Proceedings of the 21st International Conference on Principles of Knowledge Representation and Reasoning
KR in the Wild

ASP Chef: Draw and Expand

Mario Alviano , Luis Angel Rodriguez Reiners
DEMACS, University of Calabria, Via Bucci 30/B, 87036 Rende (CS), Italy
{mario.alviano, luis.reiners}@unical.it

Abstract

ASP Chef is a versatile tool built upon the principles of Answer Set Programming (ASP), offering a unique approach to problem-solving through the concept of ASP recipes. In this paper, we explore two key components of ASP Chef: the Graph ingredient and one of its extension mechanisms for registering new ingredients. The Graph ingredient serves as a fundamental feature within ASP Chef, allowing users to interpret instances of a designed predicate to construct graphs from the data. Through this capability, ASP Chef facilitates the visualization and analysis of complex relationships and structures inherent in various domains. Furthermore, ASP Chef offers a flexible extension mechanism that empowers users to register new recipes as custom ingredients. These custom ingredients, defined by sequences of mappings from interpretations to interpretations, can be stored locally within the local storage of the browser. This enables users to expand the capabilities of ASP Chef to suit their specific needs and use cases, fostering a collaborative environment where users can share and reuse custom ingredients seamlessly. Notably, the addition of new ingredients does not impose requirements on the utilization of recipes that employ them, underscoring the modular and interoperable design of ASP Chef.

1 Introduction

In the realm of Knowledge Representation and Reasoning (KRR), the ability to encode and manipulate complex domains in a computer-readable format is paramount. Such encoding enables the automation of tasks, synthesis of information, and execution of sophisticated reasoning processes by specialized engines (Balduccini, Barborak, and Ferrucci 2023). Answer Set Programming (ASP), a declarative approach to problem-solving, has garnered increasing attention from both researchers and practitioners for its ability to combine high-level linguistic constructs with advanced solving algorithms for combinatorial search and optimization (Brewka, Eiter, and Truszczynski 2011; Erdem, Gelfond, and Leone 2016; Lifschitz 2019; Kaminski et al. 2023; Alviano et al. 2023).

Despite its theoretical computational power, ASP is not designed to serve as a general-purpose programming language. Instead, ASP is most effective when employed to address specific tasks within a broader pipeline, where input and output are expected to interface with modules potentially implemented in different paradigms (Bertolucci et al. 2021). This necessity for interoperability poses a significant challenge for ASP practitioners, who often find themselves tasked with mapping data between different formats at each module of the pipeline. Recognizing this challenge, ASP CHEF[1] was recently introduced as a novel approach to ASP problem-solving (Alviano, Cirimele, and Reiners 2023). At its core lies the concept of the ASP recipe, a chain of ingredients representing distinct computational tasks. Each ingredient encapsulates operations typical of ASP engines, such as combinatorial search and optimization, as well as any other data manipulation task. By processing ASP recipes, ASP CHEF provides users with a flexible framework to employ ASP directly through logic rules or indirectly via pre-defined operations implemented using ASP engines. For example, consider the problem of *Fighting with the gang of Billy the*

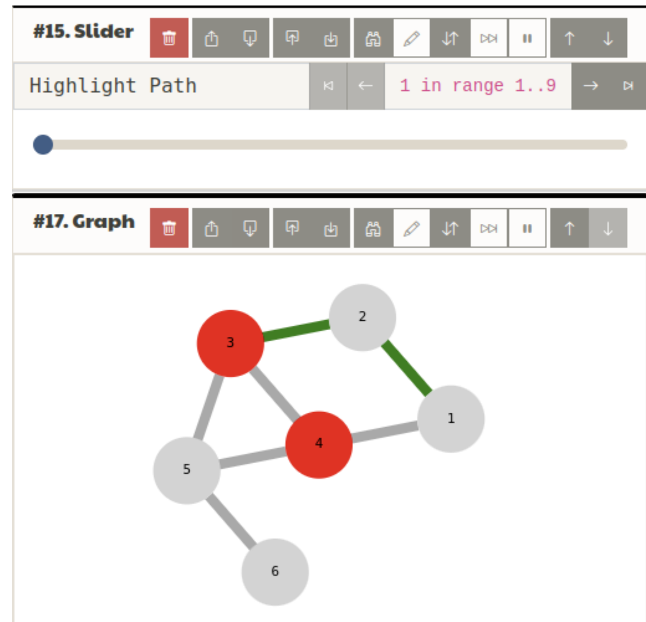

Figure 1: All 9 paths of length 3 includes at least one selected town (in red). Path 1 is highlighted in green.

[1]http://asp-chef.alviano.net/, or http://asp-chef.alviano.net/s/ for tutorials and examples

Kid[2], a scenario from the *LP/CP Programming Contest series*.[3] Here, the goal is to monitor a minimum number of towns such that each path visiting a specified number of distinct towns includes at least one monitored town (see Figure 1 for an example). While this problem can be tackled using a monolithic ASP encoding approach (using uninterpreted function symbols to compute paths at grounding time), ASP CHEF offers a simpler alternative by breaking down the problem into a series of ASP ingredients, each addressing a specific computational task: paths are computed by enumerating answer sets of a program, and provided in input to a second program that selects a minimum number of towns covering all paths.[4]

ASP CHEF adopts a uniform format for the input and output of each operation, which simplifies the design of ASP recipes and enhances their compatibility with diverse modules and data formats. Furthermore, ASP CHEF provides a web application, which serves as a platform for implementing ASP pipelines and experimenting with ASP recipes in practical settings. It is important to note that ASP CHEF is not intended to be an IDE for ASP, but instead it aims to be a low code way to define ASP-powered pipelines. ASP CHEF can be used for fast prototyping, in courses (e.g., set up tutorials, with visualizations, and ask students to fill in the encodings) and in production environments (e.g., to post-process data of other tools addressing hard computational tasks). In fact, a pipeline can process data of any sort, as soon as it is in the format of relational facts. Content that is not naturally representable in such a format can be Base64-encoded (via operations provided by ASP CHEF), and subsequently interpreted by other ingredients of the pipeline (for an analogy, email attachments are Base64-encoded in the body of the email and opened by external applications). The use of Base64 enables several interesting use cases, as for example the possibility to provide input as Comma-Separated Values (CSV) or in JavaScript Object Notation (JSON), and easily obtain a relational representation by using specialized operations available in ASP CHEF. Additionally, Base64 is used to encode templates (Alviano et al. 2024) such as the symmetric closure of a binary relation: the ingredient Base64-encodes the rules for the symmetric closure of the specified relation so that they can be used by subsequent ingredients in the recipe, such as SearchModels and Optimize, for example to enforce that a guessed relation is symmetric.

