Structured Declarative Language

Mario Alviano, Carmine Dodaro* and Ilaria R. Vasile

DeMaCS, University of Calabria, 87036 Rende (CS), Italy

Abstract

Structured Declarative Language (SDL) emerges as a powerful tool for addressing the complexities of combinatorial search and optimization problems. In this paper, we introduce SDL as a higher-level abstraction that provides a clear and intuitive specification language, with its semantics defined through translation into Answer Set Programming (ASP). SDL offers several advantages over directly writing in ASP, including flexibility in attribute handling, improved code readability, and enhanced error tolerance. Key features of SDL include the irrelevance of attribute order, seamless management of attribute arity changes, and the use of qualifying names for attribute access. Additionally, SDL incorporates features such as automatic attribute tracking and type differentiation, contributing to a more intuitive and reliable problem-solving process.

Keywords

Knowledge Representation, Answer Set Programming, Combinatorial Search and Optimization, Code Maintainability

1. Introduction

Combinatorial search and optimization represent foundational concepts within the realm of computer science and mathematics, focusing on the exploration of vast solution spaces to identify optimal configurations or arrangements. These problems arise across several domains, ranging from logistics and scheduling [1, 2] to telecommunications and bioinformatics [3, 4]. At its core, combinatorial search involves the systematic exploration of all possible combinations or permutations of a given set of elements to identify a solution that satisfies specified criteria. However, the sheer size and complexity of solution spaces in combinatorial problems often render brute-force search infeasible. Consequently, the field of optimization emerges as a crucial discipline, aiming to devise efficient algorithms and methodologies to navigate these vast solution spaces and identify optimal or near-optimal solutions.

Answer Set Programming (ASP) [5, 6] provides a robust framework for addressing the complexity inherent in combinatorial search and optimization problems. Its declarative nature and expressive syntax make it particularly well-suited for modeling and solving problems with discrete decision variables and combinatorial constraints. One of the key advantages of ASP is its ability to handle non-monotonic reasoning, allowing for the representation of incomplete or uncertain information and the formulation of default assumptions. This feature is crucial in combinatorial optimization, where decisions often need to be made under uncertainty or with partial information. Furthermore, ASP offers powerful modeling constructs for expressing complex combinatorial constraints and objectives, including aggregates and weak constraints [7]. These constructs enable the concise representation of diverse optimization problems, ranging from scheduling and planning to resource allocation and configuration [8, 9, 10]. Additionally, ASP solvers leverage efficient algorithms and optimization techniques to search for solutions within large solution spaces [11]. These solvers employ advanced search strategies, constraint propagation, and conflict analysis to efficiently explore the search space and identify optimal or near-optimal solutions.

However, when working with a long-standing ASP codebase, several challenges and limitations may arise, impacting maintenance, readability, and robustness. Here are some downsides of ASP, particularly

CILC 2024: 39th Italian Conference on Computational Logic, June 26–28, 2024, Rome, Italy

*Corresponding author.

mario.alviano@unical.it (M. Alviano); carmine.dodaro@unical.it (C. Dodaro); vasileilaria@gmail.com (I. R. Vasile)

https://alviano.net/ (M. Alviano); https://www.cdodaro.eu (C. Dodaro)

0000-0002-2052-2063 (M. Alviano); 0000-0002-5617-5286 (C. Dodaro)

concerning long-standing codebases:

- *Propagating Changes:* In a long-standing ASP codebase, the interdependencies between different parts of the code can create challenges when making modifications. Changes made to one section may necessitate updates in other areas to maintain consistency and functionality. Due to the declarative nature of ASP, where rules and facts are interconnected, it is often difficult to predict the ripple effects of a change, leading to a laborious process of identifying and updating affected components. This can introduce overhead in maintenance efforts and increase the risk of introducing unintended errors.
- *Use of Object Variables:* ASP allows the use of object variables to refer to terms, providing flexibility in rule formulation. However, in a long-standing codebase, the extensive use of object variables can lead to ambiguity and confusion. Object variables may be reused across different rules or sections of the code, making it challenging to track their meaning and scope. This ambiguity increases the likelihood of errors, such as unintentional variable reuse or misinterpretation of variable bindings, particularly when making changes or debugging the code.
- *Lack of Semantic Annotations:* ASP lacks built-in mechanisms for semantically annotating terms or sub-terms within rules. In a long-standing codebase, this absence of clear semantic annotations can hinder code readability and comprehension. Developers may struggle to discern the intended purpose or meaning of certain terms, leading to confusion and potential misinterpretation. Without explicit semantic annotations, understanding the logic and semantics of complex rules becomes more reliant on developer expertise and manual inspection, increasing the risk of errors and hindering code maintenance and evolution.

Example 1. Consider atoms of the following forms: `cab(cab_id, driver)` to represent cabs and associated drivers; `customer(cust_id, name, title)` to represent customers and their data; `assign(cust_id, cab_id)` to assign cabs to customers. Suppose the above atoms are used in the following rules (possibly part of a larger codebase):

```
% assign one cab to every customer
{assign(C,C') : cab(C',D)} = 1 :- customer(C,N,T).

% don't assign more than one customer to each cab
:- cab(C,D), #count{C' : assign(C',C)} > 1.
```

If the number or order of attributes associated with any of the above atoms change, all the codebase must be adapted. For example, if the association between cabs and drivers is moved in a different predicate, both rules above must be modified, even if neither one nor the other really use such an association. Also note that object variables are often associated with short names, and it is easy to clash with an already used name; in the first rule above, we had to use `C'` for the cab because `C` was already used for the customer (and we named variables of the second rule in a different order). Moreover, there is no clear way to distinguish the several attributes in an atom or rule, and one can unintentionally switch attributes and obtain wrong results; is the first attribute of `assign` a customer or a cab? ■

Addressing these downsides in a long-standing ASP codebase requires careful attention to code organization, documentation, and best practices. Strategies such as modularization, clear naming conventions, and documentation of variable bindings can help mitigate the challenges associated with propagating changes and managing object variables. Additionally, establishing conventions for semantic annotation or leveraging external documentation tools can enhance code readability and maintainability, facilitating the long-term management of the codebase. In this context, our contribution lies in the development and definition of *Structured Declarative Language (SDL)* as a higher-level abstraction specifically tailored to address the challenges of combinatorial search and optimization. SDL serves as a specification language that provides a clear and concise way to express optimization problems in a declarative manner, while its semantics are defined through a translation into ASP. This approach offers several advantages over directly writing in ASP:

