ASP Chef Creates Surveys

Mario Alviano, Davide Cirimele and Luis Angel Rodriguez Reiners

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

Abstract

In this paper, we present the integration of SurveyJS, an open-source JavaScript library for building dynamic and interactive forms, into ASP Chef, a web-based framework for designing and executing Answer Set Programming (ASP) pipelines known as recipes. This integration enables users to define customizable, data-driven forms using the JSON-based configuration format of SurveyJS. User input is captured as structured JSON objects, which are then seamlessly incorporated into the ASP pipeline for further reasoning, transformation, or validation. By bridging user interaction with declarative programming, this enhancement significantly expands the applicability of ASP Chef in domains requiring real-time data acquisition and interactive workflows. Furthermore, the integration complements recent extensions of ASP Chef with visualization libraries such as Chart.js and vis.js, allowing for effective representation of both user-submitted data and ASP-generated results. This work paves the way for more accessible, interactive, and versatile ASP applications in areas such as decision support, education, and data exploration.

Keywords

answer set programming, ASP Chef, visualization, data analysis, surveys

1. Introduction

Answer Set Programming (ASP) is a declarative programming paradigm [1, 2, 3, 4, 5] renowned for its efficacy in knowledge representation, reasoning, and data-centric computation [6, 7, 8, 9]. Traditionally, ASP applications have operated on static datasets, offering limited support for interactive workflows or human-in-the-loop decision-making processes (usually focused on solution visualization [10, 11, 12, 13, 14, 15]). To bridge this gap, we present an extension of ASP Chef, a web-based framework for designing and executing ASP-based pipelines, through the integration of SurveyJS (http://surveyjs.io/), an open-source JavaScript library for building dynamic, JSON-configurable forms. This integration enables two primary capabilities:

- *Data Entry within ASP Chef Pipelines:* Users can input structured data through interactive forms, which are then seamlessly incorporated into the ASP reasoning process. This facilitates real-time data acquisition and dynamic pipeline execution.
- *Visualization of Processed Data:* ASP-derived results can be displayed within SurveyJS forms, allowing users to review and interact with the outcomes of ASP computations in an intuitive manner.

This bidirectional interaction fosters a more dynamic and user-centric approach to declarative problem-solving.

Our approach is inspired by existing efforts to introduce interactivity into ASP systems. Notably, Clinguin [16] allows developers to create interactive user interfaces directly within ASP by defining UIs as sets of facts, which are then rendered to provide continuous interaction with the ASP solver Clingo [17] based on user-triggered events. Similarly, the IDP system [18] incorporates visualization and limited forms of interactive data entry through its model expansion and visualization modules [12]. While these systems have advanced the state of interactive ASP applications, our integration with SurveyJS offers a modular and web-centric solution that leverages widely adopted web technologies for form creation and data handling.

We demonstrate the applicability of this integration through two primary classes of use cases that highlight the versatility of combining declarative logic with interactive web forms. In the context of the

Tech4You project (https://ecs-tech4you.it/), which focuses on smart technologies for environmental monitoring and climate change adaptation, we employ SurveyJS forms as a front-end interface to support interactive data analysis and visualization. Instead of requiring users to encode analysis parameters (such as threshold values or selected sensor variables) directly within ASP facts, SurveyJS forms provide a more intuitive and accessible means for field operators and domain experts to configure the analysis dynamically. Users interact with configurable forms to select which environmental parameters (e.g., pH, turbidity, temperature) should be visualized, define threshold conditions to flag anomalies, and control the behavior of visual dashboards generated by ASP Chef. These inputs are captured as structured JSON objects and fed into an ASP Chef pipeline, where they influence the logic-based processing and filtering of the underlying sensor data. For instance, a user may specify via a form that all turbidity values above 1600 FNU should be highlighted; this input is then interpreted as part of an ASP rule that detects threshold violations in the sensor data stream. In this setup, SurveyJS serves not only as a data collection tool but as an interactive configuration layer for declarative analysis. It allows for real-time adjustment of logic-driven queries without modifying the ASP code itself. Visualizations rendered through integrated libraries (e.g., vis.js, Tabulator, ApexCharts) can then be tailored on-the-fly based on these user-specified criteria.

