Constraint Answer Set Solver EZCSP and Why Integration Schemas Matter

MARCELLO BALDUCCINI
Department of Decision & System Sciences
Saint Joseph’s University
(e-mail: marcello.balduccini@gmail.com)

YULIYA LIERLER
Computer Science Department
University of Nebraska at Omaha
(e-mail: ylierler@unomaha.edu)

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

Abstract

Researchers in answer set programming and constraint programming have spent significant efforts in the development of hybrid languages and solving algorithms combining the strengths of these traditionally separate fields. These efforts resulted in a new research area: constraint answer set programming. Constraint answer set programming languages and systems proved to be successful at providing declarative, yet efficient solutions to problems involving hybrid reasoning tasks. One of the main contributions of this paper is the first comprehensive account of the constraint answer set language and solver EZCSP, a mainstream representative of this research area that has been used in various successful applications. We also develop an extension of the transition systems proposed by Nieuwenhuis et al. in 2006 to capture Boolean satisfiability solvers. We use this extension to describe the EZCSP algorithm and prove formal claims about it. The design and algorithmic details behind EZCSP clearly demonstrate that the development of the hybrid systems of this kind is challenging. Many questions arise when one faces various design choices in an attempt to maximize system’s benefits. One of the key decisions that a developer of a hybrid solver makes is settling on a particular integration schema within its implementation. Thus, another important contribution of this paper is a thorough case study based on EZCSP, focused on the various integration schemas that it provides.

1 Introduction

Knowledge representation and automated reasoning are areas of Artificial Intelligence dedicated to understanding and automating various aspects of reasoning. Such traditionally separate fields of AI as answer set programming (ASP) (Niemelä, 1999; Marek and Ittetsuki, 1999; Brewka et al., 2011), propositional satisfiability (SAT) (Gomes et al.)
constraint (logic) programming (CSP/CLP) ([Rossi et al., 2008; Jaffar and Maher, 1994]) are all representatives of distinct directions of research in automated reasoning. The algorithmic techniques developed in subfields of automated reasoning are often suitable for distinct reasoning tasks. For example, ASP proved to be an effective tool for formalizing elaborate planning tasks, whereas CSP/CLP is efficient in solving difficult scheduling problems. However, when solving complex practical problems, such as scheduling problems involving elements of planning or defeasible statements, methods that go beyond traditional ASP and CSP are sometimes desirable. By allowing one to leverage specialized algorithms for solving different parts of the problem at hand, these methods may yield better performance than the traditional ones. Additionally, by allowing the use of constructs that more closely fit each sub-problem, they may yield solutions that conform better to the knowledge representation principles of flexibility, modularity, and elaboration tolerance. This has led, in recent years, to the development of a plethora of hybrid approaches that combine algorithms and systems from different AI subfields. Constraint logic programming ([Jaffar and Maher, 1994]), satisfiability modulo theories (SMT) ([Nieuwenhuis et al., 2006]), HEX-programs ([Eiter et al., 2005]), and VI-programs ([Calimeri et al., 2007]) are all examples of this current. Various projects have focused on the intersection of ASP and CSP/CLP, which resulted in the development of a new field of study, often called constraint answer set programming (CASP) ([Elkabani et al., 2004; Mellarkod et al., 2008; Gebser et al., 2008; Balduccini, 2009a; Drescher and Walsh, 2011; Lierler, 2014]).

Constraint answer set programming allows one to combine the best of two different automated reasoning worlds: (1) the non-monotonic modeling capabilities and SAT-like solving technology of ASP and (2) constraint processing techniques for effective reasoning over non-Boolean constructs. This new area has already demonstrated promising results, including the development of CASP solvers ACSOVLVR ([Mellarkod et al., 2008]), CLINGCON ([Gebser et al., 2009]), EZCSP ([Balduccini, 2009a]), IDP ([Wittocx et al., 2008]), INCA ([Drescher and Walsh, 2011]), DINGO ([Janhunen et al., 2011]), MINGO ([Lin et al., 2012]), ASPMT2SMT ([Bartholomew and Lee, 2014]), and EZSMT ([Susman and Lierler, 2016]). CASP opens new horizons for declarative programming applications. For instance, research by Balduccini (2011) on the design of CASP language EZCSP and on the corresponding solver, which is nowadays one of the mainstream representatives of CASP systems, yielded an elegant, declarative solution to a complex industrial scheduling problem.

Unfortunately, achieving the level of integration of CASP languages and systems requires nontrivial expertise in multiple areas, such as SAT, ASP and CSP. The crucial message transpiring from the developments in the CASP research area is the need for standardized techniques to integrate computational methods spanning these multiple research areas. We argue for undertaking an effort to mitigate the difficulties of designing hybrid reasoning systems by identifying general principles for their development and studying the implications of various design choices. Our work constitutes a step in this direction. Specifically, the main contributions of our work are:

1. The paper provides the first comprehensive account of the constraint answer set solver EZCSP ([Balduccini, 2009a]), a long-time representative of the CASP subfield. We define the language of EZCSP and illustrate its use on several examples. We also account for algorithmic and implementation details behind EZCSP.
2. To present the EZCSP algorithm and prove formal claims about the system, we develop an extension of the transition systems proposed by Nieuwenhuis et al. (2006) for capturing SAT/SMT algorithms. This extension is well-suited for formalizing the behavior of the EZCSP solver.

3. We also conduct a case study exploring a crucial aspect in building hybrid systems – the integration schemas of participating solving methods. This allows us to shed light on the costs and benefits of this key design choice in hybrid systems. For the case study, we use EZCSP as a research tool and study its performance with three integration schemas: “black-box”, “grey-box”, and “clear-box”. One of the main conclusions of the study is that there is no single choice of integration schema that achieves best performance in all cases. As such, the choice of integration schema should be made as easily configurable as it is the choice of particular branching heuristics in SAT or ASP solvers. The work on analytical and architectural aspects described in this paper shows how this can be achieved.

We begin this paper with a review of the ASP and CASP formalisms. In Section 3 we present the EZCSP language. In Section 4 we provide a broader context to our study by drawing a parallel between CASP and SMT solving. Then we review the integration schemas used in the design of hybrid solvers focusing on the schemas implemented in EZCSP. Section 5 provides a comprehensive account of algorithmic aspects of EZCSP. Section 7 introduces the details of the “integration schema” case study. In particular, it provides details on the application domains considered, namely, Weighted Sequence, Incremental Scheduling, and Reverse Folding. The section also discusses the variants of the encodings we compared. Experimental results and their analysis form Section 7. The conclusions are stated in Section 9.

Parts of this paper have been earlier presented at ASPOCP 2009 (Balduccini, 2009a) and at PADL 2012 (Balduccini and Lierler, 2012).

2 Preliminaries

2.1 Regular Programs

A regular (logic) program is a finite set of rules of the form

\[ a_0 \leftarrow a_1, \ldots, a_l, \text{not } a_{l+1}, \ldots, \text{not } a_m, \text{not not } a_{m+1}, \ldots, \text{not not } a_n, \]  

where \( a_0 \) is \( \perp \) (false) or an atom, and each \( a_i \) (\( 1 \leq i \leq n \)) is an atom so that \( a_i \neq a_j \) (\( 1 \leq i < j \leq l \)), \( a_i \neq a_j \) (\( l+1 \leq i < j \leq m \)), and \( a_i \neq a_j \) (\( m+1 \leq i < j \leq n \)). This is a special case of programs with nested expressions (Lifschitz et al. 1999). The expression \( a_0 \) is the head of a rule (1). If \( a_0 = \perp \), we often omit \( \perp \) from the notation. We call such rules denials. We call the right hand side of the arrow in (1) the body. If a body of a rule is empty, we call such rule a fact and omit the \( \leftarrow \) symbol. We also ignore the order of the elements in the rule. For example, rule \( a \leftarrow b, c \) is considered identical to \( a \leftarrow c, b \). If \( B \) denotes the body of (1), we write \( B^{\text{pos}} \) for the elements occurring in the positive part of the body, i.e., \( B^{\text{pos}} = \{ a_1, \ldots, a_l \} \). We frequently identify the body of (1)
with the conjunction of its elements (in which not not is dropped and not is replaced with the classical negation connective ¬):

\[ a_1 \land \ldots \land a_l \land \neg a_{l+1} \land \ldots \land \neg a_m \land a_{m+1} \land \ldots \land a_n. \]  

(2)

Similarly, we often interpret a rule (1) as a clause

\[ a_0 \lor \neg a_1 \lor \ldots \lor \neg a_l \lor a_{l+1} \lor \ldots \lor a_m \lor \neg a_{m+1} \lor \ldots \lor \neg a_n \]  

(3)

In the case when \( a_0 = \bot \) in (1), \( a_0 \) is absent in (3). Given a program \( \Pi \), we write \( \Pi^{cl} \) for the set of clauses of the form (3) corresponding to the rules in \( \Pi \).

**Answer sets** An alphabet is a set of atoms. The semantics of logic programs relies on the notion of answer sets, which are sets of atoms. A literal is an atom \( a \) or its negation \( \neg a \). We say that a set \( M \) of literals is complete over alphabet \( \sigma \) if, for any atom \( a \) in \( \sigma \), either \( a \in M \) or \( \neg a \in M \). It is easy to see how a set \( X \) of atoms over some alphabet \( \sigma \) can be identified with a complete and consistent set of literals over \( \sigma \) (an interpretation):

\[ \{ a \mid a \in X \} \cup \{ \neg a \mid a \in \sigma \setminus X \}. \]

We now restate the definition of an answer set due to Lifschitz et al. (1999) in a form convenient for our purposes. By \( At(\Pi) \) we denote the set of all atoms that occur in \( \Pi \). The reduct \( \Pi^X \) of a regular program \( \Pi \) with respect to set \( X \) of atoms over \( At(\Pi) \) is obtained from \( \Pi \) by deleting each rule (1) such that \( X \) does not satisfy its body (recall that we identify its body with (2)), and replacing each remaining rule (1) by \( a_0 \leftarrow B^{pos} \). A set \( X \) of atoms is an answer set of a regular program \( \Pi \) if it is subset minimal among the sets of atoms satisfying \( (\Pi^X)^{cl} \). For example, consider a program consisting of a single rule

\[ a \leftarrow \text{not not a}. \]

This program has two answer sets: set \( \emptyset \) and set \( \{ a \} \). Indeed, \( (\Pi^\emptyset)^{cl} \) is an empty set of clauses so that \( \emptyset \) is subset minimal among the sets of atoms that satisfies \( (\Pi^\emptyset)^{cl} \). On the other hand, \( (\Pi^\{a\})^{cl} \) consists of a single clause \( a \). Set \( \{ a \} \) is subset minimal among the sets of atoms that satisfies \( (\Pi^\{a\})^{cl} \).

A choice rule construct \( \{ a \} \leftarrow B \) (Niemelä and Simons, 2000) of the LParse language can be seen as an abbreviation for a rule \( a \leftarrow \text{not not a}, B \) (Ferraris and Lifschitz, 2005). We adopt this abbreviation in the rest of the paper.

**Example 1** Consider the regular program

\[
\begin{align*}
\{ \text{switch} \} : \\
\text{lightOn} & \leftarrow \text{switch, not am}. \\
& \leftarrow \text{not lightOn}. \\
\{ \text{am} \}. 
\end{align*}
\]

(4)

Intuitively, the rules of the program state the following:

- action \text{switch} is exogenous,
- \text{light} is on only if an action \text{switch} occurs during the non-am hours,
- it is impossible that \text{light} is not on (in other words, \text{light} must be on).
• it is either the case that these are am hours or not,

This program’s only answer set is \{switch, lightOn\}.

We now state an important result that summarizes the effect of adding denials to a program. For a set \(M\) of literals, by \(M^+\) we denote the set of positive literals in \(M\). For instance, \(\{a, c, \neg b\}^+ = \{a, c\}\).

**Theorem 1 (Proposition 2 from (Lifschitz et al., 1999))**

For a program \(\Pi\), a set \(\Gamma\) of denials, and a consistent and complete set \(M\) of literals over \(At(\Pi)\), \(M^+\) is an answer set of \(\Pi \cup \Gamma\) if and only if \(M^+\) is an answer set of \(\Pi\) and \(M\) is a model of \(\Gamma\) cl.

**Unfounded sets** For a literal \(l\), by \(\overline{l}\) we denote its complement. For a conjunction (disjunction) \(B\) of literals, \(\overline{B}\) stands for a disjunction (conjunction) of the complements of literals. For instance, \(a \land \neg b = \neg a \lor b\). We sometimes associate disjunctions and conjunctions of literals with the sets containing these literals. For example, conjunction \(\neg a \land b\) and disjunction \(\neg a \lor b\) are associated with the set \(\{\neg a, b\}\) of literals. By \(\text{ Bodies}(\Pi, a)\) we denote the set of the bodies of all rules of program \(\Pi\) with the head \(a\) (including the empty body that can be seen as \(\top\)).

A set \(U\) of atoms occurring in a program \(\Pi\) is **unfounded** (Van Gelder et al., 1991; Lee, 2005) on a consistent set \(M\) of literals with respect to \(\Pi\) if for every \(a \in U\) and every \(B \in \text{ Bodies}(\Pi, a)\), \(M \cap \overline{B} \neq \emptyset\) or \(U \cap B^\text{pos} \neq \emptyset\). We say that a consistent and complete set \(M\) of literals over \(At(\Pi)\) is a **model** of \(\Pi\) if it is a model of \(\Pi\) cl.

We now state a result that can be seen as an alternative way to characterize answer sets of a program.

**Theorem 2 (Theorem on Unfounded Sets from (Lee, 2005))**

For a program \(\Pi\) and a consistent and complete set \(M\) of literals over \(At(\Pi)\), \(M^+\) is an answer set of \(\Pi\) if and only if \(M\) is a model of \(\Pi\) and \(M\) contains no non-empty subsets unfounded on \(M\) with respect to \(\Pi\).

Theorem 2 is essential in understanding key features of modern answer set solvers. It provides a description of properties of answer sets that are utilized by so called “propagators” of solvers. Section 3 relies on these properties.

### 2.2 Logic Programs with Constraint Atoms

A constraint satisfaction problem (CSP) is defined as a triple \(\langle X, D, C \rangle\), where \(X\) is a set of variables, \(D\) is a domain – a (possibly infinite) set of values – and \(C\) is a set of constraints. Every constraint is a pair \(\langle t, R \rangle\), where \(t\) is an \(n\)-tuple of variables and \(R\) is an \(n\)-ary relation on \(D\). When arithmetic constraints are considered, it is common to replace explicit representations of relations as collections of tuples by arithmetic expressions. For instance, for a domain of three values \(\{1, 2, 3\}\) and binary-relation \(R\) consisting of ordered pairs \((1, 1), (2, 2),\) and \((3, 3)\), we can abbreviate the constraint \(\langle x, y, R \rangle\) by the expression \(x = y\). We follow this convention in the rest of the paper.

An **evaluation** of the variables is a function from the set of variables to the domain...
of values, \( \nu : X \rightarrow D \). An evaluation \( \nu \) satisfies a constraint \( \langle (x_1, \ldots, x_n), R \rangle \) if \( (v(x_1), \ldots, v(x_n)) \in R \). A solution is an evaluation that satisfies all constraints.

For a constraint \( c = \langle t, R \rangle \), where \( D \) is the domain of its variables and \( k \) is the arity of \( t \), we call the constraint \( \tau = \langle t, D^k \setminus R \rangle \) the complement of \( c \). Obviously, an evaluation of variables in \( t \) satisfies \( c \) if and only if it does not satisfy \( \tau \).

For a set \( M \) of literals and alphabet \( B \), by \( M \mid B \) we denote the set of literals over alphabet \( B \) in \( M \). For example, \( \{\neg a, b, c\} \mid \{a, b\} = \{\neg a, b\} \).

A logic program with constraint atoms (CA program) is a quadruple

\[ \langle \Pi, C, \gamma, D \rangle, \]

where

- \( C \) is an alphabet,
- \( \Pi \) is a regular logic program such that (i) \( a_0 \not\in C \) for every rule \( (1) \) in \( \Pi \) and (ii) \( C \subseteq \text{At}(\Pi) \),
- \( \gamma \) is a function from \( C \) to constraints, and
- \( D \) is a domain.

We refer to the elements of alphabet \( C \) as constraint atoms. We call all atoms occurring in \( \Pi \) but not in \( C \) regular. To distinguish constraint atoms from the constraints to which these atoms are mapped, we use bars to denote that an expression is a constraint atom. For instance, \( |x < 12| \) and \( |x \geq 12| \) denote constraint atoms. Consider alphabet \( C_1 \) that consists of these two constraint atoms and a function \( \gamma_1 \) that maps atoms in \( C_1 \) to constraints as follows: \( \gamma_1(|x < 12|) \) maps to an inequality \( x < 12 \), whereas \( \gamma_1(|x \geq 12|) \) maps to an inequality \( x \geq 12 \). Clearly, \( \gamma_1(|x < 12|) \) maps into an inequality \( x < 12 \); similarly \( \gamma_1(|x \geq 12|) \) maps into an inequality \( x < 12 \).

