Constraint Answer Set Programming: Integrational and Translational (or SMT-based) Approaches

YULIYA LIERLER

University of Nebraska Omaha
(*e-mail:* ylierler@unomaha.edu)

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

Abstract

Constraint answer set programming or CASP, for short, is a hybrid approach in automated reasoning putting together the advances of distinct research areas such as answer set programming, constraint processing, and satisfiability modulo theories. Constraint answer set programming demonstrates promising results, including the development of a multitude of solvers: ACSOLVER, CLINGCON, EZCSP, IDP, INCA, DINGO, MINGO, ASPMT2SMT, CLINGO[L,DL], and EZSMT. It opens new horizons for declarative programming applications such as solving complex train scheduling problems. Systems designed to find solutions to constraint answer set programs can be grouped according to their construction into, what we call, *integrational or translational* approaches. The focus of this paper is an overview of the key ingredients of the design of constraint answer set solvers drawing distinctions and parallels between integrational and translational approaches. The paper also provides a glimpse at the kind of programs its users develop by utilizing a CASP encoding of Travelling Salesman problem for illustration. In addition, we place the CASP technology on the map among its automated reasoning peers as well as discuss future possibilities for the development of CASP. Under consideration in Theory and Practice of Logic Programming (TPLP).

1 Introduction

Knowledge representation and automated reasoning are areas of Artificial Intelligence that pay especial attention to understanding and automating various aspects of reasoning. Such traditionally separate fields of AI as answer set programming (ASP) (Niemelä 1999; Marek and Truszczyński 1999; Brewka et al. 2011), propositional satisfiability (SAT) (Gomes et al. 2008), constraint (logic) programming (CSP/CLP) (Rossi et al. 2008; Jaffar and Maher 1994) are representatives of model search in automated reasoning. These methods have been successfully used in a myriad of scientific and industrial applications including space shuttle control (Balduccini et al. 2001; Balduccini and Gelfond 2005), scheduling (Ricca et al. 2012), planning (Kautz and Selman 1992; Rintanen 2012), hardware verification (Biere et al. 2003; Prasad et al. 2005), adaptive Linux package configuration (Gebser et al. 2011), systems biology (Gebser et al. 2010), bioinformatics (Palù et al. 2004; Palu et al. 2010), software engineering (Cohen et al. 2008; Garvin et al. 2011; Brain et al. 2012).

Often the combination of algorithmic techniques stemming from distinct subfields of automated reasoning is necessary. For instance, problems in software verification require reasoning combining propositional logic with formalizations that include, among others, theories of strings

and arrays. These observations led to studies targeting the development of hybrid (multi-logic) computational methods that put together distinct solving approaches suitable for different logics. This has led to the development of *hybrid* approaches that combine algorithms and systems from different AI subfields. Constraint logic programming (Jaffar and Maher 1994), satisfiability modulo theories (SMT) (Nieuwenhuis et al. 2006; Barrett et al. 2008; Barrett and Tinelli 2014), HEX-programs (Eiter et al. 2005), VI-programs (Calimeri et al. 2007), constraint answer set programming (CASP) (Elkabani et al. 2004; Mellarkod et al. 2008; Lierler 2014) are all examples of this trend. Constraint answer set programming is the focus of this paper.

Constraint answer set programming allows one to combine the best of two different automated reasoning worlds: (1) the non-monotonic modeling capabilities and SAT-like solving technology of ASP; and (2) constraint processing techniques for effective reasoning over non-Boolean constructs. CASP demonstrates promising results. For instance, research by Balduccini on the design of CASP language EZCSP and on the corresponding solver yields an elegant, declarative solution to a complex industrial scheduling problem (Balduccini 2011). Similarly, system CLINGO[DL] provides the basis for solving complex train scheduling problems (Abels et al. 2019). It is also due to note the development of many CASP solvers in the past decade: ACSOLVER (Mellarkod et al. 2008), CLINGCON (Gebser et al. 2009), 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), ASPMT2SMT (Bartholomew and Lee 2014), CLINGO[L,DL] (Janhunen et al. 2017), and EZSMT (Susman and Lierler 2016; Shen and Lierler 2018b). It is fair to say that CASP formalism together with the multitude of supporting tools opens new horizons for declarative programming applications.

There are two main approaches in developing CASP systems/solvers, that is, tools for processing programs in constraint answer set programming and enumerating their solutions. The first one goes after systems that, while processing CAS programs, rely on combining algorithms/solvers employed in ASP and constraint processing (Mellarkod et al. 2008; Gebser et al. 2009; Balduccini and Lierler 2017). We call this approach *integrational*. The second one transforms a CAS program into an SMT formula, whose models are in prespecified relation with answer sets of the original program (Janhunen et al. 2011; Lee and Meng 2013; Susman and Lierler 2016; Lierler and Susman 2017; Shen and Lierler 2018b). As a result a problem of finding solutions to CASP is transformed into a problem of finding models of SMT formula. We call this approach *translational*. The translational approach also includes two systems that translate CAS programs into other formalisms than SMT, namely, mixed integer programming, system MINGO (Liu et al. 2012), and answer set programming, system ASPARTAME (Banbara et al. 2015).

The focus of this paper is an overview of the key ingredients of the integrational and translational approaches towards construction of CASP systems. The paper starts with the presentation of constraint answer set programming in use to showcase the paradigm. In particular, we present a CASP formulation of Traveling Salesman Problem benchmark alongside its ASP formulation. We then proceed towards defining formal concepts of constraint answer set programming. The main part of the paper is devoted to describing details behind the integrational and translational approaches utilizing examples of two representatives of these methods — systems EZCSP and EZSMT, respectively. The paper also presents some experimental data together with an overarching comparison between the existing CASP systems in uniform terminological terms. We conclude with the discussion on future directions, opportunities, and challenges of the CASP subfield of automated reasoning. Before proceeding to the main topic of this paper we spend

CASP	=	ASP	+	Constraints
SMT	=	SAT	+	Constraints
CLP	=	LP	+	Constraints

Fig. 1: Paradigms' Content

	Programming/modeling language	Automated reasoning (solver/system/compiler)
ASP/CASP	✓	✓
SAT/SMT		✓
LP/CLP	✓	✓

Fig. 2: Ingredients of Paradigms

some time on placing CASP on the map of the automated reasoning subfield of artificial intelligence.

CASP and its Relatives

The question that comes to mind is what are the unique features of CASP in comparison to related formalisms, in particular, satisfiability modulo theories, constraint logic programming, and answer set programming. Before drawing parallels between the fields, let us recall principal ingredients of declarative programming that CASP is a good representative of. In declarative approach to programming no reference to an algorithm on how exactly to compute a solution is given. Rather a program provides a description/specification of what constitutes a solution. Automated reasoning techniques are then used to find a solution to provided specification. Thus, declarative programming paradigm provides a programmer with two ingredients:

1. Programming/modeling language to express requirements on a solution, and
2. Automated reasoning method to find a solution.

CASP vs SMT. Intuitive visualizations in Figures 1 and 2 are of use[1] when we compare CASP and SMT. Figure 1 makes it clear that the key lies in relation between ASP and SAT. Lierler provides a detailed comparison of ASP and SAT (2017). Here we reiterate the main thesis of that work:

> Answer set programming provides a declarative constraint programming language, while SAT does not.

The same claim is captured in Figure 2. Both ASP/CASP and SAT/SMT pairs provide a solid platform for solving difficult combinatorial search problems. Automated reasoning tools behind these paradigms, called solvers, share a lot in common. Yet, only ASP/CASP pair supplies its users with programming/modeling language – language of logic programs – meant to express requirements on a solution using logic programs. The DIMACS and SMT-LIB standard formats of SAT and SMT solvers, respectively, provide a uniform front end to these systems, but they are not meant for direct encoding of problems' specifications.

CASP vs CLP As Figure 1 suggests the key distinction between CASP and CLP lies in the difference of underlying paradigms of ASP and logic programming (LP). Marek and Truszczyński draw a parallel between these two declarative programming paradigms (1999). To summarize, in

[1] In Figure 1 we understand word *Constraints* as in constraint satisfaction.

original logic programming (Kowalski 1988), called Prolog, a *single* intended model is assigned to a logic program. The SLD-resolution (Kowalski 1974) is at the heart of control mechanism behind Prolog implementations. Together with a logic program, a Prolog system expects a query. This query is then evaluated by means of SLD-resolution and a given program against an intended model. In answer set programming, a *family* of intended models (possibly an empty one) is assigned to a logic program. Each member of this family forms a solution to a problem encoded by the program. Rules of a logic program formulate restrictions/constraints on solutions. A program is typically evaluated by means of a grounder-solver pair. A grounder is responsible for eliminating variables occurring in a logic program in favor of suitable object constants resulting in a propositional program. A solver – a system in spirit of SAT solvers (Lierler 2017) – is responsible for computing answer sets (solutions) of a program. Thus, even though LP and ASP share the basic language of logic programs, their programming methodologies and underlying solving/control technologies are different.

CASP vs ASP. The origin of CASP methods lies in attempts to tackle a challenge posed by the *grounding bottleneck* of ASP. Sometimes when a considered problem contains variables ranging over a large integer domain grounding required in pure ASP may result in a propositional program of a prohibitive size. CASP provides means to handle these variables within *Constraints* of the paradigm (see Figure 1). There is also an additional benefit of the paradigm. For example, some CASP dialects provide means to express constraints over real numbers whereas traditional ASP lacks this capacity. Thus, CASP offers novel modeling capabilities in comparison to these of pure ASP.

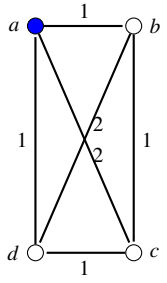
2 Constraint Answer Set Programming via Traveling Salesman Problem Formalization

Before we dive into formal definitions, we present the formalization of a variant of the Traveling Salesman Problem (Lawler et al. 1985; Gutin and Punnen 2007) in both answer set programming and constraint answer set programming (in the sequel, when we refer to this conjunction we write *(constraint) answer set programming* or (C)ASP). (Constraint) answer set programming provides a general purpose modeling language that supports elaboration tolerant solutions for search problems. We use the same notion of the search problem as Brewka et al. (2011). Quoting from their work, a *search problem P* consists of a set of instances with each *instance I* assigned a finite set $S_P(I)$ of solutions. In (constraint) answer set programming to solve a search problem P, we construct a program Π_P that captures problem specifications so that when extended with facts D_I representing an instance I of the problem, the answer sets of $\Pi_P \cup D_I$ are in one to one correspondence with members in $S_P(I)$. In other words, answer sets describe all solutions of problem P for the instance I. Thus, solving a search problem is reduced to finding a uniform encoding of its specifications by means of a logic program.

Consider the following combinatorial search problem: given an undirected weighted graph G (where weights are non-negative integers), find a Hamiltonian cycle in G with the sum of the weights of its edges at or below a given value. We can interpret this problem as a *variant* of the *Traveling Salesman Problem* (TS):

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 under *certain* maximum cost *of total time.*

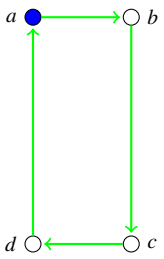
An instance with max cost 4



Encoded in ASP with facts:

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

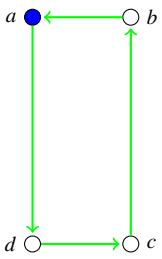
Solution 1



Only instances of route/2 in answer set 1:

```
route(a,b)
route(b,c)
route(c,d)
route(d,a)
```

Solution 2



answer set 2:

```
route(a,d)
route(d,c)
route(c,b)
route(b,a)
```

Fig. 3: Sample TS Instance and Solutions

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

Figure 3 shows an instance of the TS problem (a weighted graph) as well as its representation as a set of facts (logic rules without bodies). On the right hand side of the figure, we find two solutions to this problem.