Interestingly, an ASP recipe is essentially a sequence of mappings from interpretations to interpretations. Each ingredient within the recipe contributes to this mapping process, performing specific operations or computations on the input interpretations and producing transformed interpretations as output. Therefore, a recipe can be thought of as the composition of the mappings defined by its individual ingredients. Consider a simple analogy: just as a recipe in cooking consists of a series of steps that transform raw ingredients into a finished dish, an ASP recipe consists of a series of operations that transform input interpretations into desired output interpretations. Each ingredient within the recipe corresponds to a specific step in this transformation process. Given this perspective, it becomes natural to view a recipe itself as a new ingredient that can be used in broader recipes. Just as individual ingredients can be combined to create more complex dishes, recipes can be combined to create more sophisticated computational processes. This abstraction allows users to modularize and reuse computational logic, promoting flexibility, scalability, and maintainability in ASP-based problem-solving.

By treating recipes as ingredients, ASP CHEF enables users to build upon existing computational logic, abstracting away implementation details and focusing on higher-level problem-solving tasks. Among the recipes that users may want to reuse in several other recipes, visualizations in terms of graphs stand out as particularly valuable. Graph visualizations provide intuitive and insightful representations of data structures, relationships, and patterns, making complex information more accessible and understandable to users. In ASP CHEF, the Graph operation facilitates the generation of graph visualizations as side output within recipes. The operation takes input interpretations and constructs a graph representation based on the instances of a specified predicate as a side output. These instances specify nodes and links, with additional properties defining attributes such as color, label, shape, and directionality. The operation then translates these instances into a graphical representation, by interpreting the properties associated with nodes and links in the graph. For instance, the color property might determine the color of a node or link, while the label property could provide a textual label to annotate nodes or links.

All in all, this paper presents an extension mechanism of ASP CHEF that allows users to register new recipes as custom ingredients, thereby expanding the capabilities of the tool to address diverse computational tasks. We use the Graph operation to provide concrete examples to demonstrate the extension mechanism in action, and we illustrate how users can define custom recipes that leverage the Graph operation to generate graph visualizations from input data. In particular, we aim to provide readers with a comprehensive understanding of how ASP CHEF can be extended and customized to address a wide range of computational tasks.

2 Background

Section 2.1 introduces the ASP notions used in examples; readers familiar with ASP can skip this section. Section 2.2 provides some background on ASP CHEF that was already presented at ASPOCP'23 (Alviano, Cirimele, and Reiners 2023) and an invited speech at CILC'24.

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-

[2]https://github.com/lpcp-contest/lpcp-contest-2019/blob/master/billykid/billykid.md

[3]https://github.com/lpcp-contest

[4]A tutorial on this problem is available online at https://asp-chef.alviano.net/s/tutorials/billy-the-kid

negative integer;[5] 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* $conj(\bar{t})$ is a possibly empty sequence of literals involving the terms $\bar{t}$. An *aggregate* is of the form

$$\#func\{t_a, \overline{t'} : conj(\bar{t})\} \odot t_g \tag{1}$$

where $func \in \{\text{SUM}, \text{MIN}, \text{MAX}\}$ is an aggregation function, $\odot \in \{<, \leq, \geq, >, =, \neq\}$ is a binary comparison operator, $conj(\bar{t})$ is a conjunction, $\overline{t'}$ is a possibly empty sequence of terms, and t_g and t_a are terms. Let

$$\#\text{COUNT}\{\overline{t'} : conj(\bar{t})\} \odot t_g$$

be syntactic sugar for

$$\#\text{SUM}\{1, \overline{t'} : conj(\bar{t})\} \odot t_g.$$

A *choice* is of the form

$$t_1 \leq \{atoms\} \leq t_2 \tag{2}$$

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

$$head \ \ \texttt{:-} \ \ body. \tag{3}$$

where $head$ is an atom or a choice or a penalty, and $body$ is a possibly empty sequence of literals and aggregates.[6] 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, let $level(r)$ denote its level, and let $dist(r)$ denote its distinguishing terms; in this case, r is also called a *weak constraint*.

A variable X occurring in $B^{+}(r)$ is a *global variable*. Other variables occurring among the terms $\bar{t}$ of some aggregate in $B^{\Sigma}(r)$ of the form (1) 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 define constants, combine terms in expressions and compare expressions with binary comparators with the natural interpretation. Moreover, each atom occurring in a choice of the form (2) can be associated with a conjunctive condition, using the syntax $p(\bar{t}) : condition$; in many cases (essentially, if aggregates are not recursive), it is possible to replace $p(\bar{t}) : condition$ with $p'(\bar{t})$, where p' is a fresh predicate (a predicate not occurring elsewhere), by adding to the program the rule $p'(\bar{t}) \ \texttt{:-} \ condition$. Finally, t_1 and t_2 are optional in (2), and when absent their default values are essentially 0 and ω (the first uncountable ordinal).

Example 1. *Consider line 3 in Figure 2:*

```
{wall(X,Y)} :- grid(X,Y).
```

The above choice rule is used to define the search space of the problem, that is, it is possible to build a wall in every cell of the grid. (Note that the choice is equivalent to `0 <= {wall(X,Y)} <=` ω*, or also to* `0 <= {wall(X,Y)} <= 1`*.)*

Consider now line 1:

```
grid(X,Y) :- size(R,C), X=1..R,
Y=1..C.
```

It defines the cells of the grid given its `R`$\times$`C` *size:* `X` *is in the integer interval* `1..R`*, and* `Y` *is in* `1..C`*.*

Finally, consider line 14:

```
:~ wall(X,Y).   [ 1@2, X,Y]
```

The above weak constraint provides a penalty of 1 at the second level for each wall built in the grid. ■

An *interpretation* is a set of ground atoms.[7] For an interpretation I, relation $I \models \cdot$ is defined as follows: for a ground atom $p(\bar{c})$, $I \models p(\bar{c})$ if $p(\bar{c}) \in I$, and $I \models \texttt{not}\ p(\bar{c})$ if $p(\bar{c}) \notin I$; for a conjunction $conj(\bar{t})$, $I \models conj(\bar{t})$ if $I \models \alpha$ for all $\alpha \in conj(\bar{t})$; for an aggregate α of the form (1), the aggregate set of α w.r.t. I, denoted $aggset(\alpha, I)$, is $\{\langle t_a, \overline{t'}\rangle\sigma \mid conj(\bar{t})\sigma \in I$, for some substitution $\sigma\}$; if $func$ is SUM, $I \models \alpha$ if $(\sum_{\langle c_a, \overline{c'}\rangle \in aggset(\alpha,I)} c_a) \odot t_g$ is a true expression over integers; if $func$ is MIN, $I \models \alpha$ if $(\min_{\langle c_a, \overline{c'}\rangle \in aggset(\alpha,I)} c_a) \odot t_g$ is a true expression; if $func$ is MAX, $I \models \alpha$ 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 (2), $I \models \alpha$ 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

[5] A predicate (or function) name and its arity are possibly written as p/n, where p is a name and n an integer. Note that $p/1$, $p/2$, $p/3$ can be simultaneously members of $\mathbf{P}$ (or $\mathbf{F}$), that is, overloading of predicate (and function) names is permitted.