Example 2

Here we present a sample CA program

\[ \mathcal{P}_1 = \langle \Pi_1, C_1, \gamma_1, D_1 \rangle, \] (5)

where \( D_1 \) is a range of integers from 0 to 23 and \( \Pi_1 \) is a regular program

\begin{align*}
\{\text{switch}\}:
\text{lightOn} & \leftarrow \text{switch, not am}. \\
\text{not lightOn} & \leftarrow \text{not am}.
\{\text{am}\}:
\text{not am, } |x < 12| & \leftarrow \text{am, } |x \geq 12|.
\end{align*} (6)

The first four rules of \( \Pi_1 \) follow the lines of (3). The last two rules intuitively state that

- it is impossible that these are not \( am \) hours while variable \( x \) has a value less than 12,
- it is impossible that these are \( am \) hours while variable \( x \) has a value greater or equal to 12.

Note how \( x \) represents specific hours of a day. Also worth noting is the fact that \( x \) has a global scope. This is different from the traditional treatment of variables in CLP, Prolog, and ASP.
Let $\mathcal{P} = (\Pi, C, \gamma, D)$ be a CA program. By $\mathcal{V}_\mathcal{P}$ we denote the set of variables occurring in the constraints $\{\gamma(c) \mid c \in C\}$. For instance, $\mathcal{V}_{\mathcal{P}_1} = \{x\}$. By $\Pi[C]$ we denote $\Pi$ extended with choice rules $\{c\}$ for each constraint atom $c \in C$. We call program $\Pi[C]$ an asp-abstraction of $\mathcal{P}$. For example, an asp-abstraction $\Pi_1[C_1]$ of any CA program whose first two elements of its quadruple are $\Pi_1$ and $C_1$ consists of rules (6) and the following choice rules

\begin{align*}
\{\!\{x < 12\}\!\}, \\
\{\!\{x \geq 12\}\!\}.
\end{align*}

Let $M$ be a consistent set of literals over $At(\Pi)$. By $K_{\mathcal{P}, M}$ we denote the following constraint satisfaction problem

\[
\langle V, D, \{\gamma(c) \mid c \in M[C], c \in C\} \cup \{\gamma(c) \mid c \in M[c], c \in C\}, c \in C\rangle
\]

where $V$ is the set of variables occurring in the constraints of the last element of the triple above. We call this constraint satisfaction problem a csp-abstraction of $\mathcal{P}$ with respect to $M$. For instance, a csp-abstraction of $\mathcal{P}_1$ w.r.t. $\{\!\{x \geq 12\}\!\}, \neg\!\{\!\{x < 12\}\!\}, \text{lightOn}\}$, or $K_{\mathcal{P}_1, \{\!\{x \geq 12\}\!\}, \neg\!\{\!\{x < 12\}\!\}, \text{lightOn}\}$, is

\[
\langle \{x\}, D_1, \{x \geq 12\}\rangle.
\]

(7)

It is easy to see that $\mathcal{V}_\mathcal{P}$ consists of the variables that occur in a csp-abstractions of $\mathcal{P}$ w.r.t. any consistent sets of literals over $At(\Pi)$.

Let $\mathcal{P} = (\Pi, C, \gamma, D)$ be a CA program and $M$ be a consistent and complete set of literals over $At(\Pi)$. We say that $M$ is an answer set of $\mathcal{P}$ if

(a1) $M^+$ is an answer set of $\Pi[C]$ and

(a2) the constraint satisfaction problem $K_{\mathcal{P}, M}$ has a solution.

Let $\alpha$ be an evaluation from the set $\mathcal{V}_\mathcal{P}$ of variables to the set $D$ of values. We say that a pair $\langle M, \alpha \rangle$ is an extended answer set of $\mathcal{P}$ if $M$ is an answer set of $\mathcal{P}$ and $\alpha$ is a solution to $K_{\mathcal{P}, M}$.

Example 3

Consider sample CA program $\mathcal{P}_1 = (\Pi_1, C_1, \gamma_1, D_1)$ given in (5). Consistent and complete set

\[
M_1 = \{\text{switch}, \text{lightOn}, \neg\text{am}, \neg\!\{x < 12\}\!, \!\{x \geq 12\}\!\}\]

of literals over $At(\Pi_1)$ is such that $M_1^+$ is the answer set of $\Pi_1[C_1]$. The constraint satisfaction problem $K_{\mathcal{P}_1, M_1}$ is presented in (8). Pairs

\[
\langle M_1, x = 12\rangle
\]

and

\[
\langle M_1, x = 23\rangle
\]

are two among twelve extended answer sets of program (5).
2.3 CA Programs and Weak Answer Sets

In the previous section we introduced CA programs that capture programs that a CASP solver such as CLINGCON processes. The EZCSP solver interprets similar programs slightly differently. To illustrate the difference we introduce the notion of a weak answer set for a CA program and discuss the differences with earlier definition.

Let $\mathcal{P} = (\Pi, \mathcal{C}, \gamma, D)$ be a CA program and $X$ be a set of atoms over $At(\Pi)$. We say that $X$ is a weak answer set of $\mathcal{P}$ if

1. $X$ is an answer set of $\Pi [\mathcal{C}]$ and
2. the constraint satisfaction problem
   $$(\forall \mathcal{P}, D, \{\gamma(c)| c \in X_{\mathcal{C}}\})$$
   has a solution.

Let $\alpha$ be an evaluation from the set $\forall \mathcal{P}$ of variables to the set $D$ of values. We say that a pair $\langle X, \alpha \rangle$ is an extended weak answer set of $\mathcal{P}$ if $X$ is an answer set of $\mathcal{P}$ and $\alpha$ is a solution to (8).

The key difference between the definition of an answer set and a weak answer set of a CA program lies in their conditions (a2) and (w2). (It is obvious that we can always identify a complete and consistent set of literals with the set of its atoms.) To illustrate the difference between the two semantics, consider simple CA program:

$$\text{night} \leftarrow |x < 6|,\text{am} \leftarrow |x < 12|.$$ 

This program has three answer sets and four weak answer sets that we present in the following table.

<table>
<thead>
<tr>
<th>Answer Sets:</th>
<th>Weak Answer Sets:</th>
</tr>
</thead>
<tbody>
<tr>
<td>${\text{night}, \text{am},</td>
<td>x &lt; 6</td>
</tr>
<tr>
<td>${\neg \text{night}, \text{am}, \neg</td>
<td>x &lt; 6</td>
</tr>
<tr>
<td>${\neg \text{night}, \neg \text{am}, \neg</td>
<td>x &lt; 6</td>
</tr>
</tbody>
</table>

Note how the last weak answer set listed yields an unexpected solution, as it suggests that it is currently night but not am hours.

Another sample program is due to Sara Biavaschi and Agostino Dovier:

$$\leftarrow |x < 12|,\leftarrow |x > 10|.$$ 

This program has no answer sets, but has a weak answer set, $\emptyset$. Arguably, weak answer sets exhibit an agnostic attitude toward the values of variables associated with constraints that have no corresponding constraint atoms occurring in the answer sets.

This example is new to the online version of the paper. It substitutes the erroneous claim found in the TPLP version of the paper.
3 The EZCSP Language

The origins of the constraint answer set solver EZCSP and of its language go back to the development of an approach for integrating ASP and constraint programming, in which ASP is viewed as a specification language for constraint satisfaction problems (Balduccini, 2009a). In this approach, (i) ASP programs are written in such a way that some of their rules, and corresponding atoms found in their answer sets, encode the desired constraint satisfaction problems; (ii) both the answer sets and the solutions to the constraint problems are computed with arbitrary off-the-shelf solvers. This is achieved by an architecture that treats the underlying solvers as black boxes and relies on translation procedures for linking the ASP solver to constraint solver. The translation procedures extract from an answer set of an ASP program the constraints that must be satisfied and translate them into a constraint problem in the input language of the corresponding constraint solver. At the core of the EZCSP specification language is relation required, which is used to define the atoms that encode the constraints of the constraint satisfaction problem.

We start this section by defining the notion of propositional ez-programs and introducing their semantics via a simple mapping into CA programs under weak answer set semantics. Then, we move to describing the full language available to CASP practitioners in the EZCSP system. The tight relation between ez-programs and CA programs makes the following evident: although the origins of EZCSP are rooted in providing a simple, yet effective framework for modeling constraint satisfaction problems, the EZCSP language developed into a full-fledged constraint answer set programming formalism. This also yields another interesting observation: constraint answer set programming can be seen as a declarative modeling framework utilizing constraint satisfaction solving technology. The MiniZinc language (Marriott et al., 2008) is another remarkable effort toward a declarative modeling framework supported by the constraint satisfaction technology. It goes beyond the scope of this paper comparing the expressiveness of the constraint answer set programming and MiniZinc.

Syntax An ez-atom is an expression of the form

\[ \text{required}(\beta), \]

where \( \beta \) is an atom. Given an alphabet \( C \), the corresponding alphabet of ez-atoms \( C^{EZ} \) is obtained in a straightforward way. For instance, from an alphabet \( C_1 = \{ |x < 12|, |x \geq 12| \} \) we obtain \( C^{EZ}_1 = \{ \text{required}(|x < 12|), \text{required}(|x \geq 12|) \} \).

A (propositional) ez-program is a tuple

\[ (E, A, C, \gamma, D), \]

where

- \( A \) and \( C \) are alphabets so that \( A, C, C^{EZ} \) do not share the elements,
- \( E \) is a regular logic program so that \( \text{At}(E) = A \cup C^{EZ} \) and atoms from \( C^{EZ} \) only occur in the head of its rules,
- \( \gamma \) is a function from \( C \) to constraints, and
- \( D \) is a domain.
Semantics  We define the semantics of ez-programs via a mapping to CA programs under the weak answer set semantics. Let $\mathcal{E} = \langle E, A, C, \gamma, D \rangle$ be an ez-program. By $P_\mathcal{E}$ we denote the CA program

$$\langle \Pi, C, \gamma, D \rangle,$$

where $\Pi$ extends $E$ by two denials

$$\leftarrow \text{required}(\beta), \text{not } \beta$$
$$\leftarrow \text{not required}(\beta), \beta$$

for every ez-atom $\text{required}(\beta)$ occurring in $E$.\footnote{Formula (9) is an extension of the corresponding formula from the TPLP version of the paper, which only included the first of the two denials. The latest definition of the semantics of ez programs coincides with the semantics of these programs introduced in \citet{Balduccini:2009:ez}. The proof of this claim can be obtained in a straightforward way from the definition of reduct and its minimal models.} For a set $X$ of atoms over $\text{At}(E) \cup C$ and an evaluation $\alpha$ from the set $\mathcal{V}_P$ of variables to the set $D$ of values, we say that

- $X$ is an answer set of $E$ if $X$ is a weak answer set of $P_\mathcal{E}$;
- a pair $\langle X, \alpha \rangle$ is an extended answer set of $E$ if $\langle M, \alpha \rangle$ is an extended weak answer set of $P_\mathcal{E}$.

Example 4
We now illustrate the concept of an ez-program on our running example of the “light domain”. Let $A_1$ denote the alphabet $\{\text{switch, lightOn, am}\}$. Let $E_1$ be a collection of rules

$$\begin{align*}
\{\text{switch}\}:
\text{lightOn} & \leftarrow \text{switch, not am}.
\text{not lightOn} & \leftarrow \\
\{\text{am}\}:
\text{required}(|x \geq 12|) & \leftarrow \text{not am}.
\text{not required}(|x < 12|) & \leftarrow \text{am}.
\end{align*}$$

where $C_1^{\text{ez}}$ forms an alphabet of ez-atoms. Let $\mathcal{E}_1$ be an ez-program

$$\langle E_1, A_1, C_1, \gamma_1, D_1 \rangle.$$ 

The first member of the quadruple $P_\mathcal{E}$ is composed of the rules from (10) and of the denials

$$\leftarrow \text{required}(|x \geq 12|), \text{not } |x \geq 12|.$$  
$$\leftarrow \text{required}(|x < 12|), \text{not } |x < 12|.$$  
$$\leftarrow \text{not required}(|x \geq 12|), |x \geq 12|. $$  
$$\leftarrow \text{not required}(|x < 12|), |x < 12|. $$

Ez-program $\mathcal{E}_1$ has one answer set

$$N_1 = \{\text{switch, lightOn, required}(|x \geq 12|), |x \geq 12|\}$$

Pairs

$$\langle N_1, x = 12 \rangle$$

and $\langle N_1, x = 23 \rangle$ are two among twelve extended answer sets of ez-program $\mathcal{E}_1$.\footnote{Formula (13) is an extension of the corresponding formula from the TPLP version of the paper, which only included the first of the two denials. The latest definition of the semantics of ez programs coincides with the semantics of these programs introduced in \citet{Balduccini:2009:ez}. The proof of this claim can be obtained in a straightforward way from the definition of reduct and its minimal models.}
At the core of the EZCSP system is its solver algorithm (described in Section 5), which takes as an input a propositional ez-program and computes its answer sets. In order to allow for more compact specifications, the EZCSP system supports an extension of the language of propositional ez-programs, which we call EZ. The language is described by means of examples next. Its definition can be found in Appendix A. Also, the part of formalization of the Weighted Sequence domain presented in Section 6 illustrates the use of the so called reified constraints, which form an important modeling tool of the EZ language.

Example 5
In the EZ language, the ez-program \( \mathcal{E}_1 \) introduced in Example 3 is specified as follows:

\[
\begin{align*}
&\text{cspdomain}(fd) \\
&\text{cspvar}(x, 0, 23) \\
&\{\text{switch}\} \\
&\quad \text{lightOn} \leftarrow \text{switch}, \text{not am} \\
&\quad \quad \text{not lightOn} \\
&\quad \{\text{am}\} \\
&\quad \text{required}(x \geq 12) \leftarrow \text{not am} \\
&\quad \text{required}(x < 12) \leftarrow \text{am}.
\end{align*}
\]

The first rule specifies domain of possible csp-abstractions, which in this case is that of finite-domains. The second rule states that \( x \) is a variable over this domain ranging between 0 and 23. The rest of the program follows the lines of (13) almost verbatim.

It is easy to see that denial (9) poses the restriction on the form of the answer sets of ez-programs so that an atom of the form \( \text{required}(\beta) \) appears in an answer set if and only if an atom of the form \( \beta \) appears in it. Thus, when the EZCSP system computes answer sets for the EZ programs, it omits \( \beta \) atoms. For instance, for the program of this example EZCSP will output:

\[\{\text{cspdomain}(fd), \text{cspvar}(x, 0, 23), \text{required}(x \geq 12), \text{switch}, \text{lightOn}, x = 12\}\]

to encode extended answer set (14).

Example 6
The EZ language includes support for a number of commonly-used global constraints, such as \textit{all different} and \textit{cumulative} (more details in Appendix A). For example, a possible encoding of the classical “Send+More=Money” problem is:

\[
\begin{align*}
&\text{cspdomain}(fd). \\
&\text{cspvar}(s, 0, 9). \quad \text{cspvar}(e, 0, 9). \ldots \quad \text{cspvar}(y, 0, 9). \\
&\text{required}(s \times 1000 + e \times 100 + n \times 10 + d + m \times 1000 + o \times 100 + r \times 10 + e = m \times 10000 + o \times 1000 + n \times 100 + e \times 10 + y). \\
&\text{required}(s \neq 0). \quad \text{required}(m \neq 0). \\
&\text{required}(\text{all different}([s, e, n, d, m, o, r, y])).
\end{align*}
\]

As before, the first rule specifies the domain of possible csp-abstractions. The next set of rules specifies the variables and their ranges. The remaining rules state the main constraints of the problem. Of those, the final rule encodes an \textit{all different} constraint, which
informally requires all of the listed variables to have distinct values. The argument of the constraint is an extensional list of the variables of the CSP. An extensional list is a list that explicitly enumerates all of its elements.

A simple renaming of the variables of the problem allows us to demonstrate the intensional specification of lists:

\[
\text{cspdomain}\,(\text{fd}).
\]
\[
\text{cspvar}\,(v(s), 0, 9), \text{cspvar}\,(v(e), 0, 9), \ldots \text{cspvar}\,(v(y), 0, 9).
\]
\[
\text{required}\,(v(s) \ast 1000 + v(e) \ast 100 + v(n) \ast 10 + v(d) +
\hspace{1em} v(m) \ast 1000 + v(o) \ast 100 + v(r) \ast 10 + v(e) =
\hspace{1em} v(m) \ast 10000 + v(o) \ast 1000 + v(n) \ast 100 + v(e) \ast 10 + v(y)).
\]
\[
\text{required}\,(v(s) \neq 0). \text{required}\,(v(m) \neq 0).
\]
\[
\text{required}\,(\text{all different}(\{v/1\})).
\]

The argument of the global constraint in the last rule is intensional list \([v/1]\), which is a shorthand for the extensional list, \([v(d), v(e), v(m), v(n),\ldots]\), of all variables of the form \(v(\cdot)\).

**Example 7**

Consider a riddle:

There are either 2 or 3 brothers in the Smith family. There is a 3 year difference between one brother and the next (in order of age) for all pairs of brothers. The age of the eldest brother is twice the age of the youngest. The youngest is at least 6 years old.