1. *Abstraction and Simplification:* SDL abstracts away the complexities of ASP syntax, providing a more intuitive and concise representation of combinatorial optimization problems. By defining problems at a higher level of abstraction, SDL facilitates clearer problem specification and solution understanding.
2. *Qualifying Names for Attribute Access:* SDL enables access to attributes and their components through qualifying names, enhancing code readability and comprehension. This feature provides clarity in referencing attributes and reduces ambiguity, making it easier for developers to understand and modify SDL code. In fact, qualifying names add flexibility in attribute handling, allowing for seamless modification of attribute order and arity. The order of attributes becomes irrelevant, and modifications to arity do not necessitate alterations in rules where the affected attributes are not utilized. This enhances code maintainability and reduces the risk of errors during code evolution.
3. *Automatic Attribute Tracking:* SDL includes an automatic system for tracking the origin of attributes, such as in `edge(node(X), node(Y))`. This facilitates error detection and debugging by providing insights into attribute usage and dependencies. In particular, SDL ensures automatic adherence to the typical contract of the equivalence relation within attribute handling, maintaining consistency and integrity in problem representation. Indeed, unlike ASP, where objects cannot distinguish between different types of strings and integers, SDL provides the capability to differentiate between various types of objects. For instance, SDL allows for explicit differentiation between an attribute representing a node ID and an attribute representing a paper ID (`node(X)` vs `paper(X)`, instead of two integers).

Example 2 (Continuing Example 1)**.** In SDL, the reported portion of the codebase is encoded as follows (syntax and semantics of SDL are defined in Sections 3–4):

```
record Cab:       id: int, driver: str;
record Customer:  id: int, name: str, title: str;
record Assign:    customer: Customer, cab: Cab;

guess from Customer exactly 1
    Assign from Cab
        where Assign.customer == Customer and Assign.cab == Cab;
deny from Cab having
    count {Assign.customer from Assign where Assign.cab == Cab} > 1;
```

The structure of the processed data is declared by the record instructions, so to associate names with attributes. The number and order of attributes is completely irrelevant, as attributes are accessed via qualified names such as `Assign.customer` (or also `Assign.customer.title`). Every instruction only depends on the record parts that are explicitly mentioned, so for example the **`guess`** and **`deny`** instructions above stay the same if the `title` attribute from `Customer` is removed. The attributes of `Assign` are tracked to their origin, that is, `Customer` and `Cab`, so that it is unlikely to misuse one for the other. ■

Overall, our contribution of SDL as an abstraction layer for combinatorial search and optimization not only simplifies problem specification but also enhances code maintainability, readability, and error tolerance. By providing a clear and intuitive language for expressing optimization problems, SDL empowers users to tackle complex combinatorial challenges more effectively and efficiently.

2. Background

2.1. Answer Set Programming

All sets and sequences considered in this paper are finite if not differently specified. Let $\mathbf{P}$, $\mathbf{F}$, $\mathbf{V}$ be fixed nonempty sets of *predicate names*, *function names* and *variables*. Function and predicate names

are associated an *arity*, a non-negative integer; set $\mathbf{F}$ includes at least one function name of arity 0. *Terms* are inductively defined as follows: variables are terms; if $f \in \mathbf{F}$ has arity n, and $t_1, \ldots, t_n$ are terms, then $f(t_1, \ldots, t_n)$ is a term (parentheses are omitted if $n = 0$). A *ground term* is a term with no variables. An *atom* is of the form $p(t_1, \ldots, t_n)$, where $p \in \mathbf{P}$ has arity n. A *ground atom* is an atom with no variables. A *literal* is an atom possibly preceded by the default negation symbol `not`; they are referred to as positive and negative literals. A *conjunction* $conj(\overline{t})$ is a possibly empty sequence of literals involving the terms $\overline{t}$. An *aggregate* is of one of the forms

$$\#\text{SUM}\{t_{a,1}, \overline{t'_1} : conj_1(\overline{t_1});\ \cdots\ ;\ t_{a,n}, \overline{t'_n} : conj_n(\overline{t_n})\} \odot t_g \tag{1}$$

$$\#\text{MIN}\{t_{a,1}, \overline{t'_1} : conj_1(\overline{t_1});\ \cdots\ ;\ t_{a,n}, \overline{t'_n} : conj_n(\overline{t_n})\} \odot t_g \tag{2}$$

$$\#\text{MAX}\{t_{a,1}, \overline{t'_1} : conj_1(\overline{t_1});\ \cdots\ ;\ t_{a,n}, \overline{t'_n} : conj_n(\overline{t_n})\} \odot t_g \tag{3}$$

where $n \geq 1$, $\odot \in \{<, \leq, \geq, >, =, \neq\}$ is a binary comparison operator, each $conj_i(\overline{t_i})$ is a conjunction, each $\overline{t'_i}$ is a possibly empty sequence of terms, and t_g and each $t_{a,i}$ are terms. Let

$$\#\text{COUNT}\{\overline{t'_1} : conj_1(\overline{t_1});\ \cdots\ ;\ \overline{t'_n} : conj_n(\overline{t_n})\} \odot t_g$$

be syntactic sugar for

$$\#\text{SUM}\{1, \overline{t'_1} : conj_1(\overline{t_1});\ \cdots\ ;\ 1, \overline{t'_n} : conj_n(\overline{t_n})\} \odot t_g.$$

A *choice* is of the form

$$t_1 \leq \{atoms\} \leq t_2 \tag{4}$$

where $atoms$ is a possibly empty sequence of atoms, and t_1, t_2 are terms. Let $\bot$ be syntactic sugar for $1 \leq \{\} \leq 1$ (used for *strong constraints*, and possibly omitted to lighten the notation). A *penalty* is expressed as $[c@l]$, where c, l are terms referred to as *cost* and *level*. A *rule* is of one of the forms

$$head \ \texttt{:-}\ body. \tag{5}$$

where $head$ is an atom or a choice or a penalty, and $body$ is a possibly empty sequence of literals and aggregates. (Symbol `:-` is omitted if $body$ is empty.) For a rule r, let $H(r)$ denote the atom or choice or penalty in the head of r; let $B^{\Sigma}(r)$, $B^{+}(r)$ and $B^{-}(r)$ denote the sets of aggregates, positive and negative literals in the body of r; let $B(r)$ denote the set $B^{\Sigma}(r) \cup B^{+}(r) \cup B^{-}(r)$. If $H(r)$ is a penalty, let $cost(r)$ denote its cost and let $level(r)$ denote its level; in this case, r is also called a *weak constraint*.

