5-2016

The EZSMT Solver: Constraint Answer Set Solving meets SMT

Benjamin Susman
University of Nebraska at Omaha, bsusman@unomaha.edu

The EZSMT Solver: Constraint Answer Set Solving meets SMT

A Thesis

Presented to the

Department of Computer Science

and the

Faculty of the Graduate College

University of Nebraska

In Partial Fulfillment
of the Requirements for the Degree

Master of Science

University of Nebraska at Omaha

by

Benjamin Susman

May 2016

Supervisory Committee:

Dr. Yuliya Lierler

Dr. Abhishek Parakh

Dr. Mahadevan Subramaniam

The EZSMT System: Constraint Answer Set Solving meets SMT[1]

Benjamin Susman, M.S.
University of Nebraska, 2016

Advisor: Dr. Yuliya Lierler

Constraint answer set programming is a promising research direction that integrates answer set programming with constraint processing. It is often informally related to the field of Satisfiability Modulo Theories. Yet, the exact formal link is obscured as the terminology and concepts used in these two research areas differ. In this thesis, by connecting these two areas, we begin the cross-fertilization of not only of the theoretical foundations of both areas but also of the existing solving technologies. We present the system EZSMT, one of the first solvers of this nature, which is able to take a large class of constraint answer set programs and rewrite them into Satisfiability Modulo Theories programs so that Satisfiability Modulo Theories technology can be used to process these programs.

[1]Parts of this thesis were presented at the 3rd Workshop on Grounding, Transforming, and Modularizng Theories with Variables (GTTV'15) [Lierler and Susman, 2015] and is to be presented at the 25th International Joint Conference on Artificial Intelligence (IJCAI-16).

This thesis is dedicated to my family for their love and support.

Acknowledgements

I have had the privilege of being advised by Yuliya Lierler, whom has helped guide me through my studies at the University of Nebraska at Omaha. It has been a pleasure to work with Yuliya in many capacities, as my advisor, my teacher, my colleague, and as my friend. I will always hold dear both our research and life discussions.

I would like to thank Marcello Balduccini and Martin Brain for their discussions that have made this work possible.

I am thankful to the other thesis committee members: Abhishek Parakh and Mahadevan Subramaniam, for their helpful comments and for serving on my committee.

The University of Nebraska at Omaha (UNO) has become somewhat of a second home during my studies. The university and its faculty have made it possible for me to conduct this research through both financial and intellectual support. I am happy to had opportunities to work closely with many of UNO's students and faculty, which have provided inspiration to me throughout my studies.

I am glad to have been able to work closely with past and current members of UNO's Natural Language Processing and Knowledge Representation (NLPKR) group: Daniel Bailey, Matt Buddenhagen, Maeghan Fry, Joshua Irvin, and Nicholas Richman.

Finally, I would like to thank my friends and family. Without their lifetime of support, I would not be who I am today. Thank you for challenging me and encouraging me.

Table of Contents

List of Figures

List of Tables

Listings

1 Introduction

Constraint answer set programming (CASP) [Mellarkod *et al.*, 2008; Gebser *et al.*, 2009; Balduccini, 2009; Lierler, 2014] is a promising research direction that integrates answer set programming, a powerful knowledge representation paradigm, with constraint processing. Typical answer set programing tools start their computation with grounding, a process that substitutes variables for passing constants in respective domains. Large domains often form an obstacle for classical answer set programming. CASP enables a mechanism to model constraints over large domains so that they are processed in a non-typical way for answer set programming tools by delegating their solving to constraint solver systems specifically designed to handle large and sometimes infinite-domains. CASP solvers including CLINGCON [Gebser *et al.*, 2009] and EZCSP [Balduccini, 2009] already put CASP on the map of efficient automated reasoning tools.

Motivation Constraint answer set programming often cites itself as a related initiative to Satisfiability Modulo Theories (SMT) solving [Barrett and Tinelli, 2014]. Yet, the exact link is obscured as the terminology and concepts used in both fields differ. To add to the complexity of the picture several answer set programming modulo theories formalisms have been proposed. For instance, Liu et al. (2012), Janhunen et al. (2011), and Lee and Meng (2013) introduced logic programs modulo linear constraints, logic programs modulo difference constraints, and ASPMT programs respectively. Alongside these formalisms, there have been systems developed for either CASP or SMT solving, but there is no way to readily compare these systems' performance on equivalent programs.

Contributions This thesis attempts to unify the terminology used in CASP and SMT so that the differences and similarities of logic programs with constraints versus logic programs modulo theories become apparent. At the same time, we introduce the notion of constraint formulas, which is similar to that of logic programs with constraints. We identify a special class of SMT theories that we call "uniform". Commonly used theories in satisfiability

modulo solving such as integer linear, difference logic, and linear arithmetics belong to uniform theories. This class of theories helps us to establish precise links (i) between CASP and SMT, and (ii) between constraint formulas and SMT programs. We are able to then provide a formal description relating a family of distinct constraint answer set programming formalisms. From this description, we introduce a state-of-the-art system, EZSMT, to demonstrate the practical merits of this approach and compare it to existing CASP solvers.

We show that this unified outlook allows us not only to better understand the landscape of CASP languages and systems, but also to foster new ideas for CASP solvers design as well as SMT solvers design. For example, theoretical results of this work establish the method for using SMT systems for computing answer sets of a broad class of "tight" constraint answer set programs, which the EZSMT solver is capable of demonstrating via a complete solving pipeline. Similarly, CASP technology can be used to solve certain classes of SMT problems.

Related Work To the best of our knowledge this is the first attempt to formally relate the CASP and SMT formalisms, which is the main theoretical contribution of the thesis. The CASP solver EZSMT, the main software product of this work, is inspired by earlier solvers of this kind including systems CLINGCON [Gebser *et al.*, 2009] and EZCSP [Balduccini, 2009], MINGO [Liu *et al.*, 2012], DINGO [Janhunen *et al.*, 2011], ASPMT2SMT [Bartholomew and Lee, 2014].

Thesis Outline The outline of the thesis is as follows. We start by reviewing concepts of logic programs, completion, and (input) answer sets. We then present (i) generalized constraint satisfaction problems, (ii) constraint answer set programs, and (iii) constraint formulas. Next, we introduce satisfiability modulo theories and respective SMT programs. We define a class of uniform theories and establish links between CASP and SMT. We continue by relating a family of distinct constraint answer set programming formalisms. We then exhibit the architecture of the EZSMT system and its connection to the theoretical grounding.

Finally, we conclude by presenting experimental results comparing the performance of EZSMT to the CLINGCON and EZCSP solvers, fine representatives of CASP systems.

2 Preliminaries

This section starts by reviewing logic programs and the concept of answer set. It also introduces programs' completion. Next, the generalized constraint satisfaction problems are introduced and related to the classical constraint satisfaction problems studied in Artificial Intelligence.

2.1 Logic Programs and Completion

Syntax We begin by introducing logic programs and their syntax. 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

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

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

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

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*. We identify rule (1) with the propositional formula

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

and B with the propositional formula

$$b_1 \wedge \ldots \wedge b_\ell \wedge \neg b_{\ell+1} \wedge \ldots \wedge \neg b_m \wedge \neg\neg b_{m+1} \wedge \ldots \wedge \neg\neg b_n. \tag{4}$$

Note that (i) the order of terms in (4) is immaterial, (ii) *not* is replaced with classical negation ($\neg$), and (iii) comma is replaced by conjunction ($\wedge$). Expression $b_1 \wedge \ldots \wedge b_\ell$ in formula (4) is referred to as the *positive* part of the body and the remainder of (4) as the *negative* part of the body.

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 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 a set of facts $\{a. \mid a \in X\}$.

Semantics Now that we have introduced the syntax of logic programs, we can discuss their semantics. We say a set X *satisfies* a rule, if X satisfies the propositional formula (3). We say X satisfies a program Π, if X satisfies every rule in Π.

The *reduct* Π^X of a program Π relative to a set X of atoms is obtained by first removing all rules (1) such that X does not satisfy $\neg b_{\ell+1} \wedge \ldots \wedge \neg b_m \wedge \neg\neg b_{m+1} \wedge \ldots \wedge \neg\neg b_n$, and replacing all remaining rules with $a \leftarrow b_1, \ldots, b_\ell$. 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].

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 $\sigma \setminus X$.

Ferraris and Lifschitz (2005) showed that a choice rule $\{a\} \leftarrow B$[2] can be seen as an abbreviation for a rule $a \leftarrow \ not\ not\ a, B$. We adopt this abbreviation in the rest of the thesis.

[2]Choice rules were introduced in [Niemelä and Simons, 2000] and are commonly used in answer set programming languages.

Example 1. Consider the logic program taken verbatim from Balduccini and Lierler (2016):

$$\begin{aligned} &\{switch\}. \\ &lightOn \leftarrow\ switch, not\ am. \\ &\leftarrow not\ lightOn. \\ &\{am\}. \end{aligned} \tag{5}$$

Each rule in the program can be understood as follows:

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

This program's only answer set is $\{switch,\ lightOn\}$. This 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. □

Completion For a program Π over vocabulary σ, the *completion* of Π [Clark, 1978], denoted by $Comp(\Pi)$, is the set of classical formulas that consists of the implications

$$B \rightarrow a \tag{6}$$

for all rules (2) in Π and the implications

$$a \rightarrow \bigvee_{a \leftarrow B \in \Pi} B \tag{7}$$

for all atoms a in σ.

When set $Bodies(\Pi, a)$ is empty, the implication (7) has the form $a \rightarrow \bot$. When a rule (2) is a fact $a.$, then we identify the implication $B \rightarrow a$ with the unit clause a.

Example 2. The completion of logic program (5) consists of formulas

$$\begin{aligned} &\neg\neg switch \rightarrow switch \\ &switch \wedge \neg am \rightarrow lightOn \\ &\neg lightOn \rightarrow \bot \\ &\neg\neg am \rightarrow am \end{aligned}$$

and

$$\begin{aligned} &switch \rightarrow \neg\neg switch \\ &lightOn \rightarrow switch \wedge \neg am \\ &am \rightarrow \neg\neg am \end{aligned} \tag{8}$$

It is easy to see that this completion is equivalent to the set of formulas

$$\begin{aligned} &\neg switch \vee switch \\ &lightOn \leftrightarrow switch \wedge \neg am \\ &lightOn \\ &\neg am \vee am. \end{aligned} \tag{9}$$

The set $\{switch, lightOn\}$ is the only model of (9). Note that this set coincides with the answer set found for program (5). □

Tightness For the large class of logic programs, called *tight*, their answer sets coincide with models of their completion [Fages, 1994; Erdem and Lifschitz, 2001]. Tightness is a syntactic condition on a program that can be verified by means of program's dependency graph. The *dependency graph* of Π is the directed graph G such that (i) the vertices of G are the atoms occurring in Π, and (ii) for every rule (1) in Π whose head is not $\bot$, G has an edge from atom a to each atom $b_1, \ldots, b_\ell$. A program is called *tight* if its dependency graph is acyclic.

2.2 Generalized Constraint Satisfaction Problems

We start this section by presenting primitive constraints as introduced by Marriott and Stuckey (1998, Section 1.1) using the notation convenient for our purposes. We refer to this concept as a constraint dropping the word "primitive". We use constraints to define a notion of a generalized constraint satisfaction problem that Marriott and Stuckey refer to as "constraint". We then review constraint satisfaction problems as commonly defined in artificial intelligence literature and illustrate that they are special case of generalized constraint satisfaction problems.

2.2.1 Constraints and Generalized Constraint Satisfaction Problem

Signature, c-vocabulary, constraint atoms We adopt the following convention: for a function ν and an element x, by x^ν we denote the value that function ν maps x to (in other words, $x^\nu = \nu(x)$). A *domain* is a *nonempty* set of elements (values). A *signature* Σ is a set of *variables*, *predicate symbols*, and *function symbols (or f-symbols)*. Predicate and function symbols are associated with a positive integer called *arity*. By $\Sigma_{|v}$, $\Sigma_{|r}$, and $\Sigma_{|f}$ we denote the subsets of Σ that contain all variables, all predicate symbols, and all f-symbols respectively.

For instance, we can define signature $\Sigma_1 = \{s, r, E, Q\}$ by saying that s and r are variables, E is a predicate symbol of arity 1, and Q is a predicate symbol of arity 2. Then, $\Sigma_{1|v} = \{s, r\}$, $\Sigma_{1|r} = \{E, Q\}$, $\Sigma_{1|f} = \emptyset$.

Let D be a domain. For a set V of variables, we call a function $\nu : V \to D$ a $[V, D]$ *valuation*. For a set R of predicate symbols, we call a total function on R *an* $[R, D]$ *r-denotation*, when it maps an n-ary predicate symbol into an n-ary relation on D. For a set F of f-symbols, we call a function on F *an* $[F, D]$ *f-denotation*, when it maps an n-ary f-symbol into a function $D^n \to D$[3].