Figure 1 presents the EZ program that captures the riddle. We refer to this program as \(P_1\). Note how this program contains non-constraint variables \(B, N, B1, B2, BE,\) and \(BY\). As explained in Appendix A, the grounding process that occurs in the EZCSP system transforms these rules into propositional (ground) rules using the same approach commonly applied to ASP programs. For instance, the last rule of program \(P_1\) results in three ground rules

\[
\text{required}\,(\text{age}(1) \geq 6) \leftarrow \text{index}(1), \text{youngest}_\text{brother}(1).
\]
\[
\text{required}\,(\text{age}(2) \geq 6) \leftarrow \text{index}(2), \text{youngest}_\text{brother}(2).
\]
\[
\text{required}\,(\text{age}(3) \geq 6) \leftarrow \text{index}(3), \text{youngest}_\text{brother}(3).
\]

The ez-program that corresponds to \(P_1\) has a unique extended answer set

\[
\langle\{\text{num}_\text{brothers}(3),
\text{cspvar}\,(\text{age}(1), 1, 80), \ldots, \text{cspvar}\,(\text{age}(3), 1, 80), \ldots\},
\{\text{age}(1) = 12, \text{age}(2) = 9, \text{age}(3) = 6\}\rangle.
\]

The extended answer set states that there are 3 brothers, of age 12, 9, and 6 respectively.

---

8 The reader may notice that the program features the use of arithmetic connectives both within terms and as full-fledged relations. Although, strictly speaking, separate connectives should be introduced for each type of usage, we abuse notation slightly and use context to distinguish between the two cases.
Constraint Answer Set Solver EZCSP and Why Integration Schemas Matter

There are either 2 or 3 brothers in the Smith family.

\[ \text{num brothers}(2) \leftarrow \neg \text{num brothers}(3). \]
\[ \text{num brothers}(3) \leftarrow \neg \text{num brothers}(2). \]

\[ \text{index}(1). \text{index}(2). \text{index}(3). \]
\[ \text{is brother}(B) \leftarrow \text{index}(B), \text{index}(N), \text{num brothers}(N), B \leq N. \]
\[ \text{eldest brother}(1). \]
\[ \text{youngest brother}(B) \leftarrow \text{index}(B), \text{num brothers}(B). \]

\[ \text{cspdomain}(fd). \]
\[ \text{cspvar}(\text{age}(B), 1, 80) \leftarrow \text{index}(B), \text{is brother}(B). \]

% 3 year difference between one brother and the next.
\[ \text{required}(\text{age}(B_1) - \text{age}(B_2) = 3)) \leftarrow \]
\[ \text{index}(B_1), \text{index}(B_2), \text{is brother}(B_1), \text{is brother}(B_2), B_2 = B_1 + 1. \]

% The eldest brother is twice as old as the youngest.
\[ \text{required}(\text{age}(BE) = \text{age}(BY) \times 2) \leftarrow \]
\[ \text{index}(BE), \text{index}(BY), \text{eldest brother}(BE), \text{youngest brother}(BY). \]

% The youngest is at least 6 years old.
\[ \text{required}(\text{age}(BY) \geq 6) \leftarrow \text{index}(BY), \text{youngest brother}(BY). \]

Figure 1. The EZ program for the riddle of Example 7

4 Satisfiability Modulo Theories and its Integration Schemas

We are now ready to draw a parallel between constraint answer set programming and satisfiability modulo theories. To do so, we first define the SMT problem by following the lines of (Nieuwenhuis et al., 2006, Section 3.1). A theory \( T \) is a set of closed first-order formulas. A CNF formula \( F \) (a set of clauses) over a fixed finite set of ground (variable-free) first-order atoms is \( T \)-satisfiable if there exists an interpretation, in first-order sense, that satisfies every formula in set \( F \cup T \). Otherwise, it is called \( T \)-unsatisfiable. Let \( M \) be a set of ground literals. We say that \( M \) is a \( T \)-model of \( F \) if

\( (m1) \) \( M \) is a model of \( F \) and
\( (m2) \) \( M \), seen as a conjunction of its elements, is \( T \)-satisfiable.

The SMT problem for a theory \( T \) is the problem of determining, given a formula \( F \), whether \( F \) has a \( T \)-model. It is easy to see that in the CASP problem, \( \Pi[C] \) in condition \( m1 \) plays the role of \( F \) in \( m1 \) in the SMT problem. At the same time, condition \( m2 \) is similar to condition \( m2 \).

Given this tight conceptual relation between the SMT and CASP formalisms, it is not surprising that solvers stemming from these different research areas share several design traits even though these areas have been developing to a large degree independently (CASP being a younger field). We now review major integration schemas/methods in SMT solvers by following (Nieuwenhuis et al., 2006, Section 3.2). During the review, we discuss how different CASP solvers account for one or another method. This discussion allows us to systematize design patterns of solvers present both in SMT and CASP so that their relation becomes clearer. Such a transparent view on architectures of solvers immediately translates...
findings in one area to the other. Thus, although the case study conducted as part of our research uses CASP technology only, we expect similar results to hold for SMT, and for the construction of hybrid automated reasoning methods in general. To the best of our knowledge there was no analogous effort – thorough evaluation of effect of integration schemas on performance of systems – in the SMT community.

In every approach discussed, a formula $F$ is treated as a satisfiability formula, where each atom is considered as a propositional symbol, forgetting about the theory $T$. Such a view naturally invites an idea of lazy integration: the formula $F$ is given to a SAT solver, if the solver determines that $F$ is unsatisfiable then $F$ has no $T$-model. Otherwise, a propositional model $M$ of $F$ found by the SAT solver is checked by a specialized $T$-solver, which determines whether $M$ is $T$-satisfiable. If so, then it is also a $T$-model of $F$, otherwise $M$ is used to build a clause $C$ that precludes this assignment, i.e., $M \not= C$ while $F \cup C$ has a $T$-model if and only if $F$ has a $T$-model. The SAT solver is invoked on an augmented formula $F \cup C$. This process is repeated until the procedure finds a $T$-model or returns unsatisfiable. Note how in this approach two automated reasoning systems – a SAT solver and a specialized $T$-solver – interleave: a SAT solver generates “candidate models” whereas a $T$-solver tests whether these models are in accordance with requirements specified by theory $T$. We find that it is convenient to introduce the following terminology for the future discussion: a base solver and a theory solver, where the base solver is responsible for generating candidate models and the theory solver is responsible for any additional testing required for stating whether a candidate model is indeed a solution. In this paper we refer to lazy evaluation as black-box to be consistent with the terminology often used in CASP.

It is easy to see how the black-box integration policy translates to the realm of CASP. Given a CA program $P$, an answer set solver serves the role of base solver by generating answer sets of the asp-abstraction of $P$ (that are “candidate answer sets” for $P$) and then uses a CLP/CSP solver as a theory solver to verify whether condition (a2) is satisfied on these candidate answer sets. Originally, constraint answer set solver EZCSP embraced the black-box integration approach in its design. 10 To solve a CASP problem via black-box approach, EZCSP offers a user various options for base and theory solvers. Table II shows some of the currently available solvers. The variety of possible configurations of EZCSP illustrates how black-box integration provides great flexibility in choosing underlying base and theory solving technology in addressing problems of interest. In principle, this approach allows for a simple integration of constraint programming systems that use MiniZinc and FlatZinc as their front-end description languages. Implementing support for this interface is a topic of future research.

The Davis-Putnam-Logemann-Loveland (DPLL) procedure (Davis et al., 1962) is a backtracking-based search algorithm for deciding the satisfiability of a propositional CNF formula. DPLL-like procedures form the basis for most modern SAT solvers as well as answer set solvers. If a DPLL-like procedure underlies a base solver in the SMT and CASP tasks then it opens a door to several refinements of black-box integration. We now describe these refinements.

10 Balduccini (2009a) refers to black-box integration of EZCSP as lightweight integration of ASP and constraint programming.

11 http://www.minizinc.org/.
Constraint Answer Set Solver EZCSP and Why Integration Schemas Matter

<table>
<thead>
<tr>
<th>Base Solvers</th>
<th>Theory Solvers</th>
</tr>
</thead>
<tbody>
<tr>
<td>SMODELS (Simons et al. 2002)</td>
<td>SICSTUS PROLOG (Carlsson and Mildner 2012)</td>
</tr>
<tr>
<td>CLASP (Gebser et al. 2007)</td>
<td>BPLOGO (Zhou 2012)</td>
</tr>
<tr>
<td>CMODELS (Giunchiglia et al. 2006)</td>
<td></td>
</tr>
</tbody>
</table>

Table 1. Base and theory solvers supported by EZCSP

In the *black-box* integration approach a base solver is invoked iteratively. Consider the SMT task: a CNF formula $F_{i+1}$ of the $i+1^{th}$ iteration to a SAT solver consists of a CNF formula $F_i$ of the $i^{th}$ iteration and an additional clause (or a set of clauses). Modern DPLL-like solvers commonly implement such technique as incremental solving. For instance, incremental SAT-solving allows the user to solve several SAT problems $F_1, \ldots, F_n$ one after another (using a single invocation of the solver), if $F_{i+1}$ results from $F_i$ by adding clauses. In turn, the solution to $F_{i+1}$ may benefit from the knowledge obtained during solving $F_1, \ldots, F_i$. Various modern SAT-solvers, including MINISAT (Eén and Biere 2005; Eén and Sörensson 2003), implement interfaces for incremental SAT solving. Similarly, the answer set solver CMODELS implements an interface that allows the user to solve several ASP problems $\Pi_1, \ldots, \Pi_n$ one after another, if $\Pi_{i+1}$ results from $\Pi_i$ by adding a set of denials. It is natural to utilize incremental DPLL-like procedures for enhancing the *black-box* integration protocol: we call this refinement *grey-box* integration. In this approach, rather than invoking a base solver from scratch, an incremental interface provided by a solver is used to implement the iterative process. CASP solver EZCSP implements *grey-box* integration using the above mentioned incremental interface by CMODELS.

Nieuwenhuis et al. (2006) also review such integration techniques used in SMT as *on-line* SAT solver and theory propagation. We refer to on-line SAT solver integration as *clear-box* here. In this approach, the $T$-satisfiability of the “partial” assignment is checked, while the assignment is being built by the DPLL-like procedure. This can be done fully eagerly as soon as a change in the partial assignment occurs, or with a certain frequency, for instance at some regular intervals. Once the inconsistency is detected, the SAT solver is instructed to backtrack. The theory propagation approach extends the *clear-box* technique by allowing a theory solver not only to verify that a current partial assignment is “$T$-consistent” but also to detect literals in a CNF formula that must hold given the current partial assignment.

The CASP solver CLINGCON exemplifies the implementation of the theory propagation integration schema in CASP. It utilizes answer set solver CLASP as the base solver and constraint processing system GECODE (Schulte and Stuckey, 2008) as the theory solver. The ACSOLVER and IDP systems are other CASP solvers that implement the theory propagation integration schema. In the scope of this work, the CASP solver EZCSP was extended to implement the *clear-box* integration schema using CMODELS. It is worth noting that all of the above approaches consider the theory solver as a black box, disregarding its internal structure and only accessing it through its external API. To the best of our knowledge, no systematic investigation exists of integration schemas that also take advantage of the internal structure of the theory solver.
An important point is due here. Some key details about the grey-box and clear-box integration schemas have been omitted in the presentation above for simplicity. To make these integration schemas perform efficiently, learning – a sophisticated solving technique stemming from SAT (Zhang et al., 2001) – is used to capture the information (explanation) retrieved due to necessity to backtrack upon theory solving. This information is used by the base solver to avoid similar conflicts. Section 5.2 presents the details on the integration schemas formally and points at the key role of learning.

5 The EZCSP Solver

In this section, we describe an algorithm for computing answer sets of CA programs. A specialization of this algorithm to ez-programs is used in the EZCSP system. For this reason, we begin by giving an overview of the architecture of the EZCSP system. We then describe the solving algorithm.

5.1 Architecture

![Architecture of the EZCSP system](image)

Figure 2. Architecture of the EZCSP system

Figure 2 depicts the architecture of the system, while the narrative below elaborates on the essential details. Both are focused on the functioning of the EZCSP system while employing the black-box integration schema.

The first step of the execution of EZCSP (corresponding to the Pre-processor component in the figure) consists in running a pre-processor, which translates an input EZ program into a syntactically legal ASP program. This is accomplished by replacing the occurrences of arithmetic functions and operators in \texttt{required}(\beta) atoms by auxiliary function symbols. For example, an atom \texttt{required}(v > 2) is replaced by \texttt{required} \(\texttt{gt}(v, 2)\). A similar process is also applied to the notation for the specification of lists. For instance, an atom
required(all different([x, y])) is translated into required(all different(list(x, y))). The Grounder component of the architecture transforms the resulting program into its propositional equivalent, a regular program, using an off-the-shelf grounder such as GRINGO (Gebser et al., 2007). This regular program is then passed to the EZCSP Solver component.

The EZCSP Solver component iterates ASP and constraint programming computations by invoking the corresponding components of the architecture. Specifically, the ASP Solver component computes an answer set of the regular program using an off-the-shelf ASP solver, such as CMODELS or CLASP. If an answer set is found, the EZCSP solver runs the CLP Translator component, which maps the csp-abstraction corresponding to the computed answer set to a Prolog program. The program is then passed to the CP Solver component, which uses the CLP solver embedded in a Prolog interpreter, e.g. SicStus or BProlog, to solve the CSP instance. For example, for the sample program presented in Example 5, the EZCSP system produces the answer set:

\{ cspdomain(fd), cspvar(x, 0, 23), required(x ≥ 12), switch, lightOn \}.

The csp-abstraction of the program with respect to this answer set is translated into a Prolog rule:

\[ solve([x, V_x]) : - V_x ≥ 0, V_x ≤ 23, V_x ≥ 12, labeling([V_x]). \]

In this case, the CLP solver embedded in the Prolog interpreter will find feasible assignments for variable \( V_x \). The head of the rule is designed to return a complete solution and to ensure that the variable names used in the EZ program are associated with the corresponding values. The interested reader can refer to (Balduccini, 2009a) for a complete description of the translation process.

Finally, the EZCSP Solver component gathers the solutions to the respective csp-abstraction and combines them with the answer set obtained earlier to form extended answer sets. Additional extended answer sets are computed iteratively by finding other answer sets and the solutions to the corresponding csp-abstractions.

### 5.2 Solving Algorithm

We are now ready to present our algorithm for computing answer sets of CA programs. In earlier work, Lierler (2014) demonstrated how the CASP language CLINGCON (Gebser et al., 2009) as well as the essential subset of the CASP language AC of ACSOLVER (Mel Hankok et al., 2008) are captured by CA programs. Based on those results, the algorithm described in this section can be immediately used as an alternative to the procedures implemented in systems CLINGCON and ACSOLVER.

Usually, software systems are described by means of pseudocode. The fact that EZCSP system follows an “all-solvers-in-one” philosophy combined with a variety of integration schemas complicates the task of describing it in this way. For example, one configuration

---

13 The ASP solver to be used can be specified by command-line options.

14 The Prolog interpreter is also selectable by command-line options.

15 For illustrative purposes, we show the EZ atom required\((x ≥ 12)\) in place of the ASP atom obtained from the preprocessing phase.
of EZCSP may invoke answer set solver CLASP via black-box integration for enumerating answer sets of an asp-abstraction of CA program, whereas another may invoke CMOD-ELS via grey-box integration for the same task. Thus, rather than committing ourselves to a pseudocode description, we follow a path pioneered by Nieuwenhuis et al. (2006). In their work, the authors devised a graph-based abstract framework for describing backtrack search procedures for Satisfiability and SMT. Lierler (2014) designed a similar abstract framework that captures the EZCSP algorithm in two cases: (a) when EZCSP invokes answer set solver SMODELS via black-box integration for enumerating answer sets of asp-abstraction program, and (b) when EZCSP invokes answer set solver CLASP via black-box integration.

In the present paper we introduce a graph-based abstract framework that is well suited for capturing the similarities and differences of the various configurations of EZCSP stemming from different integration schemas. The graph-based representation also allows us to speak of termination and correctness of procedures supporting these configurations. In this framework, nodes of a graph representing a solver capture its possible “states of computation”, while edges describe the possible transitions from one state to another. It should be noted that the graph representation is too high-level to capture some specific features of answer set solvers or constraint programming tools used within different EZCSP configurations. For example, the graph incorporates no information on the heuristic used to select a literal upon which a decision needs to be made. This is not an issue, however: stand alone answer set solvers have been analyzed and compared theoretically in the literature (Anger et al., 2006), (Giunchiglia et al., 2008) (Lierler and Truszczyński, 2011) as well as empirically in biennial answer set programming competitions (Gebser et al., 2007), (Denecker et al., 2009), (Calimeri et al., 2011). At the same time, EZCSP treats constraint programming tools as “black-boxes” in all of its configurations.

5.2.1 Abstract EZCSP

Before introducing the transition system (graph) capable of capturing a variety of EZCSP procedures, we start by developing some required terminology. To make this section more self-contained we also restate some notation and definitions from earlier sections. Recall that for a set $M$ of literals, by $M^+$ we denote the set of positive literals in $M$. For a CA program $P = \langle \Pi, C, \gamma, D \rangle$, a consistent and complete set $M$ of literals over $At(\Pi)$ is an answer set of $P$ if

(a1) $M^+$ is an answer set of $\Pi[C]$ and

(a2) the constraint satisfaction problem $K_{P,M}$ has a solution.

