TPLP: Page 1–20.

doi:10.1017/S1471068425100094

ASP Chef Grows Mustache to Look Better

MARIO ALVIANO and LUIS ANGEL RODRIGUEZ REINERS
University of Calabria, Rende, Italy
(*e-mails:* mario.alviano@unical.it, luis.reiners@unical.it)

WOLFGANG FABER
University of Klagenfurt, Klagenfurt, Austria
(*e-mail:* wolfgang.faber@aau.at)

submitted 11 July 2025; revised 11 July 2025; accepted 27 July 2025

Abstract

We present ASP Chef Mustache, an extension of ASP Chef that enhances template-based rendering of answer set programming (ASP) solutions using a logic-less templating system inspired by Mustache. Our approach integrates data visualization frameworks such as Tabulator, Chart.js, and vis.js, enabling interactive representations of ASP interpretations as tables, charts, and graphs. Mustache queries in templates support advanced constructs for formatting, sorting, and multi-stage expansion, facilitating the generation of rich, structured outputs. We demonstrate the power of this framework through a series of use cases, including data analysis for the Italian VQR, visualization of blocking sets in graphs, and scheduling problems. The result is a versatile tool for bridging declarative problem solving and modern web-based visual analytics.

KEYWORDS: answer set programming, ASP chef, visualization, systems integration

1. Introduction

Answer Set Programming (ASP) is a well-established declarative paradigm for solving complex combinatorial problems, offering a clean separation between problem modeling and computation (Brewka *et al.* 2011; Erdem *et al.* 2016; Lifschitz 2019; Kaminski *et al.* 2023; Alviano *et al.* 2023). While modern ASP solvers can efficiently compute answer sets, interpreting and presenting these solutions in practical applications often requires substantial custom post-processing that motivated the development of several tools for answer set visualization (Cliffe *et al.* 2008; Kloimüllner *et al.* 2011; Lapauw *et al.* 2015; Bourneuf 2018; Hahn *et al.* 2024; Bertagnon and Gavanelli 2024). In the same spirit, ASP Chef (Alviano *et al.* 2023) was introduced to streamline the construction of pipelines that combine ASP-based search and optimization tasks with other tools for filtering, aggregation, and visualization (Alviano and Rodriguez Reiners 2024).

In previous developments, ASP Chef has demonstrated its flexibility in composing pipelines involving combinatorial search and optimization by integrating external tools through *mappings*, that is, sets of procedures that convert ASP facts into representations suitable for systems implemented in other languages. This methodology enabled meaningful collaborations with tools such as MiniZinc (Nethercote *et al.* 2007), for constraint modeling and solving, and Structured Declarative Language (SDL) (Alviano *et al.* 2024), for structured problem descriptions. These integrations (Alviano *et al.* 2024) relied on defining specific interpretations of ASP facts to match the syntax and semantics required by each target system. However, as the expressiveness and complexity of the external language increased, so did the difficulty and cost of building and maintaining the corresponding mappings. Consequently, such efforts often focused on a limited subset of the features available in the target language, hindering full exploitation of the capabilities of the integrated tool.

In this work, we introduce a different and more scalable approach to tool integration, grounded in the use of Mustache templates (a popular web templating system; Mittapalli and Arthur 2021). Rather than building logic-based rules to obtain ASP facts that are later interpreted to interact with third-party tools, users can now define output templates using the native format of the target tool, embedding variable placeholders that are filled directly using the answer sets. This shift in paradigm allows ASP developers to write plain templates in the language of the external tool, with minimal effort and no need for custom interpreters or adapters. As a result, the richness of the target system no longer translates into integration complexity. Instead, users are empowered to access the full feature set of the external tool by simply referring to its original documentation and writing the appropriate Mustache template. This template-based mechanism preserves the declarative spirit of ASP while promoting openness, modularity, and extensibility in the design of logic-based pipelines. Thanks to Mustache, we extend ASP Chef to support the embedding of interactive views using popular JavaScript libraries such as Tabulator (for dynamic tables), Chart.js (for customizable charts), and vis.js (for networks, timelines, and 3D visualizations). This enhancement empowers ASP developers to generate human-readable reports, dashboards, and exploratory interfaces with minimal additional effort, all within a declarative framework. The newly integrated libraries find application in several common and new use cases of ASP, including scheduling and data analysis.

2. Background

2.1 Mustache templating system

Mustache is a logic-less templating system designed for generating HTML, configuration files, and other structured documents. Its simplicity and flexibility make it a popular choice for developers who need to separate data from presentation while avoiding complex scripting within templates. At its core, Mustache templates use *placeholders* enclosed in

```
{{ ... }}
```

to insert dynamic content. In this work, we restrict placeholders to be *variables* that get assigned values during template rendering, or *sections* of the form

```
{{#variable}} ... {{/variable}}
```

https://doi.org/10.1017/S1471068425100094 Published online by Cambridge University Press

that enable loops and conditionals. Data are taken from a YAML or JSON file during template rendering.

Example 1.
Let us consider the following Mustache template:

```
The OS is {{ operating system }}. The list of users is the following:
{{#users}}
  - {{ username }} (id {{ userid }})
{{/users}}
```

Applying the above template to the JSON object

```
{ "operating system": "Linux",
  "users": [ { "userid": 1000, "username": "alice" },
             { "userid": 1001, "username": "bob"   } ] }
```

renders the following text (which can be part of a Markdown document):

```
The OS is Linux. The list of users is the following:
 - alice (id 1000)
 - bob (id 1001)
```

2.2 Answer set programming

A program is a set of rules defining conditions (conjunctive bodies) under which atoms must be derived (atomic heads) or guessed (choices). Programs are associated with zero or more answer sets, that is, interpretations satisfying all rules and a stability condition (Gelfond and Lifschitz 1990). Programs are extended with `#show` directives of the form

`#show` $p(\bar{t})$ `:` *conjunctive_query*.

where p is an optional predicate, $\bar{t}$ is a possibly empty sequence of terms, and *conjunctive_query* is a conjunction. Answer sets of the program are projected according to the `#show` directives. We refer the ASP-Core-2 format for details (Calimeri *et al.* 2020).