[3]When a signature contains no function symbols no reference to f-denotation is necessary.

Table 1 presents definitions of sample domain D_1, valuations ν_1, ν_2, and r-denotations ρ_1 and ρ_2.

Σ_1	$\{s, r, E, Q\}$
D_1	$\{1, 2, 3\}$
ν_1	$[\Sigma_{1\|v}, D_1]$ valuation, where $s^{\nu_1} = r^{\nu_1} = 1$
ν_2	$[\Sigma_{1\|v}, D_1]$ valuation, where $s^{\nu_2} = 2$ and $r^{\nu_2} = 1$
ρ_1	$[\Sigma_{1\|r}, D_1]$ r-denotation, where $E^{\rho_1} = \{\langle 1\rangle\},\ Q^{\rho_1} = \{\langle 1,1\rangle, \langle 2,2\rangle, \langle 3,3\rangle\}$
ρ_2	$[\Sigma_{1\|r}, D_1]$ r-denotation, where $E^{\rho_2} = \{\langle 2\rangle, \langle 3\rangle\}, Q^{\rho_2} = Q^{\rho_1}$.

Table 1: Example definitions for signature, valuation, and r-denotation

A *constraint vocabulary (c-vocabulary)* is a pair $[\Sigma, D]$, where Σ is a signature and D is a domain. A *term* over a c-vocabulary $[\Sigma, D]$ is either

- a variable in $\Sigma_{|v}$,
- a domain element in D, or
- an expression $f(t_1, \ldots, t_n)$, where f is an f-symbol of arity n in $\Sigma_{|f}$ and $t_1, \ldots, t_n$ are terms over $[\Sigma, D]$.

A *constraint atom* over a c-vocabulary $[\Sigma, D]$ is an expression

$$P(t_1, \ldots, t_n), \tag{10}$$

where P is a predicate symbol from $\Sigma_{|r}$ of arity n and $t_1, \ldots, t_n$ are terms over $[\Sigma, D]$. A *constraint literal* over a c-vocabulary $[\Sigma, D]$ is either a constraint atom (10) or an expression

$$\neg P(t_1, \ldots, t_n), \tag{11}$$

where $P(t_1, \ldots, t_n)$ is a constraint atom over $[\Sigma, D]$. For instance, expressions $\neg E(s)$, $\neg E(2)$, and $Q(r, s)$ are constraint literals over $[\Sigma_1, D_1]$.

Let $[\Sigma, D]$ be a c-vocabulary, ν be a $[\Sigma_{|v}, D]$ valuation, ρ be a $[\Sigma_{|r}, D]$ r-denotation, and ϕ be a $[\Sigma_{|f}, D]$ f-denotation. First, we define recursively a value that valuation ν assigns

to a term τ over $[\Sigma, D]$ w.r.t. ϕ. We denote this value by $\tau^{\nu,\phi}$. We define the valuations for variables, domain elements and f-symbols as follows.

- For a term that is a variable x in $\Sigma_{|v}$, $x^{\nu,\phi} = x^{\nu}$.
- For a term that is a domain element d in D, $d^{\nu,\phi}$ is d itself.
- For a term τ of the form $f(t_1, \ldots, t_n)$, $\tau^{\nu,\phi}$ is defined recursively by the formula

$$f(t_1, \ldots, t_n)^{\nu,\phi} = f^{\phi}(t_1^{\nu,\phi}, \ldots, t_n^{\nu,\phi}).$$

Second, we define what it means for valuation to be a solution of a constraint literal w.r.t. given r- and f-denotations. We say that ν *satisfies (is a solution to)* constraint literal (10) over $[\Sigma, D]$ *w.r.t.* ρ *and* ϕ when $\langle t_1^{\nu,\phi}, \ldots, t_n^{\nu,\phi} \rangle \in P^{\rho}$. Let $\mathcal{R}$ be an n-ary relation on D. By $\overline{\mathcal{R}}$ we denote *complement* relation of $\mathcal{R}$ constructed as $D^n \setminus \mathcal{R}$. Valuation ν *satisfies (is a solution to)* constraint literal of the form (11) *w.r.t.* ρ *and* ϕ when $\langle t_1^{\nu,\phi}, \ldots, t_n^{\nu,\phi} \rangle \in \overline{P^{\rho}}$.

For instance, valuation ν_1 satisfies constraint literal $Q(r, s)$ w.r.t. ρ_1 (defined in Table 1), while valuation ν_2 does not satisfy this constraint literal w.r.t. ρ_2.

Lexicon, constraints, generalized constraint satisfaction problem We are now ready to define constraints, their syntax and semantics. To begin we introduce a *lexicon*, which is a tuple $([\Sigma, D], \rho, \phi)$, where $[\Sigma, D]$ is a c-vocabulary, ρ is $[\Sigma_{|r}, D]$ r-denotation, and ϕ is $[\Sigma_{|f}, D]$ f-denotation. For a lexicon $\mathcal{L} = ([\Sigma, D], \rho, \phi)$, we call any function that is $[\Sigma_{|v}, D]$ valuation, a *valuation over* $\mathcal{L}$. We omit the last element of the tuple if the signature Σ of the lexicon contains no f-symbols. A *constraint* is defined over lexicon $\mathcal{L} = ([\Sigma, D], \rho, \phi)$. Syntactically, it is a constraint literal over $[\Sigma, D]$ (lexicon $\mathcal{L}$, respectively). Semantically, we say that valuation ν over $\mathcal{L}$ *satisfies (is a solution to)* the constraint c when ν satisfies c w.r.t. ρ and ϕ. For instance, Table 2 presents definitions of sample lexicons $\mathcal{L}_1$, $\mathcal{L}_2$, and constraints c_1, c_2, c_3, and c_4.

Valuation ν_1 from Table 1 is a solution to c_1, c_2, c_3, but not a solution to c_4. Valuation ν_2 from Table 1 is not a solution to c_1, c_2, c_3, and c_4. In fact, constraint c_4 has no solutions. We

$\mathcal{L}_1$	$([\Sigma_1, D_1], \rho_1)$
$\mathcal{L}_2$	$([\Sigma_1, D_1], \rho_2)$
c_1	a literal $Q(r, s)$ over lexicon $\mathcal{L}_1$
c_2	a literal $Q(r, s)$ over lexicon $\mathcal{L}_2$
c_3	a literal $\neg E(s)$ over lexicon $\mathcal{L}_2$
c_4	a literal $\neg E(2)$ over lexicon $\mathcal{L}_2$.

Table 2: Sample lexicons and constraints

sometimes omit the explicit mention of the lexicon when talking about constraints: we then may identify a constraint with its syntactic form of a constraint literal.

Definition 1. A *generalized constraint satisfaction problem (GCSP)* $\mathcal{C}$ is a finite set of constraints over a lexicon $\mathcal{L} = ([\Sigma, D], \rho, \phi)$. We say that a valuation ν over $\mathcal{L}$ *satisfies (is a solution to)* GCSP $\mathcal{C}$ when ν is a solution to every constraint in $\mathcal{C}$.

For example, any subset of set $\{c_2, c_3, c_4\}$ of constraints forms a GCSP. Sample valuation ν_1 over lexicon $\mathcal{L}_2$ (where ν_1 and $\mathcal{L}_2$ stem from Tables 1 and 2, respectively) satisfies the GCSP $\{c_2, c_3\}$, but does not satisfy the GCSP $\{c_2, c_3, c_4\}$.

2.2.2 From GCSP to Constraint Satisfaction Problem

In this section, we define a constraint satisfaction problem as customary in classical literature on Artificial Intelligence. We then show that it is a special case of generalized constraint satisfaction problems introduced previously.

We say that a lexicon is *finite-domain* if it is defined over a c-vocabulary that refers to a domain whose set of elements is finite. Trivially, lexicons $\mathcal{L}_1$ and $\mathcal{L}_2$ (defined in Table 2) are finite-domain. Consider a special case of a constraint of the form (10) over finite-domain lexicon $\mathcal{L} = ([\Sigma, D], \rho)$, so that each t_i is a variable. (For instance, constraints c_1, c_2, and c_3 satisfy the stated requirements, while c_4 does not.) In this case, we can syntactically identify (10) with the pair

$$\langle (t_1, \ldots, t_n), P^\rho \rangle. \tag{12}$$

A *constraint satisfaction problem* (CSP) is a set of pairs (12), where $\Sigma_{|v}$ and D of the finite-domain lexicon $\mathcal{L}$ are called variables and domain of CSP, respectively. Saying that valuation ν over $\mathcal{L}$ satisfies (10) is the same as saying that $\langle t_1^\nu, \ldots, t_n^\nu \rangle \in P^\rho$. The latter is the way in which a solution to expressions (12) in CSP is typically defined. As in the definition of semantics of GCSP, a valuation is a *solution* to a CSP problem C when it is a solution to every pair (12) in C.

In conclusion, GCSP generalizes CSP by

- elevating the restriction of finite-domain, and
- allowing us more elaborate syntactic expressions (e.g., recall f-symbols).

3 Constraint Answer Set Programs and Constraint Formulas

In this section we connect so called constraint answer set programs, rooting from the logic programs, to "constraint formulas", which are related to GCSP. First, we introduce input answer sets, followed by constraint answer set programs and input completion. Second, we present constraint formulas. Finally, we demonstrate the close connection between the constraint answer set programs and constraint formulas.

3.1 Input Answer Set

We start by introducing a generalization of the concept of an input answer set by Lierler and Truszczynski (2011). We consider input answer sets "relative to input vocabularies". We then extend the definition of completion and state the result by Erdem and Lifshitz (2001) for the case of input answer sets. The concept of an input answer set is essential for introducing constraint answer set programs. Constraint answer set programs (and constraint formulas) are defined over two disjoint vocabularies so that atoms stemming from those vocabularies "behave" differently. Input answer set semantics allows us to account for these differences.

Definition 2. For a logic program Π over vocabulary σ, a set X of atoms over σ is an

input answer set of Π relative to vocabulary $\iota \subseteq \sigma$ when X is an answer set of the program $\Pi \cup ((X \cap \iota) \setminus hd(\Pi))$.

3.2 Constraint Answer Set Program

Let σ_r and σ_i be two disjoint vocabularies. We refer to their elements as *regular* and *irregular* atoms respectively. For a program Π, by $At(\Pi)$ we denote the set of atoms occurring in it.

Definition 3. A *constraint answer set program* (CAS program) over the vocabulary $\sigma = \sigma_r \cup \sigma_i$ is a triple $\langle \Pi, \mathcal{B}, \gamma \rangle$, where Π is a logic program over the vocabulary σ such that $hd(\Pi) \cap \sigma_i = \emptyset$, $\mathcal{B}$ is a set of constraints over the same lexicon, and γ is an injective function from the set σ_i of irregular atoms to the set $\mathcal{B}$ of constraints.

For a CAS program $P = \langle \Pi, \mathcal{B}, \gamma \rangle$ over the vocabulary $\sigma = \sigma_r \cup \sigma_i$ so that $\mathcal{L}$ is the lexicon of the constraints in $\mathcal{B}$, a set $X \subseteq \sigma$ is an *answer set* of P if

- $X \subseteq At(\Pi)$
- X is an input answer set of Π relative to σ_i, and
- the following GCSP over $\mathcal{L}$ has a solution

$$\{\gamma(a) | a \in X \cap \sigma_i\} \cup \{\neg\gamma(a) | a \in (At(\Pi) \cap \sigma_i) \setminus X\}.$$

These definitions are generalizations of CAS programs introduced by Gebser et al. (2009) as they (i) refer to the concept of GCSP in place of CSP in the original definition, and (ii) allow for more general syntax of logic rules (e.g. choice rules are covered by the presented definition).

Example 3. We now consider the constraint answer set program adapted from Balduccini and Lierler (2016), defined under the CAS program syntax.

We first define Lexicon $\mathcal{L}_3$:

$$\begin{array}{ll} \Sigma_2 & \{x\} \\ D_2 & \{0, \ldots, 23\} \\ \rho_3 & [\{<, \geq\}, D_2] \\ \mathcal{L}_3 & ([\Sigma_2, D_2], \rho_3) \end{array} \tag{13}$$

where the r-denotation ρ_3 is defined with predicates symbols which take on their usual arithmetic meaning.

Second, we define a CAS program

$$P_1 = \langle \Pi_1, \mathcal{B}_1, \gamma_1 \rangle \tag{14}$$

over the lexicon $\mathcal{L}_3$, where

- Π_1 is the program

$$\begin{array}{l} \{switch\}. \\ lightOn \leftarrow \ switch, not\ am. \\ \leftarrow not\ lightOn. \\ \{am\}. \\ \leftarrow not\ am, |x < 12|. \\ \leftarrow am, |x \geq 12|. \end{array} \tag{15}$$

  The set of irregular atoms of Π_1 is $\{|x < 12|, |x \geq 12|\}$. The remaining atoms form the regular set.

