ be a set of constraints; γ be an injective function from the set of irregular literals over σ_i to $\mathcal{B}$.*

We call a triple $P = \langle \Pi, \mathcal{B}, \gamma \rangle$ an CAS program *over vocabulary $\sigma_r \cup \sigma_i$, when Π is a logic program over $\sigma_r \cup \sigma_i$ such that any rule that contains atoms in σ_i is a rule with symbol $\bot$ in its head.*

A set $X \subseteq At(\Pi)$ of atoms is an answer set *of P if*

(a) X is an input answer set of Π relative to σ_i, and
(b) the following CSP has a solution: $\{\gamma(a) | a \in X \cap \sigma_i\} \cup \{\gamma(\neg a) | a \in \sigma_i \setminus X\}$.

A pair $\langle X, \nu \rangle$ is an extended answer set *of P if X is an* answer set *of P and valuation ν is a solution to the CSP constructed in (b).*

Within this work we consider CAS programs $\langle \Pi, \mathcal{B}, \gamma \rangle$ of a particular kind. Namely,

- CAS(LIA) programs *whose set $\mathcal{B}$ of constraints is formed by LIA constraints;*
- CAS(LRA) programs *whose set $\mathcal{B}$ of constraints is formed by LRA constraints;*
- CAS(LIRA) programs *whose set $\mathcal{B}$ of constraints is formed by LIRA constraints;*
- CAS(IDL) programs *whose set $\mathcal{B}$ of constraints is formed by IDL constraints.*

It is due to note that when CAS programs are written in practice, the CASP systems permit a user listing an irregular atom in the head of the rule. Yet, that should be considered as "syntactic sugar" so that a rule of the form (2), where a is an irregular atom is seen as an abbreviation for the rule

$$\leftarrow B,\ not\ a.$$

In the presentation, we utilize vertical bars to mark the irregular atoms which will have intuitive mappings into their respective constraints. For instance, irregular atom $|x \geq 12|$ naturally maps into constraint $x \geq 12$.

Example 4. *We now exemplify the definition of a CAS program. Let Π_1 be logic program (6) extended with a denial*

$$\leftarrow a,\ |x \geq 12|,$$

where $|x \geq 12|$ denotes an irregular atoms with constraint variable x. Let $\mathcal{B}_1$ be a set of integer linear constraints $\{x \geq 12, x < 12\}$; γ_1 be an injective function from irregular literals in the signature of Π_1 to constraints

$$|x \geq 12| \rightarrow x \geq 12, \quad \neg|x \geq 12| \rightarrow x < 12.$$

CAS program $\langle \Pi_1, \mathcal{B}_1, \gamma_1 \rangle$ has three answer sets, namely,

$$\{a, b\}$$
$$\{c\}$$
$$\{c, |x \geq 12|\}$$

and infinitely many extended answer sets:

$$\{a, b, x \mapsto 11\} \quad \{a, b, x \mapsto 10\} \quad \{a, b, x \mapsto 9\} \ldots$$
$$\{c, x \mapsto 11\} \quad \{c, x \mapsto 10\} \quad \{c, x \mapsto 9\} \ldots$$
$$\{c, |x \geq 12|, x \mapsto 12\} \quad \{c, |x \geq 12|, x \mapsto 13\} \quad \{c, |x \geq 12|, x \mapsto 14\} \ldots$$

We refer to CAS program $P = \langle \Pi, \mathcal{B}, \gamma \rangle$ as *tight* when its first member Π has this property.

Lierler and Susman (2017) illustrated that for CAS programs of the four kinds considered here, one can construct an SMT formula (of the four kinds considered here) so that its models coincide with the extended answer sets of the given program. They generalized the concepts of completion and level ranking – originally introduced by Clark (1978) and Niemela (2008), respectively – which are essential in the construction of such an SMT formula. Intuitively, completion is a process that turns a CAS program into an SMT formula. This formula comes with a special guarantee that every extended answer set of the given program is a model of its completion. For the class of tight programs the reverse direction is also the case. As a result, the extended answer sets of a CAS program coincide with the models of its completion. In case of a program being nontight, so called level ranking constraints added to a completion will ensure that computed models (modulo newly introduced integer variables within level ranking constraints) are exactly the answer sets. We now provide details of that translation relevant to understanding the workings of the EZSMTV3 system.

Within the translation irregular atoms are introduced that encode level ranking constraints required to weed out models of the completion that are not answer sets. For instance, an irregular atom $|lr_a - lr_b \geq 1|$ encodes an IDL (or LIA or LIRA) constraint $lr_a - lr_b \geq 1$, where lr_a and lr_b are integer constraint variables. Let $P = \langle \Pi, \mathcal{B}, \gamma \rangle$ be a CAS program over $\sigma_r \cup \sigma_i$. If a program is not tight, for every atom $a \in \sigma_r$ that occurs in Π, we introduce an integer variable lr_a. The SMT formula $\mathcal{F}^P$ is constructed as a conjunction of the following

1. implications corresponding to rules (1) in Π;
2. for each regular atom a occurring within the given CAS program, the implication
   - $a \rightarrow \bigvee_{a \leftarrow B \in \Pi} B$, when the program is tight
   - $a \rightarrow \bigvee_{a \leftarrow B \in \Pi} \big(B \wedge \bigwedge_{b \in B^+ \setminus \sigma_i} |lr_a - lr_b| \geq 1\big)$, otherwise;
3. for each irregular atom $|c| \in \sigma_i$ occurring within the given CAS program (where c is a constraint; recall that irregular atoms are assumed to have a natural mapping into respective constraints), the equivalence $|c| \longleftrightarrow c$;
4. in case the considered program is not tight, for each irregular atom of the form $|lr_a - lr_b \geq 1|$ introduced within the translation, the equivalence
$$|lr_a - lr_b \geq 1| \longleftrightarrow lr_a - lr_b \geq 1.$$

In case of a tight CAS program P, formula $\mathcal{F}^P$ captures the completion of P.

Example 5. *Recall CAS program $\langle \Pi_1, \mathcal{B}_1, \gamma_1 \rangle$ from Example 4. Let us call it P_1. $\mathcal{F}^{P_1}$ follows*

$$\neg\neg a \rightarrow a \quad a \rightarrow b \quad \neg a \rightarrow c \quad a \wedge |x \geq 12| \rightarrow \bot$$
$$a \rightarrow \neg\neg a \quad b \rightarrow a \quad c \rightarrow \neg a$$
$$|x \geq 12| \longleftrightarrow x \geq 12$$

The models of this formula coincide with the answer sets of $\langle \Pi_1, \mathcal{B}_1, \gamma_1 \rangle$.

	Tight	Non-Tight
CAS(LIA)	SMT(LIA)	
CAS(LRA)	SMT(LRA)	SMT(LIRA)
CAS(LIRA)	SMT(LIRA)	
CAS(IDL)	SMT(IDL)	

Fig. 1. Mapping of CAS programs to respective SMT formulas.

Figure 1 summarizes the details on which kind of SMT formula system EZSMTV3 obtains during the application of the described translation process depending on the properties of the given CAS program. For instance, row 2 in the table of this figure states that given a CAS(LRA) program which is

- tight, the translation results in SMT(LRA) formula;
- non-tight, the translation results in SMT(LIRA) formula.

3 EZSMT Version 3 Language(s), Use Case, and Architecture

This section is devoted at large to the description of the language and architecture of the EZSMT Version 3 system, abbreviated as EZSMTV3. Prior to providing the details on the system's components, we articulate a bird's-eye view on the system by pointing at its major design choices. We also provide a sample use case of the system utilizing the Travelling Salesman problem. The presentation of this use case is intermixed with the details on the syntactic constructs supported by the EZSMTV3 together with their mappings into respective CAS fragments.

In a way, EZSMTV3 can be seen as a system that puts together the ideas and practices behind two CASP solvers, namely, CLINGCON version 3 (and above) (Banbara et al., 2017) and EZSMT+ (Shen and Lierler, 2018a). In particular, from CLINGCON it borrows an idea to utilize capabilities unique to the grounder GRINGO version 5 (Gebser et al., 2016). This grounder provides a possibility

- to specify the grammar of the language of constraints of interest and
- to use that newly defined language in writing programs that are subsequently grounded by GRINGO.

From EZSMT+, EZSMTV3 borrows an idea to utilize an SMT solver, such as Z3 or CVC5, as its search engine back-end after computing completion and level rankings of a given CAS program. The combination of the GRINGO version 5 front-end and an SMT solver as a back-end uniquely positions system EZSMTV3 not only as a CASP solver but also as an easily extensible framework for creating new kinds of CASP solvers. Indeed, SMT solvers support a multitude of distinct logics – languages for specifications of constraint atoms – while GRINGO version 5 allows us to specify a language of such constraint atoms and quickly incorporate these within the grounding stage of processing. The major routines of building completion and then translating that internal representation into the standard language supported by SMT solvers, namely, SMT-LIB is something that EZSMTV3 inherits from EZSMT+ and provides as part of the framework for extensions

to new logics. In the sequel, we omit the reference to version of GRINGO assuming version 5 as default.

3.1 EZSMTV3 *Language(s) and Its Use Case*

We start this section by uncovering the details of the EZSMTV3 language used for formulating programs in CAS(LIA). We then present the CAS(LIA) formalization of a variant of the Traveling Salesman (TS) Problem (Lawler et al.; Gutin and Punnen, 1985; 2007). The presented CAS(LIA) program is written in the language supported by systems EZSMTV3 (and CLINGCON). The similar formalization of the TS problem was presented by Lierler (2023a) in the language supported by the CASP solvers EZCSP (Balduccini and Lierler, 2017) (and EZSMT+). At last, we discuss the details of EZSMTV3 language used for formulating programs in CAS(LRA), CAS(LIRA), and CAS(IDL).

3.1.1 EZSMTV3 *CAS(LIA) Language*

As mentioned earlier, system GRINGO is used within EZSMTV3 as a front-end to ground a considered CAS program. Section 3.1.2 demystifies the process of grounding. It presents the Traveling Salesman problem encoding that is formalized using CAS(LIA) "schemata" rules — rules that contain "ASP" variables and hence can be seen as abbreviations for groups of corresponding ground/propositional CAS(LIA) rules such as presented in preliminaries. Within this section, we consider ground/propositional programs for simplicity.

Listing 1. *Encoding of LIA Logic in* GRINGO *version 5.*

```
#theory lia {
    linear_term {
    - : 2, unary;
    * : 1, binary, left;
    + : 0, binary, left;
    - : 0, binary, left
    };

    dom_term {
    - : 3, unary;
    + : 3, unary;
    * : 2, binary, left;
    + : 1, binary, left;
    - : 1, binary, left;
    .. : 0, binary, left
    };

    &dom/0 : dom_term, {=}, linear_term, head;
    &sum/0 : linear_term, {<=,>=,>,<,=,!=}, linear_term, any;
    &logic/1 : linear_term, head
}.
```

Consider Listing 1. It introduces the reader to the LIA language specification for grounder GRINGO used within EZSMTV3, which echoes the one utilized within CLINGCON version 5

(CLINGCONV5)[2] – the latest version of system CLINGCON rooted in the ideas by Banbara et al. (2017). Thus, any CAS(LIA) program for EZSMTV3 can be seen as a program written for CLINGCONV5 so that it can be solved by that system also. (It is due to remark that CLINGCONV5's specification has additions that for instance specify such a directive as the `&show` statement. Yet, EZSMTV3 does not support statements of the kind.) We can see the specification in Listing 1 as a collection of requirements on the kinds of statements that we expect GRINGO to process. We refer the reader to the paper by Gebser et al. (2016) for more details and intuitions behind the presented theory specification. Here we utilize examples to illustrate its purpose. In addition to syntactic restrictions on the kinds of statements supported by specifications of Listing 1, we pose additional requirements on these expressions, which have to be verified at the level when GRINGO output is being processed. Within EZSMTV3, we adopt the requirements closely related to these described by Banbara et al. (2017, Pages 12 and 13) for the constraints expressed using key words `&dom` and `&sum`. We now summarize the requirements and also discuss the nature of these constraints and how they are captured by EZSMTV3.