Example 2.
The following program solves the K-Clique problem in ASP:

```
r1 :  edge(X,Y) :- edge(Y,X).
r2 :  {in(N) : node(N)} = K :- size(K).
r3 :  :- in(X), in(Y), X < Y, not edge(X,Y).
r4 :  #show (Index+1, N) : in(N), size(K), Index = #count{N': in(N'), N > N'}.
```

Rule r_1 defines edge as a symmetrically closed relation (as the graph is undirected). The choice rule r_2 guesses K nodes (a K-clique candidate). The constraint r_3 checks that all selected nodes are linked (a valid clique). Finally, the `#show` directive r_4 projects the answer sets over the selected nodes, indexing them according to their natural ordering. Given `size(3)`, and the graph `node(a)`, `node(b)`, `node(c)`, `node(d)`, `edge(a,b)`, `edge(a,c)`, `edge(b,c)`, `edge(c,d)`, the program has one (projected) answer set, namely `(1, a)`, `(2, b)`, `(3, c)`.

https://doi.org/10.1017/S1471068425100094 Published online by Cambridge University Press

2.3 ASP chef

An *operation* O is a function receiving in input a sequence of interpretations and producing in output a sequence of interpretations. Operations may produce side outputs (e.g., a graph visualization) and accept parameters to influence their behavior. An *ingredient* is an instantiation of a parameterized operation with side output. A *recipe* is a tuple of the form(*encode*, *Ingredients*, *decode*), where *Ingredients* is a (finite) sequence $O_1\langle P_1\rangle$, $\ldots$, $O_n\langle P_n\rangle$ of ingredients, and *encode* and *decode* are Boolean values. If *encode* is true, the input of the recipe is mapped to [[`__base64__("`s`")`]], where $s = Base64(s_{in})$ (i.e., the Base64–encoding of the input string s_{in}). After that, the ingredients are applied one after another. Finally, if *decode* is true, every occurrence of `__base64__`(s) is replaced with (the ASCII string associated with) $Base64^{-1}(s)$. Among the operations supported by ASP Chef there are $Encode\langle p, s\rangle$ to extend every interpretation in input with the atom $p($`"`t`"`$)$, where $t = Base64(s)$; $Search\ Models\langle \Pi, n\rangle$ to replace every interpretation I in input with up to n answer sets of $\Pi \cup \{p(\bar{t}).\mid p(\bar{t}) \in I\}$; $Optimize\langle \Pi, n\rangle$ to replace every interpretation I in input with up to n optimal answer sets of $\Pi \cup \{p(\bar{t}).\mid p(\bar{t}) \in I\}$.

Example 3 (Continuing Example 2).
The recipe $(\mathbf{F}, [Search\ Models\langle\{r_1, \ldots, r_4\}, 1\rangle], \mathbf{F})$ addresses the K-Clique problem in a single step. With $(\mathbf{F}, [Search\ Models\langle\{r_1\}, 1\rangle, Search\ Models\langle\{r_2, r_3\}, 1\rangle, Search\ Models\langle\{r_4\}, 1\rangle], \mathbf{F})$, instead, the problem is addressed in three steps: (i) symmetric closure of `edge/2`; (ii) clique search; (iii) solution projection.

3. ASP chef mustache

We introduce the main linguistic constructs of our Mustache template system. While the original system mainly deals with variables and sections to enable loops and conditionals, our system relies on ASP terms and queries. At the core of our system there is the expansion of Mustache queries, whose results can be sorted, formatted and projected to handle duplicates (Section 3.1). More advanced constructs enable string manipulation and interpolation, the sharing of common elements among several Mustache queries, and nesting of loops and conditionals (Section 3.2).

3.1 Core functionalities

3.1.1 Mustache queries and expansion

A *Mustache query* has the form {{ Π}}, where Π is an ASP program with `#show` directives. Applying a Mustache query to an interpretation I renders one projected answer set of $\Pi \cup \{p(\bar{t}).\mid p(\bar{t}) \in I\}$ if any, and otherwise raises an error. Specifically, tuples of terms in the projected answer set are rendered, one per line and separating terms with "`, `" (comma followed by a blank space). A *template* is a text with Mustache queries.

Example 4.
Let Π be the program and I be the facts from Example 2. Applying the template

```
Here is a {{ #show (K,) : size(K). }}-clique of the given graph:
{{ Π }}
```

https://doi.org/10.1017/S1471068425100094 Published online by Cambridge University Press

to I renders

```
Here is a 3-clique of the given graph:
1, a
2, b
3, c
```

Note that the rendered text depends on the order in which tuples of terms are processed.

3.1.2 Shortcut form, sorting and basic formatting

Mustache queries are often one-liner, as

```
{{ #show (K,) : size(K). }}
```

in Example 4. Therefore, we introduce the shortcut form

{{= $p(\bar{t})$: *conjunctive_query* }}

equivalent to

{{ #show $p(\bar{t})$: *conjunctive_query*. }}.

Another shortcut is given for tuples of the form $(t,)$ with t being a number or double-quoted string, which can be equivalently written as t. The processing order of tuples of terms can be specified by creating an instance of `sort/1`, where the argument represents the index of the term used for sorting. A positive index indicates ascending order, while a negative index specifies descending order, using the absolute value of the index. Ties are broken by subsequent instances of `sort/1` (if available). The rendering of tuples of terms can be controlled with instances of `separator/1` to specify a different separator for tuples; `term_separator/1` to specify a different separator for terms; `prefix/1` and `suffix/1` to wrap output with specified text. As a side note, we discourage the use of guessing components (as choice rules) in Mustache queries, as they are better handled by other ASP Chef operations (e.g., *Search Models* and *Optimize*).

Example 5 (Revising Example 4).
Let I be the facts from Example 2 extended with `in(a)`, `in(b)`, `in(c)` (i.e., the answer set of the program from Example 2 before projection). Applying the template

```
Here is a {{= K : size(K) }}-clique of the given graph: {{
  #show (Index + 1, N) : in(N), Index = #count{N' : in(N'), N > N'}.
  #show sort(1).
  #show separator("; ").
  #show term_separator(") ").
  #show prefix("(").
}}.
```

to I renders

```
Here is a 3-clique of the given graph: (1) a; (2) b; (3) c.
```

Note that the order of tuples is specified in the Mustache query, and therefore the rendered text does not depend on the order in which tuples are produced by the underlying ASP solver.

https://doi.org/10.1017/S1471068425100094 Published online by Cambridge University Press

3.1.3 Handling duplicates

ASP follows set-semantics, meaning duplicate results do not appear naturally. On the other hand, ASP can deal with duplicate addends in sums and weak constraints thanks to an extended syntax including *distinguishing terms* (i.e., terms that are used to differentiate between equal addends). The same technique can be adopted for tuples of terms in Mustache queries: the varadic predicate `show/*` renders the first of its arguments (a tuple of terms), discarding all other arguments. All of its arguments are subject to the sorting criteria specified in the Mustache query (if any), hence enabling the possibility to sort elements based on properties that are not rendered.