- $\mathcal{B}_1 = \{x < 12, x \geq 12\}$, and
- $\gamma_1(a) = \begin{cases} \text{Inequality constraint } x < 12 & \text{if } a = |x < 12| \\ \text{Inequality constraint } x \geq 12 & \text{if } a = |x \geq 12| \end{cases}$

  and

$$\neg\gamma_1(a) = \begin{cases} \text{Inequality constraint } x \geq 12 & \text{if } a = |x < 12| \\ \text{Inequality constraint } x < 12 & \text{if } a = |x \geq 12|. \end{cases}$$

The first 4 lines of program Π_1 are identical to logic program (5). The last two lines of the program state:

- It must be am when $x < 12$, where x is understood as the hours of the day.
- It is impossible for it to be am when $x \geq 12$.

Consider the set

$$\{switch,\ lightOn, |x \geq 12|\}. \tag{16}$$

over $At(\Pi_1)$. This set is the only input answer set of Π_1 relative to irregular atoms of Π_1. Also, the GCSP with constraints $\{\gamma_1(|x \geq 12|) \cup \neg\gamma_1(|x < 12|)\} = \{x \geq 12\}$ has a solution. There are 12 valuations relative to $\mathcal{L}_3$ for x which satisfy this constraint: $x^v = 12, \ldots, x^v = 23$. It follows that set (16) is an answer set of P_1. □

3.3 Input Completion

Similar to how completion was defined in Section 2, we now define an input completion which is relative to an (input) vocabulary.

Definition 4. For a program Π over vocabulary σ, the *input-completion* of Π relative to vocabulary $\iota \subseteq \sigma$, denoted by $IComp(\Pi, \iota)$, is defined as the set of propositional formulas (formulas in propositional logic) that consists of the implications (6) for all rules (2) in Π and the implications (7) for all atoms a occurring in $(\sigma \setminus \iota) \cup hd(\Pi)$.

Example 4. The input completion of program Π_1 from Example 3 relative to a vocabulary

which consists of Π_1 irregular atoms $\{|x < 12|, |x \geq 12|\}$ consists of formulas:

$$\begin{aligned} &\neg\neg switch \rightarrow switch \\ &switch \wedge \neg am \rightarrow lightOn \\ &\neg lightOn \rightarrow \\ &\neg\neg am \rightarrow am \\ &\neg am, |x < 12| \rightarrow \bot \\ &am, |x \geq 12| \rightarrow \bot \end{aligned} \tag{17}$$

and formulas (8).

It is easy to see that $IComp(\Pi_1, \{|x < 12|, |x \geq 12||\})$ is equivalent to the formula

$$\begin{aligned} &\neg switch \vee switch \\ &lightOn \leftrightarrow switch \wedge \neg am \\ &lightOn \\ &\neg am \vee am. \\ &\neg am \wedge |x < 12| \rightarrow \bot \\ &am \wedge |x \geq 12| \rightarrow \bot \end{aligned} \tag{18}$$

The set $\{switch, lightOn, |x \geq 12|\}$ is the only model of (18). Note that this model coincides with the input answer set of Π_1 relative to the set of its irregular atoms. □

The observation that we made last in the preceding example is an instance of the general fact captured by the following theorem.

Theorem 1. For a tight program Π over vocabulary σ and vocabulary $\iota \subseteq \sigma$, a set X of atoms from σ is an input answer set of Π relative to ι if and only if X satisfies program's input-completion $IComp(\Pi, \iota)$.

3.4 Constraint Formula

Just as we defined constraint answer set programs, we can define constraint formulas. For a propositional formula F, by $At(F)$ we denote the set of atoms (propositional symbols) occurring in it.

Definition 5. A *constraint formula* over the vocabulary $\sigma = \sigma_r \cup \sigma_i$ is a triple $\langle F, \mathcal{B}, \gamma \rangle$, where F is a propositional formula over the vocabulary σ, $\mathcal{B}$ is a set of constraints over the same lexicon, and γ is an injective function from the set σ_i of irregular atoms to the set $\mathcal{B}$ of constraints.

For a constraint formula $\mathcal{F} = \langle F, \mathcal{B}, \gamma \rangle$ over the vocabulary $\sigma = \sigma_r \cup \sigma_i$ such that $\mathcal{L}$ is the lexicon of the constraints in $\mathcal{B}$, a set $X \subseteq \sigma$ is a *model* of $\mathcal{F}$ if

- $X \subseteq At(F)$
- X is a model of F, and
- the following GCSP over $\mathcal{L}$ has a solution

$$\{\gamma(a) | a \in X \cap \sigma_i\} \cup \{\neg\gamma(a) | a \in (At(F) \cap \sigma_i) \setminus X\}.$$

Example 5. Similar to the CAS program P_1 from Example 3, we can define a constraint formula

$$\mathcal{F}_1 = \langle IComp(\Pi_1, \{|x < 12|, |x \geq 12||\}), \mathcal{B}_1, \gamma_1 \rangle$$

relative to the lexicon $\mathcal{L}_3$. The set

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

is the only model of $\mathcal{F}_1$. □

Following theorem captures a relation between CAS programs and constraint formulas.

Theorem 2. For a CAS program $P = \langle \Pi, \mathcal{B}, \gamma \rangle$ over the vocabulary $\sigma = \sigma_r \cup \sigma_i$ and a

set X of atoms over σ, when Π is tight, X is an answer set of P if and only if X is a model of constraint formula $\langle IComp(\Pi, \sigma_i), \mathcal{B}, \gamma\rangle$ over $\sigma = \sigma_r \cup \sigma_i$.

We note that Example 3 and Example 5 demonstrate this property. In the future we will abuse the term "tight". We will refer to CAS program $P = \langle \Pi, \mathcal{B}, \gamma\rangle$ as tight when the first member Π has this property.

3.5 Proofs for Theorem 1 and Theorem 2

Below we present the proofs for the theorems presented in this section. By $Bodies(\Pi, a)$ we denote the set of the bodies of all rules of Π with head a.

Proof of Theorem 1. In this proof we sometimes identify rules (2) in Π with respective implications $B \to a$ so that we may write symbol Π to denote not only a set of logic rules but also a set of respective implications.

Assume X is an input answer set of a program Π relative to ι. By Definition 2, X is an answer set of $\Pi \cup ((X \cap \iota) \setminus hd(\Pi))$. Since Π is tight and $(X \cap \iota)\backslash hd(\Pi)$ only adds facts, it follows that $\Pi \cup ((X \cap \iota)\backslash hd(\Pi))$ is tight. Due to results by Fages (1994) and Erdem and Lifshitz (2001), X is a model of $Comp(\Pi \cup ((X \cap \iota)\backslash hd(\Pi)))$.

We can write $Comp(\Pi \cup ((X \cap \iota)\backslash hd(\Pi)))$ as a union of the sets:

$$\Pi \tag{19}$$

$$(X \cap \iota)\backslash hd(\Pi) \tag{20}$$

$$\{a \to \bigvee_{B \in Bodies(\Pi \cup ((X\cap\iota)\backslash hd(\Pi)), a)} B \mid a \in hd(\Pi)\} \tag{21}$$

$$\{a \to \bot \mid a \notin hd(\Pi) \text{ and } a \notin (X \cap \iota) \text{ and } a \in \sigma\} \tag{22}$$

Set X satisfies (19), (20), (21), and (22).

Similarly, we can write $IComp(\Pi, \iota)$ as the union of (19), and implications

$$\{a \rightarrow \bigvee_{B \in Bodies(\Pi, a)} B \mid a \in hd(\Pi)\} \tag{23}$$

and

$$\{a \rightarrow \bot \mid a \notin hd(\Pi) \text{ and } a \in \sigma \setminus \iota\}. \tag{24}$$

Expression (24) can be written as

$$\{a \rightarrow \bot \mid a \notin hd(\Pi) \text{ and } a \notin \iota \text{ and } a \in \sigma\}. \tag{25}$$

Note that (21) and (23) coincide since $Bodies(\Pi, a) = Bodies(\Pi \cup ((X \cap \iota) \backslash hd(\Pi)), a)$ for all atoms $a \in hd(\Pi)$. Since X satisfies (21), it trivially satisfies (23). Set (22) can be written as the union of set (25) and set

$$\{a \rightarrow \bot \mid a \notin hd(\Pi) \text{ and } a \in (\iota \setminus X) \text{ and } a \in \sigma\}. \tag{26}$$

Trivially since X satisfies (22), X also satisfies (25). Consequently, $X \models IComp(\Pi, \iota)$.

Assume $X \models IComp(\Pi, \iota)$. Trivially, X satisfies (20). Also, $IComp(\Pi, \iota)$ consists of (19), (23), and (25) by construction. Recall that (21) and (23) coincide. Thus, X satisfies (19) and (21). Trivially, X satisfies (26) as any atom that satisfies condition $a \in (\iota \setminus X)$ is such that $a \notin X$. It also satisfies (25). Consequently, X satisfies (22). We derive that $X \models Comp(\Pi \cup ((X \cap \iota) \backslash hd(\Pi)))$ that consists of (19), (20), (21), and (22). By results by Fages (1994) and Erdem and Lifshitz (2001), X coincides with the answer set of the program $\Pi \cup ((X \cap \iota) \backslash hd(\Pi))$. By Definition 2, X is an input answer set of Π. □

Proof of Theorem 2. Follows directly from Theorem 1. □

4 Satisfiability Modulo Theories versus Constraint Formulas

First, in this section we introduce the notion of a "theory" in Satisfiability Modulo Theories (SMT) as described by Barrett and Tinelli (2014). Second, we present the definition of a "restriction formula" and state the conditions under which such formulas are satisfied by a given interpretation. These formulas are syntactically restricted classical ground predicate logic formulas. The presented notions of interpretation and satisfaction are usual, but are stated in terms convenient for our purposes. In the literature on SMT, a more sophisticated syntax than restriction formulas provide is typically discussed. Yet, SMT solvers often rely on the so called propositional abstractions of predicate logic formulas [Nieuwenhuis *et al.*, 2006, Section 3.1], which, in their most commonly used case, coincide with restriction formulas discussed here.

Interpretations and Restriction formulas An *interpretation* I for a signature Σ, also referred to as Σ-interpretation, is a tuple (D, ν, ρ, ϕ), where

- D is a domain,
- ν is a $[\Sigma_{|v}, D]$ valuation,
- ρ is a $[\Sigma_{|r}, D]$ r-denotation, and
- ϕ is a $[\Sigma_{|f}, D]$ f-denotation.

For signatures that contain no f-symbols, we omit the reference to the last element of the interpretation tuple.

For a signature Σ, a Σ-*theory* is a set of interpretations over Σ. For instance, for signature Σ_1 from Table 1, we denote the following sample interpretations:

$\mathcal{I}_1$	(D_1, ν_1, ρ_1)
$\mathcal{I}_2$	(D_1, ν_2, ρ_1)

(27)

Any subset of interpretations $\{\mathcal{I}_1, \mathcal{I}_2\}$ exemplifies a unique Σ_1-theory.

In literature on predicate logic and SMT, the term "object constant" or "function symbol

of arity 0" is commonly used to refer to elements in the signature that we call variables. Here we use the terms that stem from definitions related to constraint satisfaction processing to facilitate uncovering the precise link between CASP-like formalisms and SMT-like formalisms. It is typical for predicate logic signatures to contain propositional symbols (predicate symbols of arity 0). It is easy to extend the notion of signature introduced here to allow propositional symbols. Yet it will complicate the presentation, which is the reason we avoid this extension.

A *restriction formula* over signature Σ is a finite set of constraint literals over c-vocabulary $[\Sigma, \emptyset]$. A sample restriction formula over Σ_1 follows

$$\{\neg E(s), \neg Q(r, s)\}. \tag{28}$$

We now state the semantics of restriction formulas. We begin by considering a Σ-interpretation $I = (D, \nu, \rho, \phi)$. To each term τ over a c-vocabulary $[\Sigma, \emptyset]$, I assigns a value $\tau^{\nu,\phi}$ that we denote by τ^I. We say that interpretation I *satisfies* restriction formula Φ over Σ, when ν satisfies every constraint literal in Φ w.r.t. ρ and ϕ. For instance, interpretation $\mathcal{I}_2$ satisfies restriction formula (28), while $\mathcal{I}_1$ does not satisfy (28).

We say that a restriction formula Φ over signature Σ is *satisfiable* in a Σ-theory T, or is T-satisfiable, when there is an element of the set T that satisfies Φ. For example, restriction formula (28) is satisfiable in any Σ_1-theory that contains interpretation $\mathcal{I}_2$. On the other hand, restriction formula (28) is not satisfiable in Σ_1-theory $\{\mathcal{I}_1\}$.