Domain Constraints have the form

$$\&dom\{d_1;\ldots;d_m\} = t, \tag{14}$$

where:

- d_i $(1 \leq i \leq m)$ can be u or a range $v..w$, with u, v, w being of the form (8) so that
  - a_i and x_i $(1 \leq i \leq n)$ are integers (with typical conventions such as, for instance, if one of the coefficients is in the multiplications of this expression is 1 it can be omitted) and thus, u, v, and w can be evaluated to integers); and
  - $v \leq w$.
- t is a constraint variable.

If expression (14) is such that every d_i $(1 \leq i \leq m)$ is either u or a range $v..w$, with u, v, w being *integers* we call this statement *normal*. This expression is intuitively understood as imposing the following requirement on values that constraint variable t can be mapped to. Namely, any value in the following set $\bigcup_{i=1}^{n}[d_i]$, where

$$[d] = \begin{cases} \{\text{u}\} & \text{if } d \text{ is } u \\ \{\text{v,..,w}\} & \text{if } d \text{ is } v..w, \text{ where } v \leq w. \end{cases}$$

Internally, EZSMTV3 simplifies $\&dom$ statements by evaluating possibly complex linear expressions occurring in these statements into corresponding integers resulting in the normal $\&dom$ statement. For instance, consider the following lines to occur in some EZSMTV3 program

```
&dom{1..3; 5+3*4} = x:- a, not b.
&dom{1+2..4*4} = x.
```

Internally, they will be simplified by the system into

$$\begin{aligned} &\&dom\{1..3; 17\} = x \leftarrow a,\ not\ b. \\ &\&dom\{3..16\} = x. \end{aligned} \tag{15}$$

[2] The theory specification used within CLINGCONV5 is located at https://github.com/potassco/clingcon/blob/master/libclingcon/clingcon/parsing.hh .

Syntactically, a (ground) EZSMTV3 rule containing normal $\&dom$ constraint has the form

$$D \leftarrow B, \tag{16}$$

where D is expression (14) and B is the body of this rule understood as in (2). Given the fact that within EZSMTV3 we utilize SMT(LIA) formulas behind the stage to reason over a CAS(LIA) program we present the semantics of statement (16) by means of translating it into an SMT(LIA) formula that has to be satisfied whenever statement (16) appears in the considered program. We view (16) as an abbreviation for the following SMT(LIA) implication

$$B \rightarrow \big([[d_1]] \vee \cdots \vee [[d_n]]\big),$$

where

$$[[d]] = \begin{cases} t = u & \text{if } d \text{ is } u \\ \big(t \geq v \wedge t \leq w\big) & \text{if } d \text{ is } v..w. \end{cases}$$

Recall that we identify body B with the respective conjunction. When B is empty (as, for instance, in the second line of (15)), we can simplify the implication above and identify it with an expression

$$[[d_1]] \vee \cdots \vee [[d_n]].$$

For instance, the ground EZSMTV3 rules listed in (15) are understood as the conjunction of the following SMT(LIA) formulas:

$$(a \wedge \neg b) \rightarrow \big((x \geq 1 \wedge x \leq 3) \vee x = 17\big),$$
$$(x \geq 3 \wedge x \leq 16).$$

Here it is due to note that CLINGCON is based on finite domain constraint solving so that in its implementation constraint variables over integers are considered within a default domain $-2^{30} \,..\, 2^{30}$ unless a $\&dom$ expression is provided for this variable that restricts its range; in case of EZSMTV3 no restrictions on the range of integers are considered by default.

Linear Constraints have the form

$$\&sum\{t_1; \ldots; t_n\} \bowtie t_{n+1}, \tag{17}$$

where:

- each t_i is an integer linear expression[3];
- $\bowtie$ belongs to (10).

This syntax captures expressions of the form $t_1 + t_2 + \cdots + t_m \bowtie t_{m+1}$. In turn, using standard algebraic operations this expression can be transformed into an integer linear constraint. We view (17) as an irregular atom corresponding to an underlying integer linear constraint (note that there may be multiple equivalent representations of such a constraint and any of these suffice for our purposes; indeed $x < 1$ can be seen as an equivalent representation to $x - 1 \leq 0$).

For instance, the expression of the form

```
&sum{2*2;3+x+(5+2)*z}=y
```

[3] Within the implementation, integer linear expressions are understood more liberally than defined here so that, for example 2×2 or $(5 + 2) \times z$ are considered within the realm of allowed syntax.

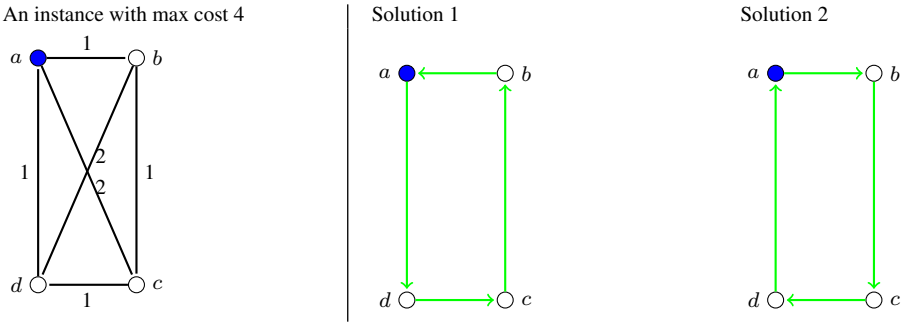

Fig. 2. Sample TS Instance and Solutions

occurring within a ground EZSMTV3 program is identified with an irregular atom

$$|x - y + 7 \times z = -7|$$

which has a natural mapping into respective LIA constraint (recall our convention to use vertical bars to denote irregular atoms).

Syntactically, a (ground) EZSMTV3 rule may contain expressions of the form (17) both in the head and the body of the rule, while EZSMTV3 identifies them with respective irregular atoms. For instance, the rule of the form

```
&sum{2*2;3+x+(5+2)*z}=y:- a, not b.
```
(18)

is understood as a denial

$$\leftarrow \neg|x - y + 7z = -7|,\ a,\ not\ b. \tag{19}$$

Example 6. *Recall a program from Example 4. Using the described* EZSMTV3 *CAS(LIA) language this program has the following form*

```
{a}.
b:-a.
c:-not a.
:-a, &sum{x}>=12.
```

3.1.2 Traveling Salesman Problem as CAS(LIA) Program

Let us state a *variant* of the *Traveling Salesman Problem*:

We are given a graph with nodes as cities and edges as roads. Each road directly connects a pair of cities, and costs a salesman some time to go through (time is expressed as a positive integer value in this variant of the problem). The salesman is supposed to pass each city exactly once. Find: a route *traversing* all the cities, yet only once visiting each one of them, under *certain* maximum cost *of total time.*

In the classical formulation of the TS problem, a route with the minimum cost is of interest. Here we state a decision problem in place of a related optimization problem. Also, in the classical formulation there are no restriction on weights over routes being integers.

Figure 2 shows an instance of the TS problem (a weighted graph). Listing 2 encodes this representation as a set of facts. On the right hand side of Figure 2, we find two solutions to this problem.

Listing 2. *Encoding of the TS Instance.*

```
city(a). city(b). city(c). city(d).
initial(a).
road(a,b).   road(b,c).   road(c,d).
cost(a,b,1). cost(b,c,1). cost(c,d,1).
road(d,a).   road(a,c).   road(b,d).
cost(d,a,1). cost(a,c,2). cost(b,d,2).
maxCost(4).
```

Listing 3 presents the CASP encoding for the TS problem, whose instances are provided in the style of the instance presented in Listing 2. This CASP encoding respects the LIA logic and supports the syntax specified by the theory specification in Listing 1. Let us start by stating intuitions behind this encoding. The first line specifies that the `road` relation is symmetric. The second line suggests that the `cost` of the road taken in either directions is the same. Line 4 specifies that for each city in the problem exactly one road that leads away from the city has to be part of the solution encoded by binary relation `route`. Line 5 specifies that for each city exactly one road that leads into this city has to be part of the solution. Lines 7 and 8 encode a notion of a reached city from an initial city. Line 10 requires that each city in the problem is identified as reached from the initial city. Lines 12 through 15 utilize constructs, whose syntax is defined within the theory specification in Listing 1. In other words, in the absence of the code within Listing 1 GRINGO would identify these lines as outside of the scope of its applicability. Line 12

- introduces irregular atoms into discourse – atoms that refer to constraint variables and
- specifies possible values for these constraint variables.

Lines 13 through 15 state requirements/constraints on these variables. In particular, Line 12 specifies a domain of possible values for the instances of constraint variables of the form `c(X,Y)`. Namely, their domains are restricted by two values: one being 0 and another being the costs associated with the roads from X to Y. Lines 13 and 14 state the conditions on when instances of constraint variables `c(X,Y)` are assigned 0 or the associated cost. Line 15 specifies an integer linear constraint that states that the sum of all possible instances of constraint variables `c(X,Y)` should not exceed the maximum cost specified.

Listing 3. *Encoding of the TS Problem.*

```
1  road(Y,X):-road(X,Y).
   cost(Y,X,C):-cost(X,Y,C).
3
   1{route(X,Y): road(X,Y)}1:-city(X).
5  1{route(X,Y): road(X,Y)}1:-city(Y).

7  reached(X):-initial(X).
   reached(Y):-reached(X), route(X,Y).
9
   :-city(X), not reached(X).
11
   &dom {0;C} = c(X,Y) :- cost(X,Y,C).
13 &sum {c(X,Y)} =0 :-cost(X,Y,C), not route(X,Y).
   &sum {c(X,Y)} =C :-cost(X,Y,C), route(X,Y).
15 :- &sum {c(X,Y):cost(X,Y,C)} > W, maxCost(W).
```

It is easy to see that the considered encoding of the TS problem contains kinds of rules that are outside of the syntax of logic programs presented in the Background section. In particular, this program uses

- ASP variables — namely, X, Y, C, and W (identifiers starting with the capital letters) — so that program's atoms are not propositional;
- aggregate expressions within Lines 4 and 5 (in fact, each of these rules is an abbreviation for two rules, where one rule contains a choice expression in the head (a choice rule) and another rule is a constraint containing count-aggregate expression in the body).