Example 6.
Suppose a cost is associated with each node in Example 2: `cost(a,1)`, `cost(b,2)`, `cost(c,1)`, `cost(d,1)`. The following weak constraint minimizes the sum of costs in the computed clique:

```
:∼ cost(Node,Cost), in(Node). [Cost@1, Node]
```

Note that variable `Node` is a distinguishing term, thanks to which the cost 1 associated with nodes `a` and `c` can be correctly counted twice in the computed solution. Applying the template

```
The cost is {{= S : S = #sum{Cost, Node : in(Node), cost(Node,Cost)} }} = {{
  #show show(Cost, Node) : in(Node), cost(Node,Cost).
  #show sort(2).
  #show separator(" + ").
}}.
```

to the clique comprising nodes `a`, `b` and `c` renders

```
The cost is 4 = 1 + 2 + 1.
```

Note that variable `Node` is a distinguish term also in the `#sum` aggregate and in the second Mustache query. Also note that costs are sorted by node.

3.2 Advanced functionalities

3.2.1 Lua string @-terms

Mustache queries have access to some interpreted functions that ease string manipulation. Among them, `@string_join(sep, ...)` to concatenate two or more strings using the provided separator, `@string_concat(...)` as a shortcut for `@string_join("", ...)`, and `@string_format(format, ...)` to format a string using the given format string and arguments. Floating-point numbers are represented in the format `real("NUMBER")`.

Example 7 (Revising Example 5).
The output shown in Example 5 can also be obtained by the following template:

```
Here is a {{= K : size(K) }}-clique of the given graph: {{
  #show show(shown_term, Index) :  in(N), Index = #count{N' : in(N'), N > N'}.
  #show sort(2).
  #show separator("; ").
}}.
```

https://doi.org/10.1017/S1471068425100094 Published online by Cambridge University Press

where *shown_term* is either `@string_concat("(", Index + 1, ")", N)` or`@string_format("(%d) %s", Index + 1, N)`.

3.2.2 Multiline strings and F-strings

Mustache queries enrich the syntax of ASP with *multiline strings* of the form {{"..."}} and *f-strings* of the form {{f"..."}}. A multiline string is mapped to a double-quoted string, representing new lines and double-quotes with the escape sequences `\n` and `\"`, respectively. F-strings additionally introduce interpolation: data are interpolated in f-strings using the syntax ${ *expression*:*format* }, where :*format* is optional (default `:%s` for string). A f-string {{f"*str*"}} is mapped to the term `@string_format(`*fmt*`,` $e_1, \ldots, e_n$`)`, where *fmt* is the double-quoted string obtained by escaping *str* and replacing interpolations with the associated formats, and $e_1, \ldots, e_n$ are the $n \geq 0$ expressions interpolated in *str*.

Example 8.
Suppose we would like to render

```
nodes: [ { id: "a", label: "a (1)", group: "in" },
         { id: "b", label: "b (2)", group: "in" },
         { id: "c", label: "c (1)", group: "in" },
         { id: "d", label: "d (1)", group: "out" } ],
```

as part of a JSON object representing the graph and the computed clique. We could rely on

```
nodes: [{{
    #show show(@string_format("{id:\"%s\", label:\"%s (%d)\", group:\"out\"}",
          X, X, C), X) : cost(X,C), not in(X).
    #show show(@string_format("{id:\"%s\", label:\"%s (%d)\", group:\"in\"}",
          X, X, C), X) : cost(X,C), in(X).
    #show sort(2).
    #show separator(",\n").
}}],
```

We observe that double-quoted strings of ASP require escaping of frequent characters in JSON (the double quote char in particular). Even worse, handling a large JSON object formatted as a single line is highly inconvenient. Below are more convenient representations:

```
nodes: [{{  #show show(@string_format({{"{
              id: "%s", label: "%s (%d)", group: "out"
            }"}}, X, X, C), X) : cost(X,C), not in(X).
            #show show({{f"{
              id: "${X}", label: "${X} (${C:%d})", group: "in"
            }"}}, X) : cost(X,C), in(X).
            #show sort(2).
            #show separator(",\n").
        }}],
```

In the template above, the f-string in the second `#show` directive essentially maps to the `@string_format` term in the first `#show` directive (modulo the value of `group`).

https://doi.org/10.1017/S1471068425100094 Published online by Cambridge University Press

3.2.3 Persistent queries

Common elements of different Mustache queries in a template can be stored in a *persistent array*, initially empty, by using the expression

`{{* Π }}`

where Π is an ASP program with `#show` directives; or using the shortcut form

`{{+` $p(\bar{t})$ `:` *conjunctive_query* `}}`

equivalent to

`{{* #show` $p(\bar{t})$ `:` *conjunctive_query*`. }}.`

The content of the persistent array is prepended to the tuples of terms and atoms obtained by evaluating subsequent Mustache queries in the template. The persistent array can be reset using the Mustache expression {{-}}.

Example 9 (Continuing Example 8).
Targeting the rendering of JSON objects, the separator is likely always `", \n"`. Instead of including the associated `#show` directive in all Mustache queries, a template could start with the Mustache expression {{+ `separator(", \n")`}}.

3.2.4 Multi-stage expansion

Mustache queries enable loops and conditionals. In some cases, nesting of Mustache queries is convenient for evaluating conditionals within loop elements or for executing inner loops. A multi-stage template expansion repeatedly processes Mustache queries until no further expansions remain.

Example 10 (Continuing Examples 8–9).
As already observed, the multiline string and the f-string in Example 8 only differ in the value of `group`. Nesting a conditional within a single loop over nodes seems natural in this case. Assuming `sort(2)` and `separator(", \n")` are in the persistent array, we can use

```
nodes: [ {{= show({{f"
    {
      id: "${X}",
      label: "${X} (${C:%d})",
      group: {{= "out" : not in(${X}) }}{{= "in" : in(${X}) }}
    }
  "}}, X) : cost(X,C)
}} ],
```

Note that the first expansion renders (four inner objects like) the following:

```
nodes: [ { id: "a", label: "a (1)",
           group: {{= "out" : not in(a) }}{{= "in" : in(a) }} }, ··· ],
```

Conditionals within each object are evaluated in the second stage to obtain the `group` values.

https://doi.org/10.1017/S1471068425100094 Published online by Cambridge University Press

Fig. 1. Side outputs associated with data from Example 6: graph with highlighted clique obtained with the *@vis.js/Network* operation; table showing node costs and computed clique obtained with the *tabulator* operation; chart reporting statistics obtained with the *Chart.js* operation. A recipe showcasing these examples is available at https://asp-chef.alviano.net/s/ICLP2025/running-example.