Example 3. Consider the following set of rules representing the encoding for the Maximal Clique problem:

$$\begin{array}{lrl}
r_1: & 0 \leq \{in(X)\} \leq 1 & \texttt{:-}\ node(X). \\
r_2: & edge(Y, X) & \texttt{:-}\ edge(X, Y). \\
r_3: & \bot & \texttt{:-}\ in(X),\ in(Y),\ X < Y,\ \texttt{not}\ edge(X, Y). \\
r_4: & size(N) & \texttt{:-}\ \#\text{COUNT}\{X : in(X)\} = N. \\
r_5: & [1@1] & \texttt{:-}\ node(X),\ \texttt{not}\ in(X).
\end{array}$$

r_1 is a choice, r_2 and r_4 are rules where $\#\text{COUNT}\{X : in(X)\} = N$ is an aggregate, r_3 is strong constraint, and r_5 is a weak constraint, with $cost(r_5) = 1$, and $level(r_5) = 1$. ■

A variable X occurring in $B^{+}(r)$ is a *global variable*. Other variables occurring among the terms $\overline{t}$ of some aggregate in $B^{\Sigma}(r)$ of the form (1)–(3) are *local variables*. And any other variable occurring in r is an *unsafe variable*. A *safe rule* is a rule with no *unsafe variables*. A *program* Π is a set of safe rules. Let Π_w denote the program comprising all and only the weak constraints of Π. Let Π_h denote the program $\Pi \setminus \Pi_w$. A substitution σ is a partial function from variables to ground terms; the application of σ to an expression E is denoted by $E\sigma$. Let $instantiate(\Pi)$ be the (infinite) set of rules obtained from rules of Π by substituting global variables with ground terms, in all possible ways; note that local variables

are still present in $instantiate(\Pi)$. The Herbrand base of Π, denoted $base(\Pi)$, is the (infinite) set of ground atoms occurring in $instantiate(\Pi)$.

The language of ASP supports a richer syntax. For the purpose of this article, we mention the possibility to combine terms in expressions and compare expressions with binary comparators with the natural interpretation. Moreover, each atom occurring in a choice of the form (4) can be associated with a conjunctive condition, using the syntax $p(\overline{t}) : condition$; in many cases (essentially, if aggregates are not recursive), it is possible to replace $p(\overline{t}) : condition$ with $p'(\overline{t})$, where p' is a fresh predicate (a predicate not occurring elsewhere), by adding to the program the rule $p'(\overline{t})$:- $condition$. Finally, t_1 and t_2 are optional in (4), and when absent their default values are essentially 0 and ω.

Example 4 (Continuing Example 3)**.** Let Π_{run} be the set of rules of Example 3 extended with the following rules:

$$\begin{aligned} r_6 &: \quad node(1). \\ r_7 &: \quad node(2). \\ r_8 &: \quad edge(1,2). \end{aligned}$$

Then, the following rules

$$\begin{aligned}
r_1 &: \quad 0 \leq \{in(1)\} \leq 1 \quad \text{:- } node(1). \\
r_2 &: \quad 0 \leq \{in(2)\} \leq 1 \quad \text{:- } node(2). \\
r_3 &: \quad edge(2,1) \quad \text{:- } edge(1,2). \\
r_4 &: \quad \bot \quad \text{:- } in(1),\ in(2),\ 1 < 2,\ \texttt{not}\ edge(1,2). \\
r_5 &: \quad size(0) \quad \text{:- } \#\textsc{count}\{1 : in(1), 2 : in(2)\} = 0. \\
r_6 &: \quad size(1) \quad \text{:- } \#\textsc{count}\{1 : in(1), 2 : in(2)\} = 1. \\
r_7 &: \quad size(2) \quad \text{:- } \#\textsc{count}\{1 : in(1), 2 : in(2)\} = 2. \\
r_8 &: \quad [1@1] \quad \text{:- } node(1),\ \texttt{not}\ in(1). \\
r_9 &: \quad [1@1] \quad \text{:- } node(2),\ \texttt{not}\ in(2).
\end{aligned}$$

are part of $instantiate(\Pi_{run})$. ■

An *interpretation* is a set of ground atoms. (Note that we are only considering finite interpretations, as those involving an infinite number of atoms are not relevant for our work.) For an interpretation I, relation $I \models \cdot$ is defined as follows: for a ground atom $p(\overline{c})$, $I \models p(\overline{c})$ if $p(\overline{c}) \in I$, and $I \models \texttt{not}\ p(\overline{c})$ if $p(\overline{c}) \notin I$; for a conjunction $conj(\overline{t})$, $I \models conj(\overline{t})$ if $I \models \alpha$ for all $\alpha \in conj(\overline{t})$; for an aggregate α of the form (1)–(3), the aggregate set of α w.r.t. I, denoted $aggset(\alpha, I)$, is $\{\langle t_{a,i}, \overline{t'_i}\rangle\sigma \mid i = 1..n, conj_i(\overline{t_i})\sigma \in I$, for some substitution $\sigma\}$; if α is of the form (1), $I \models \alpha$ if $(\sum_{\langle c_a, \overline{c'}\rangle \in aggset(\alpha,I)} c_a) \odot t_g$ is a true expression over integers; if α is of the form (2), $I \models \alpha$ if $(\min_{\langle c_a, \overline{c'}\rangle \in aggset(\alpha,I)} c_a) \odot t_g$ is a true expression; if α is of the form (3), $I \models \alpha$ if $(\max_{\langle c_a, \overline{c'}\rangle \in aggset(\alpha,I)} c_a) \odot t_g$ is a true expression; for a choice α of the form (4), $I \models \alpha$ if $t_1 \leq |I \cap atoms| \leq t_2$ is a true expression over integers; for a penalty $[w@l]$, $I \models [w@l]$ always; for a rule r with no global variables, $I \models B(r)$ if $I \models \alpha$ for all $\alpha \in B(r)$, and $I \models r$ if $I \models H(r)$ whenever $I \models B(r)$; for a program Π, $I \models \Pi$ if $I \models r$ for all $r \in instantiate(\Pi)$, or equivalently for all $r \in instantiate(\Pi_h)$. The *cost* associated with an interpretation is defined as

$$cost(\Pi, I) := \sum_{r \in instantiate(\Pi_w)\ :\ I \models B(r)} cost(r) \cdot \omega^{level(r)} \tag{6}$$

where ω is the first uncountable ordinal.