Aggregate expressions are the common constructs within the practice of answer set programming. We refer an interested reader to the work by Calimeri et al. (2020a), for instance, for more formal details on aggregates. Here let us informally discuss their roles using Line 4 within Listing 3 as an example. The expression presented on the line is an abbreviation for two rules:

```
{route(X,Y)}:- road(X,Y), city(X).
:- not #count{X,Y:route(X,Y),road(X,Y)}=1, city(X).
```

The first line can be intuitively read as *any road leading from some city* may form *a part of the route*. The word *may* points at the *choice*. The second rule contains a *count*-aggregate expression and states that *for a city exactly one tuple corresponding to a road should be considered to be part of the route*.

This is a good place to demystify the effects of the grounding process and the role of ASP variables. The process of *grounding* is defined through ensuring that ASP variables are instantiated with all possible permutations of the object constants, so that a rule with ASP variables can be seen as an abbreviation for the group of propositional rules instantiated with the object constants occurring in the program. Grounder GRINGO performs a process denoted as *intelligent grounding* that is similar to a procedure well described by Faber et al. (2012). While performing intelligent grounding a system attempts not only to instantiate given logic rules with all possible object constants of the considered program, but also to perform some simplifications and reductions that still guarantee that the produced propositional program has the same answer sets as the one that would be produced by the straight-forward instantiation of grounding. The exact procedure behind GRINGO is best documented by Kaminski (2023). Lines in Listings 2 and 4 form the output of GRINGO, when it is invoked with the flag *-t* on the code obtained by concatenating the lines within Listings 1, 2 and 3. Flag *-t* instructs GRINGO to print output in human readable form. Let us now discuss intuitions on which snippets of code within Listings 2 and 3 are relevant in producing propositional rules in Listing 4:

- Lines 1 through 4 are produced by GRINGO by relying on the facts in Listing 2 and Lines 1 and 2 in Listing 3.
- Lines 5-20 are produced by GRINGO by relying on the facts in Listing 2; Lines 1 and 3 in Listing 4; and Lines 4 and 5 in Listing 3.
- Lines 21-28 are produced by GRINGO by relying on the fact in Line 2 in Listing 2; Lines 5-20 in Listing 4 suggesting which tuples may appear in `route` relation; and Lines 7 and 8 in Listing 3.
- Lines 30-35 are due to Line 12 in Listing 3 and the `cost` relations established in Listing 2 and Lines 2 and 4 in Listing 4.

- Lines 30-35 are due to Line 12 in Listing 3 and the `cost` relations established in Listing 2 and Lines 2 and 4 in Listing 4.
- Lines 36-53 are due to Lines 13 and 14 in Listing 3; the `cost` relations established in Listing 2 and Lines 2 and 4 in Listing 4; and Lines 5-20 in Listing 4.
- Line 54 is due to Line 15 in Listing 3; the `maxCost` given in Listing 2; the `cost` relations established in Listing 2 and Lines 2, 4 in Listing 4.

How rules with `&sum` *and ASP variables connect to CAS(LIA) rules* Now that Listing 4 provides us with the propositional rendering of CAS program encoding of our running example of the TS problem let us use it to connect to the formal notions introduced in Section 2. Consider rules in Lines 36 and 54-55 in Listing 4. We can view these as corresponding to the following two rules written in the syntax discussed in Section 2

$$\begin{aligned} &\mid c(a,b) = 0 \mid \leftarrow \mathit{not}\ \mathit{route}(a,b). \\ &\leftarrow\mid c(a,b) + c(b,c) + c(c,d) + c(d,a) + c(a,c) + c(b,d) + \\ &\quad c(d,b) + c(c,a) + c(a,d) + c(d,c) + c(c,b) + c(b,a) > 4 \mid . \end{aligned}$$

In these rules two irregular atoms appear marked by vertical bars. They naturally translate into LIA constraints with twelve integer constraint variables including $c(a,b)$ and $c(b,c)$, for example.

Invoking EZSMTV3 A unique capability of EZSMTV3 lies in the fact that it provides a frontend to distinct SMT solvers, namely, CVC4, CVC5, YICES, and Z3. One may specify an SMT solver of interest at the command line. In addition, one may specify whether single or multiple answer sets (or extended answer sets) are of interest.

Let us assume the presence of the files

1. `tsp.inst` – whose content is present in Listing 2;
2. `tsp.enc` – whose content is present in Listing 3 together with an additional directive of the form `#show route/2.` This directive instructs the system to only display atoms formed with this predicate symbol in the output.

Then, the command line

```
ezsmt tsp.inst tsp.enc -s z3 -e 0 -E
```

produces the output given in Listing 5. This output matches the solutions listed in Figure 2: Answer 1 and 2 encode Solutions 1 and 2, respectively. Within this command line in addition to specifying files containing the program to process, we state

- a backend SMT solver that should be used – here, Z3 – with `-s z3`,
- a number of answer sets that should be enumerated – here, *all* – with `-e 0`,
- a request to consider extended answer sets within the enumeration process `-E`.

It is due to remark that within the code base of EZSMTV3, the specifications presented in Listing 1 are used to instruct GRINGO (invoked within) on what expressions it should find syntactically valid (Section 3.2 narrates the details on the architecture of EZSMTV3).

Let us now speak about the distinction between `-e 0` and `-e 0 -E` settings. The former is concerned with enumerating distinct answer sets disregarding the specific values that constraint variables obtain. The later will instruct EZSMTV3 to enumerate distinct extended answer sets. Let us consider a simple program presented in Listing 6. The EZSMTV3 system invoked with

Listing 4. *Part of the Grounded TS Problem with respect to TS Instance in Listing 2*

```
road(d,b).    road(c,a).    road(a,d).
cost(d,b,2).  cost(c,a,2).  cost(a,d,1).
road(d,c).    road(c,b).    road(b,a).
cost(d,c,1).  cost(c,b,1).  cost(b,a,1).
1<=#count{0,route(a,b):route(a,b);0,route(a,c):route(a,c);
                                  0,route(a,d):route(a,d)}<=1.
1<=#count{0,route(b,c):route(b,c);0,route(b,d):route(b,d);
                                  0,route(b,a):route(b,a)}<=1.
1<=#count{0,route(c,d):route(c,d);0,route(c,a):route(c,a);
                                  0,route(c,b):route(c,b)}<=1.
1<=#count{0,route(d,a):route(d,a);0,route(d,b):route(d,b);
                                  0,route(d,c):route(d,c)}<=1.
1<=#count{0,route(d,a):route(d,a);0,route(c,a):route(c,a);
                                  0,route(b,a):route(b,a)}<=1.
1<=#count{0,route(a,b):route(a,b);0,route(d,b):route(d,b);
                                  0,route(c,b):route(c,b)}<=1.
1<=#count{0,route(b,c):route(b,c);0,route(a,c):route(a,c);
                                  0,route(d,c):route(d,c)}<=1.
1<=#count{0,route(c,d):route(c,d);0,route(b,d):route(b,d);
                                  0,route(a,d):route(a,d)}<=1.
reached(a).              reached(b):-route(a,b).
reached(c):-route(a,c).  reached(d):-route(a,d).
reached(b):-route(d,b),reached(d).
reached(c):-route(d,c),reached(d).
reached(d):-route(c,d),reached(c).
reached(b):-route(c,b),reached(c).
reached(c):-route(b,c),reached(b).
reached(d):-route(b,d),reached(b).
:-not reached(b).   :-not reached(c).   :-not reached(d).
&dom{(0;1)}=(c(a,b)).   &dom{(0;1)}=(c(b,c)).
&dom{(0;1)}=(c(c,d)).   &dom{(0;1)}=(c(d,a)).
&dom{(0;2)}=(c(a,c)).   &dom{(0;2)}=(c(b,d)).
&dom{(0;2)}=(c(d,b)).   &dom{(0;2)}=(c(c,a)).
&dom{(0;1)}=(c(a,d)).   &dom{(0;1)}=(c(d,c)).
&dom{(0;1)}=(c(c,b)).   &dom{(0;1)}=(c(b,a)).
&sum{c(a,b)}=(0):-not route(a,b).
&sum{c(b,c)}=(0):-not route(b,c).
&sum{c(c,d)}=(0):-not route(c,d).
&sum{c(d,a)}=(0):-not route(d,a).
&sum{c(a,c)}=(0):-not route(a,c).
&sum{c(b,d)}=(0):-not route(b,d).
&sum{c(d,b)}=(0):-not route(d,b).
&sum{c(c,a)}=(0):-not route(c,a).
&sum{c(a,d)}=(0):-not route(a,d).
&sum{c(d,c)}=(0):-not route(d,c).
&sum{c(c,b)}=(0):-not route(c,b).
&sum{c(b,a)}=(0):-not route(b,a).
&sum{c(a,b)}=(1):-route(a,b).  &sum{c(b,c)}=(1):-route(b,c).
&sum{c(c,d)}=(1):-route(c,d).  &sum{c(d,a)}=(1):-route(d,a).
&sum{c(a,c)}=(2):-route(a,c).  &sum{c(b,d)}=(2):-route(b,d).
&sum{c(d,b)}=(2):-route(d,b).  &sum{c(c,a)}=(2):-route(c,a).
&sum{c(a,d)}=(1):-route(a,d).  &sum{c(d,c)}=(1):-route(d,c).
&sum{c(c,b)}=(1):-route(c,b).  &sum{c(b,a)}=(1):-route(b,a).
:-&sum{c(a,b); c(b,c); c(c,d); c(d,a); c(a,c); c(b,d);
       c(d,b); c(c,a); c(a,d); c(d,c); c(c,b); c(b,a)}>(4).
```

`-e 0` on this sample program produces two solutions total that correspond to distinct answer sets, while EZSMTV3 invoked with `-e 0 -E` produces three extended answer sets.

Listing 5. EZSMTV3 *output for the TS sample problem.*

```
Answer 1: route(a,d) route(d,c) route(c,b) route(b,a)
c(a,b)=0 c(b,c)=0 c(c,d)=0 c(d,a)=0 c(a,c)=0 c(b,d)=0
c(d,b)=0 c(c,a)=0 c(a,d)=1 c(d,c)=1 c(c,b)=1 c(b,a)=1
Finished round 1 in 118ms
  0ms SMT Check Satisfiability
  16ms SMT Get Values

Answer 2: route(a,b) route(b,c) route(c,d) route(d,a)
c(a,b)=1 c(b,c)=1 c(c,d)=1 c(d,a)=1 c(a,c)=0 c(b,d)=0
c(d,b)=0 c(c,a)=0 c(a,d)=0 c(d,c)=0 c(c,b)=0 c(b,a)=0
Finished round 2 in 67ms
  0ms SMT Check Satisfiability
  35ms SMT Get Values
```

Listing 6. *Sample CAS(LIA)* EZSMTV3 *program with multiple (extended) answer sets*

```
&dom{1..3}=x.
{a}.
&sum{x}=1:- a.
&sum{x}<3:- not a.
```

3.1.3 EZSMTV3 *CAS(LRA), CAS(LIRA), and CAS(IDL) Languages*

We now present the details on the encodings of the constraints supported by the EZSMTV3 system when it assumes the roles of CAS(LRA), CAS(LIRA), and CAS(IDL) solvers, respectively.