Figure 4 presents an answer set programming formalization of the traveling salesman problem using the syntax of the standard ASP-Core-2 Language (Calimeri et al. 2019). Given a program composed of the rules in Figure 4 and the facts encoding the sample instance in Figure 3, an answer set solver such as CLINGO, for example, will produce the following output

```
Answer: 1
route(a,d) route(c,b) route(d,c) route(b,a)
Answer: 2
route(a,b) route(b,c) route(c,d) route(d,a)
```

These answers correspond to the solutions of our sample instance.

Figure 5 presents a typical architecture of an answer set programming system. For example, aforementioned tool CLINGO has this architecture. A grounder is a system that replaces non-ground rules (rules with variables) by their ground counterparts (rules without variables/propositional rules) (Gebser et al. 2007; Calimeri et al. 2008). A solver is then invoked to find answer sets of a ground program. Procedures behind modern answer set solvers are close relatives of those behind SAT solvers (Lierler 2017). The process of grounding in ASP is well understood and highly optimized. For example, consider rule

```
:-W<#sum{C,X,Y:route(X,Y),cost(X,Y,C)}, maxCost(W).        (1)
```

from the ASP formalization of the TS problem and the discussed instance. A grounder of system CLINGO replaces rule (1) with the following rule:

```
:-4<#sum{1,a,b:route(a,b);1,b,c:route(b,c);1,c,d:route(c,d);
     1,d,a:route(d,a);2,b,d:route(b,d);2,a,c:route(a,c);
     1,b,a:route(b,a);1,c,b:route(c,b);1,d,c:route(d,c);
```

Encoding	Meaning
`road(Y,X):-road(X,Y).`	A road from X to Y is also a rode from Y to X.
`cost(Y,X,C):-cost(X,Y,C).`	A cost C for a road from X to Y is also a cost for the same road from Y to X.
`1{route(X,Y): road(X,Y)}1:-city(X).`	For each city, pick one route *leaving from* the city.
`1{route(X,Y): road(X,Y)}1:-city(Y).`	For each city, pick one route *going to* the city.
`reached(X):-initial(X).`	The initial city is reached.
`reached(Y):-reached(X), route(X,Y).`	If city X is reached and the route from X to Y is picked, then city Y is also reached.
`:-city(X), not reached(X).`	A city that is not reached leads to a contradiction.
`:-W<#sum{C,X,Y:route(X,Y),cost(X,Y,C)}, maxCost(W).`	The total time cost of a selected route greater than maximal cost leads to a contradiction.
`#show route/2.`	A directive to only print route predicate as output.

Fig. 4: TS: ASP encoding in the standard ASP-Core-2 Language

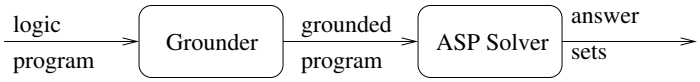


Fig. 5: Answer Set Programming System Architecture

CASP Encoding	Meaning
`cspvar(c(X,Y),0,C):-cost(X,Y,C).`	Declaration of constraint variables.
`required(c(X,Y)==0):-cost(X,Y,C), not route(X,Y).`	Time spent on a road is 0 if road is not in route.
`required(c(X,Y)==C):-cost(X,Y,C), route(X,Y).`	Time spent on a road is its cost if road is in route.
`required(sum([c/2],<=,W)):- maxCost(W).`	Total time cost must be less or equal to max cost.

Fig. 6: TS: Part of the CASP encoding in the EZ language of EZCSP

`1,a,d:route(a,d);2,d,b:route(d,b);2,c,a:route(c,a)}.`

In some cases, the time taken by grounding dominates the time taken by solving. Addressing this difficulty is one of the challenges of ASP.

We now present the formulation of the TS problem using constraint answer set programming. In particular, we obtain a CASP encoding in the language of EZCSP by taking an ASP program given in Figure 4 and replacing its rule (1) with lines presented in Figure 6. In this encoding, we introduce *constraint* variables $c(\cdot,\cdot)$ associated with each road so that when a road becomes a part of a route selected by a salesman its value is assigned to the cost of the road, while otherwise

it is 0. We then pose a constraint on these variables, which ensures that the total cost of a selected route is less than the maximal cost.

The TS problem showcases some key features that constraint answer set programming brings to the table in comparison to its parent – answer set programming:

- Consider a simple change in the statement of the TS problem, namely, *time is expressed as a **real value***. In fact, as mentioned earlier, the classical formulation of the TS problem considers weights that are real numbers. The traditional ASP framework may no longer be used to solve this problem. There is no support for real number arithmetic within grounders. Yet, CASP tools, such as, for example, EZCSP or EZSMT, can be used to find solutions to this new problem using the same program as presented here.
- ASP solvers process rules with so called sum-aggregates such as (1) by implementing specialized procedures (Niemelä and Simons 2000; Gebser et al. 2009; Lierler 2010). By replacing (1) with its CASP counterpart we allow utilization of search techniques stemming from either
  - — CSP community if we use such CASP tool as, for example, EZCSP, or
  - — SMT community if we use such CASP tool as, for example, EZSMT.

  These techniques will at times provide complementary performance. In other words, CASP allows us to utilize modeling language of ASP together with solving capabilities of SMT and CSP.

In addition,

- The grounding process of ASP may result in production of propositional programs that are of prohibitive size. This is especially the case when complex constraints over large numeric values are in place. CASP often allows us to bypass the grounding bottleneck via the reformulation of these numeric constraints using constraint atoms. Lierler et al. (2012) presents a case study on Weighted-Sequence problem (a domain inspired by a query optimization problem in relational databases), where the CASP solution is superior to its ASP counterpart as it alleviates grounding issues exhibited by an ASP solution.

Just as a typical answer set solver, a common CASP system starts its computation by performing grounding on a given program. For example, such CASP systems as EZCSP and CLINGCON utilize grounder GRINGO to produce a program composed of ground, so called, regular and irregular atoms. For instance, consider a rule

```
required(c(X,Y)==C):- cost(X,Y,C), route(X,Y).        (2)
```

from the CASP TS encoding. We can view symbols X, Y, and C as *schematic variables* that are placeholders for instances of passing constants. In the context of the CAS program composed of the sample TS instance in Figure 3 and the CASP TS encoding, rule (2) will be grounded by the EZSMT system into the rules of the kind:

```
required(c(a,b)==1):- cost(a,b,1), route(a,b).
required(c(b,a)==1):- cost(b,a,1), route(b,a).
                      ...
required(c(b,d)==2):- cost(b,d,2), route(b,d).
required(c(d,b)==2):- cost(d,b,2), route(d,b).
```

As a result, a program that a solver component of a typical CASP system processes consists of

i. "regular" ground atoms such as $cost(a,b,1)$ and $route(a,b)$, and
ii. "irregular" or constraint ground atoms such as $c(a,b) == 1$ and $c(b,d) == 2$, and
iii. ground constraint variables such as $c(a,b)$ and $c(b,d)$.

Grounding process of CASP systems mirrors that of ASP systems. Thus, we direct a reader to papers by Gebser et. al (2007) and Calimeri et al. (2008) for the details on grounding procedures. Here we focus on the unique features of CASP systems that pertain to their solving techniques. For this reason formal definitions that we present are in terms of ground/propositional CAS programs. We refer a reader, interested in the definition of syntax and semantics for non-ground CAS programs, to a paper by Bartholomew and Lee (2013).

3 Preliminaries

We now proceed towards formal preliminaries required to state the key definitions of the CASP paradigm.

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

$$a \leftarrow b_1,\ldots,b_\ell,\ not\ b_{\ell+1},\ldots,\ not\ b_m,\ \ not\ \ not\ b_{m+1},\ldots,\ not\ \ not\ b_n, \tag{3}$$

where a is an atom over σ or $\bot$, and each b_i, $1 \leq i \leq n$, is an atom in σ. We sometimes use the abbreviated form for rule (3)

$$a \leftarrow B, \tag{4}$$

where B stands for $b_1,\ldots,b_\ell,\ not\ b_{\ell+1},\ldots,\ not\ b_m,\ not\ \ not\ b_{m+1},\ldots,\ not\ \ not\ b_n$ and is also called a *body*. Syntactically, we identify rule (3) with the propositional formula

$$b_1 \wedge \ldots \wedge b_\ell \wedge \neg b_{\ell+1} \wedge \ldots \wedge \neg b_m \wedge \neg\neg b_{m+1} \wedge \ldots \wedge \neg\neg b_n \rightarrow a \tag{5}$$

and B with the propositional formula

$$b_1 \wedge \ldots \wedge b_\ell \wedge \neg b_{\ell+1} \wedge \ldots \wedge \neg b_m \wedge \neg\neg b_{m+1} \wedge \ldots \wedge \neg\neg b_n. \tag{6}$$

Note (i) the order of terms in (6) is immaterial, (ii) *not* is replaced with classical negation ($\neg$), and (iii) comma is replaced with conjunction ($\wedge$). Expression

$$b_1 \wedge \ldots \wedge b_\ell$$

in formula (6) is referred to as the *positive* part of the body and the remainder of (6) as the *negative* part of the body. Sometimes, we interpret semantically rule (3) and its body as propositional formulas, in these cases it is obvious that double negation $\neg\neg$ in (5) and (6) can be dropped.

The expression a is the *head* of the rule. When a is $\bot$, we often omit it and say that the head is empty. We call such rules *denials*. We write $hd(\Pi)$ for the set of nonempty heads of rules in Π. We call a rule whose body is empty a *fact*. In such cases, we drop the arrow. We sometimes may identify a set X of atoms with the set of facts $\{a. \mid a \in X\}$. 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 $\sigma \setminus X$. We say a set X of atoms *satisfies* rule (3), if X satisfies the

propositional formula (5). 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 denote satisfaction relation with symbol $\models$.

The *reduct* Π^X of a program Π relative to a set X of atoms is obtained by first removing all rules (3) such that X does not satisfy 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 $a \leftarrow b_1, \ldots, b_\ell$ (note that a can be $\bot$).

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).

Ferraris and Lifschitz (2005) showed that a choice rule $\{a\} \leftarrow B$ can be seen as an abbreviation for a rule $a \leftarrow$ *not not* a, B (choice rules were introduced by Niemelä and Simons (2000) and are commonly used in answer set programming languages). We adopt this abbreviation in the rest of the paper.

We now state the definition of an input answer set (Lierler and Truszczyński 2011) as it is instrumental in defining semantics for constraint answer set programs.

Definition 2 (*Input answer set*)
For a logic program Π over vocabulary σ and (*input/extensional*) 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 $\Pi \cup (X \cap \iota)$.

Example 1
Consider a logic program inspired by a running example by Balduccini and Lierler (2017):

$$\begin{array}{l} \mathit{lightOn} \leftarrow \mathit{switch},\ \mathit{not\ am}. \\ \leftarrow \mathit{not\ lightOn}. \end{array} \tag{7}$$

Take set $\{\mathit{switch}, \mathit{am}\}$ to form an input vocabulary. Intuitively, a program is evaluated relative to truth values of these input atoms that are provided at the time of the evaluation. Each rule in the program can be understood as follows:

- The light is on (*lightOn*) during the night (*not am*) when the action *switch* has occurred.
- The light must be on.

Consider set $\{\mathit{switch},\ \mathit{lightOn}\}$ of atoms. This set associates values *true* and *false* with input atoms *switch* and *am*, respectively. This set is an input answer set of program (7). Indeed, let Π be program (7) extended with the fact *switch*. Reduct $\Pi^{\{\mathit{switch},\ \mathit{lightOn}\}}$ follows:

$$\begin{array}{l} \mathit{switch}. \\ \mathit{lightOn} \leftarrow \mathit{switch}. \end{array}$$

Set $\{\mathit{switch},\ \mathit{lightOn}\}$ is an answer set of this reduct. This set is the only input answer set of sample program (7). This input answer set suggests that the only situation that satisfies the specifications of the problem is such that (i) it is currently night, (ii) the light has been switched on, and (iii) the light is on.