For a rule r of the form (5) and an interpretation I, let $expand(r, I)$ be the following set:

$$expand(r, I) := \{p(\overline{c})\ \text{ :- }\ body. \mid p(\overline{c}) \in I \text{ occurs in } H(r)\}.$$

The *reduct* of Π w.r.t. I is the program comprising the expanded rules of $instantiate(\Pi)$ whose body is true w.r.t. I, that is,

$$reduct(\Pi, I) := \bigcup_{r \in instantiate(Pi),\ I \models B(r)} expand(r, I).$$

An *answer set* of Π is an interpretation A such that $A \models \Pi$ and no $I \subset A$ satisfies $I \models \mathit{reduct}(\Pi, A)$. Let $AS(\Pi)$ be the set of answer sets of Π. A is an *optimal answer set* of Π if $A \in AS(\Pi)$ and no $I \in AS(\Pi)$ satisfies $\mathit{cost}(\Pi, I) > \mathit{cost}(\Pi, A)$. Let $AS^*(\Pi)$ be the set of optimal answer sets of Π.

Example 5 (Continuing Example 4)**.** $AS(\Pi_{run})$ comprises the following set of answer sets:

$$
\begin{array}{ll}
A_1: & \{node(1), node(2), edge(1,2), edge(2,1), size(0)\} \\
A_2: & \{node(1), node(2), edge(1,2), edge(2,1), in(2), size(1)\} \\
A_3: & \{node(1), node(2), edge(1,2), edge(2,1), in(1), size(1)\} \\
A_4: & \{node(1), node(2), edge(1,2), edge(2,1), in(1), in(2), size(2)\}
\end{array}
$$

where $\mathit{cost}(\Pi_{run}, A_1) = 2$, $\mathit{cost}(\Pi_{run}, A_2) = \mathit{cost}(\Pi_{run}, A_3) = 1$, $\mathit{cost}(\Pi_{run}, A_4) = 0$. Therefore, A_4 is an optimal answer set and $AS^*(\Pi_{run}) = \{A_4\}$. ■

3. SDL Syntax

To ease the presentation of the language, we group the supported instructions in three groups, namely *structure instructions*, *query instructions* and *model instructions*.

3.1. Structure and Query Instructions

Structure instructions are used to declare the shape of processed data. There is only one kind of structure instruction, defined next. A *record structure* is declared as

$$\textbf{record}\ \mathit{RecordName}\colon \mathit{Attributes}; \tag{7}$$

where $\mathit{RecordName}$ is a unique name and $\mathit{Attributes}$ is a comma-separated list of key-value pairs of the form $\mathit{attribute\_name} : \mathit{attribute\_type}$. In the scope of its record, every $\mathit{attribute\_name}$ is a unique name. Every $\mathit{attribute\_type}$ is either a *primitive type* among **int** and **str**, or a record name (different from $\mathit{RecordName}$).

Query instructions are used to shape the output to produce when the specification is evaluated. There is only one kind of query instruction, defined next. An *output directive* has the form

$$\textbf{show}\ \mathit{RecordNames}; \tag{8}$$

where $\mathit{RecordNames}$ is a comma-separated list of record names.

Example 6 (Continuing Example 3)**.** Consider the following structure and query instructions:

```
record Node: id: int;
record Edge: first: Node, second: Node;
record In:   node: Node;
record Size: value: int;
show In, Size;
```

They shape the records and output for Maximal Clique. ■

3.2. Model Instructions

Within each model instruction, a *record name alias* has the form

$$\mathit{RecordName}\ \textbf{as}\ \mathit{alias} \tag{9}$$

where alias is a possibly different name to identify $\mathit{RecordName}$ in the model instruction. When alias is equal to $\mathit{RecordName}$, the record name alias can be simply written as $\mathit{RecordName}$. A *signed record name alias* is a record name alias possibly preceded by the keyword **not**.

Attributes of a record can be accessed using the *dot operator* as in many object-oriented programming languages, that is, starting with the alias of the record and using a dot (`.`) followed by the attribute name. If the attribute value is a record, its attributes can be accessed as well with another dot operation. In the following, by *value* we refer to the alias of a record, a dot operation or an immediate value (integer and string constants). Values can be combined to form *value expressions* in the common way, by using binary operators (i.e., `+`, `-`, `*`, `/`) and parentheses. In the following, we use the terms *integer expression* and *string expression* based on the type of the result of the expression. A *Boolean expression* has the form

$$\mathit{ValueExpression} \odot \mathit{ValueExpression}' \tag{10}$$

where $\odot$ is a binary comparator (i.e., `<`, `<=`, `>=`, `>`, `==`, `!=`), and $\mathit{ValueExpression}$, $\mathit{ValueExpression}'$ are value expressions (of the same type).

A *simple from fragment* has the form

$$\textbf{from}\ \mathit{SignedRecordNameAliases}\ \textbf{where}\ \mathit{BooleanExpressions} \tag{11}$$

where $\mathit{SignedRecordNameAliases}$ is a comma-separated list of signed record name aliases, and $\mathit{BooleanExpressions}$ is an **and**-separated list of Boolean expressions. If $\mathit{BooleanExpressions}$ is empty, the simple from fragment can be written as follows: **from** $\mathit{SignedRecordNameAliases}$.

An *aggregate element* has the form

$$\mathit{ValueExpressions}\ \mathit{SimpleFromFragment} \tag{12}$$

where $\mathit{ValueExpressions}$ is a comma-separated list of value expressions, and $\mathit{SimpleFromFragment}$ is an optional simple from fragment. An *aggregate* has the form

$$\mathit{AggregationFunction}\ \{\mathit{AggregateElements}\} \odot \mathit{ValueExpression} \tag{13}$$

where $\mathit{AggregationFunction}$ is one of **sum**, **count**, **min**, **max**; $\mathit{AggregateElements}$ is a semicolon-separated list of aggregate elements; $\odot$ is a binary comparator; and $\mathit{ValueExpression}$ is a value expression. If $\mathit{AggregationFunction}$ is **sum**, the first value expression in every aggregate element must be an integer expression. If $\mathit{AggregationFunction}$ is either **sum** or **count**, then $\mathit{ValueExpression}$ must be an integer expression. A *from fragment* has either the form (11) or the form

$$\textbf{from}\ \mathit{SignedRecordNameAliases}\ \textbf{where}\ \mathit{BooleanExpressions}\ \textbf{having}\ \mathit{Aggregates} \tag{14}$$

where $\mathit{Aggregates}$ is an **and**-separated list of aggregates.

A *definition* has the form

$$\textbf{define}\ \mathit{RecordNameAlias}\ \mathit{FromFragment};\tag{15}$$

where $\mathit{RecordNameAlias}$ is a record name alias and $\mathit{FromFragment}$ is a from fragment.