[6] Symbol `:-` is omitted if *body* is empty. The penalty of a rule, if any, is written after the body in the actual syntax supported by mainstream ASP systems.

[7] Note that we are only considering finite interpretations, as those involving an infinite number of atoms are not relevant for our work.

```
1 grid(X,Y) :- size(R,C), X = 1..R, Y = 1..C.
2 % guess walls, leave sufficient space to all POI
3 {wall(X,Y)} :- grid(X,Y).
4 :- poi(X,Y,D), wall(X',Y'), |X-X'| + |Y-Y'| < D.
5 % attack all cells in the border
6 attack(X,Y) :- size(R,C), X = 1, Y = 1..C, not wall(X,Y).
7 attack(X,Y) :- size(R,C), X = R, Y = 1..C, not wall(X,Y).
8 attack(X,Y) :- size(R,C), Y = 1, X = 1..R, not wall(X,Y).
9 attack(X,Y) :- size(R,C), Y = C, X = 1..R, not wall(X,Y).
10 % expand the attack if no wall blocks it
11 delta(X,Y) :- X = -1..1, Y = -1..1, |X|+|Y| = 1.
12 attack(X+DX,Y+DY) :- attack(X,Y), delta(DX,DY), grid(X+DX,Y+DY), not wall(X+DX,Y+DY).
13 :- poi(X,Y,_), attack(X,Y).  % fail if a POI is under attack
14 :~ wall(X,Y).    [ 1@2, X,Y]  % minimize walls
15 :~ attack(X,Y). [-1@1, X,Y]  % break ties by maximizing the number of cells under attack
```

Figure 2: ASP encoding for the *Fortress* problem: Given a R × C grid with points-of-interest (POI), each POI requiring some area free-of-walls in order to be functional, build walls to protect all POI from external attacks. The less walls the better, the less cells inside the walls the better.

for all $r \in instantiate(\Pi_h)$. The *cost* associated with an interpretation is defined as

$$cost(\Pi, I) := \sum_{\langle c,l,\overline{t}\rangle \in penset(\Pi_w, I)} c \cdot \omega^l \tag{4}$$

where ω is the first uncountable ordinal, and

$$penset(\Pi_w, I) := \{\langle cost(r), level(r), terms(r)\rangle \mid r \in instantiate(\Pi_w),\ I \models B(r)\}.$$

For a rule r of the form (3) and an interpretation I, let $expand(r, I)$ be the following set:

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

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,

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

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

Example 2 (Continuing Example 1)**.** *Let I comprise* `size(1,2)`, `attack(1,1)`, `wall(1,2)`, *and no other atom. The instantiation of the program comprises, among other rules, the following instances of line 6 in Figure 2:*

```
attack(1,1) :- size(1,2), 1 = 1,
               1 = 1..2, not
wall(1,1).
attack(1,2) :- size(1,2), 1 = 1,
               2 = 1..2, not
wall(1,2).
```

```
size(23,23).
poi( 9, 9,3).  poi( 9,11,2).  poi(
   9,13,3).
poi( 9,15,3).  poi(10,10,1).
   poi(10,14,2).
poi(12, 9,6).  poi(12,14,3).
   poi(14,10,1).
poi(14,15,2).  poi(15, 9,3).
   poi(15,14,1).
```

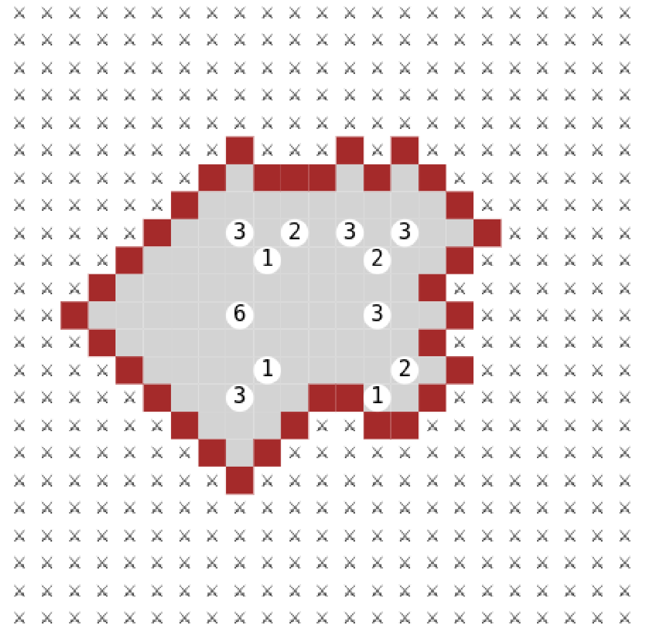

Figure 3: An instance of *Fortress* (top) and a graphical representation of its optimal solution (bottom). An ASP CHEF tutorial based on this problem is available at https://asp-chef.alviano.net/s/tutorials/fortress.

In the program reduct, the above rules are replaced by

```
attack(1,1) :- size(1,2), 1=1, 1 =
1..2.
```

According to lines 14–15, the cost associated with I is $1 \cdot \omega^2 - 1 \cdot \omega^1$. Anyhow, note that I is not a model of the program because, for example, line 11 is not satisfied. A concrete instance of Fortress *and its optimal answer set are shown in Figure 3. The cost associated with the answer set is $34 \cdot \omega^2 - 404 \cdot \omega^1$.* ■

2.2 ASP Recipes and ASP Chef

Let $Base64$ be the function associating every binary string with a longer binary string that can be interpreted as printable ASCII characters; the output string is obtained according to RFC 4648 §4.[8] Let $Base64^{-1}$ be the inverse function of $Base64$.