Input Completion. Clark (1978) introduced the notion of program's completion. The process of completion turns a logic program into a classical logic formula. When a logic program satisfies certain syntactic conditions, models of a completion formula coincide with answer sets of a logic program. In all cases, models of a completion formula include all answer sets of a logic program. Program's completion is a fundamental concept that plays an important role in the design of answer set solvers — see, for instance, the paper by Lierler and Truszczynski (2011). It is also a major building block of the translational approach to CASP solvers. We now review this concept together with the related notion of an input completion (Lierler and Susman 2017).

Let Π be a program over vocabulary σ. By $Bodies(\Pi, a)$ we denote the set of the bodies of all rules of Π with head a. The *completion of program* Π, denoted by $Comp(\Pi)$, is the set of

- classical formulas that consist of the rules (3) in Π (recall that we identify rule (3) with implication (5); when a rule (3) is a fact a, then we identify this rule with the clause consisting of a single atom a) and
- the implications

$$a \rightarrow \bigvee_{a \leftarrow B \in \Pi} B \tag{8}$$

for all atoms a in σ. When the set $Bodies(\Pi, a)$ is empty, the implication (8) has the form $a \rightarrow \bot$.

We now define an input completion that is relative to an (input) vocabulary.

Definition 3 (*Input completion*)
For a program Π over vocabulary σ, the *input-completion* of Π relative to vocabulary $\iota \subseteq \sigma$ so that $hd(\Pi) \cap \iota = \emptyset$, denoted by $IComp(\Pi, \iota)$, is defined as the set of formulas in propositional logic that consists of the rules (5) in Π and the implications (8) for all atoms a occurring in $\sigma \setminus \iota$.

Level Ranking. Niemelä (2008) characterized answer sets of "normal" logic programs in terms of program's completion and "level ranking". Normal programs consist of rules of the form (3), where $n = m$ and a is an atom. Lierler and Susman (2017) generalized a concept of a level ranking to programs introduced here. These results are fundamental in realizations of many translational approaches to (constraint) answer set programming. For instance, Niemelä developed a mapping from normal programs to the satisfiability modulo difference logic formalism (to be introduced in detail shortly). That translation paved the way towards the implementation of answer set solvers LP2DIFF (Janhunen et al. 2009) and CMODELS-DIFF (Shen and Lierler 2018a). Similarly, translational constraint answer set solvers MINGO (Liu et al. 2012), DINGO (Janhunen et al. 2011), ASPARTAME (Banbara et al. 2015), EZSMT (Shen and Lierler 2018b) rely on the concepts of completion and level ranking (and its variants, i.e, strong level ranking and strongly connected component level ranking proposed by Niemelä) in devising their translations.

We start by introducing some notation to formally define the concept of level ranking that accommodates the notion of an input vocabulary. By $\mathbb{N}$ we denote the set of natural numbers. For a rule (4), by B^+ we denote its positive part and sometimes identify it with the set of atoms that occur in it, i.e., $\{b_1, \ldots, b_l\}$ (recall that B in (4) stands for the right hand side of the arrow in rule (3)).

Definition 4 (*Level ranking*)
A function $\mathrm{lr} : X \setminus \iota \rightarrow \mathbb{N}$ is a *level ranking* of X for Π relative to vocabulary $\iota \subseteq \sigma$ so that $hd(\Pi) \cap \iota = \emptyset$, when for every atom a in $X \setminus \iota$ the following condition holds: there is B in $Bodies(\Pi, a)$ such that X satisfies B and for every $b \in B^+ \setminus \iota$ it holds that $\mathrm{lr}(a) - 1 \geq \mathrm{lr}(b)$.

We now restate Theorem 8 from Lierler and Susman (2017) that captures the relation between input answer sets of a program and models of input completion by means of level ranking.

Theorem 5
For a program Π over vocabulary σ, vocabulary $\iota \subseteq \sigma$ so that $hd(\Pi) \cap \iota = \emptyset$, and a set X of atoms over σ that is a model of input completion $ICom p(\Pi, \iota)$, X is an input answer set of Π relative to ι if and only if there is a level ranking of X for Π relative to ι.

This result is related to the characterization of answer sets of a logic program as models of its completion (Fages 1994).

Constraints. Lierler and Susman (2017) illustrated that the notion of a "constraint" (as understood in classical literature on constraint processing within the artificial intelligence realm) coincides with the notion of a ground literal of satisfiability modulo theories. Furthermore, a *constraint satisfaction problem (CSP)*, which is usually defined by a set of constraints, can be identified with the conjunction of ground literals. This conjunction is evaluated by means of first-order logic interpretations/structures representative of a particular "uniform" SMT-theory – a term introduced by Lierler and Susman (2017). An *SMT-theory* (Barrett and Tinelli 2014) is a set of interpretations/structures. A *uniform SMT-theory* (Lierler and Susman 2017) is a set of interpretations whose domain, interpretation of predicates and "interpreted" function symbols are fixed.

In practice, special forms of constraints are commonly used. *Integer linear constraints* are examples of these special cases. For instance,

$$2x + 3y > 0 \tag{9}$$

is a common abbreviation for an integer linear constraint. In line with Lierler and Susman, we identify linear integer inequality (9) with a ground atom

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

where we assume an SMT-theory called *Integer Linear Arithmetic or Linear Integer Arithmetic (ILA)* (see, for instance, the paper by Bromberger et al. (2015)). This theory is defined by the set of all possible interpretations, whose domain is the set of integers, the predicate $>$ is interpreted as an arithmetic greater relation/predicate symbol; function symbols $+$ and $\times$ are interpreted as usual in arithmetic; 0-arity function symbols 2, 3, and 0 are interpreted by mapping these into respective domain elements (identified with the same symbol). The constraint (9) contains uninterpreted 0-arity function symbols x and y that are frequently referred to as object constants (in logic literature) or variables (in constraint processing literature).

We call an interpretation satisfying a CSP, which we understand as the conjunction of ground literals, its *solutions*. We identify this interpretation with a function called *valuation* that provides a mapping for uninterpreted function symbols to domain elements. For example, one of the solutions to the CSP composed of a single constraint (9) within ILA-theory is a valuation that maps x to 0 and y to 1. Formulas composed of integer linear constraints and interpreted using SMT-theory ILA are said to be within *ILA-logic* (Barrett and Tinelli 2014).

Other commonly used SMT-theories are called *difference logic* (DL) (Nieuwenhuis and Oliveras 2005) and *linear arithmetic* (LA) (Barrett and Tinelli 2014). In difference logic the set of interpretation defining this theory is that of ILA. Yet, difference logic restricts the syntactic form of constraints to the following $x - y \leq k$, where x and y are variables and k is 0-arity function

symbol interpreted by a mapping to domain elements (integers). Linear arithmetic logic differs from ILA-logic in its SMT-theory: domain of linear arithmetic logic is a set of *real* numbers.

4 Constraint Answer Set Programs and SMT Formulas, Formally

Let σ_r, σ_e, and σ_i be three disjoint vocabularies. We refer to their elements as *regular*, *strict-irregular* atoms, and *non-strict-irregular* atoms, respectively. The terms strict and non-strict are due to Gebser et al. (2016), where the authors introduce the CASP language that permits capturing two commonly used semantics in CASP dialects.

Definition 6 (*Constraint answer set program and its answer sets*)
Let $\sigma = \sigma_r \cup \sigma_e \cup \sigma_i$ be a vocabulary so that regular atoms σ_r, strict-irregular atoms σ_e, and non-strict-irregular atoms σ_i are disjoint; $\mathscr{B}$ be a set of constraints; γ be an injective function from the set of irregular literals over $\sigma_e \cup \sigma_i$ to $\mathscr{B}$; and Π be a logic program over σ such that $hd(\Pi) \cap (\sigma_e \cup \sigma_i) = \emptyset$. We call a triple $P = \langle \Pi, \mathscr{B}, \gamma \rangle$ a *constraint answer set program* (CAS program) over vocabulary σ.

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 $\sigma_e \cup \sigma_i$, and
(b) the following CSP has a solution:

$$\{\gamma(a) \mid a \in X \cap (\sigma_e \cup \sigma_i)\} \cup \{\gamma(\neg a) \mid a \in \sigma_e \setminus X\}.$$

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).

It is now time to remark on the differences between regular, strict-irregular, and non-strict-irregular atoms. If vocabulary σ only consists of regular atoms σ_r (sets σ_e and σ_i of irregular atoms are empty) then CAS program turns into a logic program under answer set semantics. Per condition (a) all irregular atoms are part of the input/extensional vocabulary. Intuitively, irregular atoms carry additional information that goes beyond their truth value assignment. This fact culminates in the statement of the (b) condition in the definition of an answer set. The (b) condition also points at the difference between strict-irregular and non-strict-irregular atoms. While the presence of irregular atoms in set X of atoms requires a constraint of this atom to be satisfied, only the absence of a strict-irregular atom requires a constraint of its complement to be satisfied. The non-strict irregular atoms do not pose the latter restriction.

In the sequel, we utilize vertical bars to mark irregular atoms that have intuitive mappings into respective constraints. For instance, given an integer variable x, the expression $|x < 0|$ corresponds to an irregular atom that is mapped into constraint/inequality $x < 0$; similarly irregular literal $\neg|x < 0|$ is mapped into constraint/inequality $x \geq 0$.

Example 2
Let us consider CAS program $P_1 = \langle \Pi_1, \mathscr{B}_1, \gamma_1 \rangle$ from Example 3 by Lierler and Susman (2017).

Logic program Π_1 — the first element of the tuple defining P_1 — follows

$$\begin{array}{l} \{switch\}. \\ lightOn \leftarrow\ switch,\ not\ am. \\ \leftarrow not\ lightOn. \\ \{am\}. \\ \leftarrow not\ am, |x < 12|. \\ \leftarrow am, |x \geq 12|. \\ \leftarrow |x < 0|. \\ \leftarrow |x > 23|. \end{array} \tag{10}$$

The set σ_r of regular atoms of P_1 is

$$\{switch, am, lightOn\}.$$

The set σ_e of strict-irregular atoms of P_1 is

$$\{|x < 0|,\ |x < 12|,\ |x \geq 12|,\ |x > 23|\}, \tag{11}$$

where x is an integer variable (representing hours of the day). The set σ_i of non-strict-irregular atoms of P_1 is empty.

The first line of the program is understood as follows: *The action* switch *is exogenous.* The second two lines are identical to these of logic program (7). The fourth line we can intuitively read as: *It is night (*not am*) or morning (*am*)*. The last four lines of the program state:

- It must be *am* when $x < 12$.
- It is impossible for it to be *am* when $x \geq 12$.
- Variable x must be nonnegative.
- Variable x must be less than or equal to 23.

Set $\mathscr{B}_1$ consists of integer linear constraints including constraints

$$\{x < 0,\ x \geq 0,\ x < 12,\ x \geq 12,\ x > 23,\ x \leq 23\},$$

Mapping γ_1 is defined as follows

$$\gamma_1(a) = \begin{cases} \text{constraint } x < 0 & \text{if } a = |x < 0| \\ \text{constraint } x \geq 0 & \text{if } a = \neg|x < 0| \\ \text{constraint } x < 12 & \text{if } a = |x < 12| \text{ or } a = \neg|x \geq 12| \\ \text{constraint } x \geq 12 & \text{if } a = |x \geq 12| \text{ or } a = \neg|x < 12| \\ \text{constraint } x > 23 & \text{if } a = |x > 23| \\ \text{constraint } x \leq 23 & \text{if } a = \neg|x > 23|. \end{cases}$$

Consider set

$$\{switch,\ lightOn, |x \geq 12|\} \tag{12}$$

over the vocabulary of P_1. This set is the only input answer set of Π_1 relative to irregular atoms of P_1. Also, the integer linear constraint satisfaction problem formed by the constraints in

$$\begin{array}{c} \{\gamma_1(\neg|x < 0|),\ \gamma_1(\neg|x < 12|),\ \gamma_1(|x \geq 12|),\ \gamma_1(\neg|x > 23|)\} \\ = \\ \{x \geq 0,\ x \geq 12,\ x \leq 23\} \end{array}$$

has a solution. There are 12 valuations $v_1 \dots v_{12}$ for integer variable x, which satisfy this CSP, namely, $x^{v_1} = 12, \dots, x^{v_{12}} = 23$. It follows that set (12) is an answer set of P_1. Pair

$$\langle \{switch,\ lightOn,\ |x \geq 12|\}, \nu_1 \rangle$$

is one of the twelve extended answer sets of P_1.