The CAS(LRA) Language Section 3.1.1 described the CAS(LIA) language supported by EZSMTV3. The same section can be seen as the one describing the details of the CAS(LRA) language supported by the system modulo the condition that non-integer real numbers are listed using quotation marks. For instance, the expression of the form

$$\texttt{\&sum\{"2.4"*2;3+x+(5+2)*z\}=y} \tag{20}$$

occurring within a ground CAS(LRA) EZSMTV3 program is identified with an irregular atom

$$|x - y + 7 \times z = -7.8| \tag{21}$$

which has a natural mapping into a respective LRA constraint with three constraint variables over reals, namely, x, y, and z. Just as we attempted to make the fragment of the CAS(LIA) language supported by EZSMTV3 compatible with the CLINGCON language, we also attempted to make the fragment of the CAS(LRA) language supported by EZSMTV3 compatible with the CLINGO[LP] (Janhunen et al. 2017) language so that a CAS(LRA) program written for EZSMTV3 can be processed by CLINGO[LP] system (modulo omitting the directive $\&logic(lra)$. described below). It is due to note that

- CLINGO[LP] supports an additional optimization statement that is outside of the scope of EZSMTV3 and
- CLINGO[LP] is less permissive in the form of the $\&sum$ statements it allows. For example, the expression of the form (20) is considered syntactically invalid by CLINGO[LP]. Yet, recall how this expression corresponds to irregular atom (21), which we could encode in

the syntax understood by CLINGO[LP] as follows

```
&sum{x;(-1)* y;7* z}="-7.8".
```

In addition, an EZSMTV3 CAS(LRA) program may contain the following declaration

```
&logic(lra).
```

This declaration instructs EZSMTV3 that the program it is dealing with is CAS(LRA) program. Alternatively, a flag `-l 1` within the command line can be used to invoke EZSMTV3 instructing it to process a CAS(LRA) program.

The theory specification for CAS(LRA) is identical to the specification in Listing 1 modulo an additional line inserted after line 18 of that listing:

```
&logic/1 : var_term, head;
```
(22)

This additional line allows EZSMTV3 to introduce the directive `&logic(lra).`

The CAS(LIRA) Language The theory specification for CAS(LIRA) programs required by GRINGO is identical to the specification listed in Listing 1 modulo two additional lines inserted after line 18 of that listing. The first line is presented in (22) and the second line follows:

```
&type/2: var_term, head;
```

These two additional lines allow EZSMTV3 to process the directives of the following kind

```
&logic(lira).
&type{x; y}=int.
```
(23)

In this snippet of sample code, the first line declares to EZSMTV3 that the program it currently considers is within the CAS(LIRA) language; alternatively, a user may use flag `-l 2` within the command line to invoke EZSMTV3 in such a mode. Before we discuss the role of the second line let us introduce a term *functional name* of a constraint variable. Within the programs that EZSMTV3 supports a constraint variable may take one of two forms

$$v$$
$$v(t_1, \ldots, t_n).$$

In these expressions, we call v a *functional name* of a constraint variable. The sample code `&type{x; y}=int.` states a condition that any constraint variable with functional name x or y occurring in a given program is considered to be integer. Any constraint variable occurring within a program whose functional name is missing from a declaration of this kind is considered to be a constraint variable over reals.

For the remainder, Section 3.1.1 can be seen as a section describing the details of the CAS(LIRA) language supported by EZSMTV3 modulo the condition that real numbers that are not integers are listed using quotation marks. For instance, expression (20) occurring within a ground CAS(LIRA) EZSMTV3 program that contains lines in (23) and no other type-declarations is identified with an irregular atom of the form (21), which has a natural mapping into the respective LIRA constraint with integer constraint variables x and y, and real constraint variable z.

The CAS(IDL) Language The theory specification for CAS(IDL) programs is in Listing 7. This specification allows EZSMTV3 to provide support for

- difference logic constraints of the form (12); and
- the declaration

  ```
  &logic(idl).
  ```

  This directive instructs EZSMTV3 that it is dealing with CAS(IDL) program; alternatively, a user may use flag `-l 3` for the same instruction.

Listing 7. *Encoding of IDL Logic in* GRINGO *version 5.*

```
#theory idl {
    linear_term {
    - : 2, unary;
    * : 1, binary, left;
    + : 0, binary, left;
    - : 0, binary, left
    };

    dom_term {
    - : 3, unary;
    + : 3, unary;
    * : 2, binary, left;
    + : 1, binary, left;
    - : 1, binary, left;
    .. : 0, binary, left
    };

    &dom/0 : dom_term, {=}, linear_term, head;
    &diff/0 : linear_term, {<=,>=,<,>,=,!=}, linear_term, any;
    &logic/1 : linear_term, head
}.
```

For instance, expressions of the form

```
&diff{x-y}<=5
```
(24)

and

```
&diff{x}<y
```
(25)

occurring within a ground CAS(LRA) EZSMTV3 program are identified with irregular atoms

$$|x - y \leq 5|$$

and

$$|x < y|,$$

respectively. Both of these irregular atoms have a natural mapping into respective IDL constraints.

It is due to note that the CAS(IDL) EZSMTV3 program is often suitable for processing with solver CLINGO[DL] (Janhunen et al., 2017). Yet, the dialect of CAS(IDL) EZSMTV3 programs permits the following expressions that are outside the language fragment of CLINGO[DL]:

- the *&logic* directive, which specifies the IDL logic to be used within the encoding. This directive can be eliminated from programs when proper flag is used to invoke EZSMTV3.
- the *&dom* specifications for variables. These expressions are treated in the same way as described for the case of CAS(LIA) fragment, and it is easy to see that the resulting SMT

formulas are within the SMT(IDL) fragment. CLINGO[DL] bypasses the support for this language feature.

3.2 EZSMTV3 *Architecture*

Figure 3 presents the architecture of the EZSMTV3 system. This system is able to process CAS(LIA), CAS(LRA), CAS(LIRA), and CAS(IDL) utilizing the language constructs as specified in Section 3.1. We start by briefly describing the system's workings. Then we provide details for its more complex elements.

At first, the EZSMTV3 system determines which kind of program it is given – CAS(LIA), CAS(LRA), CAS(LIRA), or CAS(IDL). After that, it utilizes grounder GRINGO (Gebser et al.; Kaminski et al., 2016; 2023) to eliminate ASP variables. System GRINGO produces a ground/propositional program in the format called Answer Set Programming Intermediate Format (ASPIF) (Gebser et al.; Kaminski et al., 2016; 2023). The grounded program written in ASPIF is then read by the Reader component of the system, which stores the rules, regular and irregular atoms from the program accordingly. The logic interface is then set and the corresponding constraint variables are declared with specified types. Routines of system CMODELS(DIFF) (Shen and Lierler, 2018b) are used to compute completion and level rankings of the program. Then, the EZSMTV3 system translates the completion augmented with level rankings into SMT formulas in the syntax of the standard SMT-LIB language (Barrett et al., 2010). These formulas are then fed into an SMT solver, which finds a model of the formulas. Each found model corresponds to an extended answer set of the given program. We now provide more essential details behind each sub-component of the EZSMTV3 system depicted in Figure 3.

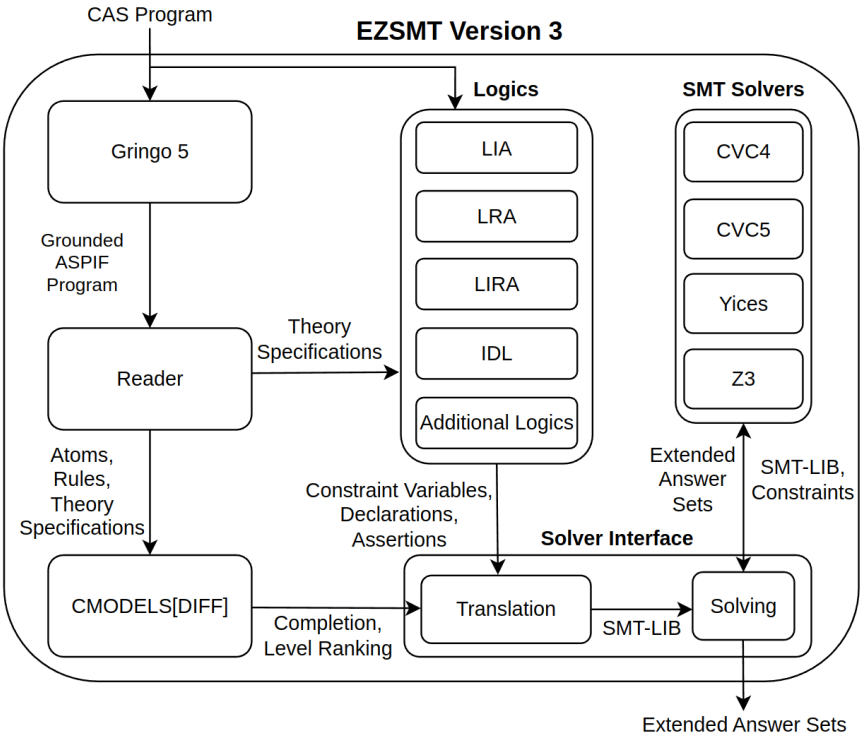

Fig. 3. EZSMTV3 Architecture

The GRINGO *5 block* Section 3.1.2 used an instance of a Traveling Salesman problem formalized as CAS(LIA) program to illustrate the process of grounding. Within the EZSMTV3 grounder GRINGO version 5 is utilized. It is a sub component of already mentioned system CLINGO 5. Sections 3.1.1 and 3.1.3 highlighted the presence and importance of theory specifications – recall, Listings 1 and 7 – that enable us to define the syntactic constructs of various CAS languages that GRINGO is then able to process. For example, the theory specification in Listing 7 instructs GRINGO that atoms of the form (24) and/or (25) are valid constructs syntactically. Given such a theory specification, GRINGO is able to ground the respective programs and encode these using the ASPIF format (Gebser et al.; Kaminski et al., 2016; 2023). This format is best documented in the appendix of the extended version of the paper by Gebser et al. (2016) available at `https://www.cs.uni-potsdam.de/wv/publications/DBLP_conf/iclp/GebserKKOSW16x.pdf` . The grounded logic program in ASPIF can be seen as a list of statements in normal form that utilize numerical values to represent program's atoms and its internal structure.

The Reader block The ASPIF statements generated by grounder GRINGO are interpreted by the Reader block. The Reader block parses ASPIF statements provided by GRINGO and stores the information about rules, regular, irregular atoms, constraint variables into the internal data structures of EZSMTV3.

It is due to note that GRINGO may recognize some expressions that are outside of the considered syntax as valid. For instance, in the realm of theory specification for IDL constraints

presented in Listings 7, GRINGO will recognize the following expression as a valid irregular atom

```
&diff{x-y}<=z
```

if it occurs within head or body of some of its rules. Intuitively this expression maps into

$$x - y < z$$

that is outside the syntax of IDL constraints. For this reason the Reader block also implements additional checks to warn the user about mistakes in thevconsidered encodings.

The Logics block Within the Logics block of EZSMTV3, we first determine whether a given program is of kind CAS(LIA), CAS(LRA), CAS(LIRA), or CAS(IDL). For that purpose $\&logic$ and/or command line $-l$ directives are used as specified in Section 3.1. In case of conflicting information between directives expressed by $\&logic$ statement within the considered program and command line, the $\&logic$ statement has higher precedence. By default, in the absence of any $\&logic$ statement or flag $-l$ in the command line the program is considered to be within the CAS(IDL) fragment. Using the information about a kind of a given program, EZSMTV3 is able to assign each constraint variable occurring within irregular atoms a proper domain.