4. JSON-based frameworks integration

Several frameworks can be configured using JSON objects. To further ease their integration in ASP Chef, we rely on *Relaxed JSON* (https://www.relaxedjson.org/), hence accepting a more permissive syntax. The new operations presented in this section have parameters $\langle p, m\rangle$, where p is a predicate and m is a Boolean. Each atom $p(s)$ in each input interpretation I produces a side output according to the JSON object rendered by applying the template $Base64^{-1}(s)$ to I, using multi-stage expansion if m is **T**.

4.1 @vis.js/network

The Network module in vis.js (https://visjs.org/) is a powerful JavaScript library for visualizing dynamic and interactive networks (graphs). Users can customize colors, shapes, labels, and border styles of nodes and edges. The module includes a physics-based layout engine, and supports real-time interactivity. Additionally, grouping and clustering allow for efficient visualization of large datasets by aggregating related nodes. The library also supports hierarchical layouts and directional edges with arrows. We extended ASP Chef with the *@vis.js/Network* operation.

Example 11 (Display data from Example 6 as a graph).
The following template renders the graph shown in Figure 1:

```
{ data: {
    nodes: [ {{= show({{f"
        {
          id: "${X}",
          label: "${X} (${C:%d})",
          group: {{= "out" : not in(${X}) }}{{= "in" : in(${X}) }}
        }
      "}}, X) : cost(X,C)
    }} ],
    edges: [
      {{= {{f"{ from: "${X}", to: "${Y}" }"}} : edge(X,Y), X<Y, in(Y) }}
      {{= {{f"{ from: "${Y}", to: "${X}" }"}} : edge(X,Y), X<Y, not in(Y) }}
      ]
    },
```

https://doi.org/10.1017/S1471068425100094 Published online by Cambridge University Press

```
    options: {
      nodes: { shape: "circle", borderWidth: 2, shadow: true },
      edges: { width: 2, shadow: true },
      groups: {
        "in": { color: { background: "yellow", border: "orange"} } }
      }
}
```

Note that nodes in the `"in"` group (and edges within the clique) are highlighted in yellow.

4.2 Tabulator

Tabulator (https://tabulator.info/) is a powerful JavaScript library that provides a wide range of customization options to create interactive tables from various sources, including JSON. In the context of ASP Chef, Tabulator can be used to render tables from input models, taking advantage of Mustache queries to configure the table and bind data from ASP facts as needed. Specifically, we extended ASP Chef with the *Tabulator* operation, and enriched the JSON configuration to accept the `download` property, allowing users to add buttons for exporting the data in various formats such as CSV, JSON, or Excel.

Example 12 (Display data from Example 6 as a table).
The following template renders the table shown in Figure 1:

```
{ data: [{{= {{f"
      {
        node: "${Node}",
        cost: ${Cost * 100 / Max},
        in: {{= "true": in(${Node}) }}{{= "false": not in(${Node}) }},
      },
    "}} : cost(Node, Cost), Max = #max{Cost' : cost(_, Cost')}
  }}],
  columns: [ { title: "Node", field: "node" },
             { title: "Cost", field: "cost", formatter: "progress" },
             { title: "In {{= K : size(K) }}-clique", field: "in",
               hozAlign: "center", formatter: "tickCross" } ],
  initialSort: [ { column: "node", dir: "asc" }, { column: "in", dir: "desc" } ],
  download: [ { color: "success", format: "csv", options: { delimiter: "\t" } } ]
}
```

Note that `data` is a list of objects whose properties are obtained interpolating ASP terms. Costs are shown as a (progress) bar, and membership in the computed clique using ticks. Rows are sorted by membership in the clique (descending), and node name (ascending). Finally, the user can export data as comma-separated values (CSV) by clicking a button.

4.3 Chart.js

Chart.js (https://www.chartjs.org/) is a lightweight yet powerful JavaScript library for creating interactive and customizable charts. It supports multiple chart types, including

https://doi.org/10.1017/S1471068425100094 Published online by Cambridge University Press

line, bar, pie, radar, and scatter plots, making it versatile for various data visualization needs. Its JSON configuration can easily customize styles, adjusting colors, fonts, and tooltips. We extended ASP Chef with the *Chart.js* operation.

Example 13 (Display statistics about Example 6).
The following template renders the mixed chart (bar plots and line) shown in Figure 1:

```
{{+ sort(2) }}
{ type: "bar",
  data: {
    labels: [{{= {{f"${Node}"}} : node(Node) }}],
    datasets: [{
      label: "Total Edges", borderWidth: 2,
      data: [{{= show(V,N) : node(N), V = #count{N' : edge(N,N')} }}],
    }, {
      label: "Clique Edges", borderWidth: 2,
      data: [{{= show(V,N) : node(N), V = #count{N' : edge(N,N'), in(N')} }}],
    }, {
      type: "line", label: "Cost",
      data: [{{= show(Value, Node) : cost(Node, Value) }}],
    }] },
  options: {
    scales: {
      y: { beginAtZero: true,  title: { display: true, text: "Number" } } }
    }
}
```

The template defines three datasets, two bars and one line, using the `show/`$*$ predicate to handle duplicate values. Also note that values are sorted by node (the second argument in `show/`$*$).

Other frameworks We extended ASP Chef with other frameworks, providing alternatives to build charts and images. The *@vis.js/Timeline* operation is specialized for temporal data. The library supports custom styling and grouping of events, making it ideal for project management, historical data visualization, and scheduling applications. The *@vis.js/Graph3D* operation can create interactive 3D visualizations of data, making it ideal for representing mathematical functions, scientific data, and geographical information. Data points are rendered on a 3D plane, either as surface plots or scatter plots. The module provides intuitive controls for zooming, rotating, and panning, allowing users to explore complex datasets from different angles. Customization options include color gradients, axis scaling, and grid styling, enabling precise data representation. The *ApexCharts* (https://apexcharts.com/) operation integrates a modern, highly customizable JavaScript charting library designed for creating interactive, responsive, and performant visualizations. It supports a wide range of chart types, including line, bar, area, pie, radar, heatmaps, and mixed charts, making it suitable for business intelligence dashboards, financial data analysis, and real-time monitoring. Interactivity is a key strength, with built-in support for tooltips, zooming and panning. The *Fabric.js* (https://fabricjs.com/) operation integrates a powerful and flexible JavaScript library for working with HTML5 canvas, enabling rich interactive graphics, image manipulation, and object-based drawing. It simplifies complex vector graphics operations by providing

https://doi.org/10.1017/S1471068425100094 Published online by Cambridge University Press