To illustrate the difference between strict and non-strict irregular atoms consider the CAS program P_1' that differs from P_1 only in sets σ_e and σ_i. In particular, the set σ_e of strict-irregular atoms of P_1' is empty. The set σ_i of non-strict-irregular atoms of P_1' is (11). Set (12) is the only input answer set of Π_1 relative to irregular atoms of P_1'. Also, the integer linear constraint satisfaction problem formed by the constraint in

$$\begin{gathered} \{\gamma_1(|x \geq 12|)\} \\ = \\ \{x \geq 12\} \end{gathered}$$

has a solution. There are indeed infinite number of valuations $v_1 \dots v_{12},\ v_{13}, \dots$ for integer variable x, which satisfy this CSP, namely, $x^{v_1} = 12, \dots, x^{v_{12}} = 23,\ x^{v_{13}} = 24, \dots$.

We direct a reader to the paper by Gebser et al. (2016), where the authors discuss in detail the rationale behind the two distinct kinds of irregular atoms.

We note that if we consider a CAS program whose set σ_i of non-strict-irregular atoms is empty then it falls into a class of programs accepted by such CASP system as CLINGCON (Gebser et al. 2009), given that constraints are in the realm of ILA. Similarly, if we consider a CAS program whose set σ_e of strict-irregular atoms is empty and whose atoms from σ_i only occur in denials, then it falls into a class of programs that such CASP system as EZCSP accepts (given that constraints are in the realm of ILA or LA).

Janhunen et al. (2017) lift the restriction on irregular atoms not to occur in the heads of program's rules. Rather, they divide all atoms into "defined" and "external" (input/extensional, if to follow the terminology of this paper), where defined atoms may occur in heads. In other words, defined irregular atoms do not longer need to be part of the input vocabulary. This is an important and an interesting extension within CASP that is utilized in the implementation of such CASP systems as CLINGO[DL] and CLINGO[LP]. To the best of our knowledge these are the only two currently available CASP systems that allow "defined" irregular atoms.

SMT Formulas. Here we state the definition of an SMT formula (Barrett and Tinelli 2014). This concept is fundamental for most translational approaches to CASP.

Definition 7 (*SMT formulas and its models*)
Let $\sigma = \sigma_r \cup \sigma_e \cup \sigma_i$ be a vocabulary (so that σ_r, σ_e, and σ_i are disjoint); $\mathscr{B}$ be a set of constraints; γ be an injective function from the set of irregular literals over $\sigma_e \cup \sigma_i$ to $\mathscr{B}$; F be a propositional formula over σ. We call a triple $\mathscr{F} = \langle F, \mathscr{B}, \gamma \rangle$ an *SMT formula* over vocabulary σ. A set $X \subseteq At(F)$ is a *model* of SMT formula $\mathscr{F}$ if

(a.1) X is a model of F, and
(b.1) the CSP constructed in (b) of Definition 6 has a solution.

A pair $\langle X, \nu \rangle$ is an *extended model* of $\mathscr{F}$ if X a *model* of $\mathscr{F}$ and ν is a solution to the CSP in (b.1).

We are now ready to provide the translation from logic programs to SMT formulas (this translation is inspired by level ranking results). We then present a translation by Lierler and Susman (2017) that maps CAS programs into SMT formulas.

As before, we utilize vertical bars to mark irregular atoms (introduced within the translation) that have intuitive mappings into respective constraints. For instance, the expression $|lr_b - 1 \geq lr_a|$ corresponds to an irregular atom that is mapped into constraint/inequality $lr_b - 1 \geq lr_a$, where lr_a and lr_b are variables over integers. To refer to the constraints corresponding to irregular literals we use superscript $\downarrow$. For example,

$$\begin{aligned} |lr_b - 1 \geq lr_a|^{\downarrow} &= lr_b - 1 \geq lr_a \\ \neg|lr_b - 1 \geq lr_a|^{\downarrow} &= lr_b - 1 < lr_a. \end{aligned}$$

Definition 8 (*Translation from a logic program to an SMT formula*)
Let Π be a logic program over vocabulary σ^Π and vocabulary $\iota \subseteq \sigma^\Pi$ such that none of ι's elements occur in the heads of rules in Π. For every atom a in $\sigma \setminus \iota$ that occurs in Π we introduce an integer variable lr_a. The SMT formula $\mathscr{F}^\Pi = \langle F^\Pi, \mathscr{B}^\Pi, \gamma^\Pi \rangle$ is constructed as follows

- formula F^Π is a conjunction of the following
  1. rules (3) in Π;
  2. for each atom $a \in \sigma^\Pi \setminus \iota$ the implication $\quad a \rightarrow \bigvee\limits_{a \leftarrow B \in \Pi} \Big(B \wedge \bigwedge\limits_{b \in B^+ \setminus \iota} |lr_a - 1 \geq lr_b| \Big)$
- set σ_r of $\mathscr{F}^\Pi$ is formed by the atoms in σ^Π; set σ_e of $\mathscr{F}^\Pi$ is formed by the irregular atoms of the form $|lr_a - 1 \geq lr_b|$ introduced in 2; and set σ_i of $\mathscr{F}^\Pi$ is empty;
- constraints in $\mathscr{B}^\Pi$ are composed of inequalities $|lr_a - 1 \geq lr_b|^{\downarrow}$ and $\neg|lr_a - 1 \geq lr_b|^{\downarrow}$ for all irregular atoms of the form $|lr_a - 1 \geq lr_b|$ introduced in 2;
- function γ^Π maps irregular literals formed from atoms of the form $|lr_a - 1 \geq lr_b|$ introduced in 2 to constraints in $\mathscr{B}^\Pi$ in a natural way captured by $\downarrow$ function.

An SMT formula $\mathscr{F}^\Pi$ has two properties: (i) it has models if and only if respective answer set program Π has answer sets and (ii) any model I of this formula is such that $I \cap \sigma$ forms an answer set of Π. This is a consequence of Theorem 9 by Lierler and Susman (2017), which follows from Theorem 5 restated here.

Definition 9 (*Translation from a CAS program to an SMT formula*)
Let $P = \langle \Pi, \mathscr{B}, \gamma \rangle$ be a CAS program over $\sigma^P = \sigma_r^P \cup \sigma_i^P \cup \sigma_e^P$. For every atom a in σ_r^P that occurs in Π we introduce an integer variable lr_a. The SMT formula $\mathscr{F}^P = \langle F^P, \mathscr{B}^P, \gamma^P \rangle$ over $\sigma = \sigma_r \cup \sigma_i \cup \sigma_e$ is constructed as follows

- the formula F^P is a conjunction consisting of formulas in 1 and 2 of Definition 8, where we understand σ^Π as σ^P and ι as $\sigma_i^P \cup \sigma_e^P$;
- set σ_r of $\mathscr{F}^P$ is formed by the atoms in σ_r^P; set σ_e of $\mathscr{F}^P$ is formed as the union of σ_e^P and the irregular atoms of the form $|lr_a - 1 \geq lr_b|$ described in Definition 8; set σ_i of $\mathscr{F}^P$ is formed by the atoms in σ_i^P;
- constraints $\mathscr{B}^P$ are composed of the elements in $\mathscr{B}$ and the elements in $\mathscr{B}^\Pi$ described in Definition 8;
- mappings of γ^P are composed of the elements in γ and the elements in γ^Π described in Definition 8.

An SMT formula $\mathscr{F}^P$ has two properties: (i) it has models if and only if respective CAS program has answer sets and (ii) any model I of this formula is such that $I \cap \sigma^P$ forms an answer set of P. This is a consequence of Theorem 10 by Lierler and Susman (2017) that follows from Theorem 5 restated here.

5 Integrational Approach via System EZCSP

As stated in the introduction, this paper presents the details behind two CASP systems, namely, EZCSP[2] and EZSMT[3]. The former is a representative of the integrational approach. The latter is a representative of a translational SMT-based approach.

Both systems, EZCSP and EZSMT, accept programs written in the language that is best documented by Balduccini and Lierler (2017). We call this language EZ. The TS problem formulation of this paper is in that language. This paragraph uses system EZCSP in its claims. The same claims are applicable for the system EZSMT. As discussed earlier, CASP systems typically start their computation by grounding a given program. System EZCSP uses grounder GRINGO (Gebser et al. 2011) for this purpose. Following our example from the end of the introduction, ground rule

```
required(c(a,b)==1):- cost(a,b,1), route(a,b).        (13)
```

exemplifies the kinds of ground rules produced by the EZCSP system at the time of grounding. Atoms of the form $\mathit{required}(\beta)$ instruct the EZCSP system that β introduces a non-strict-irregular atom. Even though "required atoms" occur in the head of rules, semantically these rules are denials with the irregular atom "complementary" to β occurring in the body.[4] For instance, rule (13) stands for the following denial, written in style used in Section 4:

$$\leftarrow \mathit{cost}(a,b,1),\ \mathit{route}(a,b),\ |c(a,b) \neq 1|.$$

Atoms such as $|c(a,b) \neq 1|$ belong to non-strict-irregular atoms of the CAS program produced by grounding of EZCSP. These CAS programs (i) contain no strict-irregular atoms and (ii) contain irregular atoms only in denials.

Figure 7 depicts the architecture of the EZCSP system. The graphic is reproduced from the paper by Balduccini and Lierler (2017). We follow the presentation by Balduccini and Lierler to state the most essential details behind the EZCSP system. The first step of the execution of EZCSP (corresponding to the *Pre-processor* component in the figure) consists in running a preprocessor, which translates an input EZ program into a syntactically legal ASP program. This is accomplished by replacing the occurrences of arithmetic functions and operators in expressions of the form $\mathit{required}(\beta)$ by auxiliary function symbols. For example, an expression $v > 2$ in $\mathit{required}(v > 2)$ is replaced by $gt(v,2)$. The *Grounder* component of the architecture transforms the resulting program into its propositional equivalent, a regular program, using an off-the-shelf grounder such as GRINGO (Gebser et al. 2007; Gebser et al. 2011). This regular program is then passed to the EZCSP *Solver* component.

The EZCSP *Solver* component iterates between ASP and constraint programming computations by invoking the corresponding components of the architecture. Specifically, the *ASP Solver* component computes an answer set of a given regular program using an off-the-shelf ASP solver, such as CMODELS or CLASP. If an answer set is found, the EZCSP solver runs the *CLP Translator* component, which maps the CSP problem corresponding to the computed answer set to a

[2] Solver EZCSP is available at `http://mbal.tk/ezcsp/` .
[3] Solver EZSMT is available at `https://www.unomaha.edu/college-of-information-science-and-technology/natural-language-processing-and-knowledge-representation-lab/software/ezsmt.php` .
[4] It is due to note that β may be a more complex expression than an irregular atom. For example, it may contain a disjunction of irregular atoms. Semantically, a rule with such β expression in its head corresponds to the denial that extends the body of this rule with the conjunction of the complementary atoms formed from β.