The CMODELS(DIFF) *block* The EZSMTV3 system incorporates a number of routines stemming from the answer set solver called CMODELS(DIFF) (Shen and Lierler, 2018b). In particular, it borrows the CMODELS(DIFF) code that determines whether a given program is tight, performs so called completion on a given program, computes level ranking formulas in case if a formula is not tight, and clausifies the resulting formulas. These routines are key to implementing the translation provided in the concluding part of Section 2.3. Indeed, bullets 1 and 2 of that translation are captured by the process of completion and construction of level ranking formulas. Just as in the case of CMODELS(DIFF), we can instruct EZSMTV3 to construct different kinds of level ranking formulas using flags `-levelRanking`, `-levelRankingStrong`, `-SCClevelRanking`, and `-SCClevelRankingStrong`. We refer the reader to the work by Shen and Lierler (2018b) for more details on the different kinds of level ranking formulas supported by CMODELS(DIFF) and EZSMTV3. The default behavior of system EZSMTV3 is captured by `-SCClevelRanking`.

By default, we set the upper bound for a level ranking variable corresponding to an atom as the number of atoms inside the strongly connected component of the program's dependency graph containing the corresponding atom. A larger upper bound can also be selected using the flag `--all-atoms-upper-bound` which sets the upper bound as the total number of atoms inside the program. Finally, the resulting formulas are stored in semi-Dimacs format documented by Susman and Lierler (2016b).

The Solver Interface block The Solver Interface block is responsible for two tasks, namely, translation and solving. In the translation phase, the formulas in semi-Dimacs form obtained from a previous block are transformed into the syntax of the Standard language for SMT solvers called SMT-LIB (Barrett et al., 2010). The translation procedure is in the style of the one described by Susman and Lierler (2016b). During this transformation, in addition to encoding the SMT formula corresponding to a given program and computed in the CMODELS(DIFF) block, the declarations for the used SMT logic, propositional atoms, and constraint variables are included.

Let us consider a simple example to illustrate transformations occurring within EZSMTV3. Assume some CAS(LIA) program that contains rule (18), which we understand as a denial (19). The CMODELS(DIFF) block will turn this denial into a group of SMT(LIA) formulas, namely,

$$|x - y + 7z = -7| \vee \neg a \vee b,$$
$$|x - y + 7z = -7| \leftrightarrow x - y + 7z = -7.$$

Within an SMT-LIB code for the considered program we will find the following lines that we annotate with the comments for readability (comments start with semicolon):

```
; Quantifier free Linear Arithmetic: SMT(LIA) language
(set-logic QF_LIA)

; Declaration of boolean and integer variables used
(declarefun a () Bool)
(declarefun b () Bool)
(declarefun x () Int)
(declarefun y () Int)
(declarefun z () Int)

; |EZSMT_THEORY(4)| is the name given within Ezsmtv3
; to irregular atom |x-y+7z=-7|
(declarefun |EZSMT_THEORY(4)| () Bool)

; SMT(LIA) formulas stated above encoded in SMT-LIB
(assert (or |EZSMT_THEORY(4)| (not a) b))
(assert (= |EZSMT_THEORY(4)| (= (+ x (* (- 1) y) (* 7 z)) (- 7))))
```

Figure 1 summarized what kind of CAS programs are mapped into what kind of SMT formulas. The table below details a logic declaration statement in SMT-LIB format appropriate to invoke a correct solving routine for the respective SMT formula:

SMT(LIA)	`(set-logic QF_LIA)`
SMT(LRA)	`(set-logic QF_LRA)`
SMT(IDL)	`(set-logic QF_IDL)`
SMT(LIRA)	`(set-logic AUFLIRA)`

In the solving phase, an SMT solver is given an obtained SMT-LIB theory. The back-and-forth communication between the Solver Interface block and the SMT solver of choice makes it possible for the system to output multiple (extended) answer sets. The command line directives can be used to provide EZSMTV3 with a specific number of solutions to be computed.

All SMT solvers currently supported by the EZSMTV3 system, namely CVC4, CVC5, YICES, and Z3 implement so called incremental solving. Within incremental solving settings one may invoke an SMT solver on a theory and instruct it to compute a model; once that model is computed the SMT solver puts its computation on hold and waits for further instructions. At that point it is possible to add more assertions to the theory already populated within the SMT solver's data structures and ask it to look for yet another model of an updated theory. This process can be repeated. EZSMTV3 utilizes incremental solving for computing multiple answer sets (or extended answer sets) by iteratively adding an assertion that is false when previously computed (extended) answer set holds. Given an answer set A of CAS program P the negation of the following formula

$$\bigwedge_{atom\ a \in A} a \ \wedge \bigwedge_{a\ occurs\ in\ P\ and\ a \notin A} \neg a \tag{26}$$

forms such an assertion. Given an extended answer set $\langle A, \nu \rangle$ of CAS program P, the negation of the formula formed by the conjunction of formulas (26) and

$$\bigwedge_{constraint\ variable\ x\ occurs\ in\ P} x = \nu(x)$$

forms such an assertion.

This way of implementing enumeration of multiple (extended) answer sets is inspired by the enumeration done by answer set solver CMODELS(DIFF) (Shen and Lierler 2018b, Section 5). Yet, utilization of incremental mode of SMT solving is unique to EZSMTV3. The EZSMT+ invoked SMT solvers from scratch each iteration.

The SMT Solvers block In our work, we implemented support within EZSMTV3 for four SMT solvers, namely, CVC4 (Barrett et al., 2011), CVC5 (Barrett et al.,), YICES (Dutertre and De Moura, 2006), and Z3 (De Moura and Bjørner, 2008). Yet, given that we use SMT-LIB to interface these solvers it requires limited effort to implement support for any other solver supporting SMT-LIB format. This implementation effort mainly has to be directed towards processing output of the solvers as their output formats are not identical.

4 Optimizations

We now turn our attention to weak constraints/optimizations supported by EZSMTV3. It is due to note that such language constructs were out of scope for the system's predecessor EZSMT+. The EZSMTV3 system supports the syntax of weak constraints as they are described by Calimeri et al. (2020b) as part of the ASP-Core2 standard language of logic programs. Here we provide the natural extension of the semantics of these statements to the CAS programs.

Calimeri et al. (2020b) present the syntax of weak constraints allowing ASP variables in the context; then, grounding is used to obtain propositional program with weak constraints and the notion of an optimal answer set is defined. Here, we present all relevant definitions using the propositional case but note that EZSMTV3 provides support for non-grounded statements that are tackled by the means of grounder GRINGO.

A *weak constraint* has the form

$$:\sim a_1, \ldots, a_j,\ not\ a_{j+1}, \ldots,\ not\ a_m[w@\ell, t_1, \ldots, t_n], \tag{27}$$

where $m > 0$ and $a_1, \ldots, a_m$ are atoms, w (weight) is an integer, l (level) is a positive integer, t_i ($n \geq 0$) are symbols. In the sequel, we abbreviate expression

$$:\sim a_1, \ldots, a_j,\ not\ a_{j+1}, \ldots,\ not\ a_m \tag{28}$$

occurring in (27) as D and identify it with the propositional formula

$$a_1 \wedge \ldots \wedge a_j \wedge\ \neg a_{j+1} \wedge \ldots \wedge\ \neg a_m. \tag{29}$$

We may refer to this formula as the *body* of a weak constraint.

Definition 6 (Optimization program or o-program)**.** *An* optimization program *(or* o-program*) over vocabulary σ is a pair (P, W), where P is a CAS program over σ and W is a finite set of weak constraints over σ. Let $\mathcal{P} = (P, W)$ be an optimization program over vocabulary σ (intuitively, P and W forms* hard *and* soft *fragments, respectively). Set X of atoms over σ is an* answer set *of $\mathcal{P}$ when it is an answer set of P.*

By $\lambda(\mathcal{P})$ we denote the set of all levels associated with optimization program $\mathcal{P}$ constructed as $\{l \mid D[w@\ell, t_1, \ldots, t_n] \in W\}$. Given an answer set X of o-program $\mathcal{P}$, we map X and Π to a set of tuples as follows

$$\text{weak}(\mathcal{P}, X) = \{(w@\ell, t_1, \ldots, t_n) \mid D[w@\ell, t_1, \ldots, t_n] \in W \text{ and } X \models D\};$$

we are now ready to define a number associated with o-program, its answer set, and a level $\ell \in \lambda(\mathcal{P})$:

$$\mathcal{P}_\ell^X = \sum_{D[w@\ell, t_1, \ldots, t_n] \in \text{weak}(\mathcal{P}, X)} w$$

Definition 7 (Optimal answer sets)**.** *Let X and X' be answer sets of $\mathcal{P}$. Answer set X is* dominated by *X' if there is some integer $\ell \in \lambda(\mathcal{P})$ such that*

$$\mathcal{P}_\ell^{X'} < \mathcal{P}_\ell^X \tag{30}$$

and

$$\mathcal{P}_{\ell'}^{X'} = \mathcal{P}_{\ell'}^X \tag{31}$$

for all integers $\ell' > \ell$.

An answer set X^* of $\mathcal{P}$ is optimal *if there is no answer set X' of $\mathcal{P}$ such that X^* is dominated by X'.*

Example 7. *We now exemplify the definitions of an optimization program and an optimal answer set. Recall the CAS program constructed in Example 4. Let us denote it as that P_1. An optimal answer set of o-program*

$$(P_1, \{:\sim a.\ [-1@1]\}) \tag{32}$$

is $\{a\ b\}$; whereas program

$$(P_1, \{:\sim a.\ [1@1]\}) \tag{33}$$

has two optimal answer sets $\{c\}$ and $\{c, |x \geq 12|\}$. An optimal answer set of another o-program

$$\begin{array}{ll} (P_1, & \{ :\sim a.\ [-1@1] \\ & \ \ :\sim |x = 12|.\ [-2@1]\}) \end{array}$$

is $\{c \quad |x = 12|\}$.