Example 3. *Let t be the following text:*

```
.,.,4,.
.,1,.,.
.,.,2,.
.,3,.,.
```

Hence, $Base64(t)$ is b := LiwuLDQsLgouLDEsLiwuCi4sLiwyLC4KLiwzLC4sLg==, and $Base64^{-1}(b) = t$. ■

In the following the term *object* refers to any finite element of a fixed universe (comprising, among other elements, strings, numbers, graphs, atoms, programs, and sequences). The notation $[x_1, \ldots, x_n]$ is used to refer to a (finite) *sequence* of $n \geq 0$ objects, where each object x_i is associated with index i. Moreover, by *space* we refer to a (possibly infinite) set of objects, and the Boolean values are denoted by $\mathbf{t}$ and $\mathbf{f}$. Abusing notation, let us associate atoms in an interpretation with indices (starting from 1), that is, in the following we will use the term interpretation for referring to a (finite) sequence of distinct atoms. Moreover, among the interpretations we include the inconsistent interpretation $\bot$ to represent errors. Let $\mathcal{I}$ be the set of all *sequences of interpretations*. An *operation* O is a function with signature $O : \mathcal{I} \longrightarrow \mathcal{I}$, that is, a function receiving in input a sequence of interpretations and producing in output a sequence of interpretations.

Example 4. *Let $\mathcal{O}$ be the operation replacing every atom of the form `__base64__(b)` as follows: if $Base64^{-1}(b)$ is interpreted as comma-separated values (CSV), for each cell in row r and column c whose value is v, atom `__cell__(r,c,v)` is added to the output. If b is the one from Example 3, then $\mathcal{O}([[$`__base64__(b)`$]])$ is $[[I]]$, where I is*

```
__cell__(1,1,".").   __cell__(1,2,".").
__cell__(1,3,4).     __cell__(1,4,".").
__cell__(2,1,".").   __cell__(2,2,1).
__cell__(2,3,".").   __cell__(2,4,".").
__cell__(3,1,".").   __cell__(3,2,".").
__cell__(3,3,2).     __cell__(3,4,".").
__cell__(4,1,".").   __cell__(4,2,3).
__cell__(4,3,".").   __cell__(4,4,".").
```

[8]https://datatracker.ietf.org/doc/html/rfc4648#section-4

Note that integers are mapped to integer terms, and everything else is mapped to string terms. ■

An *operation O with side output space $\mathcal{S}$* is a function with signature $O : \mathcal{I} \longrightarrow \mathcal{I} \times \mathcal{S}$. No particular restriction is imposed to the side output space; common cases include the set of strings, the set of graphs, and the set containing the empty set (to essentially have operations with no side output as a special case); to lighten the notation, if the side output space is $\{\emptyset\}$, we simply omit it. Let $O|_{\mathcal{I}}$ denote the operation obtained from O by discarding the side output.

Example 5. *Let $\mathcal{O}$ be the identity operation (simply forwarding the input), with side output the Base64-decoded content of any atom of the form `__base64__(b)`. For example, if b is the one from Example 3, then the side output of $\mathcal{O}([[$`__base64__(b)`$]])$ is t from Example 3.*

In ASP CHEF*, the $\mathcal{O}$ operation is called OutputEncodedContent. A similar operation, named Output, can be used to show the output produced by the preceding ingredient in the recipe. Both operations are useful to check intermediate results, even if a better alternative is to define graphical visualizations and embed them in a Recipe ingredient (see Section 4).* ■

A *parameterized operation O with parameter space $\mathcal{P}$ and side output space $\mathcal{S}$* is a function with signature $\mathcal{P} \longrightarrow (\mathcal{I} \longrightarrow \mathcal{I} \times \mathcal{S})$, that is, $O\langle P\rangle$ is an operation with side output space $\mathcal{S}$ for each parameter value $P \in \mathcal{P}$ (note that angle brackets are used to denote a *parameterized operation instantiation*). No particular restriction is imposed to the parameter space; common cases include the set of integers, the set of strings, sets of tuples, and the set containing the empty set (to have non-parameterized operations as a special case); to lighten the notation, if the parameter space is $\{\emptyset\}$, we simply omit it.

Example 6. *Let ParseCSV be the operation from Example 4, with parameters $decode_predicate$, $output_predicate$ and $separator$ to customize the processed input predicate (`__base64__` in the example), the predicate of the produced output atoms (`__cell__` in the example), and the separator of the CSV content (`,` in the example).*

Let OutputEncodedContent be the operation from Example 5, with parameters $predicate$ to customize the processed input predicate (`__base64__` in the example). ■

An *ingredient* is an instantiation of a parameterized operation with side output, that is, if O is a parameterized operation with parameter space $\mathcal{P}$ and side output space $\mathcal{S}$, and $P \in \mathcal{P}$ is a parameter value, then $O\langle P\rangle$ is an ingredient. A *recipe* is a tuple of the form $(encode, Ingredients, decode)$, where $Ingredients$ is a (finite) sequence of ingredients, and $encode$ and $decode$ are Boolean values. Intuitively, the input of a recipe is either (the string representation of) a sequence of interpretations in $\mathcal{I}$, or a string to be $Base64$-encoded. The input is processed by the pipeline of ingredients, possibly producing some side output along the way. Finally, some encoded content is possibly decoded. Such an intuition is formalized next. Let $Ingredients$ comprise $n \geq 0$ ingredients $O_1\langle P_1\rangle, \ldots, O_n\langle P_n\rangle$, and let s_{in} be the string in input. The output and side output of the

recipe $(encode, Ingredients, decode)$ given s_{in} are respectively s_{out} and $S_1, \ldots, S_n$ defined as follows:

- I_0 is one of (i) $[[$`__base64__("s")`$]]$, if *encode* is true, where $s = Base64(s_{in})$; (ii) the sequence of interpretations represented by s_{in}, where interpretations are separated by the reserved character § and atoms are represented as facts, if s_{in} conforms to this specification; (iii) $[\bot]$ otherwise, to report an error.
- For each $i = 1, \ldots, n$, let I_i be $O_i\langle P_i\rangle|_{\mathcal{I}}(I_{i-1})$, and S_i be the side output of $O_i\langle P_i\rangle(I_{i-1})$. Essentially, each ingredient of the recipe receives a sequence of interpretations from the previous computational step, and produces in output a sequence of interpretations for the next computational step. Possibly, a side output is also produced.
- Let s_{out} be the string obtained by concatenating the atoms of I_n represented as facts, one per line, and using § as the separator for interpretations. If *decode* is true, s_{out} is further processed by replacing every occurrence of `__base64__(s)` with (the ASCII string associated with) $Base64^{-1}(s)$; in this case, if $Base64^{-1}(s)$ produces an error, then s_{out} is simply the string representation of $[\bot]$.