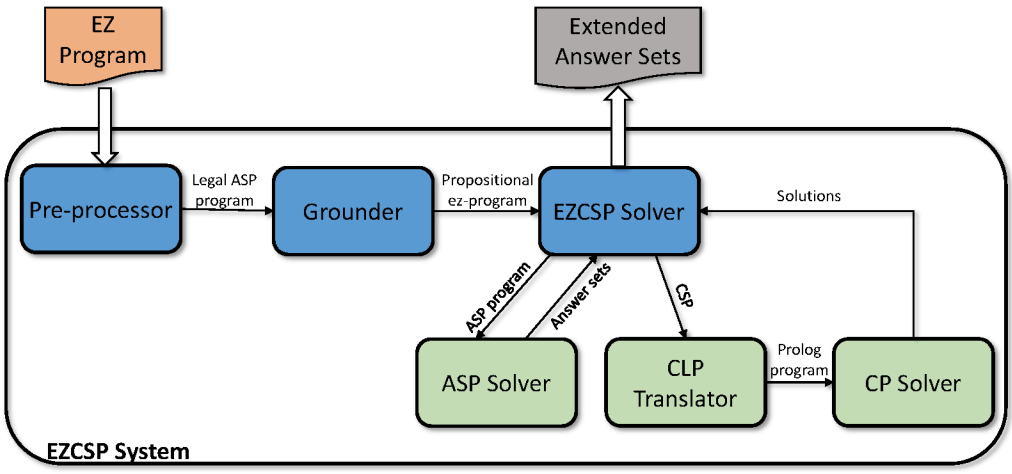


Fig. 7: Architecture of the EZCSP system

Prolog program. The program is then passed to the *CP Solver* component, which uses the CLP tools such as SICStus (Carlsson and Fruehwirth 2014), SWI Prolog (Wielemaker et al. 2012) or Bprolog (Zhou 2012), to solve the CSP instance. Recent version of EZCSP augment the *CLP Translator* component with the possibility of producing MiniZinc (Nethercote et al. 2007) formulations of CSP problems. As a result, MiniZinc solvers[5] can be used in place of CLP tools. Finally, the EZCSP *Solver* component gathers the solutions to the respective CSP problem and combines them with the answer set obtained earlier to form extended answer sets. Additional extended answer sets are computed iteratively by finding other answer sets and the solutions to the corresponding CSP problems.

It is essential to note that in integrational approaches the communication schemas between the two participating solving mechanisms are important. The presented architecture of EZCSP showcases the so called *blackbox* integration approach. The beauty of the *blackbox* approach is its flexibility in utilizing the existing technology as both answer set solver and CSP solver can be taken as they are. Yet, it is obvious that such an integration does not provide any means to rely on advances in search of an answer set solver in earlier iterations or prune the computation of an answer set solver based on information from a CSP. The EZCSP system also implements so called *grey-box and clear-box* integration, where it accommodates continuation in search and early pruning, respectively. In this capacity the EZCSP is confined to utilizing a particular answer set solver CMODELS via its internal API.

Briefing: Integrational Systems. This is a good place to speak of some other integrational systems. We first consider solver CLINGCON (Gebser et al. 2009). From the original design of the system to its latest version, its authors were proponents of a clear-box integration. Its original implementation established the clear-box communication between answer set solver CLASP (a solver of answer set system CLINGO) and constraint processing system GECODE. The second, recent, implementation of CLINGCON (Banbara et al. 2017) uses sophisticated "in house CSP"

[5] `https://www.minizinc.org/` .

propagators to replace the GECODE system. It is also a close relative to the newest representatives of the integrational approach systems CLINGO[DL] and CLINGO[LP] (Janhunen et al. 2017). Latest version of CLINGCON and these two systems are a product of a systematic effort by University of Potsdam to create an extensible infrastructure to support answer set programming based solutions. Framework CLINGO 5 (Gebser et al. 2016) provides comprehensive interfaces to assist the development of

- extensions for the language accepted by grounder GRINGO (the functionality of makes the Pre-processor component required in the architecture of EZCSP obsolete) to accommodate, for example, the irregular atoms as discussed here;
- extensions for implementing specialized propagators to accommodate, for example, the processing of irregular atoms as discussed here natively and efficiently utilizing the fact that these propagators are defined within CLINGO itself.

It is interesting to note that one can view/name constraint answer set programming as ASP modulo constraints/theories (following the tradition of SMT). An interesting related paradigm to CASP is called ASP modulo acyclicity (Bomanson et al. 2016). In this paradigm, specialized propagator is used to capture constraints specific to graph/tree problems. Bomanson et al. (2016) describe an integrational solver for ASP modulo acyclicity based on answer set solver CLASP and use Hamiltonian cycle problem as one of the benchmarks to showcase the system.

Briefing: Input Languages of CASP systems. In this paper we speak in some detail about the EZ language that is used for problem encodings to interface CASP systems EZCSP and EZSMT. The other CASP tools such as CLINGCON, CLINGO[DL], or CLINGO[LP] introduce their own ASP-like dialects to state CAS programs with schematic variables. At the moment, the task of transferring an encoding designed for one CASP system into an encoding meant for another CASP system requires a programmer experienced with dialects of these systems. An effort in spirit of the design of the standard ASP-Core-2 Language (Calimeri et al. 2019) (to interface ASP solvers) is now due for the case of CASP languages.

6 Translational Approach via System EZSMT

The concluding part of Section 4 describes how given a constraint answer set program one can construct an SMT formula whose models capture its answer sets. This construction relies on the concepts of completion and level ranking. It is worth noting that Janhunen (2006) introduced refined "strong" and "strongly connected component (SCC)" level rankings for normal logic programs (under a name of level numberings). These refined versions of level ranking can be used to reduce the size of translation from a program to an SMT formula. Shen and Lierler (2018a) generalized these results to logic programs whose rules are of the form (3). These ideas are also applicable within CAS programs and are utilized in the implementation of the CASP solver EZSMT (Shen and Lierler 2018b). We now review the key features of the EZSMT system and its building blocks to showcase a translational approach.

In a nutshell system EZSMT translates a given CAS program into an SMT formula and then utilizes an SMT solver as its search back-end to find models of the formula. Then, each model found in this way is mapped to an answer set of the given program. In addition to difference logic, ILA, and LA logics, system EZSMT can use such SMT-logics as AUFLIRA and AUFNIRA (Tinelli

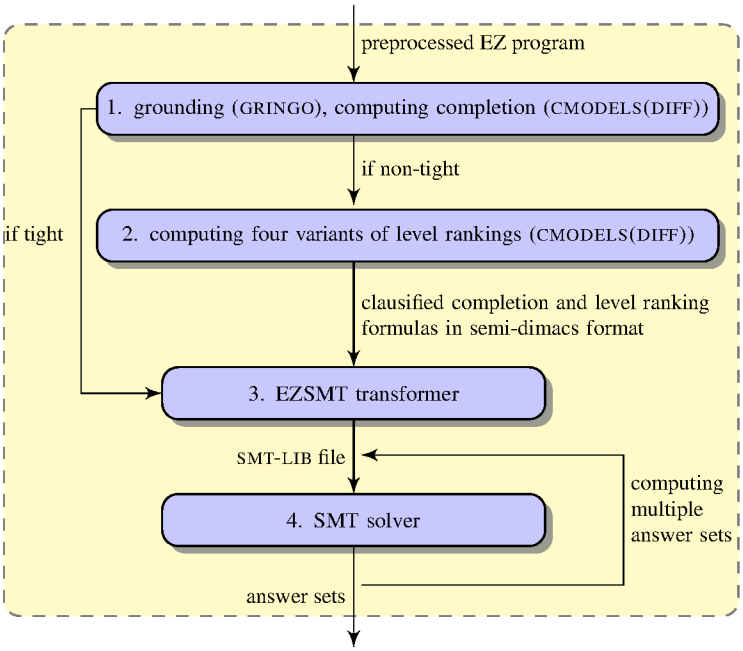


Fig. 8: System EZSMT Architecture

and Barrett 2015). Logic AUFLIRA enables us to state linear constraints that may simultaneously contain integer and real variables. Logic AUFNIRA permits nonlinear constraints, too. As mentioned earlier, the EZSMT system accepts programs in the EZ language. This language is extended by several directives that allow users to specify a domain for a constraint variable.

Figure 8 illustrates the architecture of system EZSMT. The graphic is reproduced from the paper by Shen and Lierler (2018b). The system takes an EZ program as an input. It starts by applying the Pre-processor component of system EZCSP (see Figure 7); the rationale behind the application of this component is the same as in case of the EZCSP system discussed in previous section. It then utilizes grounder GRINGO (Gebser et al. 2011) for eliminating ASP variables. Routines of system CMODELS(DIFF) (Shen and Lierler 2018a) are used to compute input completion and level rankings of the program (Steps 1 and 2). During Step 1, EZSMT also determines whether the program is "tight" or not. The tightness (Fages 1994) is a syntactic condition on a program. Intuitively, a program is tight if it has no circular dependencies between its head and positive body atoms across a program. A simple example of a non-tight program is a program with a single rule $p \leftarrow p$. In case when a program is tight it is sufficient to replace a formula in 2 of Definition 8 by a simpler formula (8) stemming from the completion to achieve a one-to-one correspondence between the answer sets of a given program and the models of the corresponding SMT formula. If the program is not tight, the corresponding level ranking formula is added. A procedure used by EZSMT to perform this task is identical to that of CMODELS(DIFF) (Shen and Lierler 2018a). System EZSMT may construct different kinds of level ranking formulas including strong level ranking formulas, SCC level ranking formulas, and strong SCC level ranking formulas, respectively. The resulting formulas are clausified to produce an output in semi-Dimacs format (Susman and Lierler 2016) (Step 3.), which is transformed into SMT-LIB syntax — a standard input language for SMT solvers (Barrett et al. 2015) — using the procedure described

by Susman and Lierler (2016) and Shen and Lierler (2018b). Finally, one of the SMT solvers CVC4 (Barrett et al. 2011), Z3 (Wintersteiger et al. 2016), or YICES (Dutertre 2017) is called to compute models (Step 4.). In fact, any other SMT solver supporting SMT-LIB can be utilized easily, too. The EZSMT system allows one to compute multiple (extended) answer sets. It utilizes ideas exploited in the implementation of CMODELS(DIFF) (Shen and Lierler 2018a, Section 5). In summary, after computing an (extended) answer set X of a program EZSMT invokes an SMT solver again by adding formulas encoding the fact that a newly computed model should be different from X. This process is repeated until the pre-specified number of solutions is enumerated or it has been established that no more solutions exist. The described process of enumerating multiple solutions is naive and begs for an improvement. Gebser et al. (2007) describe sophisticated methods for enumerating answer sets implemented within answer set solver CLASP.

Briefing: Translational Systems. We mentioned such translational constraint answer set solvers as MINGO (Liu et al. 2012), DINGO (Janhunen et al. 2011), and ASPARTAME (Banbara et al. 2015). To process CAS programs with LA and ILA logics, the MINGO system computes program's input completion extended with level ranking formulas and then translates these formulas into mixed integer programming expressions. After that it uses the CPLEX solver (IBM 2009) to solve these formulas. To process CAS programs with difference logic, system DINGO translates these programs into SMT(DL) formulas using translations in spirit of those in EZSMT and applies the SMT solver Z3 (De Moura and Bjørner 2008) to find their models. The last translational system that we mention is ASPMT2SMT (Bartholomew and Lee 2014). The ASPMT2SMT system is a close relative of EZSMT in the sense that it utilizes SMT solver Z3 for search. Solver ASPMT2SMT is nevertheless restricted to tight programs. It computes the completion of a given program and then invokes Z3 solver to enumerate the solutions. System ASPARTAME differs from all of the above as it translates the CAS programs with IL arithmetic into answer set programs.

7 Big Picture and Experimental Data

We start this section by summarizing the modern landscape of CASP technology. We then proceed towards presenting some experimental data to showcase the current computational capabilities of the field.

Figure 9 reproduces part of the table stemming from Janhunen et al. (2017) that provides a great overview of the key features and capabilities of the constraint answer set programming systems (the only difference between the original table and the one present is that system EZSMT is now marked as the one capable of processing non-tight programs; that was not the case prior). In the first row of the table, *integrational* is taken to be the complement of *translational*. The row *real numbers* refers to the ability of a solver to support constraints over real numbers. Every solver supports constraints over integers. The row *optimization* refers to the ability of a solver to support optimization statements that are valuable in designing solutions to real world problems (Andres et al. 2012). The row *non-tight* points to systems that do not require input programs to satisfy a syntactic condition of tightness. The table introduces two more systems INCA (Drescher and Walsh 2010; Drescher and Walsh 2011) and DLVHEX[CP] (Rosis et al. 2015), which have not surfaced in our discussion. System INCA implements a lazy propagation-translation approach that, along processing time, translates integer constraints of the program into logic program rules. System DLVHEX[CP] is a CLINGO-based system that utilizes CSP solver GECODE.