As noted in Section 4.3 we can view denials as clauses. Given a denial $G$, by $G^{cl}$ we will denote a clause that corresponds to $G$, e.g., $(\leftarrow \text{not } pm)^{cl}$ denotes a clause $pm$. We may sometime abuse the notation and refer to a clause as if it were a denial. For instance, a clause $pm$ may denote a denial $\leftarrow \text{not } pm$.

We now introduce notions for CA programs that parallel ”entailment” for the case of classical logic formulas. Let $P = \langle \Pi, C, \gamma, D \rangle$ be a CA program. We say that $P$ asp-entails a denial $G$ over $At(\Pi)$ when for every complete and consistent set $M$ of literals over $At(\Pi)$ such that $M^+$ is an answer set of $\Pi[C]$, $M$ satisfies $G^{cl}$. In other words, a denial is asp-
entailed if any set of literals that satisfies the condition [49] of the answer set definition is such that it satisfies this denial. CA program $\mathcal{P}$ cp-entails a denial $G$ over $At(\Pi)$ when (i) for every answer set $M$ of $\mathcal{P}$, $M$ satisfies $G^{cl}$ and (ii) there is a complete and consistent set $N$ of literals over $At(\Pi)$ such that $N^{+}$ is an answer set of $\Pi[C]$ and $N$ does not satisfy $G$. Notice that if a denial $G$ is such that a CA program $\mathcal{P}$ cp-entails $G$, then $\mathcal{P}$ does not asp-entail $G$. We say that $\mathcal{P}$ entails a denial $G$ when $\mathcal{P}$ either asp-entails or cp-entails $G$. For a consistent set $N$ of literals over $At(\Pi)$ and a literal $l$, we say that $\mathcal{P}$ asp-entails $l$ with respect to $N$, if for every complete and consistent set $M$ of literals over $At(\Pi)$ such that $M^{+}$ is an answer set of $\Pi[C]$ and $N \subseteq M$, $l \in M$.

Example 8
Recall program $\mathcal{P}_1 = (\Pi_1, C_1, \gamma_1, D_1)$ from Example 4. It is easy to check that denial $\nleftarrow \text{not lightOn}$. (or, in other words clause lightOn) is asp-entailed by $\mathcal{P}_1$. Also, $\mathcal{P}_1$ asp-entails literals switch and $\neg\text{am}$ with respect to set $\{\text{lightOn}\}$ (and also with respect to $\emptyset$).

Let regular program $\Pi_2$ extend program $\Pi_1$ from Example 4 by rules

\[
\begin{align*}
\{pm\}, \\
\leftarrow \text{not pm, } |x| \geq 12, \\
\leftarrow |x| < 12.
\end{align*}
\]

Consider a CA program $\mathcal{P}_2$ that differs from $\mathcal{P}_1$ only by substituting its first member $\Pi_1$ of quadruple $(\Pi_1, C_1, \gamma_1, D_1)$ by $\Pi_2$. Denial $\nleftarrow \text{not pm}$ (or clause pm) is cp-entailed by $\mathcal{P}_2$. Indeed, the only answer set of this program is $\{pm, \neg|x| < 12, |x| \geq 12\}$. This set satisfies $\nleftarrow \text{not pm}^{cl}$, in other words, clause pm. Consider set $\{\neg pm, \neg|x| < 12, |x| \geq 12\}$ that does not satisfy clause pm. Set of atoms $\{\neg pm, \neg|x| < 12, |x| \geq 12\}^{+} = \{|x| \geq 12\}$ is an answer set of $\Pi_2[C_1]$.

For a CA program $\mathcal{P} = (\Pi, C, \gamma, D)$ and a set $\Gamma$ of denials, by $\mathcal{P}[\Gamma]$ we denote the CA program $(\Pi \cup \Gamma, C, \gamma, D)$. The following propositions capture important properties underlying the introduced entailment notions.

**Proposition 1**
For a CA program $\mathcal{P} = (\Pi, C, \gamma, D)$ and a set $\Gamma$ of denials over $At(\Pi)$ if $\mathcal{P}$ asp-entails every denial in $\Gamma$ then (i) programs $\Pi[C]$ and $(\Pi \cup \Gamma)[C]$ have the same answer sets; (ii) CA programs $\mathcal{P}$ and $\mathcal{P}[\Gamma]$ have the same answer sets.

**Proof**
We first show that condition (i) holds. From Theorem II and the fact that $\mathcal{P}$ asp-entails every denial in $\Gamma$ it follows that programs $\Pi[C]$ and $(\Pi \cup \Gamma)[C]$ have the same answer sets. Condition (ii) follows from (i) and the fact that $K_{\mathcal{P}, M} = K_{\mathcal{P}[\Gamma], M}$ for any answer set $M$ of $\Pi[C]$ (and, consequently, for $(\Pi \cup \Gamma)[C]$).

**Proposition 2**
For a CA program $\mathcal{P} = (\Pi, C, \gamma, D)$ and a set $\Gamma$ of denials over $At(\Pi)$ if $\mathcal{P}$ cp-entails every denial in $\Gamma$ then CA programs $\mathcal{P}$ and $\mathcal{P}[\Gamma]$ have the same answer sets.
Let \( \mathcal{P} \) be a CA program \( \langle \Pi, \mathcal{C}, \gamma, D \rangle \). It is easy to see that (a) \( \Pi[\mathcal{C}] \cup \Gamma = (\Pi \cup \Gamma)[\mathcal{C}] \) and (b) \( K_{\mathcal{P}, \mathcal{M}} = K_{\mathcal{P}[\Gamma], \mathcal{M}} \).

Right-to-left: Take \( M \) to be an answer set of \( \mathcal{P} \). By the definition of an answer set, (i) \( M^+ \) is an answer set of \( \Pi[\mathcal{C}] \) and (ii) the constraint satisfaction problem \( K_{\mathcal{P}, \mathcal{M}} \) has a solution. Since \( \mathcal{P} \) cp-entails every denial in \( \Gamma \), we conclude that \( M \) is a model of \( \Gamma^{cl} \). By Theorem \( \Pi \), \( M^+ \) is an answer set of \( \Pi[\mathcal{C}] \cup \Gamma \). From (a) and (b) we derive that \( M \) is an answer set of \( \mathcal{P}[\Gamma] \).

Left-to-right: Take \( M \) to be an answer set of \( \mathcal{P}[\Gamma] \). By the definition of an answer set, (i) \( M^+ \) is an answer set of \( (\Pi \cup \Gamma)[\mathcal{C}] \) and (ii) the constraint satisfaction problem \( K_{\mathcal{P}[\Gamma], \mathcal{M}} \) has a solution. From (i) and (a) it follows that \( M^+ \) is an answer set of \( \Pi[\mathcal{C}] \cup \Gamma \). By Theorem \( \Pi \), \( M^+ \) is an answer set of \( \Pi[\mathcal{C}] \). By (b) and (ii) we derive that, \( M \) is an answer set of \( \mathcal{P} \).

Proposition 3

For a CA program \( \mathcal{P} = \langle \Pi, \mathcal{C}, \gamma, D \rangle \) and a set \( \Gamma \) of denials over At(\( \Pi \)) if \( \mathcal{P} \) entails every denial in \( \Gamma \) then (i) every answer set of \( (\Pi \cup \Gamma)[\mathcal{C}] \) is also an answer set of \( \Pi[\mathcal{C}] \); (ii) CA programs \( \mathcal{P} \) and \( \mathcal{P}[\Gamma] \) have the same answer sets.

Proof

Condition (i) follows from Theorem \( \Pi \) and the fact that \( (\Pi)[\mathcal{C}] \) and \( (\Pi \cup \Gamma)[\mathcal{C}] \) only differ in denials.

We now show that condition (ii) holds. Set \( \Gamma \) is composed of two disjoint sets \( \Gamma_1 \) and \( \Gamma_2 \) (i.e., \( \Gamma = \Gamma_1 \cup \Gamma_2 \)), where \( \Gamma_1 \) is the set of all denials that are asp-entailed by \( \mathcal{P} \) and \( \Gamma_2 \) is the set of all denials that are cp-entailed by \( \mathcal{P} \). By Proposition \( \Pi \) (ii), CA programs \( \mathcal{P} \) and \( \mathcal{P}[\Gamma_1] \) have the same answer sets. By Proposition \( \Pi \), CA programs \( \mathcal{P}[\Gamma_1] \) and \( \mathcal{P}[\Gamma_1 \cup \Gamma_2] \) have the same answer sets. It immediately follows that CA programs \( \mathcal{P} \) and \( \mathcal{P}[\Gamma_1 \cup \Gamma_2] \) have the same answer sets.

For an alphabet \( \sigma \), a record relative to \( \sigma \) is a sequence \( M \) composed of distinct literals over \( \sigma \) or symbol \( \bot \), some literals are possibly annotated by the symbol \( \Delta \), which marks them as decision literals such that:

1. the set of literals in \( M \) is consistent or \( M = M'\bot \), where the set of literals in \( M' \) is consistent and contains \( \bot \),
2. if \( M = M'\bot M'' \), then neither \( l \) nor its dual \( \bot \) is in \( M' \), and
3. if \( \bot \) occurs in \( M \), then \( M = M'\bot \) and \( M' \) does not contain \( \bot \).

We often identify records with the set of its members disregarding annotations.

For a CA program \( \mathcal{P} = \langle \Pi, \mathcal{C}, \gamma, D \rangle \), a state relative to \( \mathcal{P} \) is either a distinguished state Failstate or a triple \( M||\Gamma||\Lambda \) where \( M \) is a record relative to \( At(\Pi) \); \( \Gamma \) and \( \Lambda \) are each a set of denials that are entailed by \( \mathcal{P} \). Given a state \( M||\Gamma||\Lambda \) if neither a literal \( l \) nor \( \bot \) occurs in \( M \), then \( l \) is unassigned by the state; if \( \bot \) does not occur in \( M \) as well as for any atom \( a \) it is not the case that both \( a \) and \( \neg a \) occur in \( M \), then this state is consistent. For a state \( M||\Gamma||\Lambda \), we call \( M \), \( \Gamma \), and \( \Lambda \) the atomic, permanent, and temporal parts of the state, respectively. The role of the atomic part of the state is to track decisions (choices) as well
Figure 3. The transition rules of the graph $\text{EZ}_P$.

Figure 4. Sample path in graph $\text{EZ}_{P_1}$.

We now define a graph $\text{EZ}_P$ for a CA program $\mathcal{P}$. Its nodes are the states relative to $\mathcal{P}$.
These nodes intuitively correspond to states of computation. The edges of the graph $EZ_P$ are specified by nine transition rules presented in Figure 3. These rules correspond to possible operations by the $EZCSP$ system that bring it from one state of computation to another. A path in the graph $EZ_P$ is a description of a process of search for an answer set of $P$. The process is captured via applications of transition rules. Theorem 3 introduced later in this section makes this statement precise.

**Example 9**
Recall CA program $P_1 = \langle \Pi_1, C_1, \gamma_1, D_1 \rangle$ introduced in Example 2. Figure 4 presents a sample path in $EZ_{P_1}$ with every edge annotated by the name of a transition rule that justifies the presence of this edge in the graph.

Now we turn our attention to an informal discussion of the role of each of the transition rules in $EZ_P$.

### 5.2.2 Informal account on transition rules

We refer to the transition rules Decide, Fail, Backtrack, ASP-Propagate, CP-Propagate of the graph $EZ_P$ as basic.

The unique feature of basic rules is that they only concern the atomic part of a state. Consider a state $S = M \| \Gamma \| \Lambda$. An application of any basic rule results in a state whose permanent and temporal parts remain unchanged, i.e., $\Gamma$ and $\Lambda$ respectively (unless it is the case of Fail).

**Decide** An application of the transition rule Decide to $S$ results in a state whose atomic part has the form $M l^\Delta$. Intuitively this rule allows us to pursue evaluation of assignments that assume value of literal $l$ to be true. The fact that this literal is marked by $\Delta$ suggests that we can still reevaluate this assumption in the future, in other words to backtrack on this decision.

**Fail** The transition rule Fail specifies the conditions on atomic part $M$ of state $S$ suggesting that Failstate is reachable from $M$. Intuitively, if our computation brought us to such a state transition to Failstate confirms that there is no solution to the problem.

**Backtrack** The transition rule Backtrack specifies the conditions on atomic part of the state suggesting when it is time to backtrack and what the new atomic part of the state is after backtracking. Rules Fail and Backtrack share one property: they are applicable only when states are inconsistent.

**ASP-Propagate** The transition rule ASP-Propagate specifies the condition under which a new literal $l$ (without a decision annotation) is added to an atomic part. Such rules are commonly called propagators. Note that the condition of ASP-Propagate

$$P[\Gamma \cup \Lambda] \text{ asp-entails } l \text{ with respect to } M$$

(14)

is defined over a program extended by permanent and temporal part. This fact illustrates the role of these entities. They carry extra information aquired/learnt during the computation.
Also condition (23) is semantic. It refers to the notion of asp-entailment, which is defined by a reference to the semantics of a program. Propagators used by software systems typically use syntactic conditions, which are easy to check by inspecting syntactic properties of a program. Later in this section we present instances of such propagators, in particular, propagators that are used within the EZCSP solver.

CP-Propagate The transition rule CP-Propagate specifies the condition under which symbol ⊥ is added to an atomic part. Thus it leads to a state that is inconsistent suggesting that the search process is either ready to fail or to backtrack. The condition of CP-Propagate

\[ K, M \] has no solution

represents a decision procedure that establishes whether the CSP problem \( K, M \) has solutions or not.

We now turn our attention to non-basic rules that concern permanent and temporal parts of the states of computation.

Learn Recall the definition of the transition rule Learn

\[ M||\Gamma||\Lambda \Rightarrow M||\Gamma \cup \{ R \}||\Lambda \quad \text{if} \quad \mathcal{P}[\Gamma \cup \Lambda] \text{ entails denial } R \text{ and } R \notin \Gamma \cup \Lambda \]

An application of this rule to a state \( M||\Gamma||\Lambda \), results in a state whose atomic and temporal parts stay unchanged. The permanent part is extended by a denial \( R \). Intuitively the effect of this rule is such that from this point of computation the “permanent” denial becomes effectively a part of the program being solved. This is essential for two reasons. First, if the learnt denial \( R \) is cp-entailed then \( \Pi \cup \Gamma \cup \Lambda \) and \( \Pi \cup \Gamma \cup \Lambda \cup \{ R \} \) are programs with different answer sets. In turn, the rule ASP-Propagate may be applicable to some state \( N||\Gamma \cup \{ R \}||\Lambda \) and not to \( N||\Gamma||\Lambda \). Similarly, due to the fact that only “syntactic” instances of ASP-Propagate are implemented in solvers, the previous statement also holds for the case when \( R \) is asp-entailed.

Learn\textsuperscript{†} The role of the transition rule Learn\textsuperscript{†} is similar to that of Learn, but the learnt denials by this rule are not meant to be preserved permanently in the computation.

Restart and Restart\textsuperscript{†} The transition rule Restart allows the computation to start from “scratch” with respect to atomic part of the state. The transition rule Restart\textsuperscript{†} forces the computation to start from “scratch” with respect to not only atomic part of the state but also all temporally learnt denials. These restart rules are essential in understanding the key differences between various integration strategies that are of focus in this paper.

5.2.3 Formal properties of EZ\textsubscript{P} We call the state \( \emptyset || \emptyset || \emptyset \) — initial. We say that a node in the graph is semi-terminal if no rule other than Learn, Learn\textsuperscript{†}, Restart, Restart\textsuperscript{†} is applicable to it (or, in other words, if no single basic rule is applicable to it). We say that a path in EZ\textsubscript{P} is restart-safe when, prior to any edge \( e \) due to an application of Restart or Restart\textsuperscript{†} on this path, there is an edge \( e' \) due
to an application of Learn such that: (i) edge $e'$ precedes $e$; (ii) $e'$ does not precede any other edge $e'' \neq e$ due to Restart or Restart$^t$. We say that a restart-safe path $t$ is maximal if (i) the first state in $t$ is an initial state, and (ii) $t$ is not a subpath of any restart-safe path $t' \neq t$.

**Example 10**

Recall CA program $P_1 = \langle \Pi_1, C_1, \gamma_1, D_1 \rangle$ introduced in Example 3. Trivially a sample path in $EZP_1$ in Figure 3 is a restart-safe path. A nontrivial example of restart-safe path in $EZP_1$ follows

$$
\emptyset || \emptyset || \emptyset \quad \xrightarrow{Learn} \\
\emptyset || \{\leftarrow \text{not switch}\} || \emptyset \quad \xrightarrow{ASP-Propagate} \\
\text{lightOn} || \{\leftarrow \text{not switch}\} || \emptyset \quad \xrightarrow{Restart} \\
\emptyset || \{\leftarrow \text{not switch}\} || \emptyset.
$$

Similarly, a path that extends the path above as follows

$$
\emptyset || \emptyset || \emptyset \quad \xrightarrow{Learn} \\
\emptyset || \{\leftarrow \text{not switch}, \leftarrow \text{am}\} || \emptyset \quad \xrightarrow{ASP-Propagate} \\
\text{lightOn} || \{\leftarrow \text{not switch}, \leftarrow \text{am}\} || \emptyset \quad \xrightarrow{Restart} \\
\emptyset || \{\leftarrow \text{not switch}, \leftarrow \text{am}\} || \emptyset
$$

is restart-safe.