SMT often considers theories of a special kind. For instance, the presented definition of a theory places no restrictions on the domains, r-denotations, or f-denotations being identical across the interpretations defining the theory. In practice, such restrictions are very common. Later in the presentation we will define so called "uniform" theories that follow typical restrictions. We will then show how such restriction formulas interpreted over uniform theories can practically be seen as syntactic variants of GCSPs.

4.1 SMT and ASPT Programs

For signature Σ, domain D, and a vocabulary σ', a $[\sigma', \Sigma, D]$-*mapping* is an injective mapping μ from atoms over σ' to constraint atoms over c-vocabulary $[\Sigma, D]$. For a $[\sigma', \Sigma, D]$-mapping μ, a consistent set M of propositional literals over vocabulary $\sigma \supseteq \sigma'$ is a T-*model of* μ *(or,* T, μ*-model)* if a restriction formula

$$\{\mu(a) |\ a \in M \cap \sigma'\} \cup \{\neg\mu(a) \mid \neg a \in M \text{ and } a \in \sigma'\}$$

is satisfiable in Σ-theory T.

Definition 6. An *SMT program* P over vocabulary $\sigma = \sigma_r \cup \sigma_i$ is a triple $\langle F, T, \mu\rangle$, where F is a propositional formula over σ, μ is a $[\sigma_i, \Sigma, \emptyset]$-mapping, and T is a Σ-theory.

For an SMT program $\langle F, T, \mu\rangle$ over σ, a set $X \subseteq \sigma$ is its *model* if

1. $X \subseteq At(F)$,
2. X is a model of F, and
3. $X \cup \{\neg a \mid a \in At(F) \setminus X\}$ is a T, μ-model.

We now define the concept of logic programs modulo theories.

Definition 7. A *logic program modulo theories* (or *ASPT program*) P over vocabulary $\sigma = \sigma_r \cup \sigma_i$ is a triple $\langle \Pi, T, \mu\rangle$, where Π is a logic program over σ, μ is a $[\sigma_i, \Sigma, \emptyset]$-mapping, and T is a Σ-theory.

For an ASPT program $\langle \Pi, T, \mu\rangle$ over σ, a set $X \subseteq \sigma$ is its *model* if

1. $X \subseteq At(\Pi)$,
2. X is an input answer set of Π relative to σ_i, and
3. $X \cup \{\neg a \mid a \in At(\Pi) \setminus X\}$ is a T, μ-model.

4.2 Uniform Theories

We now identify a special class of theories that we call "uniform". We then relate the question of verifying satisfiability of formulas in such theories to the question of solving relevant generalized constraint satisfaction problems. This connection brings us to a straightforward relation between SMT program formalism over uniform theories and constraint formula formalism as well as between CAS programs and ASPT programs. In the following section, we list several common SMT formalisms such as satisfiability modulo difference logic and satisfiability modulo linear arithmetic whose theories are, in fact, uniform theories. We then use these findings to relate several ASP modulo theories approaches such as ASP(DL) introduced in [Liu *et al.*, 2012] and ASP(LC) introduced in [Liu *et al.*, 2012] to CASP approaches.

Definition 8. For a signature Σ, we call a Σ-theory T *uniform* over a lexicon $\mathcal{L} = ([\Sigma, D], \rho, \phi)$ when

(i) all interpretations in T are of the form (D, ν, ρ, ϕ) (note how valuation ν is the only not fixed element in the interpretations), and

(ii) for every possible $[\Sigma_{|v}, D]$ valuation ν, there is an interpretation (D, ν, ρ, ϕ) in T.

To illustrate a concept of a uniform theory, a table below defines sample domain D_3, valuations ν_3 and ν_4, and r-denotation ρ_4.

D_3	$\{1, 2\}$
ν_3	$[\Sigma_{1\|v}, D_3]$ valuation, where $s^{\nu_3} = 1$ and $r^{\nu_3} = 2$
ν_4	$[\Sigma_{1\|v}, D_3]$ valuation, where $s^{\nu_4} = r^{\nu_4} = 2$
ρ_4	$[\Sigma_{1\|r}, D_3]$ r-denotation, where $E^{\rho_4} = \{\langle 2\rangle\}, Q^{\rho_4} = \{\langle 1, 1\rangle, \langle 2, 2\rangle\}$

Valuations ν_1 and ν_2 from Table 1 can be seen not only as $[\Sigma_{1|v}, D_1]$ valuations, but also

as $[\Sigma_{1|v}, D_3]$ valuations. The set

$$\{(D_3, \nu_1, \rho_4), (D_3, \nu_2, \rho_4), (D_3, \nu_3, \rho_4), (D_3, \nu_4, \rho_4)\}$$

of Σ_1 interpretations is an example of a uniform theory over lexicon $([\Sigma_1, D_3], \rho_4)$. We denote this theory by T_1. On the other hand, the set

$$\{(D_3, \nu_1, \rho_4), (D_3, \nu_2, \rho_4), (D_3, \nu_3, \rho_4), (D_1, \nu_4, \rho_4)\}$$

of Σ_1 interpretations is an example of a non-uniform theory. Indeed, the condition (i) of Definition 8 does not hold for this theory: the last interpretation refers to a different domain than the others. Recall interpretations $\mathcal{I}_1$ and $\mathcal{I}_2$ given in (27). Neither Σ_1-theory $\{\mathcal{I}_1\}$ nor $\{\mathcal{I}_1, \mathcal{I}_2\}$ is uniform over lexicon $([\Sigma_1, D_1], \rho_1)$. In this case, the condition (ii) of Definition 8 does not hold.

It is easy to see that for uniform theories we can identify their interpretations of the form (D, ν, ρ, ϕ) with their second element valuation ν. Indeed, the other three elements are fixed by the lexicon over which the uniform theory is defined. In the following, we will sometimes use this convention. For example, we may refer to interpretation (D_3, ν_1, ρ_4) of uniform theory T_1 as ν_1.

For uniform Σ-theory T over lexicon $([\Sigma, D], \rho, \phi)$, we can extend the syntax of restriction formulas by saying that a *restriction formula* is defined over c-vocabulary $[\Sigma, D]$ as a finite set of constraint literals over $[\Sigma, D]$ (earlier we considered constraint literals over $[\Sigma, \emptyset]$). The earlier definition of semantics is still applicable. In the following, for the uniform theories we assume this more general syntax. Also we can extend the definition of SMT program given a constraint Σ-theory T over lexicon $([\Sigma, D], \rho, \phi)$ as follows: an ***SMT program*** P over vocabulary $\sigma = \sigma_r \cup \sigma_i$ is a triple $\langle F, T, \mu \rangle$, where F is a propositional formula over σ, μ is a $[\sigma_i, \Sigma, D]$-mapping, and T is a Σ-theory. Note how μ-mapping refers do the domain of lexicon now in place of an empty set in the earlier definition. The definition

of ASPT program can be extended in the same style. For the case of uniform theories we will assume the definition of SMT programs as stated in this paragraph. The same applies to the case of ASPT programs modulo uniform theories.

We now present a theorem that makes the connection between GCSPs over some lexicon $\mathcal{L}$ and restriction formulas interpreted using the uniform theory T over the same lexicon $\mathcal{L}$ apparent: the question whether a given GCSP over $\mathcal{L}$ has a solution translates into the question whether the set of constraint literals of a GCSP forming a restriction formula is T-satisfiable. Furthermore, any solution to such GCSP is also an interpretation in T that satisfies the respective restriction formula, and the other way around. We then relate SMT programs "modulo uniform theories" and constraint formulas, as well as ASPT programs and CAS programs.

Theorem 3. For a lexicon $\mathcal{L} = ([\Sigma, D], \rho, \phi)$, a set Φ of constraint literals defined over the c-vocabulary $[\Sigma, D]$, a uniform Σ-theory T over lexicon $\mathcal{L}$, the following holds

1. for any $[\Sigma_{|v}, D]$ valuation ν, there is an interpretation ν in T,
2. $[\Sigma_{|v}, D]$ valuation ν is a solution to GCSP Φ over lexicon $\mathcal{L}$ if and only if interpretation ν in T satisfies Φ.
3. GCSP Φ over lexicon $\mathcal{L}$ has a solution if and only if Φ is T-satisfiable.

Let $\mathcal{L}$ denote a lexicon $([\Sigma, D], \rho, \phi)$. By $\mathcal{B}_\mathcal{L}$ we denote the set of all constraints over $\mathcal{L}$ of the form (10). By $T_\mathcal{L}$ we denote the uniform Σ-theory over $\mathcal{L}$.

Theorem 4. For a signature Σ, a lexicon $\mathcal{L} = ([\Sigma, D], \rho, \phi)$, vocabularies $\sigma = \sigma_i \cup \sigma_r$, a $[\sigma_i, \Sigma, D]$-mapping μ, and a set $X \subseteq \sigma$, X is a model of an SMT program $\langle F, T_\mathcal{L}, \mu\rangle$ over σ if and only if X is a model of a constraint formula $\langle F, \mathcal{B}_\mathcal{L}, \mu\rangle$ over σ, where μ is identified with the function from irregular atoms to constraints over $\mathcal{L}$ in a trivial way.

This theorem illustrates that for uniform theories the language of SMT programs and constraint formulas coincide. Or, in other words, that the language of constraint formulas

is a special case of SMT programs that are defined over uniform theories. We now show similar relation between CAS and ASPT programs.

Theorem 5. For a signature Σ, a lexicon $\mathcal{L} = ([\Sigma, D], \rho, \phi)$, vocabularies $\sigma = \sigma_i \cup \sigma_r$, a $[\sigma_i, \Sigma, D]$-mapping μ, and a set $X \subseteq \sigma$, X is a model of an ASPT program $\langle \Pi, T_{\mathcal{L}}, \mu \rangle$ over σ if and only if X is a model of a CAS program $\langle \Pi, \mathcal{B}_{\mathcal{L}}, \mu \rangle$ over σ, where μ is identified with the function from irregular atoms to constraints over $\mathcal{L}$ in a trivial way.

4.3 Proofs for Theorem 3, Theorem 4, and Theorem 5

Below we present the proofs for the theorems presented in this section.

Proof of Theorem 3. Statement 1 trivially follows from the condition 2 of the uniform theory definition.

Proof of Statement 2. By Statement 1, interpretation ν is in T. By definition of a solution to GCSP, $[\Sigma_{|v}, D]$ valuation ν is a solution to GCSP Φ over lexicon $\mathcal{L}$ if and only if ν is a solution to every constraint in Φ. In other words, ν satisfies every constraint literal in Φ w.r.t. ρ and ϕ. By definition of interpretations satisfying formulas, the previous statement holds if and only if interpretation ν in T satisfies Φ.

Proof of Statement 3. GCSP Φ over lexicon $\mathcal{L}$ has a solution if and only if there is a $[\Sigma_{|v}, D]$ valuation ν that is a solution to GCSP Φ over lexicon $\mathcal{L}$. By statements 1 and 2, the previous statement holds if and only if there is interpretation ν in T that satisfies Φ and consequently Φ is T-satisfiable. □

Proof of Theorem 4. Set $X \subseteq \sigma$ is a model of an SMT program $\langle F, T_{\mathcal{L}}, \mu \rangle$ over σ if and only if

1. $X \subseteq At(F)$,
2. X is a model of F, and
3. $X \cup \{\neg a \mid a \in At(F) \setminus X\}$ is a $T_{\mathcal{L}}, \mu$-model.

By the definition of a $T_{\mathcal{L}}, \mu$-model, condition 3 holds if and only if restriction formula

$$\{\mu(a)|\ a \in X \cap \sigma_i\} \cup \{\neg\mu(a) \mid a \in At(F) \setminus X \text{ and } a \in \sigma_i\} \tag{29}$$

is satisfiable in Σ-theory $T_{\mathcal{L}}$ over lexicon $\mathcal{L}$. By the definition of a model of a constraint formula, to conclude the proof, we only have to illustrate that condition 3 holds if and only if the GCSP over $\mathcal{L}$

$$\{\mu(a) \mid a \in X \cap \sigma_i\} \cup \{\neg\mu(a) \mid a \in (At(F) \cap \sigma_i) \setminus X\} \tag{30}$$

has a solution. It is obvious that (29) and (30) are identical. By Statement 3 of Theorem 3 we conclude that (30) has a solution if and only if (29) is satisfiable in Σ-theory $T_{\mathcal{L}}$ (which is the case if and only if condition 3 holds). □

Proof of Theorem 5. Follows the lines of proof of Theorem 4. □

5 SMT and CASP Connection

This section starts by introducing "numeric" signatures, lexicons, and uniform theories. These definitions allow us to precisely define the languages used by various constraint answer set solvers. We conclude with the discussion of the variety of solving techniques used in logic programming community.

Let $\mathbb{Z}$ and $\mathbb{R}$ denote the sets of integers and real numbers respectively. A *numeric signature* is a signature that satisfies the following requirements

- its only predicate symbols are $<$, $>$, $\leq$, $\geq$, $=$, $\neq$ of arity 2, and
- its only f-symbols are $+$, $\times$ of arity 2.