Example 7 (Continuing Example 6)**.** Consider the following definitions:

```
define Edge as self from Edge as other
where self.first == other.second and self.second == other.first;
define Size having count {In.node from In} == Size.value;
```

The first definition is intended to enforce the symmetric closure of `Edge`. The second definition counts the `In` elements. ■

A *cardinality restriction* has one of the forms

$$\textbf{exactly}\ \mathit{IntegerExpression} \tag{16}$$

$$\textbf{at least}\ \mathit{IntegerExpression} \tag{17}$$

at most *IntegerExpression* (18)

at least *IntegerExpression* **and at most** *IntegerExpression′* (19)

where *IntegerExpression*, *IntegerExpression′* are integer expressions. A *guess element* has the form

RecordNameAlias SimpleFromFragment (20)

where *RecordNameAlias* is a record name alias and *SimpleFromFragment* is an optional simple from fragment. A *guess* instruction has the form

guess *FromFragment CardinalityRestriction GuessElements*; (21)

where *FromFragment* is an optional from fragment, *CardinalityRestriction* is an optional cardinality restriction, and *GuessElements* is a space-separated list of guess elements.

Example 8 (Continuing Example 6)**.** Consider the following guess instruction:

```
guess from Node at most 1
    In where Node == In.node;
```

It is intended to guess a subset of `Node` in the record `In`. ■

A *deny* instruction has one of the forms

deny *FromFragment*; (22)

deny *FromFragment* **or pay** *Cost* **at** *Level*; (23)

where *FromFragment* is a from fragment, *RecordNameAliases* is a **or**-separated list of record name aliases, and *Cost* and *Level* are integer expressions.

Example 9 (Continuing Example 6)**.** Consider the following deny instructions:

```
deny from In as in1, In as in2, not Edge
where in1.node == Edge.first and
      in2.node == Edge.second and
      in1.node < in2.node;

deny from Node, not In
where In.node == Node
or pay 1 at 1;
```

The first instruction is intended to ensure that `In` nodes are connected. The second instruction is intended to penalize nodes outside of the `In` selection. ■

4. SDL Semantics

Each record name occurring in a specification must be declared by exactly one record structure instruction of the form (7). The record name *depends* on all other record names occurring in the record structure instruction. The directed graph having record names as nodes, and dependencies as arcs must be acyclic. The record name is associated with a unique predicate name (usually the same name) with arity given by the number of attributes in the record structure instruction. Record name aliases and attributes are inductively mapped to terms by as follows: **int** and **str** are mapped to fresh variables (variables not occurring elsewhere); a record name alias of the form (9) is mapped to $RecordName(A_1, \ldots, A_n)$, where $A_1, \ldots, A_n$ are the terms obtained by mapping the attributes in the record structure instruction declaring $RecordName$. Signed record name aliases are mapped analogously, possibly prepending `not` if present. The dot operation is left-associative. A sequence of dot operations starts by an alias, which is associated with an atom by the mapping given above. Each dot operation $Base.AttributeName$

restricts the mapping of $Base$ (a record name alias or a sequence of dot operations) as follows: if $RecordName(A_1, \ldots, A_n)$ is the mapping of $Base$, and $AttributeName$ is the i-th attribute of the record $RecordName$, then $Base.AttributeName$ is mapped to A_i. Expressions are mapped naturally (with the identity mapping).

Example 10 (Continuing Example 6)**.** Record `Node` is mapped as `node(ID)`, whereas record `Edge` is mapped as `edge(node(ID1), node(ID2))`, where `ID`, `ID1` and `ID2` are fresh variables. The dot operation `Edge.first` is mapped to `node(ID1)`, and `Edge.first.id` is mapped to variable `ID1`. ■

A simple from fragment of the form (11) is mapped to a comma-separated list of literals (a conjunction) comprising the mappings of all elements in $SignedRecordNameAliases$ and $BooleanExpressions$. An aggregate element of the form (12) is mapped to $terms : condition$, where $terms$ are the terms mapping $ValueExpressions$, and $condition$ is the mapping of $SimpleFromFragment$. An aggregate of the form (13) is mapped to $\#AggregateFunction\{elements\} \odot term$, where $elements$ is a semi-colon separated list of the mappings of $AggregateElements$, and $term$ is the mapping of $ValueExpression$.

Example 11 (Continuing Example 7)**.** Aggregate `#count {In.node from In} == 0` is mapped to `#count{X: in(X)} = 0` ■

A from fragment of the form (14) is mapped to a comma-separated list of literals comprising the mappings of all elements in $SignedRecordNameAliases$, $Aggregates$ and $BooleanExpressions$. A *definition* of the form (15) is mapped to a rule of the form (5) such that $head$ is the mapping of $RecordNameAlias$ and $body$ is the mapping of $FromFragment$. For instance, the two definitions from Example 7 are essentially mapped to r_2 and r_4 shown in Example 3.

A guess element of the form (20) is mapped to $atom : condition$, where $atom$ is the mapping of $RecordNameAlias$ and $condition$ is the mapping of $SimpleFromFragment$ (if any, and the tautology $0 = 0$ otherwise). A guess expression of the form (21) is mapped to a rule of the form (5) such that $body$ is the mapping of $FromFragment$, and $head$ is the choice $t_1 \leq \{elements\} \leq t_2$, where t_1 and t_2 are obtained by mapping the integer expressions in the $CardinalityRestriction$ (equations 16–19), and $elements$ is a semi-colon separated list comprising the mappings of all elements in $GuessElements$. For instance, the guess instruction from Example 8 is essentially mapped to r_1 shown in Example 3.

A deny instruction of the form (22) is mapped to a rule of the form (5) such that $head$ is $\bot$, and $body$ is the mapping of $FromFragment$. The mapping of (23) is obtained similarly, but $head$ is the penalty $[c@l]$, where c is the mapping of $Cost$ and l is the mapping of $Level$. For instance, the deny instructions from Example 9 are essentially mapped to r_3 and r_5 shown in Example 3.

Given a specification S, the program comprising the rules obtained by applying the above mappings on the structure and model instructions in S is denoted Π_S. Program Π_S can be processed by modern ASP systems to address common computational tasks, among them answer set search and enumeration. The answer sets of Π_S are filtered according to the query instructions in S. Essentially, only atoms whose predicate name occurs in some output directive are shown, and all other atoms are hidden. Let $AS(S)$ and $AS^*(S)$ be obtained from $AS(\Pi_S)$ and $AS^*(\Pi_S)$ by removing all atoms whose predicate does not occur in any output directive of S.