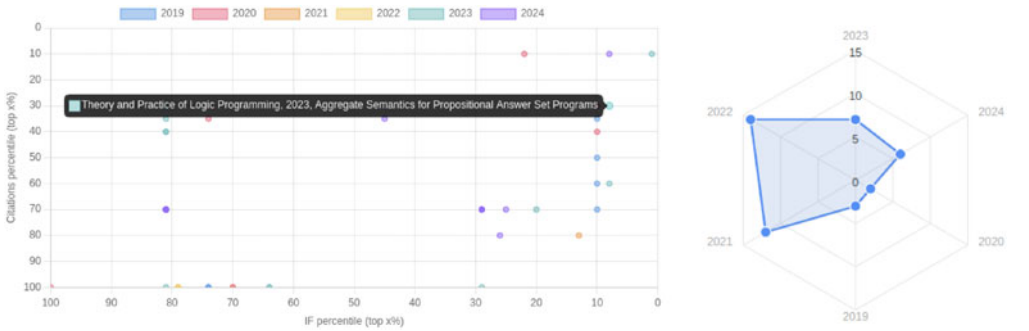

Fig. 2. VQR data analysis with *Chart.js* and *ApexChart*.

an intuitive API for creating and managing shapes, images, text, and paths. One of its standout features is object-based manipulation, allowing users to move, scale, rotate, and group elements directly on the canvas.

5 Use cases

5.1 Data analysis

Italian VQR (Valutazione della Qualità della Ricerca) is the national framework used to assess the quality and impact of research outputs through bibliometric indicators such as citation percentiles and journal impact factors. ASP finds a natural application in this context to optimize the selection of articles avoiding conflicts (see https://asp-chef.alviano.net/s/VQR2024/assegna-paper). The newly integrated libraries add several useful capabilities to analyze input data and results. Figure 2 shows a scatter plot obtained with *Chart.js*, where each point combines the percentiles on citations and impact factor to provide an insightful overview of research performance, and a radar chart from *ApexChart* to overview the number of articles per year included in the analysis. An interactive recipe starting from CSV and including a *Tabulator* representation is available at https://asp-chef.alviano.net/s/ICLP2025/vqr.

5.2 Graph analysis

The concept of *blocking sets* is also known as vertex cuts or vertex separators in graph theory: given a start and an end vertex, a blocking set is a set of vertices (without the start and end vertices) such that the end vertex is not reachable from the start vertex without passing a member of the blocking set. The identification of blocking sets has numerous applications, for example in VLSI design and cybersecurity. Using *@vis.js/Network*, ASP Chef can display blocking sets of a given directed graph, using the same layout for all computed solution to ease their understanding. A recipe is available at https://asp-chef.alviano.net/s/ICLP2025/blocking-sets. For a larger example, https://asp-chef.alviano.net/s/ICLP2025/vqr2 shows an authorship graph from which different communities of authors are easily spotted and independently optimized to select articles for the Italian VQR. Figure 3 reports networks obtained with *@vis.js/Network*

https://doi.org/10.1017/S1471068425100094 Published online by Cambridge University Press

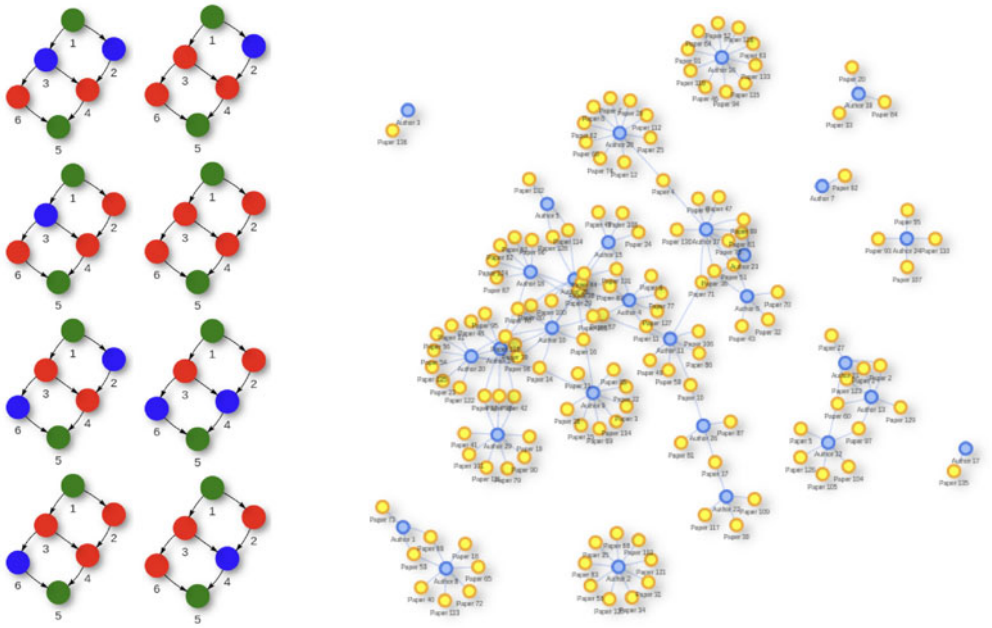

Fig. 3. Blocking sets and authorship graphs obtained with *@vis.js/Network*.

in these contextes. For blocking sets, the start and end vertices are green, vertices in the blocking sets are red.

5.3 Planning and scheduling

Incremental Scheduling is a complex problem that involves allocating jobs to specific devices with replicas, while considering deadlines, dependencies, and importance levels. The goal is to minimize the penalty for missing deadlines and the overall completion time. To visualize this intricate process, we utilize *@vis.js/Timeline* to create a dynamic and interactive schedule. The timeline ingredient showcases the computed schedule, highlighting in red the penalties incurred for missing deadlines. This visualization provides a clear and concise overview of the scheduling process, allowing users to easily identify areas of improvement and optimize their scheduling strategy. Additionally, a *@vis.js/Network* visualization is used to illustrate the dependencies between jobs, providing a comprehensive understanding of the scheduling problem. Figure 4 reports the two visualizations obtained with the recipe at https://asp-chef.alviano.net/s/ICLP2025/incremental-scheduling.