A simple path in $EZP_1$ that is not restart-safe

$$
\emptyset || \emptyset || \emptyset \quad \xrightarrow{ASP-Propagate} \\
\text{lightOn} || \emptyset || \emptyset \quad \xrightarrow{Restart} \\
\emptyset || \emptyset || \emptyset.
$$

Indeed, condition (i) of the restart-safe definition does not hold. Another example of a not restart-safe path is a path that extends path (15) as follows

$$
\text{lightOn} || \{\leftarrow \text{not switch}\} || \emptyset \quad \xrightarrow{Restart} \\
\emptyset || \{\leftarrow \text{not switch}\} || \emptyset.
$$

Indeed, condition (ii) of the restart-safe definition does not hold for the second occurrence of the Restart edge.
The following theorem captures key properties of the graph $EZ_P$. They suggest that the graph can be used for deciding whether a program with constraint atoms has an answer set.

**Theorem 3**

For any CA program $P$:

(a) every restart-safe path in $EZ_P$ is finite, and any maximal restart-safe path ends with state that is semi-terminal,

(b) for any semi-terminal state $M || \Gamma || \Lambda$ of $EZ_P$ reachable from initial state, $M$ is an answer set of $P$,

(c) state Failstate is reachable from initial state in $EZ_P$ by a restart-safe path if and only if $P$ has no answer set.

On the one hand, part (a) of Theorem 3 asserts that, if we construct a restart-safe path from initial state, then some semi-terminal state is eventually reached. On the other hand, parts (b) and (c) assert that, as soon as a semi-terminal state is reached by following any restart-safe path, the problem of deciding whether CA program $P$ has answer sets is solved.

Section 5.3 describes the varying configurations of the $EZCSP$ system.

**Example 11**

Recall `Example 9`. Since the last state in the sample path presented in Figure (4) is semi-terminal, Theorem 3 asserts that the set of literals composed of the elements of this semi-terminal state forms the answer set of CA program $P_1$. Indeed, this set coincides with the answer set $M_1$ of $P_1$ presented in `Example 3`.

In our discussion of the transition rule $ASP-Propagate$ we mentioned how the $EZCSP$ solver accounts only for some transitions due to this rule. Let $P = \langle \Pi, C, \gamma, D \rangle$ be a CA program. By $EZSM_P$ we denote an edge-induced subgraph of $EZ_P$, where we drop the edges that correspond to the application of transition rules $ASP-Propagate$ not accounted by the following two transition rules (propagators) $Unit Propagate$ and $Unfounded$:

**Unit Propagate**:

\[
M || \Gamma || \Lambda \Rightarrow M l || \Gamma || \Lambda \quad \begin{cases} 
C \lor l \in (\Pi[C] \cup \Gamma \cup \Lambda)^c, \\
M \text{ is consistent,} \\
M \models \overline{C}
\end{cases}
\]

**Unfounded**:

\[
M || \Gamma || \Lambda \Rightarrow M l || \Gamma || \Lambda \quad \begin{cases} 
M \text{ is consistent, and there is literal } l \text{ so that} \\
\overline{l} \in U \text{ for a set } U, \text{ which is} \\
\text{unfounded on } M \text{ w.r.t. } \Pi[C] \cup \Gamma \cup \Lambda
\end{cases}
\]

These two propagators rely on properties that can be checked by efficient procedures. The conditions of these transition rules are such that they are satisfied only if $P[\Gamma \cup \Lambda]$ asp-entails $l$ w.r.t. $M$. In other words, the transition rules $Unit Propagate$ or $Unfounded$ are applicable only in states where $ASP-Propagate$ is applicable. The other direction is not true. Theorem 3 holds if we replace $EZ_P$ by $EZSM_P$ in its statement. The proof of this theorem relies on the statement of Theorem 2, and is given at the end of this subsection. Graph $EZSM_P$ is only one of the possible subgraphs of the generic graph $EZ_P$ that share its key properties stated in Theorem 3. These properties show that graph $EZSM_P$ gives rise to a class of correct algorithms for computing answer sets of programs with constraints.
It provides a proof of correctness of every CASP solver in this class and a proof of termination under the assumption that restart-safe paths are considered by a solver. Note how much weaker propagators, such as Unit Propagate and Unfounded, than ASP-Propagate are sufficient to ensure the correctness of respective solving procedures. We picked the graph \( EZSM_P \) for illustration as it captures the essential propagators present in modern (constraint) answer set solvers and allows a more concrete view on the \( EZP \) framework. Yet the goal of this work is not to detail the variety of possible propagators of (constraint) answer set solvers but master the understanding of hybrid procedures that include this technology. Therefore in the rest of this section we turn our attention back to the \( EZP \) graph and use this graph to formulate black-box, grey-box, and clear-box configurations of the CASP solver EZCSP.

The rest of this subsection presents a proof of Theorem 3 as well as a proof of the similar theorem for the graph \( EZSM_P \).

Proof of Theorem 3
(a) Let \( P \) be a CA program \( \langle \Pi, C, \gamma, D \rangle \).

We first show that any path in \( EZ_P \) that does not contain \textit{Restart} or \textit{Restart} edges is finite. We name this statement \textit{Statement 1}.

Consider any path \( t \) in \( EZ_P \) that does not contain \textit{Restart} or \textit{Restart} edges.

For any list \( N \) of literals, by \(|N|\) we denote the length of \( N \). Any state \( M||\Gamma||\Lambda \) has the form \( M_0 l_1^\alpha M_1 \ldots l_p^\alpha M_p||\Gamma||\Lambda \), where \( l_1^\alpha \ldots l_p^\alpha \) are all decision literals of \( M \); we define \( \alpha(M||\Gamma||\Lambda) \) as the sequence of nonnegative integers \(|M_0|,|M_1|,\ldots,|M_p|\), and \( \alpha(\text{Failstate}) = \infty \). For any two states, \( S \) and \( S' \), of \( EZ_P \), we understand \( \alpha(S) < \alpha(S') \) as the lexicographical order. We note that, for any state \( M||\Gamma||\Lambda \), the value of \( \alpha \) is based only on the first component, \( M \), of the state. Second, there is a finite number of distinct values of \( \alpha \) for the states of \( EZ_P \) due to the fact that there is a finite number of distinct \( M \)'s over \( P \). We now define relation \textit{smaller} over the states of \( EZ_P \). We say that state \( M||\Gamma||\Lambda \) is \textit{smaller} than state \( M'||\Gamma'||\Lambda' \) when either

1. \( \Gamma \subseteq \Gamma' \), or
2. \( \Gamma = \Gamma' \), and \( \Lambda \subset \Lambda' \), or
3. \( \Gamma = \Gamma' \), \( \Lambda = \Lambda' \), and \( \alpha(M||\Gamma||\Lambda) < \alpha(M'||\Gamma'||\Lambda') \).

It is easy to see that this relation is anti-symmetric and transitive.

By the definition of the transition rules of \( EZ_P \), if there is an edge from \( M||\Gamma||\Lambda \) to \( M'||\Gamma'||\Lambda' \) in \( EZ_P \) formed by any basic transition rule or rules \textit{Learn} or \textit{Learn}'\, then \( M||\Gamma||\Lambda \) is smaller than state \( M'||\Gamma'||\Lambda' \). Observe that (i) there is a finite number of distinct values of \( \alpha \), and (ii) there is a finite number of distinct denials entailed by \( P \). Then, it follows that there is only a finite number of edges in \( t \), and, thus, Statement 1 holds.

We call a subpath from state \( S \) to state \( S' \) of some path in \( EZ_P \) restarting when (i) an edge that follows \( S \) is due to the application of rule \textit{Learn}, (ii) an edge leading to \( S' \) is due to the application of rule \textit{Restart} or \textit{Restart}' , and (iii) on this subpath, there are no other edges due to applications of \textit{Learn}, \textit{Restart}' , or \textit{Restart}, but the ones mentioned above. Using Statement 1, it follows that any restarting subpath is finite.

Consider any restart-safe path \( r \) in \( EZ_P \). We construct a path \( r' \) by dropping some finite fragments from \( r \). This is accomplished by replacing each restarting subpath of \( r \) from
state $S$ to state $S'$ by an edge from $S$ to $S'$ that we call \textit{Artificial}. It is easy to see that an edge in $r'$ due to \textit{Artificial} leads from a state of the form $M||\Gamma||\Lambda$ to a state $\emptyset||\Gamma \cup \{C\}||\Lambda'$, where $C$ is a denial. Indeed, within a restarting subpath an edge due to rule \textit{Learn} occurred introducing denial $C$. State $M||\Gamma||\Lambda$ is smaller than the state $\emptyset||\Gamma \cup \{C\}||\Lambda'$. At the same time, $r'$ contains no edges due to applications of \textit{Restart} or \textit{Restart}. Indeed, we eliminated these edges in favor of edges called \textit{Artificial}. Thus by the same argument as in the proof of Statement 1, $r'$ contains a finite number of edges. We can now conclude that $r$ is finite.

It is easy to see that maximal restart-safe path ends with a state that is semi-terminal. Indeed, assume the opposite: there is a maximal restart-safe path $t$, which ends in a non semi-terminal state $S$. Then, some basic rule applies to state $S$. Consider path $t'$ consisting of path $t$ and a transition due to a basic rule applicable to $S$. Note that $t'$ is also a restart-safe path, and that $t$ is a subpath of $t'$. This contradicts the definition of maximal.

(b) Let $M||\Gamma||\Lambda$ be a semi-terminal state so that none of the Basic rules are applicable. From the fact that \textit{Decide} is not applicable, we conclude that $M$ assigns all literals or $M$ is inconsistent.

We now show that $M$ is consistent. Proof by contradiction. Assume that $M$ is inconsistent. Then, since \textit{Fail} is not applicable, $M$ contains a decision literal. Consequently, $M||\Gamma||\Lambda$ is a state in which \textit{Backtrack} is applicable. This contradicts our assumption that $M||\Gamma||\Lambda$ is semi-terminal.

Also, $M^+$ is an answer set of $\Pi[C]$. Proof by contradiction. Assume that $M^+$ is not an answer set of $\Pi[C]$. It follows that both that $M$ is not an answer set of $\mathcal{P}$. By Proposition 8, it follows that $M$ is not an answer set of $\mathcal{P}[\Gamma \cup \Lambda]$ and $M^+$ is not an answer of $\Pi[C] \cup \Gamma \cup \Lambda$. Recall that $\mathcal{P}[\Gamma \cup \Lambda]$ asp-entails a literal $l$ with respect to $M$ if for every complete and consistent set $M'$ of literals over $At(\Pi)$ such that $M'^+P$ is an answer set of $\Pi[C] \cup \Gamma \cup \Lambda$ and $M \subseteq M'$, $l \in M'$. Since $M$ is complete and consistent set of literals over $At(\Pi)$ it follows that there is no complete and consistent set $M'$ of literals over $At(\Pi)$ such that $M \subseteq M'$ and $M'^+$ is an answer set of $\Pi[C] \cup \Gamma \cup \Lambda$. We conclude that $\mathcal{P}[\Gamma \cup \Lambda]$ asp-entails any literal $l$. Take $l$ to be a complement of some literal occurring in $M$. It follows that \textit{ASP-Propagate} is applicable in state $M||\Gamma||\Lambda$ allowing a transition to state $M \ l || \Gamma || \Lambda$. This contradicts our assumption that $M||\Gamma||\Lambda$ is semi-terminal.

CSP $K_{\mathcal{P},M}$ has a solution. This immediately follows from the application condition of the transition rule \textit{CP-Propagate} and the fact that the state $M||\Gamma||\Lambda$ is semi-terminal.

From the conclusions that $M^+$ is an answer set of $\Pi[C]$ and $K_{\mathcal{P},M}$ has a solution we derive that $M$ is an answer set of $\mathcal{P}$.

(c) We start by proving an auxiliary statement:

\textbf{Statement 2: For any CA program $\mathcal{P}$, and a path from an initial state to $l_1 \ldots l_i || \Gamma || \Lambda$ in EZ$_\mathcal{P}$, every answer set $X$ for $\mathcal{P}$ satisfies $l_j$ if it satisfies all decision literals $l^\mathcal{P}_j$ with $j \leq i$.} By induction on the length of a path. Since the property trivially holds in the initial state, we only need to prove that all transition rules of EZ$_\mathcal{P}$ preserve it.

Consider an edge $M||\Gamma||\Lambda \Rightarrow S$ where $S$ is either a fail state or state of the form $M'||\Gamma'||\Lambda'$, $M$ is a sequence $l_1 \ldots l_k$ such that every answer set $X$ of $\mathcal{P}$ satisfies $l_i$ if it satisfies all decision literals $l^\mathcal{P}_j$ with $j \leq i$.

\textit{Decide, Fail, CP-Propagate Learn, Learn', Restart, Restart'}: Obvious.

\textit{ASP-Propagate}: $M'||\Gamma'||\Lambda'$ is $M \ l_{k+1} || \Gamma || \Lambda$. Take any answer set $X$ of $\mathcal{P}$ such that $X$
satisfies all decision literals $l^A_j$ with $j \leq k + 1$. From the inductive hypothesis it follows that $X$ satisfies $M$. Consequently, $M \subseteq X$ since $X$ is a consistent and complete set of literals. From the definition of $\text{ASP-Propagate}$, $P$ as-entails $l_{k+1}$ with respect to $M$. We also know that $X^+$ is an answer set of $\Pi[C]$. Thus, $l_{k+1} \in X$.

Backtrack: $M$ has the form $P l^A_1 Q$ where $Q$ contains no decision literals. $M' || \Gamma' || \Lambda'$ has the form $P \bar{t}_i || \Gamma || \Lambda$. Take any answer set $X$ of $P$ such that $X$ satisfies all decision literals $l^A_j$ with $j \leq i$. We need to show that $X \models \bar{t}_i$. By contradiction. Assume that $X \models l_i$. By the inductive hypothesis, since $Q$ does not contain decision literals, it follows that $X$ satisfies $P l^A_i Q$, that is, $M$. This is impossible because $M$ is inconsistent. Hence, $X \models \bar{t}_i$.

Left-to-right: Since $\text{Failstate}$ is reachable from the initial state by a restart-safe path, there is an inconsistent state $M || \Gamma || \Lambda$ without decision literals such that there exists a path from the initial state to $M || \Gamma || \Lambda$. By Statement 2, any answer set of $P$ satisfies $M$. Since $M$ is inconsistent we conclude that $P$ has no answer sets.

Right-to-left: From (a) it follows that any maximal restart-safe path is a path from initial state to some semi-terminal state $S$. By (b), this state $S$ cannot be different from $\text{Failstate}$, because $P$ has no answer sets. ☐

**Theorem 4**

For any CA program $P$,

(a) every restart-safe path in $\text{EZSM}_P$ is finite, and any maximal restart-safe path ends with a state that is semi-terminal,

(b) for any semi-terminal state $M || \Gamma || \Lambda$ of $\text{EZSM}_P$ reachable from initial state, $M$ is an answer set of $P$,

(c) state $\text{Failstate}$ is reachable from initial state in $\text{EZSM}_P$ by a restart-safe path if and only if $P$ has no answer sets.

**Proof**

Let $P$ be a CA program $\langle \Pi, C, \gamma, D \rangle$.

(a) This part is proved as part (a) in proof of Theorem 3.

(b) Let $M || \Gamma || \Lambda$ be a semi-terminal state so that none of the basic rules are applicable ($\text{Unit Propagate}$ and $\text{Unfounded}$ are basic rules). As in proof of part (b) in Theorem 3 we conclude that $M$ assigns all literals and is consistent. Also, CSP $K_{P,M}$ has a solution.

We now illustrate that, $M^+$ is an answer set of $\Pi[C]$. Proof by contradiction. Assume that $M^+$ is not an answer set of $\Pi[C]$. It follows that that $M$ is not an answer set of $P$. By Proposition 3 it follows that $M$ is not an answer set of $P || \Gamma \cup \Lambda$ and $M^+$ is not an answer of $\Pi[C] \cup \Gamma \cup \Lambda$. By Theorem 3, it follows that either $M$ is not a model of $\Pi[C] \cup \Gamma \cup \Lambda$ or $M$ contains a non-empty subset unfounded on $M$ w.r.t. $\Pi[C] \cup \Gamma \cup \Lambda$. In case the former holds we derive that the rule $\text{Unit Propagate}$ is applicable in the state $M || \Gamma || \Lambda$. In case the latter holds we derive that the rule $\text{Unfounded}$ is applicable in the state $M || \Gamma || \Lambda$. This contradicts our assumption that $M || \Gamma || \Lambda$ is semi-terminal.

From the conclusions that $M^+$ is an answer set of $\Pi[C]$ and $K_{P,M}$ has a solution we derive that $M$ is an answer set of $P$. 


(c) Left-to-right part of the proof follows from Theorem 3 (c, left-to-right) and the fact that \( \text{EZSM}_P \) is a subgraph of \( \text{EZ}_P \).