Let us now consider slightly more complex o-programs. Let W_1 denote the set consisting of the following weak constraints:

$$\begin{array}{ll} :\sim a. & [-1@1] \\ :\sim b. & [-1@1] \\ :\sim a, b. & [-1@1] \\ :\sim c. & [-2@1] \end{array}$$

O-program (P_1, W_1) has two optimal answer sets, namely,

$$\{c\} \text{ and } \{c, |x \geq 12|\}. \tag{34}$$

Let W_2 denote the set consisting of the following weak constraints:

$$\begin{array}{ll} :\sim a. & [-1@1, l] \\ :\sim b. & [-1@1, m] \\ :\sim a, b. & [-1@1, n] \\ :\sim c. & [-2@1, o] \end{array}$$

O-program (P_1, W_2) *has a unique optimal answer set*

$$\{a\ b\}. \tag{35}$$

It is worth noting that an alternative syntax is frequently used by answer set programming practitioners when they expresses optimization criteria:

$$\#minimize\{w_1@\ell_1, t_{11}, \ldots, t_{1k_1} : lits_1;\ \ \ldots;\ \ w_n@\ell_n, t_{n1}, \ldots, t_{nk_n} : lits_n\}, \tag{36}$$

where $k_1, \ldots, k_n \geq 0$ and $lits_i$ is of the form $a_1, \ldots, a_j,\ not\ a_{j+1}, \ldots,\ not\ a_m$ so that $m > 0$ and $a_1, \ldots, a_m$ are atoms. This statement stands for n weak constraints

$$\begin{aligned} &:\sim lits_1[w_1@\ell_1 t_{11}, \ldots, t_{1k_1}] \\ &\ \ \ldots \\ &:\sim lits_n[w_n@\ell_n, t_{n1}, \ldots, t_{nk_n}]. \end{aligned}$$

Similarly, statement

$$\#maximize\{w_1@\ell_1, t_{11}, \ldots, t_{1k_1} : lits_1;\ \ \ldots;\ \ w_n@\ell_n, t_{n1}, \ldots, t_{nk_n} : lits_n\} \tag{37}$$

stands for n weak constraints

$$\begin{aligned} &:\sim lits_1[-1 \cdot w_1@\ell_1 t_{11}, \ldots, t_{1k_1}] \\ &\ \ \ldots \\ &:\sim lits_n[-1 \cdot w_n@\ell_n, t_{n1}, \ldots, t_{nk_n}]. \end{aligned}$$

Example 8. *Consider o-program* (32)*. The optimization requirement*

$$:\sim a.\ [-1@1]$$

of that program can be stated either as

$$\#minimize\{-1@1 : a\}$$

or as

$$\#maximize\{1@1 : a\}$$

Set W_2 *of weak constraints from Example 7 can be represented as*

$$\#minimize\{-1@1, l : a; -1@1, m : b; -1@1, n : a, b; -2@1, o : c\}.$$

4.1 EZSMT *3 implementation details*

We now turn our attention to the question of how the support for optimization statements is implemented within EZSMTV3.

We used propositional programs with weak constraints to introduce their semantics. Yet, EZSMTV3 supports weak constraints with ASP variables. As for any other language constructs, EZSMTV3 starts its processing by invoking grounder GRINGO to produce a propositional program. It is due to note that GRINGO makes additional transformations to weak constraints so that the resulting set of weak constraints has a simpler form than discussed in the earlier section. During this transformation auxiliary atoms are introduced into the program. When answer sets are computed for this new program the auxiliary atoms can be safely dropped to obtain the answer sets of the original program. The weak constraints that system EZSMTV3 is exposed to beyond

the point of grounding has one of the following forms

$$:\sim\ a.\ [w@\ell, t_1, \ldots, t_n] \tag{38}$$

$$:\sim\ not\ a.\ [w@\ell, t_1, \ldots, t_n] \tag{39}$$

where a is an atom. In addition, any weak constrain that appears within a ground program produced by GRINGO is such that the expression $w@\ell, t_1, \ldots, t_n$ appearing in that weak constraint is unique (for example, it could be used as an identifier for this constraint in the program). Let us call a program satisfying stated conditions – *gringo o-program*.

Here we avoid describing formally the procedure implemented within GRINGO for "normalizing" optimization statements. Yet, we use our sample sets W_1 and W_2 of weak constrains from Example 7 to hint at its details. Set W_1 will be rewritten by GRINGO in the following style

$$\begin{array}{ll} aux_1 \leftarrow a. & \\ aux_1 \leftarrow b. & \\ aux_1 \leftarrow a, b. & \\ :\sim aux_1. & [-1@1] \\ :\sim c. & [-2@1] \end{array}$$

whereas set W_2 will be rewritten by GRINGO as

$$\begin{array}{ll} aux_2 \leftarrow a, b. & \\ :\sim a. & [-1@1, l] \\ :\sim b. & [-1@1, m] \\ :\sim aux_2. & [-1@1, n] \\ :\sim c. & [-2@1, o] \end{array}$$

so that aux_1 and aux_2 are some fresh auxiliary atoms.

In order to process o-programs with weak constraints of the form (38) and (39), EZSMTV3 relies on the transformations proposed by Lierler (2023b; 2024) in the scope of so called w-systems. W-systems are meant as an abstraction to encapsulate various logic-based formalisms extended with optimization expressions. It is due to note that the semantic characterization of optimization statements utilized by Lierler (2023b; 2024) is in the tradition stemming from partial weighted MaxSat (Fu and Malik, 2006). We restate their semantics for the case of optimization programs studied here and point at the differences. Yet, for the case of gringo o-programs the semantics as stated here and the one studied by Lierler (2023b; 2024) coincide. Thus, the transformations that we mentioned in the beginning of the paragraph can be safely applied.

Consider a definition of a new number – ${\mathcal{P}^{\#}}_{\ell}^{X}$ – associated with o-program $\mathcal{P}$, its answer set X, and a level $\ell \in \lambda(\mathcal{P})$:

$${\mathcal{P}^{\#}}_{\ell}^{X} = \sum_{D[w@\ell,t_1,\ldots,t_n] \in W \text{ and } X \models D} w$$

We define a concept of pw-dominance and pw-optimal answer sets as in Definition 7 by replacing $\mathcal{P}$ with $\mathcal{P}^{\#}$ in equations (30) and (31).

Example 9. *Let us now illustrate the difference between optimal and pw-optimal answer sets. Consider the CAS program denoted as (P_1, W_1) in Example 7. Its two optimal answer sets are listed in (34). Its unique pw-optimal answer set is presented in (35). On the other hand, recall that (35) is the unique optimal answer set of o-program (P_1, W_2). The same set forms the unique pw-optimal answer sets of (P_1, W_2).*

In the last example when we consider o-program (P_1, W_2), it is not by chance that its optimal and pw-optimal answer sets coincide. This is a consequence of a general fact captured by the following proposition.

Proposition 2. *For o-program (P, W), if the cardinality of a set*

$$\{(w@\ell, t_1, \ldots, t_n) \mid D[w@\ell, t_1, \ldots, t_n] \in W\} \tag{40}$$

is equal to the cardinality of W then the optimal and pw-optimal answer sets of (P, W) coincide.

Note how given sets W_1 and W_2 from Example 9, the sets corresponding to (40) follow, respectively:

$$\{1@1, \quad -2@1\}$$
$$\{1@1, l \quad -1@1, m \quad -1@1, n \quad -2@1, 0\}$$

It is easy to see that any gringo o-program satisfies the if-condition of Proposition 2.

Now that we established that transformations studied by Lierler (2024) are safe for gringo o-programs we present some details on these transformations. First the weak constraints are normalized so that they only contain positive weights. Second, the weights of the week constraints are rescaled based on the factor computed for each level while taking into account the weights of smaller levels. As a result newly composed weak constraints can be considered of the same level. The first rewriting is simple. It starts by dropping all weak constraints with 0 weight. Then, any weak constraint of the form (38) that has a negative weight $w < 0$ is replaced by the following weak constraint:

$$:\sim \ not\ a.\ [-1 \cdot w@\ell, t_1, \ldots, t_n]$$

and any weak constraint of the form (39) that has a negative weight w is replaced by:

$$:\sim \ a.\ [-1 \cdot w@\ell, t_1, \ldots, t_n].$$

Note how $-1 \cdot w$ results in a positive integer. The second rewriting that eliminates all the distinct levels in favor of single level 1 is more involved and we refer the reader to Section 5.2 by Lierler (2024) for the details on the procedure. Lierler (2023b; 2024) illustrate that the described rewritings preserve the pw-optimal models of the program. System EZSMTV3 implements these rewritings.

Upon the completion of the rewriting process, the EZSMTV3 deals with the collection of weight constraints of the following form

$$\begin{aligned}
&:\sim \ a_1.\ [w_1@1, t_{11}, \ldots, t_{1n_1}]\\
&\ldots\\
&:\sim \ a_k.\ [w_k@1, t_{k1}, \ldots, t_{kn_k}]\\
&:\sim \ not\ a_{k+1}.\ [w_{k+1}@1, t_{k+11}, \ldots, t_{k+1n_{k+1}}]\\
&\ldots\\
&:\sim \ not\ a_{k+m}.\ [w_{k+m}@1, t_{k+m1}, \ldots, t_{k+mn_{k+m}}]
\end{aligned}$$

so that all weak constraints are of the same level 1 and all weights $w_1, \ldots, w_k, w_{k+1}, \ldots, w_{k+m}$ are positive numbers. Given the above collection of the weak constraints, EZSMTV3 composes

the following expression in the language of the SMT-LIB:

$$\begin{array}{lll}(assert\ (=\ val\ (+ & (ite\ a_1\ w_1\ 0) \\ & \ldots \\ & (ite\ a_k\ w_k\ 0) \\ & (ite\ (not\ a_{k+1})\ w_{k+1}\ 0) \\ & \ldots \\ & (ite\ (not\ a_{k+m})\ w_{k+m}\ 0) \\ &)))\end{array}$$

where variable val is declared as an integer and expression *ite* is intuitively evaluated as an if-then-else statement. Note how the introduction of integer variable val translates into the use of SMT(LIA) or SMT(LIRA) logics when the SMT solver is invoked as depicted in Figure 4.

It is now due to describe the iterative procedure utilized to compute optimal answer sets. When the first answer set of a given program with weak constraints is computed, the answer is inspected to collect the value v of variable val. Then the new SMT-LIB statement is composed

$$(assert\ (<\ val\ v)) \tag{41}$$

and the SMT solver of choice is instructed to continue its search with this new statement. The process of inspecting for the value of variable val and appending the statement of the form (41) is repeated till we establish that the problem becomes unsatisfiable. System EZSMTV3 implements an anytime approach for computing optimal answer sets. In other words it displays each found answer set to a user with the guarantee that each following answer set dominates the one presented earlier.

CAS(LIA)	SMT(LIA)
CAS(LRA)	SMT(LIRA)
CAS(LIRA)	SMT(LIRA)
CAS(IDL)	SMT(LIA)

Fig. 4. Mapping of logics from CAS programs with weak constraints to respective SMT formulas.

5 Experimental Analysis

In this section we present the results on comparing the performance of system EZSMTV3 with the state-of-the-art solvers such as CLINGCON (Banbara et al., 2017), CLINGO[DL] (Janhunen et al., 2017), and CLINGO[LP] (Janhunen et al., 2017). The unique part of this comparison is that all encodings used were identical for all systems involved. This also explains the choice of systems to benchmark against. Cited papers above present experimental comparison of stated systems with other related technologies.

Three benchmarks, namely, Reverse Folding (RF), Incremental Scheduling (IS), and Weighted Sequence (WS), come from the Third Answer Set Programming Competition (Calimeri et al., 2011). We obtain the CLINGCON encoding of IS from work by Banbara et al. (2017).

We include a benchmark problem called Blending (BL) from work by Biavaschi (2017). We also add a modification of this benchmark called Mixed-BL, which contains variables over both integers and reals. Three more benchmarks, namely, RoutingMin (RMin), RoutingMax (RMax), and Traveling Salesperson (TS) are obtained from work by Liu et al. (2012). The original TS benchmark is an optimization problem, and we turn it into a decision problem. The original RoutingMax and RoutingMin problems are stated as CAS(LIA) programs. It was possible to find a formulation of these problems using the CAS(IDL) language.[4] In addition, we created another variant of the RoutingMax problem encoding by re-formulating one of its integer linear constraints in the original encoding as an aggregate ($\#sum$) expression. The Labyrinth (LB) benchmark is extended from the domain presented in the Fifth Answer Set Programming Competition (Calimeri et al., 2016). This extension allows us to add integer linear constraints into the problem encoding. Also, we present results on two benchmarks from work by Balduccini et al. (2017), namely, Car and Generator (GN). It is due to remark that all encodings from the literature were inspected and when possible augmented with additional domain restrictions for their constraint variables. This change was due to an observation that systems such as CLINGCON typically benefit from prespecified tighter domain on constraint variables.