5.4 2D and 3D solution visualization

In the Skyscrapers puzzle the goal is to determine the height of skyscrapers in a grid, given clues on the number of visible skyscrapers in some directions (https://www.puzzle-skyscrapers.com/). A recipe addressing the puzzle is available at

https://doi.org/10.1017/S1471068425100094 Published online by Cambridge University Press

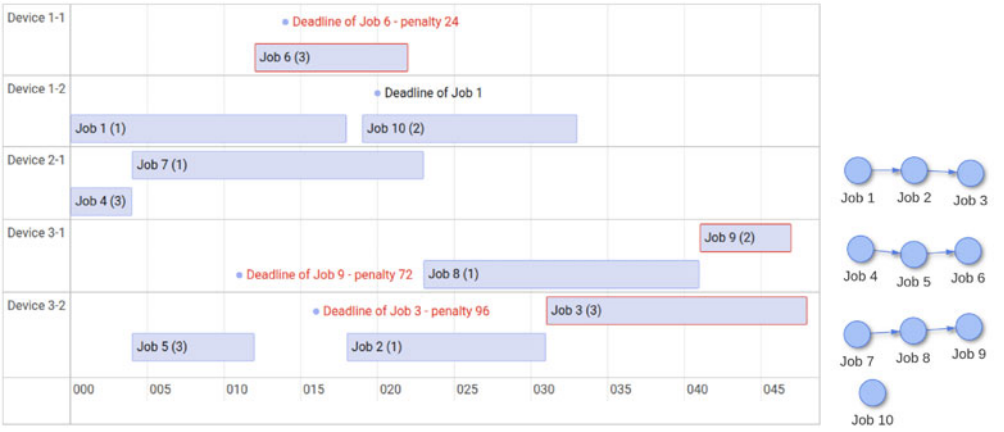

Fig. 4. Computed scheduling and job dependencies shown with *@vis.js/Timeline* and *@vis.js/Network*.

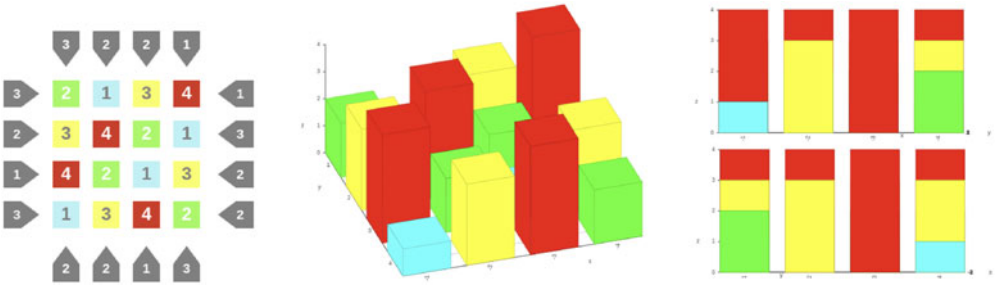

Fig. 5. Skyscrapers solution shown using *Fabric.js* and *@vis.js/Graph3d* (from different angles).

https://asp-chef.alviano.net/s/ICLP2025/skyscrapers, and produces the visualizations shown in Figure 5. The 3D visualization is particularly interesting here, as the user can use the mouse to control the camera angle to obtain an immersive experience.

6. Extending ASP chef with new operations

A key strength of ASP Chef lies in its deliberately simple and modular architecture, which enables users to extend the system by adding new operations with minimal effort. Rather than modifying core infrastructure or registering components manually, operations are implemented as self-contained (and self-registering) `.svelte` files that become instantly usable within the system. This design encourages experimentation, facilitates integration of third-party libraries, and lowers the barrier for community contributions. To implement a new Mustache-based visualization operation (such as a charting or table component) developers typically create two files within `src/lib/operations`: (i) a file named `<Operation>.svelte` that defines UI controls for parameters of the operation (e.g., `predicate` and `multi-stage`), and (ii) a companion file named `+<Operation>.svelte`

https://doi.org/10.1017/S1471068425100094 Published online by Cambridge University Press

that imports and renders the third-party library, feeding it the result of the Mustache template as configuration.

For example, for the *ApexCharts* operation, the main `ApexCharts.svelte` file contains

```
<Operation {id} {operation} {options}>
  <Input type="text" bind:value={options.predicate} on:input={edit} />
  <Button outline="{!options.multistage}" on:click={() => {
      options.multistage = !options.multistage; edit(); }}>Multi-Stage</Button>
  {#each models.flatMap(model =>
              model.filter(atom => atom.predicate === options.predicate)) as cfg}
    <ApexCharts part={model} cfg_atom={cfg} multistage={options.multistage} />
  {/each}
</Operation>
```

This snippet instantiates the operation with UI controls to select the ASP predicate that encodes the chart configuration and to toggle options such as multi-stage. The `#each` loop adds instances of `+ApexCharts.svelte` for each instance of the selected predicate. The companion file `+ApexCharts.svelte` integrates the ApexCharts library and renders the chart:

```
<script>
    import ApexCharts from 'apexcharts'; import {Utils} from "$lib/utils";
    import {Base64} from "js-base64";    import {onMount} from "svelte";

    export let part, cfg_atom, multistage;
    let chart;
    onMount(async () => {
        const content = Base64.decode(cfg_atom.terms[0].string);
        const expanded_content = await Utils.expand_mustache_queries(
                                                part, content, multistage);
        const configuration = Utils.parse_relaxed_json(expanded_content);
        await (new ApexCharts(chart, configuration)).render();
    });
</script>

<div class="chart" bind:this={chart}></div>
```

This fragment decodes the Mustache template in the configuration atom, expands it using the current interpretation (`part`) as context, parses it as JSON, and invokes the rendering engine.

Despite their functionality, these files are concise: (i) `ApexCharts.svelte` is 73 lines; (ii) `+ApexCharts.svelte` is 44 lines. Similar patterns are followed for other operations: *Chart.js* (73 + 47 lines) replicates the same logic with a different charting library; *Tabulator* (74 + 70 lines) extends slightly to add features like export buttons.