Let $R(s_{in}) = [s_{out}, S_1, \ldots, S_n]$ denote the fact that s_{out} and $S_1, \ldots, S_n$ are the output and side output of a recipe R given an input string s_{in}.

Example 7. *Let R be $(\mathbf{t}, Ingr, \mathbf{f})$, where Ingr comprises* OutputEncodedContent⟨`__base64__`⟩ *and* ParseCSV⟨`__base64__`, `__cell__`, `,`⟩. *Let b, t be from Example 3, and I be from Example 4. Hence, $R(t)$ is $[[I]], t, \emptyset]$. Indeed, the CSV in input is first mapped to* `__base64__(b)`*, then decoded and shown as a side output, and finally mapped to facts.* ■

ASP CHEF supports more than 80 operations, enabling fast prototyping of pipelines to address combinatorial tasks. Before moving to the next sections, where we focus on the Graph and Recipe operations, let us give an example of the frequently used SearchModels and Optimize operations.

Example 8. *Let SearchModels be the operation with parameters rules and number, replacing each interpretation I in input with up to number answer sets of rules extended with the facts in I. For example, if the recipe from Example 7 is extended with the ingredient* SearchModels$\langle\Pi, 1\rangle$*, where Π is*

```
given((X,Y),V) :- __cell__(X,Y,V), V !=
    ".".
size(S) :- S = #count{X :
    __cell__(X,_,_)}.
square(S') :- size(S), S' = 1..S, S'*S' =
    S.
```

then the interpretation I from Example 4 is extended with

```
given((1,3),4).   given((2,2),1).
given((3,3),2).   given((4,2),3).
size(4).          square(2).
```

Similarly, Optimize has the same parameters but produces optimal answer sets. ■

3 Graph Operation

The Graph operation takes two parameters, namely *predicate* (a predicate name in $\mathbf{P}$) and *echo* (a Boolean value). If *echo* is $\mathbf{t}$, the sequence of interpretations in input is forwarded in output. Otherwise, if *echo* is $\mathbf{f}$, instances of *predicate* are removed from the input interpretations before forwarding them in output. If the input comprises exactly one interpretation, the side output produced by the Graph operation is obtained by processing instances of *predicate*; otherwise, an error is shown as side output. In ASP CHEF, the Graph operation uses D3.js force graphs, but it is possible to fix the position of nodes to obtain other types of visualization.

Instances of *predicate* are expected to specify a graph representation of the interpretation in input. The arity of *predicate* is not fixed, that is, all atoms of the form $predicate(\bar{t})$ in input are processed, regardless of the length of $\bar{t}$. The first term in $\bar{t}$ must be one of `node(ID)`, `link(SOURCE,TARGET)` and `defaults`. The other terms in $\bar{t}$ have the form $property(VALUE)$. Currently, the following properties can be specified for a node:

- `label`, with *VALUE* being a single term to be used as the label of the node (or `none` to associate the node with no label);
- `image`, with *VALUE* being the URL of an image to be drawn on top of the node;
- `color`, with *VALUE* being an HTML color code to be used to draw the node;
- `text_color`, with *VALUE* being an HTML color code for the label of the node (if any);
- `font`, with *VALUE* being a CSS font family for the label of the node (if any);
- `shape`, with *VALUE* being `circle`, `square` or a sequence of (pairs of) integers defining a polygon; the shape is used to draw the node;
- `radius`, with *VALUE* being a positive integer defining the size of the node (or 0 for the default radius) in case `shape` is `circle` or `square`;
- `opacity`, with *VALUE* being an integer giving the percentage of opacity of the node;
- `fx` and `fy`, with *VALUE* being an integer, to fix the position of the node (or `none` to let the position be determined by applying forces);
- `draggable`, with *VALUE* being empty, to let the node be dragged when clicked with the pointer;
- `undraggable`, with *VALUE* being empty, to inhibit dragging of the node.

Regarding links, the following properties can be specified:

- `label`, with *VALUE* being a single term to be used as the label of link (or `none` to associate the link with no label);
- `color`, with *VALUE* being an HTML color code to be used to draw the link;

```
block((row, Row), (Row, Col))
  :- Row = 1..S, Col = 1..S, size(S).
block((col, Col), (Row, Col))
  :- Row = 1..S, Col = 1..S, size(S).
block((sub, Row', Col'), (Row, Col))
  :- Row  = 1..Size,     Col  = 1..Size,
     Row' = (Row-1) / S, Col' = (Col-1) /
    S,
     size(Size), square(S).
```

Figure 4: ASP program to produce the blocks of $s^2 \times s^2$ Sudoku, where s and s^2 are given by predicates `square` and `size`.

- `text_color`, with *VALUE* being an HTML color code for the label of the link (if any);
- `opacity`, with *VALUE* being an integer giving the percentage of opacity of the link;
- `directed`, with *VALUE* being empty, to draw a directed link;
- `undirected`, with *VALUE* being empty, to draw an undirected link.

Finally, the properties to set `defaults` are the following:

- `node_image`, `node_color`, `node_text_color`, `node_font`, `node_shape`, `node_radius`, `node_opacity`, `node_draggable` and `node_undraggable`;
- `link_color`, `link_text_color`, `link_opacity`, `directed` and `undirected`.

If a property does not meet the above format, it is reported and ignored.