Right-to-left part of the proof follow the lines of Theorem 3 (c, right-to-left). □

5.3 Integration Configurations of EZCSP

We can characterize the algorithm of a specific solver that utilizes the transition rules of the graph \( \text{EZ}_P \) by describing a strategy for choosing a path in this graph.

**black-box:** A configuration of EZCSP that invokes an answer set solver via black-box integration for enumerating answer sets of an asp-abstraction program is captured by the following strategy in navigating the graph \( \text{EZ}_P \)

1. Restart never applies,
2. rule CP-Propagate never applies to the states where one of these rules are applicable: Decide, Backtrack, Fail, ASP-Propagate,
3. Learn\(^t\) may apply anytime with the restriction that the denial \( R \) learnt by the application of this rule is such that \( P \) asp-entails \( R \),
4. single application of Learn follows immediately after an application of the rule CP-Propagate. Furthermore, the denial \( R \) learnt by the application of this rule is such that \( P \) cp-entails \( R \),
5. Restart\(^t\) follows immediately after an application of the rule Learn. Restart\(^t\) does not apply under any other condition.

It is easy to see that the specifications of the strategy above forms a subgraph of the graph \( \text{EZ}_P \). Let us denote this subgraph by \( \text{EZ}^b_P \). Theorem 3 holds if we replace \( \text{EZ}_P \) by \( \text{EZ}^b_P \) in its statement:

**Theorem 5**

For any CA program \( P \),

(a) every restart-safe path in \( \text{EZ}^b_P \) is finite, and any maximal restart-safe path ends with a state that is semi-terminal,

(b) for any semi-terminal state \( M||\Gamma||\Lambda \) of \( \text{EZ}^b_P \) reachable from initial state, \( M \) is an answer set of \( P \),

(c) state \( \text{Failstate} \) is reachable from initial state in \( \text{EZ}^b_P \) by a restart-safe path if and only if \( P \) has no answer sets.

**Proof**

Let \( P \) be a CA program \( \langle \Pi, \mathcal{C}, \gamma, D \rangle \).

(a) This part is proved as part (a) in proof of Theorem 3.

(b) Graph \( \text{EZ}^b_P \) is the subgraph of \( \text{EZ}_P \). At the same time it is easy to see that any non semi-terminal state in \( \text{EZ}_P \) is also a non semi-terminal state in \( \text{EZ}^b_P \). Thus, claim (b) follows from Theorem 3 (b).

(c) Left-to-right part of the proof follows from Theorem 3 (c, left-to-right) and the fact that \( \text{EZ}^b_P \) is a subgraph of \( \text{EZ}_P \).

Right-to-left part of the proof follows the lines of Theorem 3 (c, right-to-left). □
grey-box: A configuration of EZCSP that invokes an answer set solver via grey-box integration for enumerating answer sets of asp-abstraction program is captured by the strategy in navigating the graph \( EZP \) that differs from the strategy of black-box in rules 1 and 5 only. Below we present only these rules.

1. \( \text{Restart}^t \) never applies,
5. \( \text{Restart} \) follows immediately after an application of the rule Learn. \( \text{Restart} \) does not apply under any other condition.

clear-box: A configuration of EZCSP that invokes an answer set solver via clear-box integration for enumerating answer sets of asp-abstraction program is captured by the following strategy in navigating the graph \( EZP \)

- \( \text{Restart}^t \) and \( \text{Restart} \) never apply.

Similar to the black-box case, the specifications of the grey-box and clear-box strategies form subgraphs of the graph \( EZP \). Theorem 5 holds if we replace \( EZP \) by these subgraphs. We avoid stating formal proofs as they follow the lines of proof for Theorem 5.

We note that the outlined strategies provide only a skeleton of the algorithms implemented in these systems. Generally, any particular configurations of EZCSP can be captured by some subgraph of \( EZP \). The provided specifications of black-box, grey-box, and clear-box scenarios allow more freedom than specific configurations of EZCSP do. For example, in any setting of EZCSP it will never follow an edge due to the transition \( \text{Decide} \) when the transition \( \text{ASP-Propagate} \) is available. Indeed, this is a design choice of all available answer set solvers that EZCSP is based upon. The provided skeleton is meant to highlight the essence of key differences between the variants of integration approaches. For instance, it is apparent that any application of \( \text{Restart}^t \) forces us to restart the search process by forgetting about atomic part of a current state as well as some previously learnt clauses. The black-box integration architecture is the only one allowing this transition.

As discussed earlier, the schematic rule \( \text{ASP-Propagate} \) is more informative than any real propagator implemented in any answer set solver. These solvers are only able to identify some literals that are asp-entailed by a program with respect to a state. Thus if a program is extended with additional denials a specific propagator may find additional literals that are asp-entailed. This observation is important in understanding the benefit that \( \text{Restart} \) provides in comparison to \( \text{Restart}^t \). Note that applications of these rules highlight the difference between black-box and grey-box.

6 Application Domains

In this work we compare and contrast different integration schemas of hybrid solvers on three application domains that stem from various subareas of computer science: weighted-sequence (Lierler et al., 2012), incremental scheduling (Balduccini, 2011), reverse folding. The weighted-sequence domain is a handcrafted benchmark, whose key features are inspired by the important industrial problem of finding an optimal join order by cost-based query optimizers in database systems. The problem is not only practically relevant but proved to be hard for current ASP and CASP technology as illustrated in (Lierler et al.)
The incremental scheduling domain stems from a problem occurring in commercial printing. CASP offers an elegant solution to it. The reverse folding problem is inspired by VLSI design – the process of creating an integrated circuit by combining thousands of transistors into a single chip.

This section provides a brief overview of these applications. All benchmark domains are from the Third Answer Set Programming Competition – 2011 (ASPCOMP) (Calimeri et al., 2011), in particular, the Model and Solve track. We chose these domains for our investigation for several reasons. First, these problems touch on applications relevant to various industries. Thus, studying different possibilities to model and solve these problems is of value. Second, each one of them displays features that benefit from the synergy of computational methods in ASP and CSP. Each considered problem contains variables ranging over a large integer domain thus making grounding required in pure ASP a bottleneck. Yet, the modeling capabilities of ASP and availability of such sophisticated solving techniques such as learning makes ASP attractive for designing solutions to these domains. As a result, CASP languages and solvers become a natural choice for these benchmarks making them ideal for our investigation.

Three Kinds of CASP Encodings: Hybrid languages such as CASP combine constructs and processing techniques stemming from different formalisms. As a result, depending on how the encodings are crafted, one underlying solver may be used more heavily than the other. For example, any ASP encoding of a problem is also a CASP formalization of it. Therefore, the computation for such encoding relies entirely on the base solver and the features and performance of the theory solver are irrelevant to it. We call this a pure-ASP encoding. At the other end of the spectrum are pure-CSP encodings: encodings that consist entirely of ez-atoms. From a computational perspective, such an encoding exercises only the theory solver. (From a specification perspective, the use of CASP is still meaningful, as it allows for a convenient, declarative, and at the same time executable specification of the constraints.) In the middle of the spectrum are true-CASP encodings, which, typically, are non-stratified and include collections of ez-atoms expressing constraints whose solution is non-trivial.

An analysis of these varying kinds of encodings in CASP gives us a better perspective on how different integration schemas are affected by the design choices made during the encoding of a problem. At the same time considering the encoding variety allows us to verify our intuition that true-CASP is an appropriate modeling and solving choice for the explored domains. We conducted experiments on encodings falling in each category for all benchmarks considered.

The weighted-sequence (WSEQ) domain is a handcrafted benchmark problem. Its key features are inspired by the important industrial problem of finding an optimal join order by cost-based query optimizers in database systems. Lierler et al. (2012) provides a complete description of the problem itself as well as the formalization named SEQ++ that became the encoding used in the present paper.

In the weighted-sequence problem we are given a set of leaves (nodes) and an integer \( m \) – maximum cost. Each leaf is a pair \((weight, cardinality)\) where \(weight\) and \(cardinality\) are integers. Every sequence (permutation) of leaves is such that all leaves but the first are assigned a \textit{color} that, in turn, associates a leaf with a \textit{cost} (via a cost formula). A colored se-
sequence is associated with the cost that is a sum of leaves costs. The task is to find a colored sequence with cost at most $m$. We refer the reader to (Lierler et al., 2012) for the details of pure-ASP encoding SEQ++. The same paper also contains the details on a true-CASP variant of SEQ++ in the language of CLINGCON. We further adapted that encoding to the EZ language by means of simple syntactic transformations. Here we provide a review of details of the SEQ++ formalizations using pure-ASP and the EZ language that we find most relevant to this presentation. The reader can refer to Appendix A for details on the syntax used. The non-domain predicates of the pure-ASP encoding are $\text{leafPos}$, $\text{posColor}$, $\text{posCost}$. Intuitively, $\text{leafPos}$ is responsible for assigning a position to a leaf, $\text{posColor}$ is responsible for assigning a color to each position, $\text{posCost}$ carries information on costs associated with each leaf. Some rules used to define these relations are given in Figure 5.

% Give each leaf a location in the sequence
1{leafPos(L, N) : location(N)}1 ← leaf(L).

% No sharing of locations
← leafPos(L1, N), leafPos(L2, N), location(N), L1 ≠ L2.

% green if (weight(right) + card(right)) < (weight(left) + leafCost(right))
posColor(1, green) ← leafPos(L1, 0), leafPos(L2, 1),
leafWeightCardinality(L1, WL, CL),
leafWeightCardinality(L2, WR, CR),
leafCost(L2, W3),
W1 = WR + CR, W2 = WL + W3,
W1 < W2.

% posCost for first coloredPos
posCost(1, W) ← posColor(1, green), leafPos(L, 1),
leafWeightCardinality(L, WR, CR),
max_total_weight(MAX),
W = WR + CR, W ≤ MAX.

posCost(1, W) ← not posColor(1, green), leafPos(L1, 0), leafPos(L2, 1),
leafWeightCardinality(L1, WL, CL), leafCost(L2, WR),
max_total_weight(MAX),
W = WL + WR, W ≤ MAX.

% Acceptable solutions
acceptable ← #sum\{nWeight(P, W) = W : coloredPos(P)\} MAX,
max_total_weight(MAX).
← not acceptable.

Figure 5. Some typical rules of the pure-ASP language formalization of WSEQ.

The first two rules in Figure 5 assign a distinct location to each leaf. The next rule is part of the color assignment. The following two rules are part of the cost determination. The final two rules ensure that the total cost is within the specified limit.

The main difference between the pure-ASP and true-CASP encodings is in the treatment of the cost values of the leaves. We first note that cost predicate $\text{posCost}$ in the pure-ASP encoding is “functional”. In other words, when this predicate occurs in an answer set, its first argument uniquely determines its second argument. Often, such functional predicates
in ASP encodings can be replaced by ez-atoms\textsuperscript{24} in CASP encodings. Indeed, this is the case in the weighted-sequence problem. Thus in the true-CASP encoding, the definition of $posCost$ is replaced by suitable ez-atoms, making it possible to evaluate cost values by CSP techniques. This approach is expected to benefit performance especially when the cost values are large. Some of the corresponding rules follow:

\begin{verbatim}
% posCost for first coloredPos
required(posCost(1) = W) ← posColor(1, green), leafPos(L, 1),
leafWeightCardinality(L, WR, CR), W = WR + CR.
required(posCost(1) = W) ← not posColor(1, green),
leafPos(L1, 0), leafPos(L2, 1),
leafWeightCardinality(L1, WL, CL), leafCost(L2, WR),
W = WL + WR.

% Acceptable solutions
required(sum([posCost/1, <=, MV]) ← max_total_weight(MV).

The first two rules are rather straightforward translations of the ASP equivalents. The last rule uses a global constraint to ensure acceptability of the total cost.

The pure-CSP encoding is obtained from the true-CASP encoding by replacing the definitions of $leafPos$ and $posColor$ predicates by constraint atoms. The replacement is based on the observation that $leafPos$ and $posColor$ are functional.

\begin{verbatim}
% green if (weight(right) + card(right)) < (weight(left) + leafCost(right))
is_green(1, L1, L2) ← leafWeightCardinality(L1, WL, CL),
leafWeightCardinality(L2, WR, CR),
leafCost(L2, W3),
W1 = WR + CR, W2 = WL + W3,
W1 < W2.
required(posColor(1) = green ← (leafPos(L1) = 0 ∧ leafPos(L2) = 1)) ←
leaf(L1), leaf(L2), is_green(1, L1, L2).
\end{verbatim}

As shown by the last rule, color assignment requires the use of reified constraints. It is important to note that symbol $←$ within the scope of required stands for material implication. Color names are mapped to integers by introducing additional variables. For example, variable \texttt{green} is associated with value 1 by a variable declaration \texttt{cspvar(green, 1, 1)}. Interestingly, no ez-atoms are needed for the definition of $leafPos$. The role of the choice rule above is implicitly played by the variable declaration

\begin{verbatim}
cspvar(leafPos(L), 0, N - 1) ← leaf(L), location(N).
\end{verbatim}

The \textbf{incremental scheduling} (IS) domain stems from a problem occurring in commercial printing. In this domain, a schedule is maintained up-to-date with respect to jobs being added and equipment going offline. A problem description includes a set of devices, each with predefined number of instances (slots for jobs), and a set of jobs to be produced. The

\textsuperscript{24} We abuse the term ez-atom and refer to “non-ground” atoms of the EZ language that result in ez-atoms by the same name.
penalty for a job being late is computed as \( td \cdot imp \), where \( td \) is the job’s tardiness and \( imp \) is a positive integer denoting the job’s importance. The total penalty of a schedule is the sum of the penalties of the jobs. The task is to find a schedule whose total penalty is no larger than the value specified in a problem instance. We direct the reader to (Balduccini, 2011) for more details on this domain. We start by describing the pure-CSP encoding and then illustrate how it relates to the true-CASP encoding.

The pure-CSP encoding used in our experiments is the official competition encoding submitted to ASPCOMP by the EZCSP team. In that encoding, constraint atoms are used for (i) assigning start times to jobs, (ii) selecting which device instance will perform a job, and (iii) calculating tardiness and penalties. Core rules of the encoding are shown in Figure 6.

The ez-atom of the first rule uses a global constraint to specify that the start times must be assigned in such a way as to ensure that no more than \( n_d \) jobs are executed at any time, where \( n_d \) is the number of instances of a given device \( d \). The ez-atom of the second rule uses reified constraints with the \( \lor \) connective (disjunction) to guarantee that at most one job is executed on a device instance at every time. The ez-atom of the third rule uses a global constraint to define total penalty. The last rule restricts total penalty to be within the allowed maximum value.

The true-CASP encoding was obtained from the pure-CSP encoding by introducing a new relation \( on\_instance(j, i) \), stating that job \( j \) runs on device-instance \( i \). The rules formalizing the assignment of device instances in the pure-CSP encoding were replaced by ez-atoms. For example, the second rule from Figure 6 was replaced by:

\[
1 \{ on\_instance(J, I) : instance\_of(D, I) \} 1 \leftarrow job\_device(J, D) .
\]
The main difference with respect to the ez-atom of the pure-CSP encoding is the introduction of a choice rule to select an instance $I$ for a job $J$. The constraint that each instance processes at most one job at a time is still encoded using an ez-atom. Finally, the pure-ASP encoding was obtained from the true-CASP encoding by introducing suitable new relations, such as $\text{start}(j, s)$ and $\text{penalty}(j, p)$, to replace all remaining ez-atoms. The rules that replace the first rule in Figure 6 follow:

$$1\text{(start}(J, S) : \text{time}(S))\{1 \leftarrow \text{job}(J).$$

$$\leftarrow \text{on\_instance}(J1, I), \text{on\_instance}(J2, I), J1 \neq J2,$$
$$\text{job\_device}(J1, D), \text{job\_device}(J2, D),$$
$$\text{start}(J1, S1), \text{job\_len}(J1, L1), \text{start}(J2, S2),$$
$$S1 \leq S2, S2 < S1 + L1.$$  

The last two rules in Figure 6 are replaced by the rules in the pure-ASP encoding:

$$\text{tot\_penalty}(TP) \leftarrow TP \{ \text{penalty}(J, P) = P \} TP.$$

$$\leftarrow \text{not } [\text{penalty}(J, P) = P]\text{Max, max\_total\_penalty}(Max).$$

In the reverse folding (RF) domain, one manipulates a sequence of $n$ pairwise connected segments located on a 2D plane in order to take the sequence from an initial configuration to a goal configuration. The sequence is manipulated by pivot moves: rotations of a segment around its starting point by 90 degree in either direction. A pivot move on a segment causes the segments that follow to rotate around the same center. Concurrent pivot moves are prohibited. At the end of each move, the segments in the sequence must not intersect. A problem instance specifies the number of segments, the goal configuration, and required number of moves denoted by $t$. The task is to find a sequence of exactly $t$ pivot moves that produces the goal configuration.