7. Related work

Several tools and frameworks supporting the development of ASP have been introduced in the literature, among them Integrated Development Environments (IDEs) such as ASPIDE (Febbraro *et al*. 2011), SeaLion (Busoniu *et al.* 2013), and LoIDE (Calimeri *et al.* 2018). Unlike these tools, ASP Chef is designed as a platform for experimenting with ASP solutions and their integration with external tools. ASP Chef is powered

https://doi.org/10.1017/S1471068425100094 Published online by Cambridge University Press

by CLINGO-WASM and enables the development of shareable examples and interactive demos, a feature especially suited to scientific communication, as shown in Costantini and Formisano (2024). Several research efforts have explored the modular use of ASP through microservices or pipeline-based compositions (Calimeri and Ianni 2006; Costantini and Gasperis 2018; Cabalar *et al.* 2020; Costantini *et al.* 2021; Cabalar *et al.* 2023). These works resonate with the guiding principles of ASP Chef, which promotes the decomposition of ASP workflows into manageable and reusable components. Prior versions of ASP Chef supported integration with external systems, such as MiniZinc and Structured Declarative Language (SDL), by exchanging ASP facts. However, this fact-based approach often incurred non-trivial implementation costs to interpret ASP data in the target language, especially when the target system had a rich and complex syntax. To address this limitation, the present work introduces a more direct and extensible mechanism for integration, based on Mustache templates. Instead of interpreting facts, the user defines templates that interpolate ASP data into the desired target format or language, facilitating smoother and more expressive interactions with external libraries and systems. This enables seamless integration of advanced visualization and reporting tools, without needing dedicated wrappers or converters.

A first embryonic version of templated queries was introduced in our earlier work *ASP Chef Chats with Large Language Models* (Alviano *et al.* 2025), where a restricted form of Mustache query was proposed to synthesize structured Markdown snippets from ASP results and feed them into LLM prompts. In that version, the output was a projection of matching atoms, formatted via special atoms like `ol/1`, `ul/1`, `th` and `tr`. The goal was to enable Markdown generation suitable for LLM consumption, with Base64-encoded answers retrieved from predicates such as `__base64__/1`. In contrast, the current work generalizes this idea into a full-featured templating system, decoupled from LLM usage and capable of targeting arbitrary external libraries. By adopting standard Mustache templates and extending them with ASP-aware data injection, we shift from producing static Markdown to dynamically generating JSON configurations, HTML fragments, and other consumable formats suitable for integration with a wide range of front-end and visualization frameworks.

Visual representation of ASP results has been studied through tools such as ASPVIZ (Cliffe *et al.* 2008), IDPD3 (Lapauw *et al.* 2015), and KARA (Kloimüllner *et al.* 2011), all of which rely on ASP facts to describe the visual layout of answer sets. In the same spirit, more recent approaches like CLINGRAPH (Hahn *et al.* 2022) and ASPECT (Bertagnon *et al.* 2023) offer exportable, high-quality visualizations suited for publication in LaTeX. While these systems provide expressive rendering capabilities, they generally depend on external installations and do not focus on browser-native interaction. Moreover, their development and maintenance must face the complexity of the underlying rendering engines. The visualization operations introduced in this work, based on Tabulator, Chart.js, and vis.js, complement these efforts by enabling fully browser-based, interactive visualizations such as scatter plots, timelines, and network diagrams. Building on our experience with the previously developed *Graph* operation (Alviano and Rodriguez Reiners 2024), we found that using Mustache templates to implement the *@vis.js/Network* and *Fabric.js* operations not only significantly reduced development time but also led to more powerful and expressive functionalities.

https://doi.org/10.1017/S1471068425100094 Published online by Cambridge University Press

A particularly relevant application of ASP-based visualization in complex domains is presented by Gebser *et al.* (2018), who propose an ASP-based framework for experimenting with robotic intra-logistics domains. Their work highlights the value of integrating high-level reasoning with sophisticated visualization and execution environments, and shows how ASP can serve as a flexible backbone for modeling, simulation, and visualization in complex logistical settings. ASP Chef shares this ambition but places stronger emphasis on interactive, browser-native visualizations that support rapid prototyping and deployment via web technologies. Finally, we mention clinguin (Beiser *et al.* 2025) for developing graphical user interfaces directly in ASP. Future applications of Mustache in ASP Chef may explore convergence points between these systems, particularly in the direction of form-based input via frameworks like SurveyJS.

8. Conclusion

We presented an extension of ASP Chef that uses Mustache templates and popular JavaScript libraries to support the creation of interactive, data-driven views from ASP results. By simplifying the integration of external visualization tools, our approach enables developers to build rich, user-friendly interfaces while staying within the declarative paradigm.

Beyond its functionality, a key strength of ASP Chef is its simple and modular design. New Mustache-based operations can be added with minimal effort by creating a `.svelte` file that defines UI parameters and links to a third-party library using the configuration generated from the expanded template. These operations are immediately usable in recipes, without the need for manual registration or system modification. This plugin-like architecture lowers the barrier to contribution and invites experimentation. In future iterations, we plan to provide formal guidelines to support community-driven extensions and integrations.

Future work will explore the use of Mustache templates to integrate frameworks for user interface definition, particularly form-based interactions. This would allow ASP developers not only to present data, but also to collect and structure user input within a seamless and declarative pipeline. In addition, we envision integrating Controlled Natural Language (CNL) processing capabilities to allow domain experts to write specifications in a restricted natural language that can be automatically translated into ASP code (Caruso *et al.* 2024). The combination of Mustache templating, structured input collection, and CNL2ASP translation would broaden the accessibility of ASP Chef, making it easier for non-programmers to contribute to the knowledge engineering process.

Acknowledgments

This work was supported by the 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 the Italian Ministry of Health (MSAL) under POS projectsCAL.HUB.RIA (CUP H53C22000800006) and RADIOAMICA (CUP H53C22000650006); by the Italian Ministry of Enterprises and Made in Italy under project STROKE 5.0 (CUPB29J23000430005); under PN RIC project ASVIN "Assistente Virtuale Intelligente di Negozio" (CUP B29J24000200005); and by the LAIA lab (part of the SILA labs). This research was also funded in part by the Austrian Science Fund (FWF) within projects 10.55776/COE12 and 10.55776/PIN8782623. Mario Alviano is member of Gruppo Nazionale Calcolo Scientifico-Istituto Nazionale di Alta Matematica (GNCS-INdAM).

References

Alviano, M., Cirimele, D. and Rodriguez Reiners, L. A. 2023. Introducing ASP recipes and ASP chef. In *ICLP Workshops*. CEUR Workshop Proceedings, Vol. 3437. https://ceur-ws.org/Vol-3437/.

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

Alviano, M., Dodaro, C. and Vasile, I. R. 2024. Structured declarative language. In *Proceedings of the 39th Italian Conference on Computational Logic, Rome, Italy, June 26–28, 2024*, E. D. Angelis and M. Proietti, Eds. CEUR Workshop Proceedings, CEUR-WS.org, Rome, Italy, Vol. 3733. https://ceur-ws.org/Vol-3733/.