Example 9. *A $s^2 \times s^2$ Sudoku is a $s^2 \times s^2$ grid partially filled with digits in the interval $1..s^2$. The goal is to fill in the remaining cells avoiding duplicates in each block (row, column and non-overlapping $s \times s$ box). The CSV in Example 3 actually encodes a 4×4 Sudoku with four given clues. Example 8 provides a fact representation of the instance. Blocks can be materialized by the program in Figure 4. A graphical representation can be obtained by providing to the Graph operation the answer set of the program in Figure 5. Lines 4–7 define the following defaults: nodes are drawn as squares with radius 10. Lines 8–13 define a background node, black, whose size and position are determined from the size of the Sudoku. Lines 14–17 define a node for each cell, and determine its position. Lines 18–20 add labels to the cells with a given clue; the default color of labels is black. Lines 21–24 add blue labels to the guessed cells. The graphical representation is shown in Figure 7, where the solution is obtained by computing the answer set of the program in Figure 6.* ■

4 Recipe Operation and Serialization

A recipe is serialized in a compressed JSON object to obtain a URL that can be interpreted by ASP CHEF. The JSON object comprises the list of ingredients, with their parameters, among other elements such as the Boolean flags *encode*

```
1 #const radius = 10.
2 #const size = 2*radius.
3 #const grid = size + 1.
4 __graph__(defaults,
5   node_shape(square),
6   node_radius(radius)
7 ).
8 __graph__(node(background),
9   color(black),
10  fx(((N+1) * grid + S) / 2),
11  fy(((N+1) * grid + S) / 2),
12  radius((N * grid) / 2 + S)
13 ) :- size(N), square(S).
14 __graph__(node((Row,Col)),
15  fx(Col * grid + BCol),
16  fy(Row * grid + BRow)
17 ) :- block((sub,BRow,BCol), (Row,Col)).
18 __graph__(node(Cell),
19  label(Value)
20 ) :- given(Cell, Value).
21 __graph__(node(Cell),
22  label(Value),
23  text_color(blue)
24 ) :- assign(C,V), not given(C,V).
```

Figure 5: ASP program producing the facts for the Graph ingredient drawing $s^2 \times s^2$ Sudoku, where s and s^2 are given by predicates `square` and `size`, given clues are encoded by predicate `given`, blocks are materialized by the program in Figure 4, and assigned values are given by predicate `assign`.

```
% guess values, matching the given clues
1 <= {assign((R,C), V) : V = 1..S} <= 1
  :- size(S), R = 1..S; C = 1..S.
:- given(C,V), not assign(C,V).
% no duplicates in any block
:- block(B,C), block(B,C'), C != C',
   assign(C,V), assign(C',V).
% all values in every block
:- block(B,_), size(S), V = 1..S,
   #count{C: block(B,C), not
    assign(C,V)}=0.
```

Figure 6: ASP program to solve $s^2 \times s^2$ Sudoku, where s and s^2 are given by predicates `square` and `size`, given clues are encoded by predicate `given`, and blocks are materialized by the program in Figure 4.

		4	
	1		
		2	
	3		

3	2	4	1
4	1	3	2
1	4	2	3
2	3	1	4

Figure 7: Side output of the Graph ingredient processing the facts produced by the programs in Figures 4–5 combined with the facts in Example 8 (left), and with its solution obtained with the program in Figure 6 (right).

and *decode*, and the input for the recipe. This way, the URL can be shared to move a recipe and its input in a different browser, without the need to interact with a remote server (if not for loading the ASP CHEF application). Additionally, URLs can be shortened by storing them in a GitHub repository; the short URL is obtained from the path of the file storing the original URL. This way, the sharing of recipes is further simplified, and ASP CHEF recipes inherit the versioning mechanism of GitHub.

Example 10. *The JSON representation of the recipe (and input) from Example 7 is the following:*

```
{
  "input":
    ".,.,4,.\n.,1,.,.\n.,.,2,.\n.,3,.,.",
  "encode_input": true,
  "decode_output": false,
  "recipe": [
    {
      "operation": "Output Encoded
    Content",
      "options": {
        "predicate": "__base64__"
      }
    },
    {
      "operation": "Parse CSV",
      "options": {
        "decode_predicate": "__base64__",
        "output_predicate": "__cell__",
        "separator": ","
      }
    }
  ]
}
```

Above we simplified the format to ease the presentation. The actual URL encoding the recipe from Example 7 takes 730 characters, and starts with https://asp-chef.alviano.net/#eJzFk9tymz... We stored the recipe in our GitHub repository, and it can be accessed with the short URL https://asp-chef.alviano.net/s/example@KR2024. ■

The Recipe operation takes two parameters, namely *name* (an optional name to give to the recipe ingredient) and *Ingredients* (the actual sequence of ingredients composing the recipe ingredient itself); in the implementation, the list of ingredients is given by the URL serialization of a recipe, possibly shortened, from which Boolean flags and input are simply ignored. Input parameters of the recipe itself must be provided by facts. The sequence of interpretations in input traverses all ingredients of the recipe ingredient, and is therefore processed as the ingredients of the recipe ingredient were part of the main recipe. Formally, if $(\mathbf{f}, \mathit{Ingredients}, \mathbf{f})(s_{in}) = [s_{out}, S_1, \ldots, S_n]$, then the recipe ingredient maps the sequence s_{in} to the sequence s_{out}, and produces the side output $[S_1, \ldots, S_n]$.

Example 11. *Consider a recipe that Base64-encodes the input and whose only ingredient is the recipe https://asp-chef.alviano.net/s/example@KR2024. For the input*

```
foo,bar
```

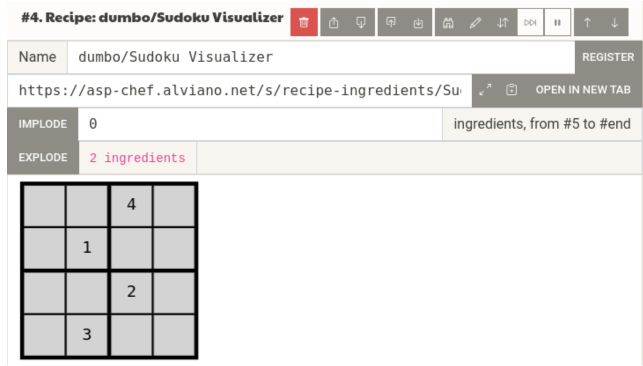

Figure 8: The SudokuVisualizer recipe ingredient. By clicking REGISTER the user has the possibility to add the ingredient as a registered operation.

Operations

su

Meta Supported Models	[⏎]
Symmetric Closure	[1]
Transitive Closure	[2]
&r/dumbo/Sudoku Visualizer	[3]

Figure 9: Registered recipe ingredients are available in the list of operations, so that they can be added to the main recipe as any other operation.