	CLINGO [DL]	CLINGO [LP]	CLING CON	ASPAR TAME	INCA	EZ CSP	EZ SMT	MINGO	DINGO	ASPMT 2SMT	DLVHEX [CP]
translational	✗	✗	✗	✓	✗	✗	✓	✓	✓	✓	✗
real numbers	✗	✓	✗	✗	✗	✓	✓	✓	✗	✓	✗
optimization	✗	✓	✓	✓	✓	✗	✗	✗	✗	✗	✓
non-tight	✓	✓	✓	✓	✓	✓	✓	✓	✓	✗	✓

Fig. 9: Solvers and their features

The experimental data presented here is reproduced from the paper by Shen and Lierler (2018b). It focuses on the performance of three systems EZSMT, EZCSP and CLINGCON (Banbara et al. 2017; Ostrowski 2018). The benchmarks are posted at the EZSMT website (see Footnote 3). In conclusion of this section we remark on how these systems compare to other CASP solvers.

We now point to the origins of the considered benchmarks. 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 CLINGCON and EZSMT encodings of IS from Banbara et al. (2017). We include a benchmark problem called Blending (BL) (Biavaschi 2017) and extend it to BL*, which contains variables over both integers and reals. Also, we use the Bouncing Ball (BB) domain (Bartholomew 2016). It is important to remark that the encoding for BB domain results in a tight program. This domain uses nonlinear constraints over real numbers. Three more benchmarks, namely, RoutingMin (RMin), RoutingMax (RMax), and Travelling Salesperson (TS) are obtained from Liu et al. (2012). The obtained TS benchmark is an optimization problem that we turn into a TS variant considered in the introduction. 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. The next benchmark, Robotics (RB), comes from Young et al. (2017). Also, we present results on two benchmarks from Balduccini et al. (2017), namely, Car and Generator (GN).

All benchmarks are run on an Ubuntu 16.04.1 LTS (64-bit) system with an Intel core i5-4250U processor. The resource allocated for each benchmark is limited to one CPU core and 4GB RAM. We set a timeout of 1800 seconds. No problems are solved simultaneously. The systems that we use to compare the performance of variants of EZSMT (invoking SMT solver Z3 v. 4.5.1; YICES v. 2.5.4) are CLINGCON v. 3.3.0 and the variants of EZCSP v. 2.0.0 (invoking ASP solvers CLASP v. 3.2.0; and constraint solver SWIprolog v. 7.4.1 or MINIZINC v. 2.0.2). The GRINGO system v. 4.5.3 is used as grounder for EZSMT and EZCSP with one exception: GRINGO v. 3.0.5 is utilized for EZCSP for the Reverse Folding benchmark (due to some incompatibility issues).

Figure 10 summarizes the main results. In this figure, we use EZSMT(Z3) and EZSMT(YICES) to denote two variants of EZSMT. Acronym EZCSP-CLASP-SWI (EZCSP-CLASP-MZN) stands for a variant of EZCSP, where CLASP is utilized as the answer set solver and SWIprolog (MINIZINC, respectively) is utilized as a constraint solver.

In Figure 10, we present cumulative time in seconds of all instances for each benchmark with numbers of unsolved instances due to timeout or insufficient memory inside parentheses. The "/" sign indicates that this solver or its variant does not support the kinds of constraints occurring in the encoding. For example, CLINGCON does not support constraints over real numbers or nonlinear constraints. The total number of used instances is shown in parentheses after a benchmark name. All the steps involved, including grounding and transformation, are reported as parts of the solving time. The benchmarks are divided into categories by double separations. Figure 11

Category	Benchmark	CLINGCON	EZSMT(Z3) SCC	EZSMT(Z3) SCCStrong	EZSMT(YICES) SCC	EZSMT(YICES) SCCStrong	EZCSP CLASP-SWI	EZCSP CLASP-MZN
NT-IL	RMin (100)	**4.68**	8.76	11.8	5.81	7.57	126007(70)	126002(70)
——	RMax (100)	3144	459	**22.4**	5190	5945	180000(100)	180000(100)
——	TS (30)	455	7347(4)	43620(24)	1881(1)	75.2	**14.3**	54000(30)
——	LB (22)	**3002(1)**	9510(1)	10089(2)	4399(2)	5512(2)	12558(6)	12638(6)
T-IL	RF (50)	326	6058(2)		27840(14)		**101**	7218(4)
——	IS (30)	**9080(5)**	9200(5)		9098 (5)		41446(21)	39458(21)
——	WS (30)	52.5	29.2		**5.23**		54000(30)	54000(30)
T-INL	Car (8)	/	0.32		**0.25**		10.1	2.34
T-RL	BL (30)	/	88.4		**47.4**		18322(9)	/
——	GN (8)	/	0.58		**0.48**		5641(3)	/
——	RB (8)	/	0.4		**0.39**		2.04	/
T-RNL	BB (5)	/	3663(2)		**0.98**		9000(5)	/
T-ML	BL* (30)	/	**5573(2)**		/		/	/

Fig. 10: Experimental Data

T	NT	I	R	M	L	NL
tight	non-tight	integer	real	mixed	linear	non linear

Fig. 11: Meaning of the Category Column Letters

presents the readings of the letters in the category column, where the first two letters refer to the syntactic condition on a logic program; the middle three letters refer to the domains of constraint variables of the program; and the last two letters refer to the kinds of constraints.

Systems CLINGCON, EZCSP-CLASP-SWI, and EZCSP-CLASP-MZN are run in their default settings. For non-tight programs, system EZSMT with strongly connected components level rankings (flags `-SCClevelRanking` and `-SCClevelRankingStrong`) show best performance.

In summary, we observe that CLINGCON achieves first positions in three benchmarks. EZCSP-CLASP-SWI and EZSMT(Z3) win in two benchmarks, respectively. EZSMT(YICES) ranks first in six benchmarks. The EZSMT(YICES) system displays the best overall results. Utilizing different SMT solvers may improve the performance of EZSMT in the future.

On anticipated performance of related systems. CASP solver CLINGO[LP] (Janhunen et al. 2017) handles linear constraints over integers or reals. The experimental analysis presented by Janhunen et al. (2017) only considers programs with constraints over integers. On these benchmarks, CLINGCON outperforms CLINGO[LP]. Susman and Lierler (2016) compare the performance of MINGO (Liu et al. 2012) and EZSMT on tight programs. The latter consistently has better performance. The ASPMT2SMT (Bartholomew and Lee 2014) system is a close relative of EZSMT in the sense that it utilizes SMT solver Z3 for search. We expect that EZSMT(Z3) times mimic these of ASPMT2SMT on tight programs.

8 Discussion and Future Directions

This article was meant to construct a compelling tale of constraint answer set programming developments of the past decade supplying the interested reader with birds-eye view of the area and enough literature links to acquire details when needs be. This concluding section lists open questions and possible directions of the field.

Gebser et al. (2016) point out how an ASP-based problem solving frequently requires capabilities going beyond classical ASP language and systems. They observe that ASP system CLINGO and/or its grounder GRINGO and/or its solver CLASP often serve as important building blocks of more complex systems (including such systems as constraint answer set solvers). The fundamental contribution by Gebser et al. (2016) was to conceive CLINGO 5 framework that provides a general purpose interface which helps to make extensions of GRINGO/CLASP systems a routine and systematic process. This interface also targets facilitation of streamlining communication between theory/constraint propagation and answer set solving propagation as well as other advanced techniques such as conflict driven learning implemented in CLASP. As such CLINGO 5 can be seen as one of the key contributions to the CASP community. It provides a general purpose platform for bootstrapping unique constraint answer set programming solutions.

Automated reasoning spans areas such as satisfiability solving, answer set programming, satisfiability modulo theories solving, integer (mixed) programming, constraint answer set programming, and constraint processing. The relation between answer set solvers and satisfiability solvers is well understood, see, for example, the paper by Lierler (2017). Also, the relation between different instances of answer set solvers has been studied: see the paper by Lierler and Truszczynski (2011). Several representatives of the integration approach to constraint answer programming have been contrasted and compared, see, for instance, the paper by Lierler (2014). Yet, a deeper understanding of how various solving techniques in theory solving of SMT compare to these of constraint processing CSP/CLP is missing. Similarly, the following is an open question: how do techniques in mixed (integer) programming compare to these in SMT, CSP, and/or integer linear programming. At the moment the best we can do is to use constraint answer set programming and its various implementations that include translational approaches to conduct experimental analysis that spans a variety of automated reasoning communities (Lierler and Susman 2017; Janhunen et al. 2017). Dovier et al. (2009) provide us with insights on how CLP solutions to combinatorial search problems compare to these with ASP solutions. Deeper understanding of differences and similarities between algorithms used in these traditionally different areas of AI is due.

Papers by Erdoğan and Lifschitz (2004), Lifschitz (2017), Fandinno et al. (2020), Cabalar et al. (2020), Bomanson et al. (2020), to name a few, provide the techniques for analysing and arguing program correctness in traditional answer set programming. To the best of our knowledge

there were no attempts to lift these methods to the scope of constraint answer set programming. As hybrid answer set programming approaches are making their pronounced way into practice the methodologies for arguing the correctness of such solutions are due. Alternative definitions of CAS programs and their semantics as studied by Lin and Wang (2008), Balduccini (2013), and Bartholomew and Lee (2013) may suit the purpose of the formal arguments of correctness better than the definition presented here.

One way to view translational methods in constraint answer set programming is as an attempt to utilize existing technology from distinct automated reasoning subfields for solving CASP formulations of solutions to problems. Another way to view these is as an attempt to provide a programming front end of logic programming under answer set semantics to the variety of tools that otherwise possess only limited modeling capabilities. For example, despite the existence of the common standard SMT-LIB language for formulating SMT problems one may not call that a full-fledged programming language. Just as the DIMACS format – standard for communicating with the satisfiability solvers – does not constitute a suitable language for modeling solutions to problems in it directly. As mentioned earlier, CLINGO 5 framework provides infrastructure for bootstrapping novel hybrid answer set programming solutions. It remains to be seen if this framework is sufficient for establishing a full-fledged front end for the translational approach targeting the utilization of SMT solvers that goes beyond traditional (integer) linear arithmetic. For example, particular SMT fragments provide vector and array arithmetics. It is still to be established whether expressions of these logics may prove to be convenient modeling tools (backed up by specialized efficient search techniques of SMT).

MiniZinc[6] is a free and open-source constraint modeling language. In the past decade it became a standard front end for accessing a conglomerate of CSP tools. The translational approaches of constraint answer set solvers looked into utilizing SMT solvers via SMT-LIB so far. In a similar manner, the language of MiniZinc can be utilized for accessing CSP tools that support the MiniZinc language. MiniZinc-based translational approach to constraint answer set solving is still to see the light.

At the closing of Section 5, we mention that no efforts by the research community have been taken to produce a standard input language for CASP solvers. The maturity of the field suggests it is time for such an effort. Possibly, an even more ambitious effort is due. This paper makes it clear that many automated reasoning paradigms — SMT, ASP, CASP, CSP, CLP – are geared towards solving difficult combinatorial search problems. We named several case studies, where researchers attempt to experimentally compare these methods by designing solutions to problems in distinct paradigms and then studying behaviors of respective solvers on these solutions. Providing a standard language to interface tools from distinct communities will allow us to benefit from portfolio approaches (Nudelman et al. 2004) originated in SAT by tapping into a broad spectrum of solving techniques. We trust that the standard language for CASP together with translational techniques that are able to transform CAS programs into the specifications in languages of related paradigms is a promising directions of research. To this end the question of a mature programming methodology for utilizing CASP is in need. At the moment, typical users of ASP are the ones that practice CASP. They borrow so called generate define and test methodology of ASP (Lifschitz 2002; Denecker et al. 2019) that accounts for logic programming aspect of CASP. Yet, the methodology that naturally accounts for constraints is still to come.