We use the symbol $\mathcal{A}$ to denote a numeric signature.

Let $\rho_{\mathbb{Z}}$ and $\phi_{\mathbb{Z}}$ be $[\{<, >, \leq, \geq, =, \neq\}, \mathbb{Z}]$ r-denotation and $[\{+, \times\}, \mathbb{Z}]$ f-denotation respectively, where they map their predicate and function symbols into usual arithmetic

relations and operations over integers. Similarly, $\rho_{\mathbb{R}}$ and $\phi_{\mathbb{R}}$ denote $[\{<, >, \leq, \geq, =, \neq\}, \mathbb{R}]$ r-denotation and $[\{+, \times\}, \mathbb{R}]$ f-denotation respectively, defined over the reals. We can now define the following lexicons

- an *integer lexicon* of the form $([\mathcal{A}, \mathbb{Z}], \rho_{\mathbb{Z}}, \phi_{\mathbb{Z}})$ and
- a *numeric lexicon* of the form $([\mathcal{A}, \mathbb{R}], \rho_{\mathbb{R}}, \phi_{\mathbb{R}})$.

A *(numeric) linear expression* has the form

$$a_1 x_1 + \cdots + a_n x_n, \tag{31}$$

where $a_1, \ldots, a_n$ are real numbers and $x_1, \ldots, x_n$ are variables over real numbers. When $a_i = 1$ $(1 \leq i \leq n)$ we may omit it from the expression. We view expression (31) as an abbreviation for the following term

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

over some c-vocabulary $[\mathcal{A}, \mathbb{R}]$, where $\mathcal{A}$ contains $x_1, \ldots, x_n$ as its variables. For instance, $2x_2 + 3x_3$ is an abbreviation for the expression $+(\times(2, x_2), \times(3, x_3))$.

An *integer linear expression* has the form (31), where $a_1, \ldots, a_n$ are integers, and $x_1, \ldots, x_n$ are variables over integers.

We call a constraint *linear (integer linear)* when it is defined over some numeric (integer) lexicon and has the form

$$\bowtie (e, k) \tag{32}$$

where e is a linear (integer linear) expression, k is a real number (an integer), and $\bowtie$ belongs to $\{<, >, \leq, \geq, =, \neq\}$. We can write (32) as an expression in usual infix notation $e \bowtie k$.

We call a GCSP a *(integer) linear constraint satisfaction problem* when it is composed of (integer) linear constraints. For instance, consider integer linear constraint satisfaction problem composed of two constraints $x > 4$ and $x < 5$ (here signature $\mathcal{A}$ is implicitly

defined by restricting its variable to contain x). It is easy to see that this problem has no solutions. On the other hand, linear constraint satisfaction problem composed of the same two constraints $x > 4$ and $x < 5$ has infinite number of solutions, including valuation that assigns x to 4.1.

By $T^{\mathcal{A}}_{\mathbb{Z}}$ and $T^{\mathcal{A}}_{\mathbb{R}}$ we denote uniform $\mathcal{A}$-theories over integer and numeric lexicons respectively. We call a theory $T^{\mathcal{A}}_{\mathbb{Z}}$ *integer*. Similarly, we call a theory $T^{\mathcal{A}}_{\mathbb{R}}$ *numeric*.

Two examples of integer theories commonly implemented in SMT solvers are *linear integer arithmetic* and *difference logic arithmetic*. *Linear real arithmetic* is an example of numeric theory commonly supported by SMT solver.

We note that difference logic is a special case of integer linear arithmetic that introduces syntactic requirements on considered expressions [Nieuwenhuis and Oliveras, 2005]. SMT solver CVC4[4] [Barrett *et al.*, 2011] supports all three mentioned arithmetics.

We are now ready to define SMT(DL), SMT(IL), and SMT(L) programs that capture common SMT formalisms used in practice. For a vocabulary σ, we say that $[\sigma, \mathcal{A}, \mathbb{Z}]$-mapping μ is *integer linear (difference-logic) friendly* if every element in σ is mapped to an integer linear formula (difference logic formula) over integer lexicon whose signature is $\mathcal{A}$. Similarly, for vocabulary σ, we say that $[\sigma, \mathcal{A}, \mathbb{R}]$-mapping μ is *linear friendly* if every element in σ is mapped to a linear formula over numeric lexicon whose signature is $\mathcal{A}$.

We call an SMT program $\langle F, T^{\mathcal{A}}_{\mathbb{Z}}, \mu\rangle$ an *SMT(DL)* or *SMT(IL)* program when μ is difference-logic friendly or integer linear friendly respectively. We call a SMT program $\langle F, T^{\mathcal{A}}_{\mathbb{R}}, \mu\rangle$ a *SMT(L)* program when μ is linear friendly. In the same style, we can define the notions of ASPT(DL), ASPT(IL), ASPT(L) programs.

We say that a CAS program $\langle \Pi, \mathcal{B}, \gamma\rangle$ is *integer* when $\mathcal{B}$ is the set of all integer linear constraints of the form (10) over some integer lexicon. Similarly, we say that a CAS program $\langle \Pi, \mathcal{B}, \gamma\rangle$ is *numeric* when $\mathcal{B}$ is the set of all linear constraints of the form (10) over some numeric lexicon. From Theorem 5, it follows that integer CAS programs and numeric CAS

[4] http://cvc4.cs.nyu.edu/web/

Solver	Language
CLINGCON [Gebser *et al.*, 2009]	ASPT(IL)*
EZCSP [Balduccini, 2009]	ASPT(IL)* ASPT(IL) ASPT(L)
MINGO [Liu *et al.*, 2012]	ASPT(L)
DINGO [Janhunen *et al.*, 2011]	ASPT(DL)

Table 3: Solvers Categorization

programs are in fact the same objects as ASPT(IL) and ASPT(L) programs respectively.

Obviously, Theorems 2 and 4 pave the way for using SMT systems that solve SMT(IL), SMT(L) problems as is for solving ASPT(IL), ASPT(L) programs whose first member of a triple is a tight logic program. It is sufficient to compute the input completion of the program relative to irregular atoms. This observation has been utilized in work by Lee and Meng (2013) and Janhunen et al. (2011).

Outlook on Constraint Answer Set Solvers Table 3 presents the landscape of current constraint answer set solvers using the unified terminology of this section. The star $*$ annotating language ASPT(IL) denotes that the solver supporting this language requires the specification of finite ranges for its variables (since finite-domain constraint solvers are used as underlying solving technology). This notation will be used for the rest of the paper.

At a high-level abstraction, one may summarize the architectures of the CLINGCON and EZCSP solvers as *ASP-based solvers plus theory solver*. Given a CAS program $\langle \Pi, \mathcal{B}, \gamma \rangle$, both CLINGCON and EZCSP first use an answer set solver to compute an input answer set of Π. Second, they contact a theory solver to verify whether respective constraint satisfaction problem has a solution. In case of CLINGCON, finite-domain constraint solver GECODE is used as a theory solver. System EZCSP uses constraint logic programming tools such as BPROLOG [Zhou, 2012], SICSTUS PROLOG [Carlsson and Fruehwirth, 2014], and SWI PROLOG [Wielemaker *et al.*, 2012]. These tools provide EZCSP with the ability to work with three different kinds of constraints: finite-domain integer, integer-linear, and linear

constraints. To process ASPT(L) programs, the solver MINGO translates the formalism into mixed integer programming formula and then uses the solver CPLEX [IBM, 2009] to solve these formulas. To process ASPT(DL) programs DINGO translates the formalism into an SMT(DL) program and applies the SMT solver Z3 [De Moura and Bjørner, 2008].

The diversity of solving approaches used in CASP paradigms suggests that solutions of the kind are available for SMT technology. Typical SMT architecture is in a style somewhat different than the systems CLINGON and EZCSP. One difference is that a satisfiability solver is used as a base solver. Another difference is that theory solvers are typically implemented within an SMT solver and are as such custom solutions. The fact that CLINGON and EZCSP use tools available from the constraint programming community suggests that these tools could be of use in SMT community also. The solution exhibited by system MINGO, where mixed integer programming is used for solving ASPT(L) programs, hints that a similar strategy can be implemented for solving SMT(L) programs. These ideas have recently been explored in [King *et al.*, 2014].

6 The EZSMT Solver

By Theorem 2, we know that answer sets of a tight CAS program coincide with models of a constraint formula that corresponds to the input completion of the CAS program relative to its irregular atoms. Theorem 4 further demonstrates that models of such a constraint formula coincide with models of a respective SMT program. Finally, an SMT solver can be used to find models of this SMT program, which correspond to answer sets of the original CAS program. The EZSMT solver is a software system that was developed in the scope of this thesis that relies on these results.

The EZSMT system takes a *tight* program written in the EZCSP language (an input language accepted by the CASP solver EZCSP [Balduccini, 2009]) and produces an "equivalent" program written in SMT-LIB (a common input language for SMT solvers [Barrett *et al.*, 2015]). Subsequently, EZSMT runs a compatible SMT solver, such as CVC4 or Z3, to

compute models of the program.

In this section, we introduce the EZCSP and SMT-LIB languages. Then, we discuss the EZSMT architecture. We use the CAS program from Example 3 to illustrate a sample workflow of the EZSMT solver.

6.1 The EZCSP and SMT-LIB Languages

The EZCSP language is a fine representative of CASP language used in practice. Balduccini (2009) describes the syntax of the EZCSP language.

We illustrate the constructs of this language using our running sample CAS program P_1 from Example 3. In EZCSP syntax P_1 has the form

```
cspdomain(fd).
cspvar(x,0,23).
{switch}.
lightOn ← switch not am.
← not lightOn.
{am}.
required(x ≥ 12) ← not am.
required(x < 12) ← am.
```

Listing 1: EZCSP Program

The first line of the program states that the CAS program is a ASPT(IL)* program. The second line of the program specifies that the domain of the constraint variable $x \in \Sigma_{2|v}$ is the range $[0..23]$. In the EZCSP language, all irregular atoms are enclosed in a "required" statement. For instance, see the last two rules in Listing 1. Appendix A introduces details on the fragment of the EZCSP language constructs that are supported by the EZSMT system.

SMT-LIB acts as a standard language for a majority of SMT solvers [Barrett *et al.*, 2015]. SMT-LIB allows the SMT community to develop benchmarks and run solving competitions in a standard language. Barret et al. (2015) and Cok (2011) define the syntax and usage of SMT-LIB.

As opposed to CASP languages, which are regarded as declarative *programming* languages, SMT-LIB is a low-level specification language. SMT-LIB is not intended to be a

modeling language, but geared in large part to be easily interpretable by SMT solvers.

6.2 EZSMT Architecture

The EZSMT system takes a program given in the EZCSP language and outputs a program in the SMT-LIB version 2 language. This SMT-LIB program can be run by such SMT solvers as CVC4 and Z3. The program transformation process has been fairly modularized from the actual solving process, which can be conducted by any existing SMT solver that supports SMT-LIB as an input format. The current system runs in a pipeline presented in Figure 1. Subsequent sections are devoted to the steps of the pipeline.