Add &r/dumbo/Sudoku Visualizer Operation

Visualize a Sudoku instance or solution encoded by predicates `given/2` and `assign/2`. Blocks are expected to be already materialized by the following program (where `size/1` and `square/1` give the size of the Sudoku):

```
block((row, Row), (Row, Col)) :- Row = 1..Size, Col = 1.
block((col, Col), (Row, Col)) :- Row = 1..Size, Col = 1.
block((sub, Row', Col'), (Row, Col)) :- Row = 1..Size; C
```

Figure 10: Documentation associated with the registered recipe.

```
Sudoku Visualizer
https://asp-chef.alviano.net/s/recipe-ingredients/Sudoku+Visualizer
Visualize a Sudoku instance or solution encoded by predicates `given/2` and `assign/2`.
Blocks are expected to be already materialized by the following program (where `size/1`
    and `square/1` give the size of the Sudoku):
```asp
block((row, Row), (Row, Col)) :- Row = 1..Size, Col = 1..Size, size(Size).
block((col, Col), (Row, Col)) :- Row = 1..Size, Col = 1..Size, size(Size).
block((sub, Row', Col'), (Row, Col)) :- Row = 1..Size, Col = 1..Size, Row' = (Row-1) / S,
    Col' = (Col-1) / S, size(Size), square(S).
```
```

Figure 11: URL and documentation to register the Sudoku Visualizer operation via the RegisterRecipes operation.

it produces

`__cell__(1,1,"foo").  __cell__(1,2,"bar").`

(and the input text as side output).

For a more interesting example, let us consider the input t from Example 3. If the sequence of ingredients is extended with (i) a SearchModels ingredient with the program in Example 8, (ii) a SearchModels ingredient with the program in Figure 4, (iii) a SearchModels ingredient with the program in Figure 5, and (iv) a Graph ingredient with predicate `__graph__`*, the graphical representation shown in Figure 7 (left) is produced as a side output.*

Further extending the recipe with (v) a SearchModels ingredient with the program in Figure 6, (vi) a SearchModels ingredient with the program in Figure 5, and (vii) a Graph ingredient with predicate `__graph__`*, the graphical representation shown in Figure 7 (right) is also produced as a side output. It can be observed that (iii)–(iv) and (vi)–(vii) are the same sequence of ingredients. They can be packed as a recipe, associated with a short URL (https://asp-chef.alviano.net/s/recipe-ingredients/Sudoku+Visualizer) and replaced by two recipe ingredients pointing to the short URL. The simplified recipe is accessible at https://asp-chef.alviano.net/s/sudoku-example@KR2024.* ■

Recipe ingredients can be registered as new operations to expand the list of operations supported by ASP CHEF. A registered recipe is stored in the local storage of the browser, so to ease the reuse of the same recipe ingredient in multiple recipes. When a registered recipe is added to the main recipe, the list of ingredients of the main recipe is extended with a recipe ingredient pointing to the URL of the registered recipe. This way, the main recipe does not dependent on the local storage of the browser, so that the main recipe can still be shared with any other browser, even with no knowledge about the registered recipes. Moreover, if the registered recipe uses a short URL, in case the recipe is updated (i.e., the URL stored in GitHub is modified), all recipes using the registered ingredient are automatically updated as well.

Example 12. *The visualization of Sudoku instances and their solutions is a good candidate for expanding the list of operations supported by ASP CHEF, at least for those users interested in Sudoku puzzles. Figure 8 shows a Recipe ingredient pointing to the URL storing the Sudoku Visualizer. The Recipe ingredient is assigned the name* `dumbo/Sudoku Visualizer`*, which is also shown in the header of the ingredient. If the user click the REGISTER button, the list of operations is extended with the* `&r/dumbo/Sudoku Visualizer` *operation, as shown in Figure 9.* ■

Finally, ASP CHEF provides the RegisterRecipes operation, with parameters $prefix$ and $predicate$, to register recipes stored in atoms of the form $predicate(Base64)$. Here, $Base64$ is a Base64-encoded text whose first line is the name of the recipe ingredient, the second line is its URL, and any remaining line is used to provide the documentation for the registered recipe. The $prefix$ is prepended to the name of the recipe (to obtain a namespace mechanism to further ease the sharing of recipes).

Example 13. *Let the text shown in Figure 11 be the input of a recipe that Base64-encodes its input and whose sequence of ingredients comprises only a RegisterRecipes ingredient with prefix* `dumbo` *and predicate* `__base64__`*. The new operation* `&r/dumbo/Sudoku Visualizer` *is added to ASP CHEF, and associated with the documentation shown in Figure 10.* ■

5 Related Work

Various tools and frameworks supporting the development of Answer Set Programming (ASP) programs have been presented in the literature, catering to different aspects of ASP-based problem-solving. Notably, Integrated Development Environments (IDEs) such as ASPIDE (Febbraro, Reale, and Ricca 2011), SEALION (Busoniu et al. 2013), and LOIDE (Calimeri et al. 2018) have been designed to facilitate the creation and management of ASP programs. While ASPIDE and SEALION are desktop applications, LOIDE stands out as a web-based IDE. In contrast, ASP CHEF, while not intended to be an IDE, offers a unique browser-based environment for ASP-based problem-solving, leveraging technologies such as CLINGO-WASM to run ASP computations entirely within the browser without the need for a backend server; among the advantages of such a serverless approach there is the possibility to develop proof-of-concept solvers and interactive examples that can be shared within a scientific article—e.g., see the fast prototyping approach proposed in (Costantini and Formisano 2024) to address reduct-based semantics for Epistemic Logic Programs. Addition-

ally, several works have focused on simplifying the development of ASP modules and microservices, aligning with the goal of employing ASP in a non-monolithic manner (Calimeri and Ianni 2006; Costantini and Gasperis 2018; Cabalar, Fandinno, and Lierler 2020; Costantini, Gasperis, and Lauretis 2021; Cabalar et al. 2023). These efforts share common ground with ASP CHEF in promoting modular and flexible ASP-based problem-solving approaches.

Visualization of ASP output has been addressed by various tools and frameworks, including ASPVIZ (Cliffe et al. 2008), IDPD3 (Lapauw, Dasseville, and Denecker 2015), and KARA (Kloimüllner et al. 2011), all of which utilize predicates to describe graphical representations of ASP solutions. Although these tools offer rich visualization capabilities, they typically require additional software installations and may lack browser-based accessibility. More recent endeavors, such as CLINGRAPH (Hahn et al. 2022) and ASPECT (Bertagnon, Gavanelli, and Zanotti 2023), aim to produce high-quality graphical representations of ASP solutions, exportable in LaTeX. The Graph operation introduced in Section 3 aligns with the aforementioned efforts in visualizing ASP output. While the current state of the Graph operation may not match the richness of other visualization tools in terms of features like animation, it offers distinct advantages such as browser-based accessibility and interactivity. Furthermore, as a side output within ASP CHEF recipes, the Graph operation provides flexibility in showcasing intermediate states of computational pipelines, enhancing transparency and understanding in ASP-based problem-solving processes and possibly reducing the number of debug sessions with tools like DWASP-GUI (Dodaro et al. 2019). As a final remark, we observe that ASP CHEF has the possibility to interact with external servers, and some proof-of-concept servers are available in the GitHub repository[9] to interact with CLINGRAPH and ASPECT.