Alviano, M., Guarasci, P., Reiners, L. A. R. and Vasile, I. R. 2024. Integrating structured declarative language (SDL) into ASP chef. In *Logic Programming and Nonmonotonic Reasoning - 17th International Conference, LPNMR 2024, Dallas, TX, USA, October 11–14, 2024, Proceedings*, C. Dodaro, G. Gupta and M. V. Martinez, Eds. Lecture Notes in Computer Science, Springer, Dallas, TX, USA, Vol. 15245, 387–392.

Alviano, M., Macrì, P. and Rodriguez Reiners, L. A. 2025. ASP chef chats with large language models. In *Proceedings of the Thirty-Fourth International Joint Conference on Artificial Intelligence, IJCAI 2025*, ijcai.org, Montreal, Canada. August 16–22, 2025. https://www.ijcai.org/all_proceedings.

Alviano, M. and Rodriguez Reiners, L. A. 2024. ASP Chef: Draw and expand. In *Proceedings of the 21st International Conference on Principles of Knowledge Representation and Reasoning, KR 2024*, P. Marquis, M. Ortiz and M. Pagnucco, Eds. Hanoi, Vietnam, November 2-8, 2024, 720–730. https://proceedings.kr.org/2024/68/.

Beiser, A., Hahn, S. and Schaub, T. 2025. Asp-driven user-interaction with clinguin. In *Proceedings 40th International Conference on Logic Programming*, ICLP 2024, University of Texas at Dallas, Dallas Texas, USA, October 14–17, 2024, P. Cabalar, F. Fabiano, M. Gebser, G. Gupta, and T. Swift, Eds. EPTCS, Vol. 416, 215–228. https://arxiv.org/abs/1511.00928.

Bertagnon, A. and Gavanelli, M. 2024. ASPECT: Answer set representation as vector graphics in latex. *Journal of Logic and Computation* 34, 8, 1580–1607.

Bertagnon, A., Gavanelli, M. and Zanotti, F. 2023. ASPECT: answer set representation as vector graphics in latex. In *CILC*. CEUR Workshop Proceedings, Vol. 3428. https://ceur-ws.org/Vol-3428/.

Bourneuf, L. 2018. An answer set programming environment for high-level specification and visualization of FCA. In FCA4AI 2018, *Stockholm, Sweden, July 13, 2018,* S. O. Kuznetsov, A. Napoli and S. Rudolph, Eds. CEUR Workshop Proceedings, CEUR-WS.org, Stockholm, Sweden, Vol. 2149, 9–20.

Brewka, G., Eiter, T. and Truszczynski, M. 2011. Answer set programming at a glance. *Communications of the 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 and Practice of Logic Programming* 13, 657–673.

Cabalar, P., Fandinno, J. and Lierler, Y. 2020. Modular answer set programming as a formal specification language. *Theory and Practice of Logic Programming* 20, 5, 767–782.

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

Calimeri, F., Faber, W., Gebser, M., Ianni, G., Kaminski, R., Krennwallner, T., Leone, N., Maratea, M., Ricca, F. and Schaub, T. 2020. ASP-Core-2 input language format. *Theory and Practice of Logic Programming* 20, 2, 294–309.

Calimeri, F., Germano, S., Palermiti, E., Reale, K. and Ricca, F. 2018. Developing ASP programs with ASPIDE and LoIDE. *KI - Künstliche Intelligenz* 32, 185–186.

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

Caruso, S., Dodaro, C., Maratea, M., Mochi, M. and Riccio, F. 2024. CNL2ASP: Converting controlled natural language sentences into ASP. *Theory and Practice of Logic Programming* 24, 2, 196–226.

Cliffe, O., Vos, M. D., Brain, M. and Padget, J. A. 2008. ASPVIZ: Declarative visualisation and animation using answer set programming. In *ICLP*, Springer, Vol. 5366, 724–728

Costantini, S. and Formisano, A. 2024. Solver fast prototyping for reduct-based ELP semantics. In *Proceedings of the 39th Italian Conference on Computational Logic, Rome, Italy, June 26–28, 2024*, E. D. Angelis and M. Proietti, Eds. CEUR Workshop Proceedings, CEUR-WS.org, Rome, Italy, Vol. 3733. https://ceur-ws.org/Vol-3428/.

Costantini, S. and Gasperis, G. D. 2018. Dynamic goal decomposition and planning in MAS for highly changing environments. In CILC. CEUR Workshop Proceedings, CEUR-WS.org, Vol. 2214, 40–54. https://ceur-ws.org/Vol-2214/.

Costantini, S., Gasperis, G. D. and Lauretis, L. D. 2021. An application of declarative languages in distributed architectures: ASP and DALI microservices. *International Journal of Interactive Multimedia and Artificial Intelligence* 6, 5, 66–78.

Erdem, E., Gelfond, M. and Leone, N. 2016. Applications of answer set programming. *AI Magazine* 37, 3, 53–68.

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

Gebser, M., Obermeier, P., Otto, T., Schaub, T., Sabuncu, O., Nguyen, V. and Son, T. C. 2018. Experimenting with robotic intra-logistics domains. *Theory and Practice of Logic Programming* 18, 502–519.

Gelfond, M. and Lifschitz, V. 1990. Logic Programming. In *Proceedings of the Seventh International Conference*, Jerusalem, Israel, June 18-20, 1990. MIT Press 1990, ISBN 0-262-73090-1.

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

Hahn, S., Sabuncu, O., Schaub, T. and Stolzmann, T. 2024. *Clingraph* : A system for ASP-based visualization. *Theory and Practice of Logic Programming* 24, 3, 533–559.

Kaminski, R., Romero, J., Schaub, T. and Wanko, P. 2023. How to build your own ASP-based system?!. *Theory and Practice of Logic Programming* 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*. Lecture Notes in Computer Science, Vol. 7773, Springer, 325–344.

https://doi.org/10.1017/S1471068425100094 Published online by Cambridge University Press

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

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

Mittapalli, J. S. and Arthur, M. P. 2021. Survey on template engines in java. In *ITM Web of Conferences*, EDP Sciences, Vol. 37, 01007.

Nethercote, N., Stuckey, P. J., Becket, R., Brand, S., Duck, G. J. and Tack, G. 2007. MiniZinc: Towards a standard CP modelling language. In *CP 2007, Providence, RI, USA, September 23-27, 2007, Proceedings*, C. Bessiere, Ed. LNCS, Springer, Providence, RI, USA, Vol. 4741, 529–543.

https://doi.org/10.1017/S1471068425100094 Published online by Cambridge University Press