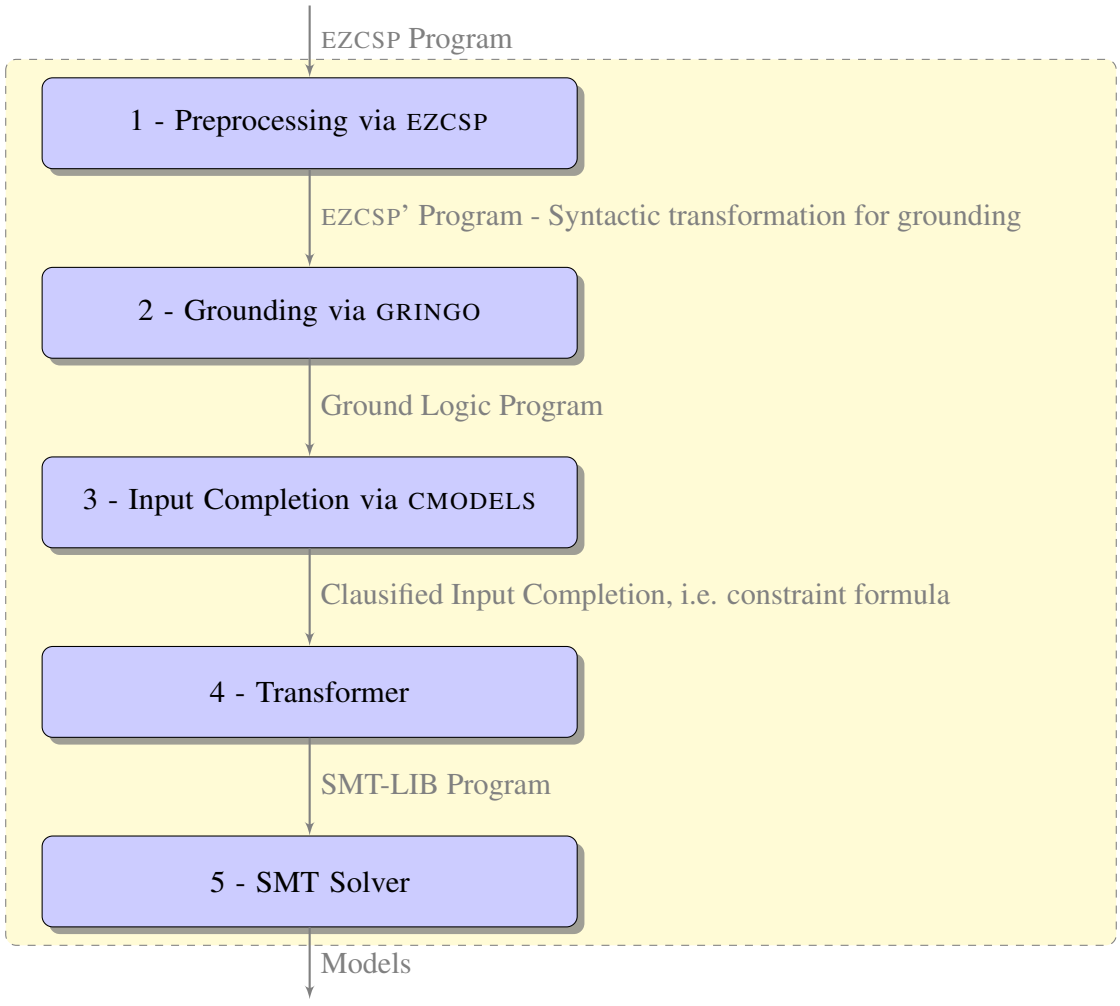

Figure 1: EZSMT Pipeline

6.2.1 Preprocessing and Grounding

In this thesis, we formally introduced CAS programs over a signature that allows propositional atoms or irregular atoms. In practice, CAS languages, just as traditional answer set programming languages, allow non-irregular atoms with schematic-variables. The process of eliminating these variables is referred to as *grounding* [Gebser *et al.*, 2007]. It is a well understood process in answer set programming and off the shelf grounders exist, e.g., the GRINGO system[5] [Gebser *et al.*, 2011]. The EZSMT system also allows schematic-variables (as they are part of the EZCSP language) and relies on GRINGO to eliminate these variables.

Prior to applying GRINGO, all irregular atoms in the input program must be identified in order to be preserved through the grounding process. The "required" keyword in the EZCSP language allows us to achieve this so that the rules with the "required" expression in the head are converted into an intermediate language. The invocation of the EZCSP system with the `--preparse-only` flag performs the conversion. Later in the pipeline, this intermediate language will undergo a semantics preserving transformation into SMT-LIB syntax. The preprocessing performed by EZCSP results in a valid input program for the grounder GRINGO, while preserving the constraint relations.

For instance, the application of EZCSP with `--preparse-only` flag on the program in Listing 1 results in the following:

```
cspdomain(fd).
cspvar(x,0,23).
{switch}.
lightOn ← switch not am.
← not lightOn.
{am}.
required(ezcsp__geq(x, 12)) ← not am.
required(ezcsp__lt(x, 12)) ← am.
```

Listing 2: Preprocessed EZCSP Program

[5] http://potassco.sourceforge.net

6.2.2 Program's Completion

The third block in the pipeline in Figure 1 is responsible for three tasks. First, it determines whether the program is tight or not. Given a non tight programs the system will exit with the respective message. (Recall that EZSMT only accepts tight programs.) Second, it computes the input completion of a given program (recall, that this input completion can be seen as an SMT program). Third, the input completion is clausified using Tseitin transformations [Tseitin, 1968] so that the resulting formula is in conjunctive normal form. This transformation preserves the models of the completion modulo original vocabulary. The output from this step is a file in a DIMACS[6]-inspired format. System CMODELS[7] [Lierler, 2005] is used to perform the described steps. This system is invoked with the `--cdimacs` flag.

For example, given the grounding produced by GRINGO for the program in Listing 2, CMODELS will produce the output presented in Listing 3. This output encodes the clausified input completion of the CAS program in Example 3 and can be viewed as an SMT program.

```
smt cnf 7 10
-switch switch 0
-switch lightOn 0
-lightOn switch 0
cspdomain(fd) 0
cspvar(x,0,23) 0
switch 0
lightOn 0
-5 0
required(ezcsp__geq(x,12)) 0
7 0
```

Listing 3: Completion of EZCSP Program

The first line in Listing 3 states that there are 7 atoms and 10 clauses in the computed formula. Each other line stands for a clause, for instance, line

```
-switch switch 0
```

[6] http://www.satcompetition.org/2009/format-benchmarks2009.html
[7] http://www.cs.utexas.edu/users/tag/cmodels/

stands for a clause

$$\neg switch \vee switch.$$

Note that there are atoms named 5 and 7 that appear in Listing 2. These are auxiliary atoms introduced during the clausification process.

6.2.3 Transformation

The output program from CMODELS serves as input to the Transformer block in the EZSMT pipeline. Transformer converts the given SMT program into the SMT-LIB syntax. Mappings for irregular atoms in the EZCSP language format to SMT-LIB format are outlined in Appendix A.

For instance, given the SMT program presented in Listing 2, the Transformer produces the following SMT-LIB code.

```
1   (set-option :interactive-mode true)
2   (set-option :produce-models true)
3   (set-option :produce-assignments true)
4   (set-option :print-success false)
5   ; --- END HEADER ---
6
7   ;Quantifier Free Linear Arithmetic
8   (set-logic QF_LIA)
9
10  (declare-fun |lightOn| () Bool)
11  (declare-fun |required(ezcsp__geq(x,12))| () Bool)
12  (declare-fun |switch| () Bool)
13  (declare-fun |5| () Bool)
14  (declare-fun |cspvar(x,0,23)| () Bool)
15  (declare-fun |7| () Bool)
16  (declare-fun |cspdomain(fd)| () Bool)
17  (assert (or (not |switch|) |switch|))
18  (assert (or (not |switch|) |lightOn|))
19  (assert (or (not |lightOn|) |switch|))
20  (assert |cspdomain(fd)|)
21  (assert |cspvar(x,0,23)|)
22  (assert |switch|)
23  (assert |lightOn|)
24  (assert (not |5|))
25  (assert |required(ezcsp__geq(x,12))|)
26  (assert |7|)
27  (declare-fun |x| () Int)
28  (assert (<= 0 |x|))
```

```
29    (assert (>= 23 |x|))
30
31    (assert (=> |required(ezcsp__geq(x,12))| (>= |x|  12)))
32
33    ; Check satisfiability
34    (check-sat)
35    ; Comment if unsat occurs.
36    (get-model)
```

The resultant program can be described as follows:

(i) Lines 1-5 and 33-36 represent the program header and footer respectively. They are responsible for setting directives necessary to indicate to a solver that it should find a model of the program after satisfiability is determined [Barrett *et al.*, 2015]. Note that if input is unsatisfiable, that implies no such model exists.

(ii) Given our sample ASPT(IL) program, the Transformer instructs an SMT solver to use quantifier-free linear integer arithmetic (*QF_LIA*) [Barrett *et al.*, 2015] using the code in Lines 7-8. More information about arithmetics/logics in SMT-LIB is available in Appendix A.

(iii) Lines 10-16 are declarations of all atoms in our sample program as boolean variables (called functions in the SMT-LIB parlance). There are 7 declared variables in total.

(iv) Lines 17-26 assert all ten clauses from Listing 3 to be true.

(v) Line 27 declares variable x to be an integer.

(vi) Lines 28-29 declares the domain of variable x to be in range from 0 to 23 (recall how `cspvar(x,0,23)` from Listing 1 encodes this information).

(vii) Line 31 expresses that if the irregular atom `required(ezcsp__geq(x,12))` holds then the constraint $x \geq 12$ must also hold. In other words, it plays a role of a mapping γ_1 from Example 3.

6.2.4 SMT Solver

The final step is to use an SMT solver that accepts SMT-LIB compliant programs for findings its models.

The output produced by CVC4 with the SMT program listed in Section 6.2.3 is presented in Listing 4.

```
sat
(model
(define-fun lightOn () Bool true)
(define-fun |required(ezcsp__geq(x,12))| () Bool true)
(define-fun switch () Bool true)
(define-fun 5 () Bool false)
(define-fun |cspvar(x,0,23)| () Bool true)
(define-fun 7 () Bool true)
(define-fun |cspdomain(fd)| () Bool true)
(define-fun x () Int 12)
)
```

Listing 4: Output of CVC4

The first line of the output indicates that a satisfying assignment exists. The subsequent lines indicate a model that satisfies the SMT-LIB program.

Recall that the answer set for the Example 3 program was $\{switch, lightOn, |x \geq 12|\}$ with twelve valuations relative to the lexicon that satisfy the constraint. It is clear that this answer set corresponds to the model of Listing 4. Note that the solver correctly identified one of the possible valuations for x that maps x to 12.

We note that the output format of the SMT solver Z3 is of the same style as that of CVC4.

6.2.5 Limitations

Due to the fact that the EZSMT solver accepts programs in the EZCSP language, it is natural to compare the system to the EZCSP solver. The EZSMT system faces some limitations relative to EZCSP. Some of these exist due to implementation challenges and others are inherit to its architecture.

The EZSMT solver accepts only a subset of the EZCSP language. In particular, it supports

a limited set of its global constraints [Balduccini and Lierler, 2016]. Only, the global constraints *all_different* and *sum* are fully supported by EZSMT. Also, EZSMT can only be used on tight EZCSP programs. Yet, this is a large class of programs.

It is important to notice that, currently, we do not have a way to compute all possible models as this is not a typical objective of most current SMT solvers. For instance, the SMT-LIB language does not provide a directive to instruct an SMT solver to find all models for a problem.

EZSMT assumes the logic of input programs to be quantifier-free linear integer arithmetic. It could be worthwhile to extend EZSMT to support quantifier-free linear real arithmetic (*QF_LRA*), thereby allowing the solving of SMT(L) programs.

7 Experimental Results

In order to demonstrate the efficacy of the system and to provide a comparison to other existing CASP solvers, three CAS programs have been used to benchmark EZSMT. These benchmarks stem from the Third Answer Set Programming Competition, 2011 (ASPCOMP) [Calimeri *et al.*, 2011]. These CASP formalizations have been used to benchmark other CASP solvers and have been shown to not scale when using traditional ASP solvers [Balduccini and Lierler, 2016]. The selected encodings are: Reverse-Folding [Balduccini and Lierler, 2012], Weighted Sequence [Lierler *et al.*, 2012], and Incremental Scheduling [Balduccini, 2011].

7.1 Benchmark Descriptions

Since these same encodings have already been demonstrated to not scale with current ASP solving technology [Balduccini and Lierler, 2016], no ASP solver performance is presented in this work as we expect each instance would timeout.

All experiments were conducted on a computer with an Intel Core i7-940 processor running Ubuntu 14.04 LTS (64-bit) operating system. Each benchmark was allocated 4

GB RAM, a single processor core, and given an 1,800 second timeout (30 minutes). No benchmarks were run simultaneously.

Originally, 1 GB RAM was allocated for each benchmark, but this appeared to be somewhat prohibitive for running the CVC4 and Z3 solvers on some of the benchmarks. As such the memory-allocation was increased to 4GB.

Four CASP solvers were benchmarked for each instance/encoding pair:

- EZSMT with CVC4 version 1.4 as the SMT solver (EZSMT- CVC4)
- EZSMT with Z3 version 4.4.2 - 64 bit as the SMT solver (EZSMT- Z3)
- CLINGCON version 2.0.3, and
- EZCSP version 1.6.20b49-r3345 with B-Prolog version 7.4 #3.

The best performing EZCSP configuration, as demonstrated Balduccini and Lierler (2016), was used for each run of EZCSP.

We define a *uniform encoding* as an encoding where the problem specification and the problem instance data are separated [Lierler, 2015]. For the rest of the thesis, we will refer to the problem specification as an *encoding* and the problem instance as an *instance*. The encodings and instances used for benchmarking are available at: http://unomaha.edu/nlpkr/software/ezsmt/experiments/bsusman-thesis/experiments.zip. Each set of encodings and instances are separated by benchmark. Unless otherwise mentioned, the encoding for the CLINGCON system is CLINGCON.CL and the encoding used for EZCSP and EZSMT is TRUE-CASP.EZ.

Both CVC4 and Z3 were ran under their default configurations. The only parameter used to run the SMT encodings specified that the encodings were written in SMT-LIB version 2 format.

Presented in this section is only the abbreviated results of the benchmarking. Appendix B contains more instance-level information and may be used for a more complete picture of each systems' performance.