Example 12 (Continuing Examples 6–9)**.** The specification is essentially mapped to the Π_{run} program, whose answer sets are shown in Example 5. Actually, the mapping includes a few function names to track the meaning and scope of terms. Hence, the actual program defining the semantics of the specification is the following:

```
node(1).  node(2).  edge(node(1), node(2)).

edge(node(SF), node(SS)) :- edge(node(OF), node(OS)),
  node(SF) == node(OS), node(SS) == node(OF).

size(Value) :- #count{node(Id) : in(node(Id))} = Value.

0 <= {in(node(Id')) : node(Id) == node(Id')} <= 1 :- node(Id).
```

```
:- in(node(Id)), in(node(Id')), not edge(node(F), node(S)),
   node(Id) == node(F), node(Id') == node(S), node(Id) < node(Id').
[1@1] :- node(Id), not in(node(Id')), node(Id') == node(Id).
```

According to the query instructions in the specification, the answer sets are filtered as follows: $\{size(0)\}$, $\{in(node(2)), size(1)\}$, $\{in(node(1)), size(1)\}$, $\{in(node(1)), in(node(2)), size(2)\}$. ■

5. Use Case

In this section, we show some practical applications of SDL by presenting case studies showcasing its usage in modeling various combinatorial problems, namely Graph Coloring, Hamiltonian Path, and Nurse Scheduling. The SDL specifications reported in this section, along with their translations to ASP, are available at https://github.com/dodaro/SDL/tree/main/examples.

5.1. Graph Coloring

The Graph Coloring problem is a classic combinatorial problem in graph theory. Given an undirected graph, the goal is to assign colors to its vertices in such a way that two adjacent vertices do not share the same color. The following is a SDL specification for solving the Graph Coloring problem:

```
1  record Node: id: int;
2  record Edge: first: Node, second: Node;
3  record Color: value: str;
4  record Assign: node: Node, color: Color;
5  guess from Node
6     exactly 1
7     Assign
8        from Color
9           where Assign.node == Node and Assign.color == Color;
10 deny from Assign as a1, Assign as a2, Edge
11    where a1.node != a2.node and
12        a1.color == a2.color and
13        Edge.first == a1.node and Edge.second == a2.node;
14 show Assign;
```

The definition of the records is provided in lines 1–4. Lines 5–9 define how colors are assigned to nodes, with each node being assigned exactly one color. Lines 10–13 are used to prohibit the assignments where two different connected nodes share the same color. Finally, line 14 is used to filter the output to records `Assign`.

5.2. Hamiltonian Cycle

The Hamiltonian cycle problem is a well-known problem in graph theory that involves finding a cycle that traverses every vertex in a graph exactly once, returning to the starting vertex.

The following is a SDL specification for solving the Hamiltonian Cycle problem:

```
1  record Node: id: int;
2  record Arc: first: Node, second: Node;
3  record InCycle: first: Node, second: Node;
4  record Reached: node: Node;
5  record Start: node: Node;
6  guess from Node
7     exactly 1
```

```
8        InCycle
9           from Arc
10             where InCycle.first == Arc.first and
11                InCycle.second == Arc.second and
12                Node == Arc.first;
13    deny from Node
14       having count {
15          InCycle.first from InCycle where Node == InCycle.second
16       } != 1;
17    define Reached
18       from Start
19          where Start.node == Reached.node;
20    define Reached as r1
21       from Reached as r2, InCycle
22          where r2.node == InCycle.first and r1.node == InCycle.second;
23    deny from Node, not Reached
24       where Node == Reached.node;
25    show InCycle;
```

The definition of the records is outlined in lines 1 through 5. Lines 6–12 establish criteria for selecting arcs to be included in the path, ensuring that each node has precisely one outgoing arc within the path. Lines 13–16 are used to prohibit any node from having two or more incoming arcs within the path. Lines 17–22 define delineate the conditions under which nodes are considered reached. Specifically, the starting node is always reached (lines 17–19), and a node X is reached if there exists an arc in the path from a previously reached node Y to X (lines 20–22). Lines 23–24 assert that each node must be reached. Finally, line 25 is used to filter the output to records `InCycle`.

5.3. Nurse Scheduling

The Nurse Scheduling problem (NSP) consists of allocating nurses to shifts, satisfying some requirements (here we focus on the variant analyzed in [12]). The schedule spans one year. There are three working shifts, namely *morning* (id 1) lasting 7 hours, *afternoon* (id 2) lasting 7 hours, and *night* (id 3) lasting 10 hours. There are three non-working shifts, namely *special rest after two consecutive nights* (id 4), *rest* (id 5), and *holiday* (id 6). Moreover, the schedule must comply with hospital, nurse and balance requirements. Hospital requirements specify cardinality restrictions on the number of nurses in each shift. Nurse requirements impose cardinality restrictions on the number of working hours per year, ensuring that each nurse receives 30 days of holidays, and that the starting time of a shift is at least 24 hours later than the previous shift. Additionally, each nurse must have at least two rest days within each fourteen-day window. Moreover, after two consecutive working nights, one special rest day is allocated, distinct from the regular rest days. Balance requirements specify cardinality restrictions on the number of times a nurse can be assigned to morning, afternoon, and night shifts. Below we provide a SDL specification.

```
1  record Nurse: id: int;
2  record Day: id: int;
3  record Shift: id: int, hours: int;
4  record HoursLimits: min: int, max: int;
5  record DayLimits: shift: Shift, min: int, max: int;
6  record NurseLimits: shift: Shift, min: int, max: int;
7  record Assign: nurse: Nurse, shift: Shift, day: Day;
8  show Assign;
```

```
9   guess from Day, Nurse exactly 1
10     Assign from Shift
11     where Assign.nurse == Nurse and Assign.shift == Shift and
          Assign.day == Day;
```

The definition of the records is outlined in lines 1–7, line 8 is used to filter the output as seen before, and lines 9–11 are used to assign nurses to exactly one shift for each day. Hospital requirements are encoded as follows:

```
1   deny from Day, NurseLimits
2      having count {
3         Assign.nurse from Assign
4            where Assign.shift == NurseLimits.shift and Assign.day == Day
5      } > NurseLimits.max;
6   deny from Day, NurseLimits
7      having count {
8         Assign.nurse from Assign
9            where Assign.shift == NurseLimits.shift and Assign.day == Day
10     } < NurseLimits.min;
```