[6] https://www.minizinc.org/

Optimization statements are important in (constraint) answer set programming. As one can see in Figure 9 no translational approach supports these statements. Utilizing MiniZinc/CSP solvers would allow to elevate this restriction as CSP solvers typically provide support for optimization problems. Also, MaxSMT (Robinson et al. 2010) concerns SMT solving that provides means to formulate optimization statements. It is yet another direction of work on connecting MaxSMT together with optimization statements of constraint answer set programming.

Acknowledgments

I would like to acknowledge and cordially thank many of my collaborators with whom we had a chance to contribute to an exciting field of Constraint Answer Set Programming and many of my colleagues who have fostered my understanding of the subject matter: Marcello Balduccini, Broes De Cat, Marc Denecker, Martin Gebser, Michael Gelfond, Tomi Janhunen, Roland Kaminski, Martin Nyx Brain, Joohyung Lee, Ilkka Niemelä, Max Ostrowski, Torsten Schaub, Peter Schueller, Da Shen, Benjamin Susman, Cesare Tinelli, Miroslaw Truszczynski, Philipp Wanko, Yuanlin Zhang. Thank you for the years of an incredible journey. Also, I would like to thank the anonymous reviewers for their valuable feedback, which helped to bring the article to this form. This work was partially supported by the NSF 1707371 grant.

References

ABELS, D., JORDI, J., OSTROWSKI, M., SCHAUB, T., TOLETTI, A., AND WANKO, P. 2019. Train scheduling with hybrid ASP. In *Logic Programming and Nonmonotonic Reasoning - 15th International Conference, LPNMR 2019, Philadelphia, PA, USA, June 3-7, 2019, Proceedings*, M. Balduccini, Y. Lierler, and S. Woltran, Eds. Lecture Notes in Computer Science, vol. 11481. Springer, 3–17.

ANDRES, B., KAUFMANN, B., MATHEIS, O., AND SCHAUB, T. 2012. Unsatisfiability-based optimization in clasp. In *Technical Communications of the 28th International Conference on Logic Programming (ICLP'12)*, A. Dovier and V. S. Costa, Eds. Leibniz International Proceedings in Informatics (LIPIcs), vol. 17. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 211–221.

BALDUCCINI, M. 2011. Industrial-size scheduling with ASP+cp. In *Logic Programming and Nonmonotonic Reasoning - 11th International Conference (LPNMR)*. Lecture Notes in Computer Science, vol. 6645. Springer-Verlag, 284–296.

BALDUCCINI, M. 2013. ASP with non-herbrand partial functions: a language and system for practical use. *Theory and Practice of Logic Programming 13*, 547–561.

BALDUCCINI, M. AND GELFOND, M. 2005. Model-based reasoning for complex flight systems. In *Proceedings of Infotech@Aerospace (American Institute of Aeronautics and Astronautics)*.

BALDUCCINI, M., GELFOND, M., NOGUEIRA, M., WATSON, R., AND BARRY, M. 2001. An A-Prolog decision support system for the Space Shuttle. In *Working Notes of the AAAI Spring Symposium on Answer Set Programming*.

BALDUCCINI, M. AND LIERLER, Y. 2017. Constraint answer set solver EZCSP and why integration schemas matter. *Theory and Practice of Logic Programming 17,* 4, 462–515.

BALDUCCINI, M., MAGAZZENI, D., MARATEA, M., AND LEBLANC, E. C. 2017. CASP solutions for planning in hybrid domains. *Theory and Practice of Logic Programming 17,* 4, 591–633.

BANBARA, M., GEBSER, M., INOUE, K., OSTROWSKI, M., PEANO, A., SCHAUB, T., SOH, T., TAMURA, N., AND WEISE, M. 2015. aspartame: Solving constraint satisfaction problems with answer set programming. In *Logic Programming and Nonmonotonic Reasoning*, F. Calimeri, G. Ianni, and M. Truszczynski, Eds. Springer International Publishing, Cham, 112–126.

BANBARA, M., KAUFMANN, B., OSTROWSKI, M., AND SCHAUB, T. 2017. Clingcon: The next generation. *Theory and Practice of Logic Programming (TPLP) 17,* 4, 408–461.

BARRETT, C., CONWAY, C. L., DETERS, M., HADAREAN, L., JOVANOVIĆ, D., KING, T., REYNOLDS, A., AND TINELLI, C. 2011. Cvc4. In *Proceedings of the 23rd International Conference on Computer Aided Verification (CAV'11), volume 6806 of LNCS*. Springer.

BARRETT, C., FONTAINE, P., AND TINELLI, C. 2015. The SMT-LIB Standard: Version 2.5. Tech. rep., Department of Computer Science, The University of Iowa. Available at `www.SMT-LIB.org`.

BARRETT, C., SEBASTIANI, R., SESHIA, S. A., AND TINELLI, C. 2008. Satisfiability modulo theories. In *Handbook of Satisfiability*, A. Biere, M. Heule, H. van Maaren, and T. Walsch, Eds. IOS Press, 737–797.

BARRETT, C. AND TINELLI, C. 2014. Satisfiability modulo theories. In *Handbook of Model Checking*, E. Clarke, T. Henzinger, and H. Veith, Eds. Springer.

BARTHOLOMEW, M. 2016. Answer set programming modulo theories. Ph.D. thesis, Arizona State University.

BARTHOLOMEW, M. AND LEE, J. 2013. Functional stable model semantics and answer set programming modulo theories. In *Proceedings of International Joint Conference on Artificial Intelligence (IJCAI)*. 718—-724.

BARTHOLOMEW, M. AND LEE, J. 2014. *Logics in Artificial Intelligence: 14th European Conference, JELIA 2014, Funchal, Madeira, Portugal, September 24-26, 2014. Proceedings*. Springer International Publishing, Cham, Chapter System aspmt2smt: Computing ASPMT Theories by SMT Solvers, 529–542.

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

BIERE, A., CIMATTI, A., CLARKE, E. M., STRICHMAN, O., AND ZHU, Y. 2003. Bounded model checking. *Advances in Computers 58*, 117–148.

BOMANSON, J., GEBSER, M., JANHUNEN, T., KAUFMANN, B., AND SCHAUB, T. 2016. Answer set programming modulo acyclicity. *Fundamenta Informaticae 147*, 63–91.

BOMANSON, J., JANHUNEN, T., AND NIEMELÄ, I. 2020. Applying visible strong equivalence in answer-set program transformations. *ACM Trans. Comput. Log. 21,* 4, 33:1–33:41.

BRAIN, M., ERDEM, E., INOUE, K., OETSCH, J., PUEHRER, J., TOMPITS, H., AND YILMAZ, C. 2012. Event-sequence testing using answer-set programming. *International Journal on Advances in Software 5,* 3&4.

BREWKA, G., EITER, T., AND TRUSZCZYŃSKI, M. 2011. Answer set programming at a glance. *Communications of the ACM 54(12)*, 92–103.

BROMBERGER, M., STURM, T., AND WEIDENBACH, C. 2015. Linear integer arithmetic revisited. In *Automated Deduction - CADE-25*, A. P. Felty and A. Middeldorp, Eds. Springer International Publishing, Cham, 623–637.

CABALAR, P., FANDINNO, J., AND LIERLER, Y. 2020. Modular answer set programming as a formal specification language. *Theory and Practice of Logic Programming 20,* 5, 767–782.

CALIMERI, F., COZZA, S., AND IANNI, G. 2007. External sources of knowledge and value invention in logic programming. *Annals of Mathematics and Artificial Intelligence 50,* 3-4, 333–361.

CALIMERI, F., COZZA, S., IANNI, G., AND LEONE, N. 2008. Computable functions in ASP: theory and implementation. In *Proceedings of International Conference on Logic Programming (ICLP)*. 407–424.

CALIMERI, F., FABER, W., GEBSER, M., IANNI, G., KAMINSKI, R., KRENNWALLNER, T., LEONE, N., MARATEA, M., RICCA, F., AND SCHAUB, T. 2019. ASP-core-2 input language format. *Theory and Practice of Logic Programming 20,* 2 (Dec), 294–309.

CALIMERI, F., GEBSER, M., MARATEA, M., AND RICCA, F. 2016. Design and results of the fifth answer set programming competition. *Artificial Intelligence 231*, 151 – 181.

CALIMERI, F., IANNI, G., RICCA, F., ALVIANO, M., BRIA, A., CATALANO, G., COZZA, S., FABER, W., FEBBRARO, O., LEONE, N., MANNA, M., MARTELLO, A., PANETTA, C., PERRI, S., REALE, K., SANTORO, M. C., SIRIANNI, M., TERRACINA, G., AND VELTRI, P. 2011. 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)*. Springer-Verlag, Berlin, Heidelberg, 388–403.

CARLSSON, M. AND FRUEHWIRTH, T. 2014. *SICStus PROLOG User's Manual 4.3*. Books On Demand - Proquest.

CLARK, K. 1978. Negation as failure. In *Logic and Data Bases*, H. Gallaire and J. Minker, Eds. Plenum Press, New York, 293–322.

COHEN, M. B., DWYER, M. B., AND SHI, J. 2008. Constructing interaction test suites for highly-configurable systems in the presence of constraints: A greedy approach. *IEEE Transactions on Software Engineering 34,* 5, 633–650.

DE MOURA, L. AND BJØRNER, N. 2008. Z3: An efficient SMT solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. 337–340.

DENECKER, M., LIERLER, Y., TRUSZCZYNSKI, M., AND VENNEKENS, J. 2019. The informal semantics of answer set programming: A tarskian perspective.

DOVIER, A., FORMISANO, A., AND PONTELLI, E. 2009. An empirical study of constraint logic programming and answer set programming solutions of combinatorial problems. *Journal of Experimental & Theoretical Artificial Intelligence 21,* 2, 79–121.

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

DRESCHER, C. AND WALSH, T. 2011. Translation-based constraint answer set solving. In *Proceedings of the 22nd International Joint Conference on Artificial Intelligence (IJCAI)*. AAAI Press, 2596–2601.

DUTERTRE, B. 2017. yices. `http://yices.csl.sri.com/`. [Accessed: 2018].

EITER, T., IANNI, G., SCHINDLAUER, R., AND TOMPITS, H. 2005. A uniform integration of higher-order reasoning and external evaluations in answer set programming. In *Proceedings of International Joint Conference on Artificial Intelligence (IJCAI)*. Professional Book Center, 90–96.

ELKABANI, I., PONTELLI, E., AND SON, T. C. 2004. Smodels with clp and its applications: A simple and effective approach to aggregates in ASP. In *ICLP*, B. Demoen and V. Lifschitz, Eds. Lecture Notes in Computer Science, vol. 3132. Springer-Verlag, 73–89.

ERDOĞAN, S. AND LIFSCHITZ, V. 2004. Definitions in answer set programming. In *Proceedings of International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR)*. 114–126.

FAGES, F. 1994. Consistency of Clark's completion and existence of stable models. *Journal of Methods of Logic in Computer Science 1*, 51–60.

FANDINNO, J., LIFSCHITZ, V., LÜHNE, P., AND SCHAUB, T. 2020. Verifying tight logic programs with Anthem and Vampire. *Theory Pract. Log. Program. 20,* 5, 735–750.

FERRARIS, P. AND LIFSCHITZ, V. 2005. Weight constraints as nested expressions. *Theory and Practice of Logic Programming 5*, 45–74.

GARVIN, B. J., COHEN, M. B., AND DWYER, M. B. 2011. Evaluating improvements to a meta-heuristic search for constrained interaction testing. *Empirical Software Engineering 16,* 1, 61–102.