7.2 Results Analysis

A potential limitation to the following results rests on issues involved with system configurations. The only solver which was run with optimized configurations for the given benchmarks was the EZCSP system. It is entirely possible that different configurations of CLINGCON, CVC4, or Z3 could lead to further optimized results, although the generic configurations of each system is demonstrated to be sufficient for the scope of this work.

The format of Table 4, Table 5, and Table 6 are identical. The first column of each table indicates the solver being benchmarked. The second column indicates the combined total runtime of all instances. It is important to note that any instance which timed-out is represented in this column by adding the maximum allowed time for an instance (1,800 seconds) to the total runtime. If an instance timed-out, the Total Runtime is not equivalent to the time to run all instances to completion. The final column of each table lists the number of timeouts which occurred during benchmarking.

7.2.1 Reverse Folding

The reverse folding benchmark consists of 50 instances in total. Abbreviated results of benchmarking are presented in Table 4.

In this first benchmark set, the difference between SMT solvers used for EZSMT becomes very apparent. In this case, the Z3 solver performed better by an order of magnitude than that of CVC4 on identical SMT encodings. This underlines both the importance of solver selection and difference between SMT solvers. The flexibility about EZSMT is that users are free to select different SMT solvers as appropriate to the instances and encodings. Overall, we can see that EZCSP is the only system which did not timeout while running any instance.

Both CVC4 and Z3 timed-out on some instances due to memory limitations. It is possible if the memory-size allowed during benchmarking were increased that these solvers could have run to completion on more instances.

Solver	Total Runtime (secs)	Total Timeouts
EZSMT- CVC4	47948.41	22
EZSMT- Z3	4872.87	2
CLINGCON	2014.10	1
EZCSP	559.42	0

Table 4: Reverse Folding Results

Solver	Total Runtime (secs)	Total Timeouts
EZSMT- CVC4	24.18	0
EZSMT- Z3	23.28	0
CLINGCON	187.46	0
EZCSP	13878.58	0

Table 5: Weighted Sequence Results

7.2.2 Weighted Sequence

The abbreviated results of the weighted sequence benchmarks can be found in Table 5. The weighted sequence benchmark consists of 30 instances in total.

In the weighted sequence benchmark, we first note that no instance timed out. This helps in giving a clearer picture of the time it takes for each solver to run all instances to completion. Because no timeouts occurred, we can safely compare the runtimes of each solver. In this case, we note the considerable speedup featured by EZSMT. EZSMT noticeably outperformed CLINGCON and outperformed EZCSP by orders of multiple orders of magnitude.

7.2.3 Incremental Scheduling

The results of the incremental scheduling benchmarks can be found in Table 6. The incremental scheduling benchmark consists of 30 instances in total. Two instance types were available to benchmark against: “EASY” and “HARD”. Only the 30 “HARD” instances were selected for benchmarking. The original EZCSP encoding included a global constraint, cumulative, which is not currently supported by EZSMT. To resolve this issue, the encoding

Solver	Total Runtime (secs)	Total Timeouts
EZSMT- CVC4	10276.90	5
EZSMT- Z3	9135.48	5
CLINGCON	20416.60	11
EZCSP	37332.12	20
EZCSP with Cumulative	26690.77	14

Table 6: Incremental Scheduling Results

was rewritten to mimic methods used in the CLINGCON encoding without the use of the cumulative global constraint. As such, EZSMT- CVC4, EZSMT- Z3, EZCSP in Table 6 represent instances run on the rewritten encoding. EZCSP *with Cumulative* represents the original encoding which includes the global constraint.

Opposite of the weighted sequence results in Table 5, we first note that each solver timed-out for at least one instance in the incremental scheduling benchmark set. We do not give particular credence to the total runtimes, but more towards the total timeouts. We note that EZSMT times out the least, followed by CLINGCON timing out on over one-third the instances, and finally EZCSP, which timed out on about half the instances. It is worth noting that the use of the cumulative global constraint allowed EZCSP to run more instances to completion. All solvers timed out on the same 5 instances which EZSMT- CVC4 and EZSMT- Z3 timed out on.

7.3 Overall EZSMT Results

Overall, the benchmarks reveal many important aspects of the EZSMT solver. First, as demonstrated by the reverse folding results in Table 4, the underlying SMT solving technology selected for the SMT-LIB program produced by EZSMT is important. Next, we note that the weighted sequence results in Table 5 and the incremental scheduling results in Table 6 demonstrate the efficacy of this approach. EZSMT has been demonstrated to outperform both modern CASP solvers CLINGCON and EZCSP in terms of Total Runtime and in terms of Total Timeouts for these benchmarks.

We note that non-surprisingly a majority of the solving time for EZSMT in each benchmark occurs in the actual SMT solver. The process of EZCSP preparsing, GRINGO grounding, and CMODELS performing the completion on the program for the given benchmarks often take less than a second to complete.

8 Conclusions

In this thesis we unified the terminology stemming from the fields of CASP and SMT solving. This unification helped us identify the special class of so called uniform theories widely used in SMT practice. Given such theories, CASP and SMT solving share more in common than meets the eye.

Based on this unification, we also developed the EZSMT system, which is able to take tight constraint answer set programs and rewrite them into SMT-LIB programs. The EZSMT solver opens the doors for writing programs in the CASP formalism, while allowing SMT solving technologies to be utilized. This system is capable of outperforming other cutting-edge CASP solvers as illustrated by our experimental results.

As future work, we hope to compare Bartholomew and Lee (2014)'s ASPMT2SMT system to the EZSMT solver, but the semantic link between these two systems is yet to be uncovered. We also hope that future work can be done to help lift the tightness constraint in a style demonstrated by Liu et al. (2012) and Janhunen et al. (2011). We would like to further explore other common theories in SMT such as bit vectors and determine if they are indeed uniform theories as well. As more uniform theories are identified, it would be worthwhile to determine under what circumstances these theories can operate together while preserving the uniformity of theory. This would be directly applicable to the currently identified integer and numeric uniform theories.

Also, future work directions include the following. Extending EZSMT to completely support the entire EZCSP language, including global constraints, to allow a seamless transformation for capable CAS programs. Extending EZSMT so that it is able to enumerate

all models of a given program, as is desired in many CAS program domains. Adding support for more specific logics in EZSMT such as quantifier-free integer difference logic and quantifier-free linear real arithmetic, thereby allowing faster solving methods and the ability to solve different types of problems. Exploring the difference between solving techniques used by SMT solvers such as CVC4 and Z3 in relation to their effectiveness of solving CAS programs.

Overall, we expect this work to be a strong building block that will bolster the cross-fertilization between three different, even if related, automated reasoning communities: CASP, constraint (satisfaction processing) programming, and SMT.

A Transformation Language Reference.

The end of this appendix presents the transformations implemented by the EZCSP solver and the Transformer of the EZSMT system. Each transformation is described as follows:

Description $Inputs \rightarrow Output$

E EZCSP Language

I Intermediate Preprocessed Form

S SMT-LIB

Presented transformations are based on documentation found in [Balduccini and Lierler, 2016], [Barrett *et al.*, 2015], and in EZCSP source code. We note that the constraint domain and selected logic for the SMT solver restrict where and if variables may occur in particular operations. Furthermore, the presented list should not be considered complete, but should be used as a reference for the mapping assumed by EZSMT between EZCSP and SMT-LIB. As this aspect is theory dependent, we note the operations described below are assumed to be defined by the integer or numeric theory described in Section 5 and to correspond to *Core*, *Integer*, and *Real* theories described by Barret et al.(2015) and Cok (2011). Implementation-wise, EZSMT is currently restricted to the program's logic to be quantifier-free linear integer arithmetic, referred to as *QF_LIA* in SMT, and does not formally support other logics at this time. The logic of quantifier-free linear integer arithmetic utilizes the *Core* and *Integer* theories and all terms declared with the *Int* type must be linear. Future implementations of EZSMT may include support for other logics.

And $Bool \times Bool \rightarrow Bool$

E l /\r, l #/\r

I ezcsp__and(l,r)

S (and l r)

Or $Bool \times Bool \rightarrow Bool$

E l \/ r, l #\/ r

I ezcsp__or(l,r)

S (or l r)

Xor $Bool \times Bool \rightarrow Bool$

E l r, l # r

I ezcsp__xor(l,r)

S (xor l r)

Implication $Bool \times Bool \rightarrow Bool$

E l -> r, l #-> r

I ezcsp__impl_r(l,r)

S (=> l r)

Equality $Bool \times Bool \rightarrow Bool, Int \times Int \rightarrow Bool$

E l = r, l #= r, l == r, l #== r

I ezcsp__eq(l,r)

S (= l r)

Not Equal $Int \times Int \rightarrow Bool$

E l != r, l #!= r, l \= r, l #\= r

I ezcsp__neq(l,r)

S (distinct l r)

Greater Than $Int \times Int \rightarrow Bool$

E l > r, l #> r

I ezcsp__gt(l,r)

S (> l r)

Less Than $Int \times Int \rightarrow Bool$

E l < r, l #< r

I ezcsp__lt(l,r)

S (< l r)

Greater Than/Equal to $Int \times Int \rightarrow Bool$

E l >= r, l #>= r

I ezcsp__geq(l,r)

S (>= l r)

Less Than/Equal to $Int \times Int \rightarrow Bool$

C l <= r, l #<= r, l =< r, l #=< r

I ezcsp__leq(l,r)

S (<= l r)

Maximum $Int \times Int \rightarrow Int$

C max(l, r)

I ezcsp__max(l,r)

S (ite (< l r) r l))

Minimum $Int \times Int \rightarrow Int$

C min(l,r)

I ezcsp__min(l,r)

S (ite (< l r) l r))

Addition $Int \times Int \rightarrow Int$

C l + r

I ezcsp__pl(l,r)

S (+ l r)

Subtraction $Int \times Int \rightarrow Int$

C l - r

I ezcsp__mn(l,r)

S (- l r)

Multiply $Int \times Int \rightarrow Int$

C l * r

I ezcsp__tm(l,r)

S (* l r)

Divide $Int \times Int \rightarrow Int$

C l / r

I ezcsp__dv(l,r)

S (/ l r)

If an operator is not mentioned here, then it is currently not supported or experimentally supported. The only supported global constraints from EZCSP are *all_different* and *sum*.

B Complete Benchmark Results

B.1 Reverse Folding

Instance Name	EZSMT- CVC4	EZSMT- Z3	CLINGCON	EZCSP
SAT Instances				
01-reverse_folding-0-0	0.27	0.15	0.04	0.06
02-reverse_folding-0-0	27.40	1.44	0.48	0.29
03-reverse_folding-0-0	Timeout	Timeout	7.77	3.09
04-reverse_folding-0-0	Timeout	Timeout	Timeout	51.68
08-reverse_folding-0-0	34.70	1.64	0.43	1.18
09-reverse_folding-0-0	1582.10	2.67	0.85	1.62
10-reverse_folding-0-0	Timeout	5.39	1.87	3.75
11-reverse_folding-0-0	3.40	0.56	0.14	0.26
12-reverse_folding-0-0	1.72	0.51	0.13	0.33
13-reverse_folding-0-0	3.43	0.75	0.21	0.34
14-reverse_folding-0-0	6.93	0.70	0.19	0.32
15-reverse_folding-0-0	8.45	0.73	0.21	0.37
16-reverse_folding-0-0	12.51	1.09	0.31	0.67
17-reverse_folding-0-0	42.28	2	0.40	0.48
18-reverse_folding-0-0	24.58	0.97	0.30	1.07
19-reverse_folding-0-0	106.29	0.99	0.32	0.51
20-reverse_folding-0-0	402.82	4.55	0.94	1.27
21-reverse_folding-0-0	191.42	3.93	1.05	2.15
22-reverse_folding-0-0	992.65	3.52	1.15	1.45
23-reverse_folding-0-0	714.29	2.67	0.95	1.41
24-reverse_folding-0-0	1545.69	12.82	1.60	5.22
25-reverse_folding-0-0	Timeout	20.92	1.88	1.81
26-reverse_folding-0-0	1555.96	4.63	1.53	6.46
27-reverse_folding-0-0	Timeout	5.66	1.86	5.69
28-reverse_folding-0-0	1437.41	4.66	1.52	1.95
29-reverse_folding-0-0	Timeout	13.48	2.17	2.85
30-reverse_folding-0-0	Timeout	7.60	2.03	2.74
31-reverse_folding-0-0	1196.41	24.65	2.54	15.21
32-reverse_folding-0-0	Timeout	5.04	1.55	2.05
33-reverse_folding-0-0	Timeout	5.8	2.19	3.01
34-reverse_folding-0-0	Timeout	28.52	3.20	14.65
35-reverse_folding-0-0	Timeout	51.94	4.48	9.47