System EZSMTV3 and all used benchmarks are hosted at `https://github.com/ylierler/ezsmtv3` .

All benchmarks are run on an Ubuntu 20.04.6 LTS (64-bit) system with an Intel® Core™ i7-7700 CPU @ 3.60GHz with 31.2 GiB RAM. The resource allocated for each benchmark instance is limited to one CPU core and 4 GiB of RAM. We set a timeout of 1800 seconds for each instance. Systems that we use to compare the performance of variants of EZSMTV3 (invoking SMT solver CVC4 v. 1.8; CVC5 v. 1.0.8; YICES v. 2.6.4; Z3 v. 4.8.7) are CLINGCON v. 5.2.1, CLINGO[LP] v. 0.2.0 and CLINGO[DL] v. 1.5.0. The GRINGO system v. 5.4.0 is used as a grounder for EZSMTV3.

Within all presented figures, all of the steps involved, including grounding and translation, are reported as part of the total solving time. Letter $\mathcal{E}$ stands for EZSMTV3; C-CON stand for CLINGCON; and C[DL] stands for CLINGO[DL]. The number in parenthesis after the name of the benchmark specifies how many instances were used in experiments. The time reported is the cumulative time of all the instances of the particular benchmark. The number of unsolved instances due to timeout or insufficient memory is put inside parentheses. The cumulative time mentioned in bold font is the least time taken for that particular benchmark problem using the corresponding solvers. The "-" symbol is used to show that the considered solver does not support this particular encoding. The benchmarks are divided into categories. The acronyms T and NT in the category names indicate that the programs are tight and non-tight, respectively. The second part of the category name indicates the logic used to formulate the CAS encoding of the considered problems. For non-tight (NT) programs, more solving options are possible, such as the use of different level ranking formulas using flags. Shen and Lierler (2018a) highlights the impact of the level ranking flags on the performance of EZSMT+. Similar impact is expected on EZSMTV3.

Before presenting individual results let us mention that EZSMTV3 can be seen as a more versatile system than its mentioned peers CLINGCON, CLINGO[DL], and CLINGO[LP]. Indeed, EZSMTV3 supports programs of four kinds, namely, CAS(LIA), CAS(IDA), CAS(LIA), and CAS(LIRA). Systems CLINGCON, CLINGO[DL], and CLINGO[LP] support CAS(LIA),

[4] Such a reformulation was suggested by Max Ostrowski.

CAS(IDA), CAS(LIA) programs, *respectively*. This fact explains why figures that follow contain benchmark data for different subsets of systems.

We start the discussion of the experimental analysis with the presentation of Figure 5. This figure is meant to illustrate the uniqueness of the EZSMTV3 system. Unlike its other peer systems geared to support a specific logic, EZSMTV3 implements various logics including LIRA. Thus, EZSMTV3 is capable of solving new kinds of domains. In the future, we envision extensions of the system to more logics provided by the SMT-solving portfolio. Another special feature of EZSMTV3 is that it can be seen as a multitude of systems. Indeed, each SMT solver invoked by the system provides us with different computational capabilities. Figure 5 illustrates that SMT solvers CVC4 and CVC5 are superior to Z3 for the case of the considered benchmark. The same figure does not present timings for EZSMTV3 invoking YICES. This is due to the fact that SMT solver YICES provides no support for LIRA logic.

Category	Benchmark	$\mathcal{E}$(Z3)	$\mathcal{E}$(YICES)	$\mathcal{E}$(CVC4)	$\mathcal{E}$(CVC5)
T-LIRA	Mixed-BL (30)	2957.33	-	**93.3**	140.66

Fig. 5. Summary of Experimental Data on CAS(LIRA) Encodings

Figure 6 presents the comparison between CLINGO[LP] and EZSMTV3 on CAS(LRA) encodings available. Figure 7 presents the comparison between CLINGCON and EZSMTV3 on CAS(LIA) encodings. It is due to note that system CLINGCON is a mature tool that has been under development for close to a decade, whereas CLINGO[LP] has been developed to illustrate the versatility of CLINGO series 5 that provides capabilities to bootstrap nontrivial extensions. These figures seem to indicate that relying on SMT solvers as a backend is a viable approach. We see how EZSMTV3 variants are competitive or superior with respect to CLINGO[LP]. At the same time it is obvious that when a technology is specifically geared towards solving CAS(LIA) programs such as CLINGCON then the efforts are paid off. On several of the benchmarks EZSMTV3 is comparable in its performance with CLINGCON, but often enough CLINGCON exhibits superior performance.

Category	Benchmark	CLINGO[LP]	$\mathcal{E}$(Z3)	$\mathcal{E}$(YICES)	$\mathcal{E}$(CVC4)	$\mathcal{E}$(CVC5)
T-LRA	BL (30)	11640.43(2)	62.39	**37.81**	43.52	54.74
	GN (8)	4.19	4.23	**4.12**	5.04	4.78

Fig. 6. Summary of Experimental Data on CAS(LRA) Encodings

Last but not least we present Figure 8 that summarizes the results for CAS(IDL) encodings. In the same table we add lines from the earlier figure that showcase the results for the same problems encoded as CAS(LIA) programs and solved by different technologies. This table points at the possibility to improve EZSMTV3 by exploring other translations of aggregate expressions ($\#sum$, in this case) than these currently implemented within EZSMTV3 (these routines the system inherits from answer set solver CMODELS Giunchiglia et al. (2006)). The experimental data points at the superiority of specialized propagators for processing aggregate expressions.

Category	Benchmark	C-CON	$\mathcal{E}$(Z3)	$\mathcal{E}$(YICES)	$\mathcal{E}$(CVC4)	$\mathcal{E}$(CVC5)
NT-LIA	RMin (100)	**1.17**	95.81	91.88	110.67	107.61
	RMax(#sum) (100)	**20.34**	35166.79	10802.86	16754	10707.14
	RMax(&sum) (100)	**10.62**	2087.3	592.44	180000 (100)	1743.65
	TS (30)	**45.5**	43763.86(22)	2117.4(1)	3129.99(1)	3035.31
	LB (22)	**4445.42(1)**	7648.74(1)	7309.14(1)	7397.58(1)	8361.58(2)
T-LIA	RF (50)	**105.63**	8417.93(1)	7610.42(1)	48819.14(20)	11852.58(1)
	IS (30)	**9060.7(5)**	9150.36(5)	9212.7(5)	10074.78(5)	9331.48(5)
	WS (30)	**17.22**	57.55	45.51	57.3	57.57

Fig. 7. Summary of Experimental Data on CAS(LIA) Encodings

Category	Benchmark 100	C-CON	C[DL]	$\mathcal{E}$(Z3)	$\mathcal{E}$(YICES)	$\mathcal{E}$(CVC4)	$\mathcal{E}$(CVC5)
NT-LIA	RMax(&sum)	**10.62**	-	2087.3	592.44	180000 (100)	1743.65
	RMax(#sum)	**20.34**	-	35166.79	10802.86	16754	10707.14
NT-IDL	RMax DL	-	**4.38**	22682.92	9687.78	10620.85	16277.06
NT-LIA	RMin	**1.17**	-	95.81	91.88	110.67	107.61
NT-IDL	RMin DL	-	**1.24**	110.75	100.94	122.15	115.02

Fig. 8. Summary of Experimental Data on Variants of Routing Problems: CAS(IDL) and CAS(LIA) encodings combined

6 Conclusions

This paper gives a detailed account of the EZSMTV3 system. A central focus of our work was the development of a robust and extensible CASP software framework which may significantly advance declarative programming and knowledge representation by offering both enhanced modeling expressiveness and access to cutting-edge solver performance. We aimed to emulate and expand upon the success of extensible platforms such as the CLINGO 5 series (Gebser et al., 2019) and the influential SAT solver MINISAT (Eén and Sörensson, 2003), both of which served as blueprints for designing modular, API-driven solvers. Also, the extensibility of MINISAT led to more than a decade of impactful developments in SAT and related technologies, including its use in solvers like CMODELS (Giunchiglia et al., 2006) and MINISAT(ID) (de Cat et al., 2014). Similarly, the flexibility of CLINGO 5 enabled the rapid prototyping of new CASP solvers such as CLINGO[LP] and CLINGO[DL] (Janhunen et al., 2017).

To validate our claim that the EZSMTV3 system is capable to support the rapid development of new CASP technologies we bootstrap four distinct CASP solvers, one that support linear integer constraints, another one that supports constraints over reals, then one that supports mixed real integer constraints and difference logic constraints. All of these were implemented using the same streamlined methodology that we carefully document here. One of the intentions of this description is to attract broader community involvement with the EZSMTV3 framework with the potential of seeing new solvers. We focused on developing a clear and accessible interface for expressing and reasoning with different kinds of constraints. This required the introduction of new language features, and streamlined integration with SMT solver technology via an incremental solving interface. The experiment section articulates the validity of the approach and properly places the system among its peers.

One more observation is due. The EZSMTV3 system can be seen as an alternative to SMT-LIB front-end to SMT solvers. As such, it provides a declarative programming language based on logic programming conventions to this automated reasoning technology. A similar idea was explored by the EZSCP system (Balduccini and Lierler, 2017) in the scope of constraint satisfaction processing. That system utilized CSP solvers to process CAS programs. An alternative view to that work was simplifying utilization of CSP solvers by providing them with the convenient interface though declarative programming languages based on logic programming.

Acknowledgments We are grateful to Nicholas Wilson for his contributions to the original prototype of EZSMTV3 as part of his Master Project (thesis equivalent) at the University of Nebraska Omaha. We are thankful to Zachary Hansen for his assistance to build a sand-basket for EZSMTV3 available on the University of Nebraska Omaha server and providing his valuable remarks on the complete draft of the paper.

Competing interests The author(s) declare none

References

Ilkka Niemelä. Logic programs with stable model semantics as a constraint programming paradigm. *Annals of Mathematics and Artificial Intelligence*, 25:241–273, 1999.

Victor Marek and Mirosław Truszczyński. Stable models and an alternative logic programming paradigm. In *The Logic Programming Paradigm: a 25-Year Perspective*, pages 375–398. Springer Verlag, 1999.

Gerhard Brewka, Thomas Eiter, and Mirosław Truszczyński. Answer set programming at a glance. *Communications of the ACM*, 54(12):92–103, 2011.

Francesca Rossi, Peter van Beek, and Toby Walsh. Constraint programming. In Frank van Harmelen, Vladimir Lifschitz, and Bruce Porter, editors, *Handbook of Knowledge Representation*, pages 181–212. Elsevier, 2008.

J. Jaffar and M.J. Maher. Constraint logic programming: A survey. *Journal of Logic Programming*, 19(20): 503–581, 1994.

Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. Solving SAT and SAT modulo theories: From an abstract Davis-Putnam-Logemann-Loveland procedure to DPLL(T). *Journal of the ACM*, 53(6):937–977, 2006.