GEBSER, M., KAMINSKI, R., KAUFMANN, B., OSTROWSKI, M., SCHAUB, T., AND WANKO, P. 2016. Theory Solving Made Easy with Clingo 5. In *Technical Communications of the 32nd International Conference on Logic Programming (ICLP 2016)*, M. Carro, A. King, N. Saeedloei, and M. D. Vos, Eds. OpenAccess Series in Informatics (OASIcs), vol. 52. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 2:1–2:15.

GEBSER, M., KAMINSKI, R., KAUFMANN, B., AND SCHAUB, T. 2009. *On the Implementation of Weight Constraint Rules in Conflict-Driven ASP Solvers*. Springer Berlin Heidelberg, Berlin, Heidelberg, 250–264.

GEBSER, M., KAMINSKI, R., KÖNIG, A., AND SCHAUB, T. 2011. Advances in gringo series 3. In *Proceedings of International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR)*. Springer, 345–351.

GEBSER, M., KAUFMANN, B., NEUMANN, A., AND SCHAUB, T. 2007. Conflict-driven answer set enumeration. In *Proceedings of the 9th International Conference on Logic Programming and Nonmonotonic Reasoning*. LPNMR'07. Springer-Verlag, Berlin, Heidelberg, 136–148.

GEBSER, M., KÖNIG, A., SCHAUB, T., THIELE, S., AND VEBER, P. 2010. The bioASP library: ASP solutions for systems biology. In *22nd IEEE International Conference on Tools with Artificial Intelligence, ICTAI 2010*. IEEE Computer Society, 383–389.

GEBSER, M., LEE, J., AND LIERLER, Y. 2011. On elementary loops of logic programs. *Theory and Practice of Logic Programming 11(6)*, 953–988.

GEBSER, M., OSTROWSKI, M., AND SCHAUB, T. 2009. Constraint answer set solving. In *Proceedings of 25th International Conference on Logic Programming (ICLP)*. Springer-Verlag, 235–249.

GEBSER, M., SCHAUB, T., AND THIELE, S. 2007. Gringo: A new grounder for answer set programming. In *Proceedings of the Ninth International Conference on Logic Programming and Nonmonotonic Reasoning*. 266–271.

GOMES, C. P., KAUTZ, H., SABHARWAL, A., AND SELMAN, B. 2008. Satisfiability solvers. In *Handbook of Knowledge Representation*, F. van Harmelen, V. Lifschitz, and B. Porter, Eds. Elsevier, 89–134.

GUTIN, G. AND PUNNEN, A., Eds. 2007. *The Traveling Salesman Problem and Its Variations*. Springer-Verlag.

IBM 2009. *IBM ILOG AMPL Version 12.1 User's Guide*. IBM. `http://www.ibm.com/software/commerce/optimization/cplex-optimizer/`.

JAFFAR, J. AND MAHER, M. 1994. Constraint logic programming: A survey. *Journal of Logic Programming 19(20)*, 503–581.

JANHUNEN, T. 2006. Some (in)translatability results for normal logic programs and propositional theories. *Journal of Applied Non-Classical Logics*, 35–86.

JANHUNEN, T., KAMINSKI, R., OSTROWSKI, M., SCHELLHORN, S., WANKO, P., AND SCHAUB, T. 2017. Clingo goes linear constraints over reals and integers. *Theory Pract. Log. Program. 17,* 5-6, 872–888.

JANHUNEN, T., LIU, G., AND NIEMELÄ, I. 2011. Tight integration of non-ground answer set programming and satisfiability modulo theories. In *Working notes of the 1st Workshop on Grounding and Transformations for Theories with Variables*.

JANHUNEN, T., NIEMELÄ, I., AND SEVALNEV, M. 2009. Computing stable models via reductions to difference logic. In *Logic Programming and Nonmonotonic Reasoning*. Springer Berlin Heidelberg, 142–154.

KAUTZ, H. AND SELMAN, B. 1992. Planning as satisfiability. In *Proceedings of European Conference on Artificial Intelligence (ECAI)*. 359–363.

KOWALSKI, R. A. 1974. Predicate logic as programming language. In *Proceedings of International Federation of Information Processing Conference*, J. L. Rosenfeld, Ed. North–Holland, Stockholm, Sweden, 569–574.

KOWALSKI, R. A. 1988. The early years of logic programming. *Commun. ACM 31,* 1 (Jan.), 38–43.

LAWLER, E. L., LENSTRA, J. K., KAN, A. H. G. R., AND SHMOYS, D. B., Eds. 1985. *The Traveling Salesman Problem: A Guided Tour of Combinatorial Optimization*. Wiley.

LEE, J. AND MENG, Y. 2013. Answer set programming modulo theories and reasoning about continuous changes. In *Proceedings of the 23rd International Joint Conference on Artificial Intelligence (IJCAI-13), Beijing, China, August 3-9, 2013*.

LIERLER, Y. 2010. SAT-based answer set programming. Ph.D. thesis, University of Texas at Austin.

LIERLER, Y. 2014. Relating constraint answer set programming languages and algorithms. *Artificial Intelligence 207C*, 1–22.

LIERLER, Y. 2017. What is answer set programming to propositional satisfiability. *Constraints 22*, 307–337.

LIERLER, Y., SMITH, S., TRUSZCZYŃSKI, M., AND WESTLUND, A. 2012. Weighted-sequence problem: ASP vs cASP and declarative vs problem oriented solving. In *Proceedings of the 14th International Symposium on Practical Aspects of Declarative Languages, PADL 2012*, C. V. Russo and N.-F. Zhou, Eds. Lecture Notes in Computer Science, vol. 7149. Springer-Verlag, 63–77.

LIERLER, Y. AND SUSMAN, B. 2017. On relation between constraint answer set programming and satisfiability modulo theories. *Theory and Practice of Logic Programming 17,* 4, 559–590.

LIERLER, Y. AND TRUSZCZYŃSKI, M. 2011. Transition systems for model generators — a unifying approach. *Theory and Practice of Logic Programming, 27th Int'l. Conference on Logic Programming (ICLP) Special Issue 11,* 4-5, 629–646.

LIFSCHITZ, V. 2002. Answer set programming and plan generation. *Artificial Intelligence 138*, 39–54.

LIFSCHITZ, V. 2017. Achievements in answer set programming. *TPLP 17,* 5-6, 961–973.

LIFSCHITZ, V., TANG, L. R., AND TURNER, H. 1999. Nested expressions in logic programs. *Annals of Mathematics and Artificial Intelligence 25*, 369–389.

LIN, F. AND WANG, Y. 2008. Answer set programming with functions. In *Proceedings of International Conference on Principles of Knowledge Representation and Reasoning (KR)*. 454–465.

LIU, G., JANHUNEN, T., AND NIEMELÄ, I. 2012. Answer set programming via mixed integer programming. In *Principles of Knowledge Representation and Reasoning: Proceedings of the 13th International Conference*. AAAI Press, 32–42.

MAREK, V. AND TRUSZCZYŃSKI, M. 1999. Stable models and an alternative logic programming paradigm. In *The Logic Programming Paradigm: a 25-Year Perspective*. Springer Verlag, 375–398.

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

NETHERCOTE, N., STUCKEY, P., BECKET, R., BRAND, S., DUCK, G., , AND TACK, G. 2007. MiniZinc: Towards a standard cp modelling language. In *Proceedings of the 13th International Conference on Principles and Practice of Constraint Programming*. 529–543.

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

NIEMELÄ, I. 2008. Stable models and difference logic. *Annals of Mathematics and Artificial Intelligence 53*, 313–329.

NIEMELÄ, I. AND SIMONS, P. 2000. Extending the Smodels system with cardinality and weight constraints. In *Logic-Based Artificial Intelligence*, J. Minker, Ed. Kluwer, 491–521.

NIEUWENHUIS, R. AND OLIVERAS, A. 2005. DPLL(T) with exhaustive theory propagation and its application to difference logic. In *Proceedings of the 17th International Conference on Computer Aided Verification (CAV'05), volume 3576 of LNCS*. Springer.

NIEUWENHUIS, R., OLIVERAS, A., AND TINELLI, C. 2006. 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.

NUDELMAN, E., DEVKAR, A., SHOHAM, Y., LEYTON-BROWN, K., AND HOOS, H. 2004. SATzilla: An algorithm portfolio for SAT. In *SAT Competition*.

OSTROWSKI, M. 2018. Modern constraint answer set solving. Ph.D. thesis, University of Potsdam.

PALU, A. D., DOVIER, A., AND PONTELLI, E. 2010. Computing approximate solutions of the protein structure determination problem using global constraints on discrete crystal lattices. *Int. J. Data Min. Bioinformatics 4,* 1 (Jan.), 1–20.

PALÙ, R. D., DOVIER, A., AND FOGOLARI, F. 2004. Constraint logic programming approach to protein structure prediction. *BMC Bioinformatics 5*, 2004.

PRASAD, M. R., BIERE, A., AND GUPTA, A. 2005. A survey of recent advances in SAT-based formal verification. *STTT 7,* 2, 156–173.

RICCA, F., GRASSO, G., ALVIANO, M., MANNA, M., LIO, V., IIRITANO, S., AND LEONE, N. 2012. Team-building with answer set programming in the gioia-tauro seaport. *Theory and Practice of Logic Programming 12,* 3, 361–381.

RINTANEN, J. 2012. Planning as satisfiability: Heuristics. *Artificial Intelligence 193*, 45–86.

ROBINSON, N., GRETTON, C., PHAM, D.-N., AND SATTAR, A. 2010. Cost-optimal planning using weighted MaxSAT. In *ICAPS 2010 Workshop on Constraint Satisfaction Techniques for Planning and Scheduling (COPLAS10)*.

ROSIS, A. F. D., EITER, T., REDL, C., AND RICCA, F. 2015. Constraint answer set programming based on hex-programs.

ROSSI, F., VAN BEEK, P., AND WALSH, T. 2008. Constraint programming. In *Handbook of Knowledge Representation*, F. van Harmelen, V. Lifschitz, and B. Porter, Eds. Elsevier, 181–212.

SHEN, D. AND LIERLER, Y. 2018a. SMT-based answer set solver CMODELS-DIFF (system description). In *Proceedings of the 34th International Conference on Logic Programming (ICLP)*.

SHEN, D. AND LIERLER, Y. 2018b. SMT-based constraint answer set solver EZSMT+ for non-tight programs. In *Proceedings of the 16th International Conference on Principles of Knowledge Representation and Reasoning (KR)*.

SUSMAN, B. AND LIERLER, Y. 2016. SMT-based constraint answer set solver EZSMT (system description). In *Proceedings of 32th International Conference on Logic Programming (ICLP)*. Dagstuhl Publishing, OpenAccess Series in Informatics (OASIcs).

TINELLI, C. AND BARRETT, C. 2015. AUFLIRA. `http://smtlib.cs.uiowa.edu/logics-all.shtml#AUFLIRA`. [Accessed: 2018].

WIELEMAKER, J., SCHRIJVERS, T., TRISKA, M., AND LAGER, T. 2012. SWI-Prolog. *Theory and Practice of Logic Programming 12,* 1-2, 67–96.

WINTERSTEIGER, C. M., BJØRNER, N., AND DE MOURA, L. 2016. Z3. `https://github.com/Z3Prover/z3/releases/tag/z3-4.5.0`. [Accessed: 2018].

WITTOCX, J., MARIËN, M., AND DENECKER, M. 2008. 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)*. electronic, 153–165. available at `https://lirias.kuleuven.be/bitstream/123456789/229814/1/lash08.pdf`.

YOUNG, R., BALDUCCINI, M., AND ISRANEY, A. 2017. CASP for robot control in hybrid domains. In *Workshop on Answer Set Programming and Other Computing Paradigms (ASPOCP17)*.

ZHOU, N. 2012. The language features and architecture of B-Prolog. *Theory and Practice of Logic Programming 12,* 1-2 (Jan.), 189–218.