The true-CASP encoding used for our experiments is from the official ASPCOMP 2011 submission package of the EZCSP team. In this encoding, relation $\text{pivot}(s, i, d)$ states that at step $s$ the $i^{th}$ segment is rotated in direction $d$. The effects of pivot moves are described by ez-atoms, which allows us to carry out the corresponding calculations with CSP techniques.

$$\text{pivot}(1, I, D) \leftarrow \text{first}(I), \text{requiredMove}(I, D).$$

$$\text{pivot}(N1, I1, D1) \leftarrow \text{pivot}(N2, I2, D2), N1 = N2 + 1,$$
$$\text{requiredMove}(I1, D1), \text{requiredMove}(I2, D2),$$
$$\text{next}(I1, I2).$$
The first two rules are some of the rules used for determining the pivot rotations. The determination is based on the technique described in (Balduccini and Lierler, 2012). The last two rules are part of the calculation of the effects of pivot moves. Note that \( tfoldx(s, i) \) and \( tfoldy(s, i) \) denote the \( x \) and \( y \) coordinates of the start of segment \( i \) at step \( s \).

The pure-ASP encoding was obtained from the true-CASP encoding by adopting an ASP-based formalization of the effects of pivot moves. This was accomplished by introducing two new relations, \( tfoldx(s, i, x) \) and \( tfoldy(s, i, y) \), stating that the new start of segment \( i \) at step \( s \) is \( \langle x, y \rangle \). The definition of the relations is provided by suitable ASP rules, such as:

\[
\begin{align*}
tfoldy(S + 1, I, Y^2) & \leftarrow tfoldx(S, I, X_1), pivot(S, P, D), I \geq P, \\
tfoldx(S, P, XP), tfoldy(S, P, YP), X_0 = X_1 - XP, \\
rotatedx(D, X_0, Y_0), Y^2 = Y_0 + YP. \\
xcoord(-2 \cdot N \cdot 2 + N) & \leftarrow length(N).
\end{align*}
\]

Differently from the previous domains, for RF we were unable to formulate a pure-CSP variant of the true-CASP encoding. Thus, we resorted to the encoding described in (Dovier et al., 2011). This encoding leverages a mapping from action language \( B \) (Gelfond and Lifschitz, 1998) statements to numerical constraints, which are then solved by a CLP system.

### 7 Experimental Results

The experimental comparison of the integration schemas was conducted on a computer with an Intel Core i7 processor at 3GHz and running Fedora Core 16. The memory limit for each process and the timeout were set to 1 GB RAM\(^{27}\) and 6,000 seconds respectively. A single processor core was used for every experiment.

The version of EZCSP used in the experiments was 1.6.20b49. This version implements the black-box, grey-box, and clear-box integration schemas, when suitable API interfaces are available in the base solver. One answer set solver that provides such interfaces is CMODELS, which for this reason was chosen as base solver for the experiments. It is worth noting that the development of the API in CMODELS was greatly facilitated by the API provided by MINISAT v. 1.12b supporting non-clausal constraints (Flin and Sörensson, 2003) (MINISAT forms the main inference mechanism of CMODELS). In the experiments, we used

\(^{27}\)The instances that resulted in an out-of-memory were also tested with 4 GB RAM, with no change in the outcome.
CMODELS version 3.83 as the base solver and BPROLOG 7.4 as the theory solver. Answer set solver CMODELS 3.83 (with the inference mechanism of MINISAT v. 1.12b) was also

We note that BPROLOG is the default theory solver of EZCSP. Command-line option --solver cmodels-3.83 instructs EZCSP to invoke CMODELS 3.83 using the black-box integration schema. Command-line options --cmodels-incremental and --cmodels-feedback instruct EZCSP to use, respectively, the grey-box and clear-box integration schema. In these two cases, CMODELS 3.83 is automatically selected as the base solver.

Figure 7. Performance on WSEQ domain: total times in logarithmic scale

Figure 8. Performance on WSEQ domain: cumulative view (grey-box and clear-box)
used for the experiments with the pure-ASP encodings. Unless otherwise specified, for all solvers we used their default configurations.

The executables used in the experiments and the encodings can be downloaded, respectively, from

- http://www.mbal.tk/ezcsp/intschemas/ezcsp-binaries.tgz
- http://www.mbal.tk/ezcsp/intschemas/experiments.tgz

In order to provide a frame of reference with respect to the state of the art in CASP, the results also include performance information for CLINGCON 2.0.3 on the true-CASP encodings adapted to the language of CLINGCON. We conjecture that the choice of constraint solver by CLINGCON (namely, GECODE) together with theory propagation is the reason for CLINGCON’s better performance in a number of the experiments. Yet, in the context of our experiments, the performance of CLINGCON w.r.t. EZCSP is irrelevant. Our work is a comparative study of the impact of the different integration schemas for a fixed selection of a base and theory solver pair. System EZCSP provides us with essential means to perform this study.

In all figures presented: CASP Black, CASP Grey, CASP Clear denote EZCSP implementing respectively black-box, grey-box and clear-box, and running a true-CASP encoding; Pure-CSP denotes EZCSP implementing black-box running a pure-CSP encoding (note that for pure-CSP encodings there is no difference in performance between the integration schemas); ASP denotes CMODELS running a pure-ASP encoding; Clingcon denotes CLINGCON running a true-CASP encoding. Each configuration is associated with the same color in all figures. A pattern is applied to the filling of the bars whenever the bar goes off-chart. The numbers in the overlaid boxes report the time in seconds and, in parentheses, the total number of timeouts and out-of-memory.

We begin our analysis with WSEQ (Figures 7 and 8). The total times across all the instances for all solvers/encodings pairs considered are shown in Figure 7. Because of the large difference between best and worst performance, a logarithmic scale is used. For uniformity of presentation, in the charts out-of-memory conditions and timeouts are both rendered as out-of-time results. The instances used in the experiments are the 30 instances available via ASPCOMP. Interestingly, answer set solver CMODELS on the pure-ASP encoding has excellent performance, comparable to the best performance obtained with CASP encodings by EZCSP. Of the CASP encodings, the true-CASP encoding running in black-box times out on every instance. Figure 8 thus focuses on the cumulative run times of clear-box and grey-box (on the true-CASP encoding). The numbers on the horizontal axis identify the instances, while the vertical axis is for the cumulative run time, that is, the value for instance $n$ is the sum of the run times for instances $1 \ldots n$. Cumulative times were chosen for the per-instance figures because they make for a more readable chart when there is large variation between the run times for the individual instances. As shown in Figure 8, the true-CASP encoding running in clear-box performs substantially better than grey-box. This demonstrates that, for this domain, the tight integration schema has an advantage.

In case of the Is domain we considered two sets of experiments. In the first one (Figures 9 and 10), we used the 50 official instances from ASPCOMP. We refer to these instances as easy, since the corresponding run times are rather small. Figure 9 provides a comparison
of the total times. Judging by the total times, tight integration schemas appear to have an advantage, allowing the true-CASP encoding to outperform the pure-CSP encoding. As one might expect, the best performance for the true-CASP encoding is from the clear-box integration schema. In this case the early pruning of the search space made possible by the clear-box architecture seems to yield substantial benefits. As expected, grey-box is also faster than black-box, while CMODELS on the pure-ASP encoding runs out of memory in all the instances.

The second set of experiments for the IS domain (Figures 11 and 12) consists of 30 in-
stances that we generated to be substantially more complex than the ones from ASPCOMP, and that are thus called hard. As discussed below, this second set of experiments reveals a remarkable change in the behavior of solver/encodings pairs when the instances require more computational effort. The process we followed to generate the 30 hard instances consisted in (1) generating randomly 500 fresh instances; (2) running the true-CASP encoding with the grey-box integration schema on them with a timeout of 300 seconds; (3) selecting randomly, from those, 15 instances that resulted in timeout and 15 instances that were solved in 25 seconds or more. The numerical parameters used in the process were selected
with the purpose of identifying more challenging instances than those from the easy set and were based on the results on that set. The execution times reported in Figure [11] clearly indicate the level of difficulty of the selected instances (once again, CMODELS runs out of memory). Remarkably, these more difficult instances are solved more efficiently by the pure-CSP encoding that relies only on the CSP solver. In fact, the pure-CSP encoding outperforms every other method of computation (including CLINGCON on true-CASP encoding). More specifically, solving the instances with the true-CASP encoding takes between 30% and 50% longer than with the pure-CSP encoding. This was not the isolated effect of a few instances, but rather a constant pattern throughout the experiment. A possible explanation for this phenomenon is that domain IS is overall best suited to the CSP solving procedures. It seems natural for the difference in performance to become more evident as the problem instances become more challenging, when other factors such as overhead play less of a role. This conjecture is compatible with the difference in performance observed earlier on the easy instances.

Another remarkable aspect highlighted by Figure 12 is that clear-box is outperformed by grey-box. This is the opposite of what was observed on the easy instances and highlights the fact that there is no single-best integration schema, even when one focuses on true-CASP encodings. We hypothesize this to be due to the nature of the underlying scheduling problem, which is hard to solve, but whose relaxations (obtained by dropping one or more constraints) are relatively easy. Under these conditions, the calls executed by clear-box to the theory solver are ineffective at pruning the search space and incur a non-negligible overhead. (The performance of CLINGCON is likely affected by the same behavior.) In grey-box, on the other hand, no time is wasted trying to prune the search space of the base solver, and all the time spent in the theory solver is dedicated to solving the final CSP. The performance of black-box is likely due to the minor efficiency of its integration schema compared to grey-box.

The final experiment focuses on the RF domain (Figures 13 and 14). The instances used in this experiment are the 50 official instances from ASPCOMP. The total execution times are presented in Figure 13. Although the instances for this domain are comparatively easy, as suggested by the black-box and grey-box times, some of the configurations have high total execution times. The clear-box encoding is also off-chart, due to timeouts on 19 instances. This is a substantial difference in performance compared to the other true-CASP configurations, upon which we expand later in this section. Surprisingly, the total time of CLINGCON is also close to off-chart. Upon closer inspection, we have found this to be due to 3 instances for which CLINGCON runs out of memory. This is an interesting instance of the trade-off between speed of execution and performance stability, considering that on the other instances CLINGCON is very fast. The per-instance execution times for grey-box and black-box are detailed in Figure 14. The figure highlights the very similar performance of the two schemas, with black-box losing only in the final 10% of the instances in spite of its higher overhead. This is likely due to the simplicity of the RF problem: most extended answer sets can be found with little backtracking between base and theory solver, and thus the difference between the two schemas has little bearing on the execution times. Similarly to the hard instances of the IS domain, the better performance of black-box and grey-box in comparison to clear-box can be explained by the fact that, in this domain, frequent checks with the theory solver add overhead but are overall ineffective at pruning the search.
Figure 13. Performance on RF domain: total times (detail of 0-20,000sec execution time range, clear-box and pure-ASP off-chart)

Figure 14. Performance on RF domain: cumulative view (black-box and grey-box)

8 A Brief Account on Related Systems

In the introduction we mentioned solvers ACSOLVER (Mellarkod et al., 2008), CLINGCON (Gebser et al., 2009), IDP (Wittecoq et al., 2008), INCA (Drescher and Walsh, 2011), DINGO (Janhunen et al., 2011), MINGO (Lau et al., 2012), ASPMT SMT (Bartholomew and Lee, 2013), and EZSMT (Susman and Lierler, 2016). In this section we briefly remark on
Table 2. Solvers used by state-of-the-art CASP systems

<table>
<thead>
<tr>
<th>System</th>
<th>Base Solver</th>
<th>Theory Solver</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACSOLVER</td>
<td>SMODELS</td>
<td>constraint logic programming systems</td>
</tr>
<tr>
<td>CLINGCON</td>
<td>CLASP</td>
<td>GECODE (Schulte and Stuckey, 2008)</td>
</tr>
<tr>
<td>IDP</td>
<td>MINISAT(ID)</td>
<td>GECODE (Schulte and Stuckey, 2008)</td>
</tr>
<tr>
<td>INCA</td>
<td>CLASP</td>
<td>its own CP solver</td>
</tr>
</tbody>
</table>

this variety of CASP systems. This is not intended as a detailed comparison between the systems, but as a quick summary.

At a high-level abstraction, one may easily relate the architectures of the CLINGCON, ACSOLVER, IDP, and INCA to that of EZCSP. Given a CASP program, all of these systems first utilize an answer set solver to compute a part of an answer set for an asp-abstraction and then utilize a constraint programming system to solve a resulting csp-abstraction. All of these systems implement the clear-box integration schema. Table 2 provides a summary of base solvers and theory solvers utilized by them.

A few remarks are due. Unlike its peers, ACSOLVER does not implement learning as its base solver SMODELS does not support this technique. The fact that system INCA implements its own CP solver, or, in other words, a set of its in house CP-based propagators allows this system to take advantage of some sophisticated techniques stemming from CP. In particular, it implements so called “lazy nogood generation”. This technique allows one to transfer some of the information stemming from CP-based propagations into a propositional logic program extending the original input to a base solver. We also note that the latest version of CLINGCON, as well, bypasses the use of GECODE by implementing its own CP-based propagators. All of the above systems are focused on finite domain integer linear constraints. Some of them allow for global constraints.

System DINGO translates CA programs into SMT modulo difference logic formulas and applies the SMT solver Z3 (De Moura and Bjørner, 2008) to find their models. Rather than arbitrary integer linear constraints, the system only handles those that fall into the class of difference logic. On the other hand, the system does not pose the restriction of finite domain. The EZSMT (Susman and Lierler, 2016) solver and the ASPMT2SMT (Bartholomew and Lee, 2014) solvers utilize SMT solvers to process CA programs. Both of these systems may only deal with tight programs. They allow for arbitrary integer linear constraints. None of the SMT-based CASP solvers allow for global constraints in their programs due to the underlying solving technology.

Last but not least, the solver MINGO translates CA programs into mixed integer programming expressions and then utilizes IBM ILOG CPLEX system to find solutions.

Susman and Lierler (2016) provide an experimental analysis of systems from all of the families mentioned.

9 Conclusions

In this paper, we have addressed in a principled way the integration of answer set solving techniques and constraint solving techniques in CASP solvers and, in particular, in the realm of the constraint answer set solver EZCSP. To begin, we defined logic programs with constraint atoms (CA programs). To bridge the ASP and the constraint programming aspects of such programs, we introduced the notions of asp-abstractions and csp-abstractions, which allow for a simple and yet elegant way of defining the extended answer sets of CA programs.

Next, we described the syntax of the CASP language used by the constraint answer set solver EZCSP, which we call EZ. It is worth noting that this paper contains the first detailed and principled account of the syntax of the EZ language. We relate programs written in the EZCSP language and CA programs. The tight relation between EZ programs and CA programs makes it evident that the EZ language is a full-fledged constraint answer set programming formalism. Recall that the EZCSP system originated as an attempt to provide a simple, flexible framework for modeling constraint satisfaction problems. This yields an interesting observation: constraint answer set programming can be seen as a declarative modeling framework utilizing constraint satisfaction solving technology.

In this paper we also drew a parallel between CASP and SMT. We used this connection to introduce three important kinds of integration of CASP solvers: black-box integration, grey-box integration, and clear-box integration. We introduced a graph-based abstract framework suitable for describing the EZCSP solving algorithm. The idea of using graph-based representations for backtrack-search procedures was pioneered by the SAT community. Compared to the use of pseudocode for describing algorithms, such a framework allows for simpler descriptions of search algorithms, and is well-suited for capturing the similarities and differences of the various configurations of EZCSP stemming from different integration schemas.

Finally, we presented an experimental comparison of the various integration schemas, using the implementation of EZCSP as a testbed. For the comparison, we used three challenging benchmark problems from the Third Answer Set Programming Competition – 2011 (Calimeri et al., 2011). The experimental analysis takes into account how differences in the encoding of the solutions may influence overall performance by exploiting the components of the solver in different ways. The case study that we conducted clearly illustrates the influence that integration methods have on the behavior of hybrid systems. The main attractive feature of the black-box integration schema is the ease of inception of a new system. In realm of CASP, one may take existing off-the-shelf ASP and CSP tools and connect them together by simple intermediate translation functions. This facilitates fast implementation of a prototypical CASP solver. One can then move towards a grey-box or clear-box architecture in the hope of increased performance when a prototype system proves to be promising. Yet, our experiments demonstrate that different integration schemas may be of use and importance for different domain, and that, when it comes to performance, there is no single-best integration schema. Thus, systematic means ought to be found for facilitating building hybrid systems supporting various coupling mechanisms. Just as the choice of a particular heuristic for selecting decision literals is often configurable in SAT or ASP solvers via command line parameters, the choice of integration schema in
hybrid systems should be configurable. Experimental results also indicate a strong need for theory propagation. Standard interfaces in both base and theory systems are required in order to easily build hybrid systems to support this feature.

Building clear and flexible APIs allowing for various types of interactions between the solvers seems a necessary step towards making the development of hybrid solvers effective. This work provides evidence for the need of an effort towards this goal. Many SAT solvers and SMT solvers already come with APIs that aim at facilitating extensions of these complex software systems. We argue for this practice to be adopted by other automated reasoning communities.

Finally, our study was performed in the realm of CASP technology, but it translates to SMT as well, given the discussed links between the two technologies. Incidentally, this also brings to the surface the importance of establishing means of effective communication between the two communities of constraint answer set programming and SMT solving.

References


ÉÉN, N. AND BIERE, A. 2005. Effective preprocessing in SAT through variable and clause elimination. In SAT.