C. Barrett, R. Sebastiani, S.A. Seshia, and C. Tinelli. Satisfiability modulo theories. In A. Biere, M. Heule, H. van Maaren, and T. Walsch, editors, *Handbook of Satisfiability*, pages 737–797. IOS Press, 2008.

Clark Barrett and Cesare Tinelli. Satisfiability modulo theories. In Edmund Clarke, Tom Henzinger, and Helmut Veith, editors, *Handbook of Model Checking*. Springer, 2014.

Islam Elkabani, Enrico Pontelli, and Tran Cao Son. Smodels with clp and its applications: A simple and effective approach to aggregates in ASP. In Bart Demoen and Vladimir Lifschitz, editors, *ICLP*, volume 3132 of *Lecture Notes in Computer Science*, pages 73–89. Springer-Verlag, 2004. ISBN 3-540-22671-0.

Veena S. Mellarkod, Michael Gelfond, and Yuanlin Zhang. Integrating answer set programming and constraint logic programming. *Annals of Mathematics and Artificial Intelligence*, 53(1-4):251–287, 2008.

Yuliya Lierler. Relating constraint answer set programming languages and algorithms. *Artificial Intelligence*, 207C:1–22, 2014.

Martin Gebser, Max Ostrowski, and Torsten Schaub. Constraint answer set solving. In *Proceedings of 25th International Conference on Logic Programming*, pages 235–249. Springer, 2009a.

Marcello Balduccini and Yuliya Lierler. Constraint answer set solver ezcsp and why integration schemas matter. *Theory and Practice of Logic Programming*, 17(4):462–515, 2017. doi: 10.1017/S1471068417000102.

Johan Wittocx, Maarten Mariën, and Marc Denecker. The IDP system: a model expansion system for an extension of classical logic. In *Proceedings of Workshop on Logic and Search, Computation of Structures from Declarative Descriptions (LaSh)*, pages 153–165. electronic, 2008. available at https://lirias.kuleuven.be/bitstream/123456789/229814/1/lash08.pdf.

Christian Drescher and Toby Walsh. A translational approach to constraint answer set solving. *Theory and Practice of Logic programming (TPLP)*, 10(4-6):465–480, 2010.

Tomi Janhunen, Guohua Liu, and Ilkka Niemela. Tight integration of non-ground answer set programming and satisfiability modulo theories. In *Proceedings of the 1st Workshop on Grounding and Transformations for Theories with Variables*, 2011.

Guohua Liu, Tomi Janhunen, and Ilkka Niemela. Answer set programming via mixed integer programming. In *Knowledge Representation and Reasoning Conference*, 2012. URL https://www.aaai.org/ocs/index.php/KR/KR12/paper/view/4516.

Michael Bartholomew and Joohyung Lee. System aspmt2smt: Computing aspmt theories by smt solvers. In *European Conference on Logics in Artificial Intelligence, JELIA*, pages 529–542. Springer, 2014. ISBN 978-3-319-11558-0. doi: 10.1007/978-3-319-11558-0_37. URL http://dx.doi.org/10.1007/978-3-319-11558-0_37.

Tomi Janhunen, Roland Kaminski, Max Ostrowski, Torsten Schaub, Sebastian Schellhorn, and Philipp Wanko. Clingo goes linear constraints over reals and integers. *CoRR*, abs/1707.04053, 2017.

Benjamin Susman and Yuliya Lierler. SMT-Based Constraint Answer Set Solver EZSMT (System Description). In *Technical Communications of the 32nd International Conference on Logic Programming (ICLP 2016)*, volume 52, pages 1:1–1:15, 2016a.

Da Shen and Yuliya Lierler. Smt-based constraint answer set solver ezsmt+ for non-tight programs. In *Sixteenth International Conference on Principles of Knowledge Representation and Reasoning*, 2018a.

Martin Gebser, Roland Kaminski, Benjamin Kaufmann, and Torsten Schaub. Multi-shot asp solving with clingo. *Theory and Practice of Logic Programming*, 19(1):27–82, 2019. doi: 10.1017/S1471068418000054.

Roland Kaminski, Javier Romero, Torsten Schaub, and Philipp Wanko. How to build your own asp-based system?! *Theory and Practice of Logic Programming*, 23(1):299–361, 2023.

Martin Gebser, Ronald Kaminski, Max Ostrowski, Torsten Schaub, and Sven Thiele. On the input language of ASP grounder *Gringo*. In E. Erdem, F. Lin, and T. Schaub, editors, *Proceedings of the Tenth International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'09)*, volume 5753 of *Lecture Notes in Artificial Intelligence*, pages 502–508. Springer-Verlag, 2009b.

Martin Gebser, Roland Kaminski, Benjamin Kaufmann, Javier Romero, and Torsten Schaub. Progress in CLASP series 3. In *Proceedings of the Thirteenth International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'15)*, 2015.

Roland Kaminski. *Complex reasoning with answer set programming*. doctoralthesis, University of Potsdam, 2023.

Clark Barrett, Christopher L Conway, Morgan Deters, Liana Hadarean, Dejan Jovanović, Tim King, Andrew Reynolds, and Cesare Tinelli. cvc4. In *Computer Aided Verification: 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings 23*, pages 171–177. Springer, 2011.

Clark Barrett, Haniel Barbosa, Martin Brain, Gereon Kremer, Makai Mann, Abdalrhman Mohamed, Mudathir Mohamed, Aina Niemetz, Andres Nötzli, Alex Ozdemir, et al. Cvc5 at the smt competition 2021.

Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

Bruno Dutertre and Leonardo De Moura. The yices smt solver. *Tool paper at http://yices. csl. sri. com/tool-paper. pdf*, 2(2):1–2, 2006.

Yuliya Lierler. Constraint answer set programming: Integrational and translational (or smt-based) approaches. *Theory and Practice of Logic Programming*, 23(1):195–225, 2023a. doi: 10.1017/S1471068421000478.

Vladimir Lifschitz, Lappoon R. Tang, and Hudson Turner. Nested expressions in logic programs. *Annals of Mathematics and Artificial Intelligence*, 25:369–389, 1999.

Yuliya Lierler and Benjamin Susman. On relation between constraint answer set programming and satisfiability modulo theories. *Theory and Practice of Logic Programming*, 17(4):559–590, 2017.

Clark Barrett, Aaron Stump, and Cesare Tinelli. The Satisfiability Modulo Theories Library (SMT-LIB). `www.SMT-LIB.org`, 2010.

Keith Clark. Negation as failure. In *Logic and Data Bases*, pages 293–322. Plenum Press, 1978.

Ilkka Niemela. Stable models and difference logic. *Annals of Mathematics and Artificial Intelligence*, 53: 313–329, 2008.

Mutsunori Banbara, Benjamin Kaufmann, Max Ostrowski, and Torsten Schaub. Clingcon: The next generation. *Theory and Practice of Logic Programming (TPLP)*, 17(4):408–461, 2017.

Martin Gebser, Roland Kaminski, Benjamin Kaufmann, Max Ostrowski, Torsten Schaub, and Philipp Wanko. Theory solving made easy with clingo 5. In *Technical Communications of the 32nd International Conference on Logic Programming (ICLP 2016)*. Schloss-Dagstuhl-Leibniz Zentrum für Informatik, 2016.

E. L. Lawler, Jan Karel Lenstra, A. H. G. Rinnooy Kan, and D. B. Shmoys, editors. *The Traveling Salesman Problem: A Guided Tour of Combinatorial Optimization*. Wiley, 1985.

G. Gutin and A.P. Punnen, editors. *The Traveling Salesman Problem and Its Variations*. Springer-Verlag, 2007.

Francesco Calimeri, Wolfgang Faber, Martin Gebser, Giovambattista Ianni, Roland Kaminski, Thomas Krennwallner, Nicola Leone, Marco Maratea, Francesco Ricca, and Torsten Schaub. ASP-Core-2 input language format. *Theory and Practice of Logic Programming*, 20(2):294–309, 2020a. doi: 10.1017/S1471068419000450.

Wolfgang Faber, Nicola Leone, and Simona Perri. In Esra Erdem, Joohyung Lee, Yuliya Lierler, and David Pearce, editors, *Correct Reasoning: Essays on Logic-Based AI in Honor of Vladimir Lifschitz*, chapter The Intelligent Grounder of DLV, pages 247–264. Springer-Verlag, 2012.

Da Shen and Yuliya Lierler. Smt-based answer set solver cmodels-diff (system description). In *Technical Communications of the 34th International Conference on Logic Programming (ICLP 2018)*, 2018b.

Benjamin Susman and Yuliya Lierler. Smt-based constraint answer set solver ezsmt (system description). In *Technical Communications of the 32nd International Conference on Logic Programming (ICLP 2016)*. Schloss-Dagstuhl-Leibniz Zentrum für Informatik, 2016b.

Francesco Calimeri, Wolfgang Faber, Martin Gebser, Giovambattista Ianni, Roland Kaminski, Thomas Krennwallner, Nicola Leone, Marco Maratea, Francesco Ricca, and Torsten Schaub. ASP-Core-2 input language format. *Theory And Practice Of Logic Programming*, 20(2):294–309, 2020b. doi: 10.1017/s1471068419000450.

Yuliya Lierler. Unifying framework for optimizations in non-boolean formalisms. *Theory Pract. Log. Program.*, 23(6):1248–1280, 2023b. doi: 10.1017/S1471068422000400. URL https://doi.org/10.1017/s1471068422000400.

Yuliya Lierler. An abstract view on optimizations in propositional frameworks. *Ann. Math. Artif. Intell.*, 92(2):355–391, 2024. doi: 10.1007/S10472-023-09914-6. URL https://doi.org/10.1007/s10472-023-09914-6.

Zhaohui Fu and Sharad Malik. On solving the partial max-sat problem. In Armin Biere and Carla P. Gomes, editors, *Theory and Applications of Satisfiability Testing - SAT 2006*, pages 252–265, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg. ISBN 978-3-540-37207-3.

Francesco Calimeri, Giovambattista Ianni, Francesco Ricca, and et al. The third answer set programming competition: Preliminary report of the system competition track. In *Proceedings of the International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR)*, pages 388–403, Berlin, Heidelberg, 2011. Springer-Verlag.

Sara Biavaschi. Automated reasoning methods in hybrid systems, 2017. Annual Report of "Scuola Superiore dell'Università di Udine".

Francesco Calimeri, Martin Gebser, Marco Maratea, and Francesco Ricca. Design and results of the fifth answer set programming competition. *Artif. Intell.*, 231(C):151–181, 2016.

Marcello Balduccini, Daniele Magazzeni, Marco Maratea, and Emily C. Leblanc. Casp solutions for planning in hybrid domains. *Theory and Practice of Logic Programming*, 17(4):591–633, 2017. doi: 10.1017/S1471068417000187.

Enrico Giunchiglia, Yuliya Lierler, and Marco Maratea. Answer set programming based on propositional satisfiability. *Journal of Automated Reasoning*, 36:345–377, 2006.

N. Eén and N. Sörensson. An extensible SAT solver. In *Proceedings of SAT-2003*, pages 502–518, 2003.

Broes de Cat, Bart Bogaerts, Maurice Bruynooghe, and Marc Denecker. Predicate logic as a modelling language: The IDP system. *CoRR*, abs/1401.6312, 2014. URL http://arxiv.org/abs/1401.6312.