6 Conclusion

ASP CHEF is a versatile tool built upon the principles of ASP. Here we focused on one of its extension mechanism, using the Graph operation as a practical example and use case. The extension mechanism empowers users to register new recipes as custom ingredients, expanding the capabilities of the tool to accommodate a wide range of problem domains and computational workflows. Looking ahead, ASP CHEF has the potential for further development and refinement. Future efforts could focus on expanding the library of available operations and ingredients, enhancing the usability and user experience of the tool, and fostering a vibrant community of users and contributors. For example, we started to collect recipe ingredients to form a library[10] whose operations can be selectively added to local ASP CHEF instantiations.

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 is a member of the Gruppo Nazionale Calcolo Scientifico-Istituto Nazionale di Alta Matematica (GNCS-INdAM).

[9]https://github.com/alviano/asp-chef

[10]https://asp-chef.alviano.net/s/recipe-ingredients

References

Alviano, M.; Dodaro, C.; Fiorentino, S.; Previti, A.; and Ricca, F. 2023. ASP and subset minimality: Enumeration, cautious reasoning and muses. *Artif. Intell.* 320:103931.

Alviano, M.; Ianni, G.; Pacenza, F.; and Zangari, J. 2024. Rethinking answer set programming templates. In Gebser, M., and Sergey, I., eds., *Practical Aspects of Declarative Languages - 26th International Symposium, PADL 2024, London, UK, January 15-16, 2024, Proceedings*, volume 14512 of *Lecture Notes in Computer Science*, 82–99. Springer.

Alviano, M.; Cirimele, D.; and Reiners, L. A. R. 2023. Introducing ASP recipes and ASP chef. In *ICLP Workshops*, volume 3437 of *CEUR Workshop Proceedings*. CEUR-WS.org.

Balduccini, M.; Barborak, M.; and Ferrucci, D. A. 2023. Pushing the limits of clingo's incremental grounding and solving capabilities in practical applications. *Algorithms* 16(3):169.

Bertagnon, A.; Gavanelli, M.; and Zanotti, F. 2023. ASPECT: answer set representation as vector graphics in latex. In *CILC*, volume 3428 of *CEUR Workshop Proceedings*. CEUR-WS.org.

Bertolucci, R.; Capitanelli, A.; Dodaro, C.; Leone, N.; Maratea, M.; Mastrogiovanni, F.; and Vallati, M. 2021. Manipulation of articulated objects using dual-arm robots via answer set programming. *Theory Pract. Log. Program.* 21(3):372–401.

Brewka, G.; Eiter, T.; and Truszczynski, M. 2011. Answer set programming at a glance. *Commun. ACM* 54(12):92–103.

Busoniu, P.; Oetsch, J.; Pührer, J.; Skocovsky, P.; and Tompits, H. 2013. Sealion: An eclipse-based IDE for answer-set programming with advanced debugging support. *Theory Pract. Log. Program.* 13(4-5):657–673.

Cabalar, P.; Fandinno, J.; Schaub, T.; and Wanko, P. 2023. On the semantics of hybrid ASP systems based on clingo. *Algorithms* 16(4):185.

Cabalar, P.; Fandinno, J.; and Lierler, Y. 2020. Modular answer set programming as a formal specification language. *Theory Pract. Log. Program.* 20(5):767–782.

Calimeri, F., and Ianni, G. 2006. Template programs for disjunctive logic programming: An operational semantics. *AI Commun.* 19(3):193–206.

Calimeri, F.; Germano, S.; Palermiti, E.; Reale, K.; and Ricca, F. 2018. Developing ASP programs with ASPIDE and LoIDE. *Künstliche Intell.* 32(2-3):185–186.

Cliffe, O.; Vos, M. D.; Brain, M.; and Padget, J. A. 2008. ASPVIZ: declarative visualisation and animation using answer set programming. In *ICLP*, volume 5366 of *Lecture Notes in Computer Science*, 724–728. Springer.

Costantini, S., and Formisano, A. 2024. Solver fast prototyping for reduct-based ELP semantics. In Angelis, E. D., and Proietti, M., eds., *Proceedings of the 39th Italian Conference on Computational Logic, Rome, Italy, June 26-28, 2024*, volume 3733 of *CEUR Workshop Proceedings*. CEUR-WS.org.

Costantini, S., and Gasperis, G. D. 2018. Dynamic goal decomposition and planning in MAS for highly changing environments. In *CILC*, volume 2214 of *CEUR Workshop Proceedings*, 40–54. CEUR-WS.org.

Costantini, S.; Gasperis, G. D.; and Lauretis, L. D. 2021. An application of declarative languages in distributed architectures: ASP and DALI microservices. *Int. J. Interact. Multim. Artif. Intell.* 6(5):66–78.

Dodaro, C.; Gasteiger, P.; Reale, K.; Ricca, F.; and Schekotihin, K. 2019. Debugging non-ground ASP programs: Technique and graphical tools. *Theory Pract. Log. Program.* 19(2):290–316.

Erdem, E.; Gelfond, M.; and Leone, N. 2016. Applications of answer set programming. *AI Mag.* 37(3):53–68.

Febbraro, O.; Reale, K.; and Ricca, F. 2011. ASPIDE: integrated development environment for answer set programming. In *LPNMR*, volume 6645 of *Lecture Notes in Computer Science*, 317–330. Springer.

Hahn, S.; Sabuncu, O.; Schaub, T.; and Stolzmann, T. 2022. Clingraph: ASP-based visualization. In *LPNMR*, volume 13416 of *Lecture Notes in Computer Science*, 401–414. Springer.

Kaminski, R.; Romero, J.; Schaub, T.; and Wanko, P. 2023. How to build your own asp-based system?! *Theory Pract. Log. Program.* 23(1):299–361.

Kloimüllner, C.; Oetsch, J.; Pührer, J.; and Tompits, H. 2011. Kara: A system for visualising and visual editing of interpretations for answer-set programs. In *INAP/WLP*, volume 7773 of *Lecture Notes in Computer Science*, 325–344. Springer.

Lapauw, R.; Dasseville, I.; and Denecker, M. 2015. Visualising interactive inferences with IDPD3. *CoRR* abs/1511.00928.

Lifschitz, V. 2019. *Answer Set Programming*. Springer.