Above, lines 1–5 enforce that for each day and shift, the number of working nurses does not exceed a given threshold. Similarly, lines 6–10 are used to enforce that the number of working nurses is not below a specified minimum. Nurse requirements are encoded as follows:

```
1   deny from Nurse, HoursLimits
2      having sum {
3            Shift.hours, Assign.day from Assign, Shift
4               where Assign.nurse == Nurse and Assign.shift == Shift
5      } > HoursLimits.max;
6   deny from Nurse, HoursLimits
7      having sum {
8            Shift.hours, Assign.day from Assign, Shift
9               where Assign.nurse == Nurse and Assign.shift == Shift
10     } < HoursLimits.min;
11  deny from Nurse
12     having count {
13        Assign.day from Assign
14           where Assign.nurse == Nurse and Assign.shift.id == 6
15     } != 30;
16  deny from Nurse, Assign as a1, Assign as a2
17     where a1.nurse == Nurse and a2.nurse == Nurse and a2.shift.id <
          a1.shift.id and a1.day.id == a2.day.id+1 and a2.shift.id <= 3
          and a1.shift.id <= 3;
18  deny from Nurse, Day
19     where Day.id <= 352
20     having count {
21        Assign.day from Assign
22           where Assign.nurse == Nurse and Assign.shift.id == 5 and
                Assign.day.id >= Day.id and Assign.day.id < Day.id+14
23     } < 2;
24  deny from Assign as a1, Assign as a2, Assign as a3
25     where a1.nurse.id == a2.nurse.id and a1.nurse.id == a3.nurse.id and
```

```
        a1.shift.id == 4 and a2.shift.id == 3 and a3.shift.id == 3 and
        a2.day.id == a1.day.id-2 and a3.day.id == a1.day.id-1;
26  deny from Assign as a1, not Assign as a2
27    where a1.nurse == a2.nurse and a1.shift.id == 4 and a2.shift.id ==
        3 and a2.day.id == a1.day.id-2;
28  deny from Assign as a1, not Assign as a2
29    where a1.nurse == a2.nurse and a1.shift.id == 4 and a2.shift.id ==
        3 and a2.day.id == a1.day.id-1;
```

Above, lines 1–5 (resp. 6–10) enforce that a nurse does not exceed a maximum (resp. minimum) number of working hours per year. Lines 11–15 ensure that each nurse is allocated precisely 30 days of holidays per year (recall that the shift id for holidays is 6). Lines 16–17 guarantee that the starting time of a shift is at least 24 hours later than the previous shift. Lines 18–23 establish that each nurse receives a minimum of two rest days within every fourteen-day period (recall that the shift for rest is 5). Finally, lines 24–29 ensure that a nurse is assigned to a special rest day shift if only if they were assigned to the night shift during the previous two days (recall that the shift ids for night and special rest day are 3 and 4, respectively). Balance requirements are encoded as follows:

```
1   deny from Nurse, DayLimits
2     having count {
3       Assign.day from Assign where Assign.shift == DayLimits.shift and
          Assign.nurse == Nurse
4     } > DayLimits.max;
5   deny from Nurse, DayLimits
6     having count {
7       Assign.day from Assign where Assign.shift == DayLimits.shift and
          Assign.nurse == Nurse
8     } < DayLimits.min;
```

Above, lines 1–4 (resp. 5–8) ensure that a nurse cannot be assigned to a given shift for a number of days greater (resp. lower) than a predetermined threshold.

6. Related Work

Numerous frameworks and methodologies have been developed to address combinatorial search and optimization problems, often leveraging declarative programming paradigms. Among these, ASP and OntoDLV [13] stand out as significant contributors to the field of knowledge representation and reasoning. ASP represents a declarative programming paradigm tailored for solving combinatorial search and optimization problems. ASP solvers, including CLINGO [11] and DLV [14], employ efficient algorithms and optimization techniques to explore solution spaces. ASP's expressive syntax and non-monotonic reasoning capabilities make it suitable for modeling complex problem domains and reasoning about incomplete or uncertain information. OntoDLV extends the DLV system by integrating support for reasoning with ontologies expressed in Description Logics (DL). This integration enables OntoDLV to provide a powerful platform for ontology-driven reasoning and query answering. By combining DL-based ontology reasoning with ASP, OntoDLV offers enhanced expressiveness and reasoning capabilities. It facilitates seamless integration of ontological knowledge with ASP rules, particularly beneficial in domains such as the semantic web, bioinformatics, and data integration.

While ASP and OntoDLV have significantly contributed to combinatorial optimization, our work on SDL introduces distinctive features and advantages. SDL serves as a higher-level abstraction specifically designed to address combinatorial search and optimization challenges. It offers a clear and intuitive specification language with semantics defined through translation into ASP. SDL simplifies problem specification by abstracting away complexities of ASP syntax, enhancing code readability

and comprehension. Notably, SDL provides flexibility in attribute handling, including irrelevance of attribute order and seamless management of attribute arity changes. It offers qualifying names for attribute access, improving code maintainability. Additionally, SDL incorporates features for enhanced error tolerance, such as automatic attribute tracking and type differentiation, thereby facilitating a more intuitive and reliable problem-solving process.

Regarding the downsides of ASP mentioned in the introduction, we observe that OntoDLV can refer attributes by their name, so that the order of attributes is often immaterial. Moreover, changing the structure of a record does not necessarily require to modify portions of the knowledge base that are not directly interested by the change. However, OntoDLV heavily depends on the use of object variables, which SDL can completely avoid thanks to the use of qualifying names. Finally, while OntoDLV opted for extending the syntax of ASP with additional constructs, SDL introduces a new structured language. Hence, in order to use OntoDLV, a programmer must first have a clear understanding of ASP syntax, which is instead not required to use SDL (as a full understanding of relational algebra is not required to be proficient with SQL).

Finally, we mention that the syntax of SDL is inspired by SQL [15]. Specifically, both languages share keywords such as **`from`**, **`where`**, **`as`**, **`having count`**, and **`having sum`**. However, SQL is designed for managing and manipulating relational databases, enabling users to create, read, update, and delete data, as well as perform tasks like querying for specific information, adding new records, updating existing data, and generating reports. In contrast, SDL is used to specify and describe complex combinatorial and optimization problems.