Instance Name	EZSMT- CVC4	EZSMT- Z3	CLINGCON	EZCSP
Reverse Folding SAT Instances (continued)				
36-reverse_folding-0-0	Timeout	30.13	15.66	8.45
37-reverse_folding-0-0	Timeout	64.89	20.90	10.37
38-reverse_folding-0-0	Timeout	34.18	6.36	78.58
39-reverse_folding-0-0	Timeout	94.24	5.32	15.04
40-reverse_folding-0-0	Timeout	64.88	7.38	9.94
41-reverse_folding-0-0	Timeout	193.24	8.48	11.72
42-reverse_folding-0-0	Timeout	32.65	6.83	10.36
43-reverse_folding-0-0	Timeout	40.59	22.92	89.04
44-reverse_folding-0-0	Timeout	123.66	9.82	14.04
45-reverse_folding-0-0	84.55	141.01	34.55	144.45
46-reverse_folding-0-0	85.97	221.75	27.79	16.50
47-reverse_folding-0-0	462.71	2.98	1.14	2.48
48-reverse_folding-0-0	580.98	3.06	1.08	1.76
49-reverse_folding-0-0	807.70	3.24	1.08	1.92
50-reverse_folding-0-0	33.57	1.60	0.31	0.53
UNSAT Instances				
05-reverse_folding-0-0	0.08	0.06	0.01	0.12
06-reverse_folding-0-0	0.51	0.18	0.04	0.09
07-reverse_folding-0-0	1.63	0.55	0.14	0.62
Totals Runtime (secs)	47948.41	4872.87	2014.10	559.42

B.2 Weighted Sequence

Instance Name	EZSMT- CVC4	EZSMT- Z3	CLINGCON	EZCSP
SAT Instances				
01-treeWeight-495-8leaves	0.59	0.89	6.10	299.45
02-treeWeight-570-8leaves	0.26	1.05	14.12	1246.86
03-treeWeight-679-8leaves	0.32	0.98	7.64	67.47
04-treeWeight-560-8leaves	0.31	0.42	0.15	84.52
05-treeWeight-497-8leaves	1.35	1.12	2.93	87.54
06-treeWeight-739-8leaves	0.8	0.25	2.94	160.35
07-treeWeight-569-8leaves	0.44	0.77	11.19	293.18
08-treeWeight-591-8leaves	0.66	0.34	14.82	1040.67
09-treeWeight-729-8leaves	0.67	0.36	5.89	470.60
10-treeWeight-445-8leaves	1.08	0.14	5.49	629.19
11-treeWeight-302-8leaves	1.23	0.77	8.05	872.98
12-treeWeight-651-8leaves	0.35	1.30	10.24	51.60
13-treeWeight-513-8leaves	1.06	0.39	16.55	516.89
14-treeWeight-537-8leaves	0.81	1.22	3.58	614.21
15-treeWeight-228-8leaves	0.66	0.23	1.66	20.06
16-treeWeight-281-8leaves	0.38	0.73	5.37	497.42
17-treeWeight-351-8leaves	0.27	0.92	12.93	83.29
18-treeWeight-375-8leaves	1.07	0.15	0.78	725.08
19-treeWeight-772-8leaves	0.74	0.42	7.24	273.24
20-treeWeight-544-8leaves	0.31	1.35	3.23	323.91
21-treeWeight-714-8leaves	1.11	1.78	1.19	82.82
22-treeWeight-641-8leaves	0.83	0.54	3.67	227.43
23-treeWeight-542-8leaves	1.12	0.76	7.96	1193.92
24-treeWeight-533-8leaves	0.64	1.25	0.26	20.25
25-treeWeight-539-8leaves	1.88	1.66	2.17	356.31
26-treeWeight-528-8leaves	0.67	0.66	0.59	1216.85
27-treeWeight-542-8leaves	0.92	0.47	9.45	163.24
28-treeWeight-411-8leaves	0.62	0.29	2.33	873.62
29-treeWeight-431-8leaves	1.48	0.90	2.52	273.14
30-treeWeight-486-8leaves	1.55	1.17	16.42	1112.49
Totals Runtime (secs)	24.18	23.28	187.46	13878.58

B.3 Incremental Scheduling

Note that the instance name has been abbreviated from the original, but is still identifiable by the numbers at the beginning and end of the instance name.

Instance Name	EZSMT- CVC4	EZSMT- Z3	CLINGCON	EZCSP	EZCSP- Cumulative
SAT Instances					
105-is-17280-0.dimacs	5.11	0.58	22.93	Timeout	480.34
115-is-17280-0.dimacs	5.51	0.34	Timeout	Timeout	Timeout
135-is-17280-0.dimacs	1.12	1.61	Timeout	Timeout	60.59
138-is-17280-0.dimacs	18.97	1.06	412.25	Timeout	28.12
140-is-18576-0.dimacs	99.73	1.45	12.88	Timeout	Timeout
141-is-17280-0.dimacs	22.12	0.46	Timeout	Timeout	Timeout
153-is-17280-0.dimacs	0.35	0.53	8.25	Timeout	21.13
182-is-17280-0.dimacs	67.60	7.53	Timeout	Timeout	Timeout
214-is-27000-0.dimacs	957.90	17.79	Timeout	Timeout	Timeout
219-is-28080-0.dimacs	3.00	0.68	Timeout	Timeout	56.32
UNSAT Instances					
064-is-15092-0.dimacs	20.46	20.18	41.36	Timeout	Timeout
090-is-15444-0.dimacs	0.21	0.09	0.38	27.42	35.45
099-is-15631-0.dimacs	0.51	0.12	0.35	707.85	28
102-is-18468-0.dimacs	0.71	0.23	0.86	Timeout	193.29
165-is-18180-0.dimacs	1.97	0.27	1.17	Timeout	Timeout
170-is-18108-0.dimacs	0.38	0.12	0.87	Timeout	Timeout
241-is-28320-0.dimacs	0.39	0.15	0.14	256.63	22.54
251-is-40518-0.dimacs	0.38	0.18	0.04	62.88	86.32
258-is-40698-0.dimacs	67.63	80.84	114.80	Timeout	Timeout
264-is-40968-0.dimacs	0.39	0.18	0.04	125.56	176.31
298-is-40824-0.dimacs	0.40	0.18	0.04	8.06	16.72
305-is-50100-0.dimacs	0.47	0.21	0.06	9.89	19.68
329-is-50260-0.dimacs	0.45	0.20	0.05	73.04	130.58
332-is-50480-0.dimacs	0.44	0.20	0.05	32.43	63.69
379-is-78325-0.dimacs	0.70	0.30	0.08	28.36	71.69
All Timeout Instances					
289-is-38880-0	Timeout	Timeout	Timeout	Timeout	Timeout
295-is-40626-0	Timeout	Timeout	Timeout	Timeout	Timeout
359-is-75000-0	Timeout	Timeout	Timeout	Timeout	Timeout
360-is-75000-0	Timeout	Timeout	Timeout	Timeout	Timeout
383-is-75000-0	Timeout	Timeout	Timeout	Timeout	Timeout
Totals Runtime (secs)	10276.90	9135.48	20416.60	37332.12	26690.77

References

Marcello Balduccini and Yuliya Lierler. Practical and methodological aspects of the use of cutting-edge asp tools. pages 78–92, 2012.

Marcello Balduccini and Yuliya Lierler. Constraint answer set solver ezcsp and why integration schemas matter. Unpublished draft, 2016.

Marcello Balduccini. Representing constraint satisfaction problems in answer set programming. *In* Proceedings of ICLP Workshop on Answer Set Programming and Other Computing Paradigms (ASPOCP), https://www.mat.unical.it/ASPOCP09/, 2009.

Marcello Balduccini. *Logic Programming and Nonmonotonic Reasoning: 11th International Conference, LPNMR 2011, Vancouver, Canada, May 16-19, 2011. Proceedings*, chapter Industrial-Size Scheduling with ASP+CP, pages 284–296. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011.

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

Clark Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanović, Tim King, Andrew Reynolds, and Cesare Tinelli. Cvc4. In *Proceedings of the 23rd International Conference on Computer Aided Verification (CAV'11), volume 6806 of LNCS*. Springer, 2011.

Clark Barrett, Pascal Fontaine, and Cesare Tinelli. The SMT-LIB Standard: Version 2.5. Technical report, Department of Computer Science, The University of Iowa, 2015. Available at `www.SMT-LIB.org`.

Michael Bartholomew and Joohyung Lee. *Logics in Artificial Intelligence: 14th European Conference, JELIA 2014, Funchal, Madeira, Portugal, September 24-26, 2014. Proceed-*

ings, chapter System aspmt2smt: Computing ASPMT Theories by SMT Solvers, pages 529–542. Springer International Publishing, Cham, 2014.

Francesco Calimeri, Giovambattista Ianni, Francesco Ricca, Mario Alviano, Annamaria Bria, Gelsomina Catalano, Susanna Cozza, Wolfgang Faber, Onofrio Febbraro, Nicola Leone, Marco Manna, Alessandra Martello, Claudio Panetta, Simona Perri, Kristian Reale, Maria Carmela Santoro, Marco Sirianni, Giorgio Terracina, and Pierfrancesco Veltri. *Logic Programming and Nonmonotonic Reasoning: 11th International Conference, LPNMR 2011, Vancouver, Canada, May 16-19, 2011. Proceedings*, chapter The Third Answer Set Programming Competition: Preliminary Report of the System Competition Track, pages 388–403. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011.

Mats Carlsson and Thom Fruehwirth. *Sicstus PROLOG User's Manual 4.3*. Books On Demand - Proquest, 2014.

Keith Clark. Negation as failure. In Herve Gallaire and Jack Minker, editors, *Logic and Data Bases*, pages 293–322. Plenum Press, New York, 1978.

David R. Cok and Grammatech Inc. The smt-libv2 language and tools: A tutorial. *GrammaTech, Inc., Version*, 2011.

Leonardo De Moura and Nikolaj Bjørner. 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*, pages 337–340, 2008.

Esra Erdem and Vladimir Lifschitz. Fages' theorem for programs with nested expressions. In *Proceedings of International Conference on Logic Programming (ICLP)*, pages 242–254, 2001.

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

Paolo Ferraris and Vladimir Lifschitz. Weight constraints as nested expressions. *Theory and Practice of Logic Programming*, 5:45–74, 2005.

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

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

Martin Gebser, Roland Kaminski, Arne König, and Torsten Schaub. *Logic Programming and Nonmonotonic Reasoning: 11th International Conference, LPNMR 2011, Vancouver, Canada, May 16-19, 2011. Proceedings*, chapter Advances in gringo Series 3, pages 345–351. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011.

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

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

Tim King, Clark Barrett, and Cesare Tinelli. Leveraging linear and mixed integer programming for smt. In *Proceedings of the 14th Conference on Formal Methods in Computer-Aided Design*, FMCAD '14, pages 24:139–24:146, Austin, TX, 2014. FMCAD Inc.

Joohyung Lee and Yunsong Meng. 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*, 2013.

Yuliya Lierler and Benjamin Susman. Constraint answer set programming versus satisfiability modulo theories or constraints versus theories. In *Proceedings of the 3rd Workshop on Grounding, Transforming, and Modularizing Theories with Variables*, 2015.

Yuliya Lierler and Miroslaw Truszczynski. Transition systems for model generators — a unifying approach. *Theory and Practice of Logic Programming, 27th International Conference on Logic Programming (ICLP'11) Special Issue*, 11, issue 4-5, 2011.

Yuliya Lierler, Shaden Smith, Miroslaw Truszczynski, and Alex Westlund. *Practical Aspects of Declarative Languages: 14th International Symposium, PADL 2012, Philadelphia, PA, USA, January 23-24, 2012. Proceedings*, chapter Weighted-Sequence Problem: ASP vs CASP and Declarative vs Problem-Oriented Solving, pages 63–77. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.

Yuliya Lierler. Cmodels: SAT-based disjunctive answer set solver. In *Proceedings of International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR)*, pages 447–452, 2005.

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

Yuliya Lierler. What is answer set programming to propositional satisfiability. Unpublished draft, 2015.

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

Guohua Liu, Tomi Janhunen, and Ilkka Niemela. Answer set programming via mixed integer programming. In *Knowledge Representation and Reasoning Conference*, 2012.

Kim Marriott and Peter J. Stuckey. *Programming with Constraints: An Introduction*. MIT Press, 1998.

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

Ilkka Niemelä and Patrik Simons. Extending the Smodels system with cardinality and weight constraints. In Jack Minker, editor, *Logic-Based Artificial Intelligence*, pages 491–521. Kluwer, 2000.

Robert Nieuwenhuis and Albert Oliveras. 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, 2005.

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

G. S. Tseitin. On the complexity of derivations in the propositional calculus. *Studies in Mathematics and Mathematical Logic*, Part II:115–125, 1968.

Jan Wielemaker, Tom Schrijvers, Markus Triska, and Torbjörn Lager. SWI-Prolog. *Theory and Practice of Logic Programming*, 12(1-2):67–96, 2012.

Neng-fa Zhou. The language features and architecture of b-prolog. *Theory and Practice of Logic Programming*, 12(1-2):189–218, January 2012.