SUSMAN, B. AND LIELER, Y. 2016. SMT-based constraint answer set solver EZSMT (system description). In *International Conference on Logic Programming (ICLP).*


Appendix A  
EZ – The Language of EZCSP

The EZ language is aimed at a convenient specification of a propositional ez-program $\mathcal{E} = \langle E, A, C, \gamma, D \rangle$. To achieve this, the language supports an explicit specification of domains and variables, the use of non-ground rules, and a compact representation of lists in constraints. We begin by describing the syntax of the language. Next, we define a mapping from EZ programs to propositional ez-programs.

Let $\Sigma_{\text{EZ}} = \langle C_{\text{EZ}}, V_{\text{EZ}}, F_{\text{EZ}}, R_{\text{EZ}} \rangle$ be a signature, where $C_{\text{EZ}}$, $V_{\text{EZ}}$, $F_{\text{EZ}}$, and $R_{\text{EZ}}$ denote pairwise disjoint sets of constant symbols, non-constraint variable symbols, function symbols, and relation symbols respectively. Set $C_{\text{EZ}}$ includes symbols for integers and predefined constants ($fd$, $q$, $r$), denoting CSP domains. We use common convention in logic programming and denote non-constraint variable symbols in $V_{\text{EZ}}$ by means of upper case letters. Function and relation symbols are associated with a non-negative integer called arity. The arity of function symbols is always greater than 0. Set $F_{\text{EZ}}$ includes pre-defined symbols that intuitively correspond to arithmetic operators (e.g., $+$), reified arithmetic connectives (e.g., $<$, $=$), reified logical connectives (see Table A1), list delimiters ([] and ]) and names of global constraints (discussed later in this section). Set $R_{\text{EZ}}$ contains pre-defined symbols cspdomain, cspvar, required.

The notions of terms, atoms, literals, and rules are defined over $\Sigma_{\text{EZ}}$ similarly to ASP, although the notion of term is slightly expanded. Specifically, a term over signature $\Sigma_{\text{EZ}} = \langle C_{\text{EZ}}, V_{\text{EZ}}, F_{\text{EZ}}, R_{\text{EZ}} \rangle$ is defined as:

1. a constant symbol from $C_{\text{EZ}}$.
2. a variable symbol from $V_{\text{EZ}}$.
3. an expression of the form $f(t_1, \ldots, t_k)$, \hspace{1cm} (A1)
   where $f$ is a function symbol in $F_{\text{EZ}}$ of arity $k$ and $\langle t_1, \ldots, t_k \rangle$ are terms from cases 1–3 (If a function symbol is a pre-defined arithmetic operator, arithmetic connective, or logical connective, then common infix notation is used.)
4. an extensional list, i.e., an expression of the form $[t_1, t_2, \ldots, t_k]$ where $t_i$’s are terms from cases 1–3.
5. an intensional list, i.e., an expression of the form $[g/k]$ where $g \in F_{\text{EZ}}$ or (with slight abuse of notation) $g \in R_{\text{EZ}}$ and $k$ is an integer.
6. a global constraint, i.e., an expression of the form $f(\lambda_1, \lambda_2, \ldots, \lambda_k)$, where $f \in F_{\text{EZ}}$ and each $\lambda_i$ is a list.

Pre-defined arithmetic and logical connectives from $F_{\text{EZ}}$ are dedicated to the specification of constraints. The connectives are reified to enable their use within atoms of the form required($\beta$). Furthermore, the logical connectives enable the specification of so called “reified constraints” such as:

$$x \geq 12 \lor y < 3,$$ \hspace{1cm} (A2)

37 In constraint satisfaction, global constraints are applied to lists of terms of arbitrary length, while local constraints, such as $x > y$, apply to a fixed number of arguments. For simplicity, in the definition of the language we disregard special cases of global constraints, whose arguments are not lists.
An EZ program is a pair \( \langle \Sigma_{EZ}, \Pi \rangle \), where \( \Pi \) is a set of rules over signature \( \Sigma_{EZ} \). Every EZ program is required to contain exactly one fact, whose head is \( cspdomain(fd) \), \( cspdomain(q) \), or \( cspdomain(r) \). Following common practice, we denote a program simply by the set of its rules, and let the signature be implicitly defined.

Similarly to ASP, a non-ground rule is a rule containing one or more non-constraint variables. A non-ground rule is interpreted as a shorthand for the set of propositional (ground) rules obtained by replacing every non-constraint variable in the rule by suitable terms not containing non-constraint variables. The process of replacing non-ground rules by their propositional counterparts is called grounding and is well understood in ASP (Gebser et al., 2007; Calimeri et al., 2008). For this reason, in the rest of this section we focus on ground EZ programs.

We now define a mapping from a (ground) EZ program \( \Pi \) to a propositional ez-program \( E = \langle E, A, C, \gamma, D \rangle \). We assume that \( \gamma \), a function from \( C \) to constraints, is defined along the lines of Section 2.2 and given. Recall that only one fact formed from relation \( cspdomain \) is allowed in a program \( \Pi \). The fact’s head is mapped to the constraint domain \( D \) by mapping \( \mu_D \):

\[
\mu_D(\Pi) = \begin{cases} 
FD \text{ (finite domains)} & \text{if } cspdomain(fd) \cdot \in \Pi \\
Q & \text{if } cspdomain(q) \cdot \in \Pi \\
R & \text{if } cspdomain(r) \cdot \in \Pi 
\end{cases}
\]

Atoms formed from relation \( cspvar \) specify the set \( V_{PE} \) of variables (recall that \( V_{PE} \) is the set of constraint variables that appear in csp-abstractions corresponding to \( E \)). The corresponding atoms take two forms, \( cspvar(v) \) and \( cspvar(v, l, u) \), where \( v \) is a term from \( \Sigma_{EZ} \) and \( l, u \) belong to \( C_{EZ} \cap D \). The latter form allows one to provide a range for the variable. Specifically, set \( V_{PE} \) is obtained from facts containing the above atoms as follows:

\[
V_{PE} = \{v | cspvar(v) \cdot \in \Pi \text{ or } cspvar(v, l, u) \cdot \in \Pi\}.
\]

The constraints that specify the range of the variables are generated by mapping \( \mu_V \):

\[
\mu_V(\Pi) = \{ \text{required}(v \geq l) \cdot | cspvar(v, l, u) \cdot \in \Pi \} \cup \{ \text{required}(v \leq u) \cdot | cspvar(v, l, u) \cdot \in \Pi \}.
\]
Next, we address the specification of lists. Let us begin by introducing some needed terminology. If a term is of the form \( (A) \), then we refer to \( A \) as its lexord arguments have prefix \( \lambda \).

As a practical example of the use of intensional lists, suppose that, above, relation \( \Sigma \) gives information on cumulative and other global constraints. The choice of a particular order is due to the fact that global constraints that accept multiple lists typically have the form \( \Sigma \). Consider an expression \( \{ f(t_1, t_2, \ldots, t_m) / k \} \), where \( f \in F_{\Sigma} \). Given a set \( S \), \( \text{lexord}(\Pi) \) denotes the facts in \( \Pi \). Finally, given a set \( S \), \( \text{atom}(r) \) denotes a list \( \{ a_1, a_2, \ldots, a_n \} \) enumerating the elements of \( S \) in such a way that \( a_i \leq a_{i+1} \). We can now define mappings \( \lambda_v \) and \( \lambda_r \) from the two forms of intensional lists to corresponding extensional lists:

- Given an expression of the form \( [f(t_1, t_2, \ldots, t_m) / k] \), where \( f \in F_{\Sigma} \), \( t_i \)'s are terms, and \( 0 \leq m \leq k \), its extensional representation is the list:
  \[
  \lambda_v([f(t_1, t_2, \ldots, t_m) / k]) = \text{lexord}(\text{terms}(f, k, \{ t_1, t_2, \ldots, t_m \})) \cap V_r
  \]
  of all variables with functor \( f \), arity \( k \), and whose arguments have prefix \( \{ t_1, t_2, \ldots, t_m \} \). For example, given a set of variables
  \[
  X_1 = \{ v(1), v(2), v(3), w(a, 1), w(a, 2), w(b, 1) \},
  \]
  the expression \( [w(a)/2] \) denotes the list \( \lambda_v(w, 2, \{ a \}) = \{ w(a, 1), w(a, 2) \} \). When \( m = 0 \), the expression is abbreviated \( [f/k] \). For instance, given set \( X_1 \) as above, the expression \( [v/1] \) denotes \( [v(1), v(2), v(3)] \).

- Consider an expression \( [r(t_1, t_2, \ldots, t_m) / k] \), where \( r \) is not a pre-defined relation from \( R_{\Sigma} \) and \( 0 \leq m \leq k \). Let \( \{ a_1, a_2, \ldots, a_n \} \) denote list \( \text{lexord}(\text{facts}(\Pi) \cap \text{atom}(r, k, \{ t_1, t_2, \ldots, t_m \})) \) and let \( \alpha_i^k \) denote the \( i \)-th argument of \( a_i \). Then, the extensional representation, \( \lambda_r([r(t_1, t_2, \ldots, t_m) / k]) \), of \( [r(t_1, t_2, \ldots, t_m) / k] \) is:
  \[
  \lambda_r([r(t_1, t_2, \ldots, t_m) / k]) = [\alpha_1^k, \alpha_2^k, \ldots, \alpha_n^k].
  \]
  For example, given a relation \( r' \) defined by facts \( r'(a, 1, 1), r'(a, 2, 1), r'(b, 5, 7) \), the expression \( [r'(a)/3] \) denotes the list \( \{ 3, 1 \} \) and the expression \( [r'(a, 2)/3] \) denotes \( [1] \). Similarly to the previous case, when the list of arguments is empty, the expression can be abbreviated as \( [r/k] \). For instance, given a relation \( r'' \) for which we are given facts \( r''(a, 3), r''(b, 1), r''(c, 2) \), the expression \( [r''/2] \) denotes \( \{ 3, 1, 2 \} \).

As a practical example of the use of intensional lists, suppose that, above, relation \( r'' \) denotes the amount of resources required for a job and suppose that we are given facts \( d(a, 1), d(b, 1), d(c, 1) \), specifying that jobs \( a, b, c \) have duration 1. Additionally, variables \( st(a), st(b), st(c) \) represent the start time of the jobs. A cumulative constraint for this scenario can be written as

\[
\text{required} \left( \text{cumulative} \left( \{ st/1, [d/2], [r''/2], 4 \} \right) \right),
\]

39 The choice of a particular order is due to the fact that global constraints that accept multiple lists typically expect the elements in the same position throughout the lists to be in a certain relation. More sophisticated techniques for the specification of lists are possible, but in our experience, this method gives satisfactory results.

40 \text{EZCSP} gives information on cumulative and other global constraints.
which is an abbreviation of

\[
\text{required}(\text{cumulative}([\text{st}(a), \text{st}(b), \text{st}(c)], [1, 1, 1], [3, 2, 1], 4))
\]

and means that values should be assigned to variables \text{st}(a), \text{st}(b), \text{st}(c) so that each job, of duration 1 and requiring amounts of resources 3, 2, 1 respectively, can be executed on a machine that can provide at most 4 resources at any given time.

Next, let \( \mu_R \) be a function that maps an atom of the form \text{required}(\beta) to an atom \text{required}(\beta') by:

- Replacing every occurrence of \([f(t_1, \ldots, t_m)/k] \) in \( \beta \) by \( \lambda_v([f(t_1, \ldots, t_m)/k]) \);
- Replacing every occurrence of \([r(t_1, \ldots, t_m)/k] \) in \( \beta \) by \( \lambda_r([f(t_1, \ldots, t_m)/k]) \).

The mapping is easily extended to rules and to programs as follows:

\[
\mu_R(a \leftarrow B.) = \begin{cases} 
\mu(a) \leftarrow B. & \text{if } a \text{ is of the form } \text{required}(\beta) \\
 a \leftarrow B. & \text{otherwise}
\end{cases}
\]

where \( B \) denotes the body of a rule.

\[
\mu_R(\Pi) = \bigcup_{r \in \Pi} \mu_R(r)
\]

Finally, let \( \mu_A(\Pi) \) and \( \mu_C(\Pi) \) denote mappings from \( \Pi \) to alphabets \( A \) and \( C \), which are straightforward given the above construction. Thus, given an EZ program \( \Pi \), the corresponding propositional EZ-program is:

\[
\mathcal{E}(\Pi) = \{ \mu_V(\Pi) \cup \mu_R(\Pi), \mu_A(\Pi), \mu_C(\Pi), \gamma, \mu_D(\Pi) \}.
\]

### A.1 Global Constraints in Language EZ

The global constraints supported by the EZ language include:

- \text{all.different}(V)\footnote{Note that the first argument is of the type \([f(t_1, t_2, \ldots, t_m)/k] \) while the other two are of type \([r(t_1, t_2, \ldots, t_m)/k] \), hence the different expansions.}, where \( V \) is a list of variables. This constraint, available only in the \( fd \) domain, ensures that all the variables in \( V \) are assigned unique values. Typically, the implementation of the corresponding algorithm found in constraint solvers is incomplete. Global constraint \text{all.distinct}(V)\footnote{See for example \url{http://sicstus.sics.se/sicstus/docs/3.7.1/html/sicstus_33.html}}, which provides a complete implementation of the algorithm, is also supported.
- \text{assignment}(X, Y)\footnote{Typically, the implementation of the corresponding algorithm found in constraint solvers is incomplete. Global constraint \text{all.distinct}(V)\footnote{See for example \url{http://sicstus.sics.se/sicstus/docs/3.7.1/html/sicstus_33.html}}, which provides a complete implementation of the algorithm, is also supported.}, where \( X \) and \( Y \) are lists of \( n \) variables whose domain is \( 1 \cdot n \). The constraint is satisfied if, for every \( i, j \), \( X_i = j \) if and only if \( Y_j = i \).
- \text{circuit}(V)\footnote{Typically, the implementation of the corresponding algorithm found in constraint solvers is incomplete. Global constraint \text{all.distinct}(V)\footnote{See for example \url{http://sicstus.sics.se/sicstus/docs/3.7.1/html/sicstus_33.html}}, which provides a complete implementation of the algorithm, is also supported.}, where \( V \) is a list of \( n \) variables whose domain is \( 1 \cdot n \). The constraint is satisfied by an assignment \( V_1 = v_1, V_2 = v_2, \ldots, V_n = v_n \) if the directed graph with nodes \( 1 \ldots n \) and arcs \( (1, v_1), (2, v_2), \ldots, (n, v_n) \) forms a Hamiltonian cycle.
- \text{count}(M, V, \circ, E)\footnote{Typically, the implementation of the corresponding algorithm found in constraint solvers is incomplete. Global constraint \text{all.distinct}(V)\footnote{See for example \url{http://sicstus.sics.se/sicstus/docs/3.7.1/html/sicstus_33.html}}, which provides a complete implementation of the algorithm, is also supported.}, where \( M \) is an integer or variable, \( V \) a list of variables, \( \circ \) an arithmetic comparison operator, and \( E \) an integer or variable. This constraint is satisfied if the number, \( c \), of elements of \( V \) that equal \( M \) is such that \( c \circ E \).
• cumulative \((S, D, R, L)\), where \(S\) is a list of variables, \(D\) and \(R\) are lists of non-negative integers matching the length of \(S\), and \(L\) is an integer or a variable. This constraint, which is only available in the \(fd\) domain, is typically used in scheduling problems. In that context, \(S\) represent the start times of a set of jobs, \(D\) provides the duration of those jobs, and \(R\) the resources they require. \(L\) is the amount of resources available at any time step. Intuitively, the constraint assigns start times to the jobs so that, at any time, no more than an amount \(L\) of resources is used.

• disjoint \(2(X, W, Y, H)\), where \(X, Y\) are lists of variables and \(W, H\) are lists of integers defining the coordinates and dimensions of rectangles. For example, if \(X = [x_1, \ldots], Y = [y_1, \ldots], W = [w_1, \ldots], H = [h_1, \ldots]\), one of the rectangles they describe has top-left vertex \((x_1, y_1)\) and bottom-right vertex \((x_1 + w_1, y_1 + h_1)\). This constraint is only available in the \(fd\) domain, and assigns values to the variables so that the corresponding rectangles do not overlap.

• element \((I, V, E)\), where \(I\) is an integer or variable, \(V\) a list of variables, and \(E\) an integer or variable. This constraint is satisfied if the \(I^{th}\) element of \(V\) is \(E\).

• minimum \((M, V)\) and maximum \((M, V)\), where \(M\) is a variable or integer and \(V\) is a list of variables. These constraints are satisfied if minimum or maximum of \(V\) equals \(M\).

• scalar product \((C, X, \circ, E)\), where \(C\) is a list of integers, \(X\) is a list of variables, \(\circ\) is an arithmetic comparison operator, and \(E\) is an integer or variable. The intuitive meaning of this constraint is that the scalar product, \(p\), of the elements of \(C\) and \(X\) must be such that \(p \circ E\).

• serialized \((S, D)\), where \(S\) is a list of variables and \(D\) is a list of integers, intuitively denoting start time and duration of jobs. The constraint assigns start times to the jobs so that their execution does not overlap, and can be viewed as a special case of cumulative.

• sum \((V, \circ, E)\), where \(V\) is a list of variables, \(\circ\) an arithmetic comparison operator, and \(E\) is an integer or a variable. This constraint assigns value to the variables so that \((\sum_{v \in V} v) \circ E\) is satisfied.