7. Conclusion

In this paper, we have introduced Structured Declarative Language (SDL) as a higher-level abstraction specifically designed to address the intricacies of combinatorial search and optimization. By providing a clear and intuitive specification language, SDL offers a powerful framework for expressing optimization problems in a declarative manner, while its semantics are defined through translation into Answer Set Programming (ASP). Throughout our exploration of SDL, we have highlighted several key advantages that distinguish it from directly writing in ASP. The flexibility in attribute handling, including the irrelevance of attribute order and the seamless management of attribute arity changes, enhances code maintainability and reduces the risk of errors during code evolution. Additionally, the use of qualifying names for attribute access improves code readability and comprehension, facilitating easier understanding and modification of SDL code. Furthermore, SDL incorporates features to enhance error tolerance and robustness in problem specification, such as automatic attribute tracking and differentiation between object types. These features contribute to a more intuitive and reliable problem-solving process, empowering users to tackle complex combinatorial challenges with greater efficiency and confidence. As future work, further research and development efforts can focus on extending the capabilities of SDL, refining its syntax, and exploring additional optimization techniques to address an even broader range of real-world challenges. Moreover, in this paper, we have shown the semantics of SDL through its translation to ASP. For future work, we plan to define a formal semantics for SDL and develop additional translations to alternative formalisms, such as Constraint Programming [16] and Satisfiability Modulo Theories [17]. Finally, we mention that a preliminary implementation of a tool for translating SDL to ASP is available at https://github.com/dodaro/SDL.

Acknowledgments

This work was partially supported by Italian Ministry of University and Research (MUR) under PRIN project PRODE “Probabilistic declarative process mining”, CUP H53D23003420006 under PNRR project FAIR “Future AI Research”, CUP H23C22000860006, under PNRR project Tech4You “Technologies for climate change adaptation and quality of life improvement”, CUP H23C22000370006, and under PNRR project SERICS “SEcurity and RIghts in the CyberSpace”, CUP H73C22000880001; by Italian Ministry of

Health (MSAL) under POS projects CAL.HUB.RIA (CUP H53C22000800006) and RADIOAMICA (CUP H53C22000650006); by Italian Ministry of Enterprises and Made in Italy under project STROKE 5.0 (CUP B29J23000430005); and by the LAIA lab (part of the SILA labs). Mario Alviano and Carmine Dodaro are members of Gruppo Nazionale Calcolo Scientifico-Istituto Nazionale di Alta Matematica (GNCS-INdAM).

References

[1] D. Abels, J. Jordi, M. Ostrowski, T. Schaub, A. Toletti, P. Wanko, Train scheduling with hybrid answer set programming, Theory Pract. Log. Program. 21 (2021) 317–347.

[2] M. M. S. El-Kholany, M. Gebser, K. Schekotihin, Problem decomposition and multi-shot ASP solving for job-shop scheduling, Theory Pract. Log. Program. 22 (2022) 623–639.

[3] M. Gebser, T. Schaub, S. Thiele, P. Veber, Detecting inconsistencies in large biological networks with answer set programming, Theory Pract. Log. Program. 11 (2011) 323–360.

[4] N. Leone, F. Ricca, Answer set programming: A tour from the basics to advanced development tools and industrial applications, in: Reasoning Web, volume 9203 of *LNCS*, Springer, 2015, pp. 308–326.

[5] G. Brewka, T. Eiter, M. Truszczynski, Answer set programming at a glance, Commun. ACM 54 (2011) 92–103. URL: https://doi.org/10.1145/2043174.2043195. doi:10.1145/2043174.2043195.

[6] M. Gelfond, V. Lifschitz, Classical negation in logic programs and disjunctive databases, New Gener. Comput. 9 (1991) 365–386. URL: https://doi.org/10.1007/BF03037169. doi:10.1007/BF03037169.

[7] F. Calimeri, W. Faber, M. Gebser, G. Ianni, R. Kaminski, T. Krennwallner, N. Leone, M. Maratea, F. Ricca, T. Schaub, Asp-core-2 input language format, Theory Pract. Log. Program. 20 (2020) 294–309. URL: https://doi.org/10.1017/S1471068419000450. doi:10.1017/S1471068419000450.

[8] E. Erdem, M. Gelfond, N. Leone, Applications of answer set programming, AI Mag. 37 (2016) 53–68. URL: https://doi.org/10.1609/aimag.v37i3.2678. doi:10.1609/AIMAG.V37I3.2678.

[9] P. Cappanera, M. Gavanelli, M. Nonato, M. Roma, Logic-based benders decomposition in answer set programming for chronic outpatients scheduling, Theory Pract. Log. Program. 23 (2023) 848–864. URL: https://doi.org/10.1017/s147106842300025x. doi:10.1017/S147106842300025X.

[10] M. Mochi, G. Galatà, M. Maratea, Master surgical scheduling via answer set programming, J. Log. Comput. 33 (2023) 1777–1803. URL: https://doi.org/10.1093/logcom/exad035. doi:10.1093/LOGCOM/EXAD035.

[11] M. Gebser, B. Kaufmann, T. Schaub, Conflict-driven answer set solving: From theory to practice, Artif. Intell. 187 (2012) 52–89. URL: https://doi.org/10.1016/j.artint.2012.04.001. doi:10.1016/J.ARTINT.2012.04.001.

[12] C. Dodaro, M. Maratea, Nurse scheduling via answer set programming, in: M. Balduccini, T. Janhunen (Eds.), Proc. of LPNMR, volume 10377 of *LNCS*, Springer, 2017, pp. 301–307. URL: https://doi.org/10.1007/978-3-319-61660-5_27. doi:10.1007/978-3-319-61660-5_27.

[13] F. Ricca, L. Gallucci, R. Schindlauer, T. Dell'Armi, G. Grasso, N. Leone, Ontodlv: An asp-based system for enterprise ontologies, J. Log. Comput. 19 (2009) 643–670.

[14] W. T. Adrian, M. Alviano, F. Calimeri, B. Cuteri, C. Dodaro, W. Faber, D. Fuscà, N. Leone, M. Manna, S. Perri, F. Ricca, P. Veltri, J. Zangari, The ASP system DLV: advancements and applications, Künstliche Intell. 32 (2018) 177–179.

[15] E. F. Codd, A relational model of data for large shared data banks, Commun. ACM 13 (1970) 377–387. URL: https://doi.org/10.1145/362384.362685. doi:10.1145/362384.362685.

[16] F. Rossi, P. van Beek, T. Walsh (Eds.), Handbook of Constraint Programming, volume 2 of *Foundations of Artificial Intelligence*, Elsevier, 2006. URL: https://www.sciencedirect.com/science/bookseries/15746526/2.

[17] C. W. Barrett, R. Sebastiani, S. A. Seshia, C. Tinelli, Satisfiability modulo theories, in: Handbook of Satisfiability - Second Edition, volume 336 of *FAIA*, IOS Press, 2021, pp. 1267–1329. URL: https://doi.org/10.3233/FAIA201017. doi:10.3233/FAIA201017.