In the educational domain, we use SurveyJS to construct self-evaluation quizzes integrated into digital learning environments. These quizzes, embedded within course modules, allow students to reflect on and assess their comprehension of lecture material in an interactive and self-guided manner. The quiz questions are defined using SurveyJS, while the assessment logic (including answer validation, feedback generation, and error detection) is handled declaratively using ASP rules. Once a student completes a quiz, their responses are passed into an ASP Chef pipeline, which processes the answers and computes an evaluation. This enables the system to generate automated, personalized feedback tailored to the performance of the student. The feedback may include targeted explanations, additional reading resources, or suggestions for specific topics to review, helping students consolidate their understanding. A key advantage of this approach over more conventional quiz platforms (such as Google Forms) is the flexibility in question serialization and loading. Instead of relying on web-based graphical interfaces to manually configure quizzes, instructors can define questions using a custom textual format, aligned with the JSON schema of SurveyJS. This allows rapid quiz creation via simple copy-and-paste operations using any text editor. The resulting quiz content is lightweight, portable, and easily versioned or generated programmatically, making it especially suitable for logic-based pipelines and large-scale deployment across course modules.

These use cases illustrate how the integration of SurveyJS significantly enhances the flexibility and usability of ASP-based pipelines. By allowing domain experts and end users to guide logic-driven diagnostics, configure visualization parameters, and contribute structured input through intuitive web interfaces, the system removes the need for direct interaction with ASP syntax or low-level encodings. More broadly, the integration of SurveyJS into ASP Chef enables novel forms of interaction between declarative reasoning and human input, fostering a tighter loop between logic inference and interface-driven customization. This design facilitates the development of accessible, dynamic, and domain-aware applications of ASP in real-world settings. By bridging the gap between declarative computation and interactive user interfaces, this work opens new pathways for building flexible, user-centric, and logic-driven tools in diverse domains such as decision support, environmental monitoring, educational technology, and data analytics.

2. Background

2.1. Answer Set Programming (ASP)

An ASP program is a finite set of rules. Each rule typically has a head, representing a conclusion (which may be atomic or a choice), and a body, representing a set of conditions that must hold (a conjunction of literals, aggregates and inequalities). Formally, an ASP program Π induces a collection (zero or more) of answer sets (also known as stable models), which are interpretations that satisfy all the rules in Π while also fulfilling the stability condition (i.e., the models must be supported and minimal in a specific formal sense [19]). Quantitative preferences can be expressed in terms of weak constraints. The intended output of a program can be specified using `#show` directives of the form

$$\texttt{\#show}\ p(\bar{t})\ \texttt{:}\ \mathit{conjunctive\_query}\,\texttt{.}$$

Here, p denotes an optional predicate symbol, $\bar{t}$ is a (possibly empty) sequence of terms, and $\mathit{conjunctive\_query}$ is a conjunction of literals serving as a condition for displaying instances of $p(\bar{t})$. Answer sets are then printed accordingly. For a detailed specification of syntax and semantics, including `#show` and other directives, we refer to the ASP-Core-2 standard format [20].

Example 1. Consider a scenario where we want to assign each student 1–2 lectures given their preferences (rankings), without exceeding lecture capacity. The following ASP program models this assignment problem:

```
r1 :  1 <= {assigned(S,L) : lecture(L,_)} <= 2 :- student(S,_).
r2 :  :- lecture(L,_), capacity(L,C), #count{S : assigned(S,L)} > C.
r3 :  :~ student(S,_), lecture(L,_), not assigned(S,L). [1@1, S,L]
r4 :  :~ preference(S,L,P), not assigned(S,L). [1@-P, S,L]
r5 :  #show (N, L) : assigned(S,L), student(S,N).
```

Rule r_1 is a *choice rule* that assigns 1–2 lectures to each student. Rule r_2 is a *constraint* ensuring no overbooking. Rules r_3 and r_4 are *weak constraints* preferring top-ranked lectures. The `#show` directive in r_5 ensures that only the assignments appear in the output (projecting the answer sets accordingly). Given the input facts

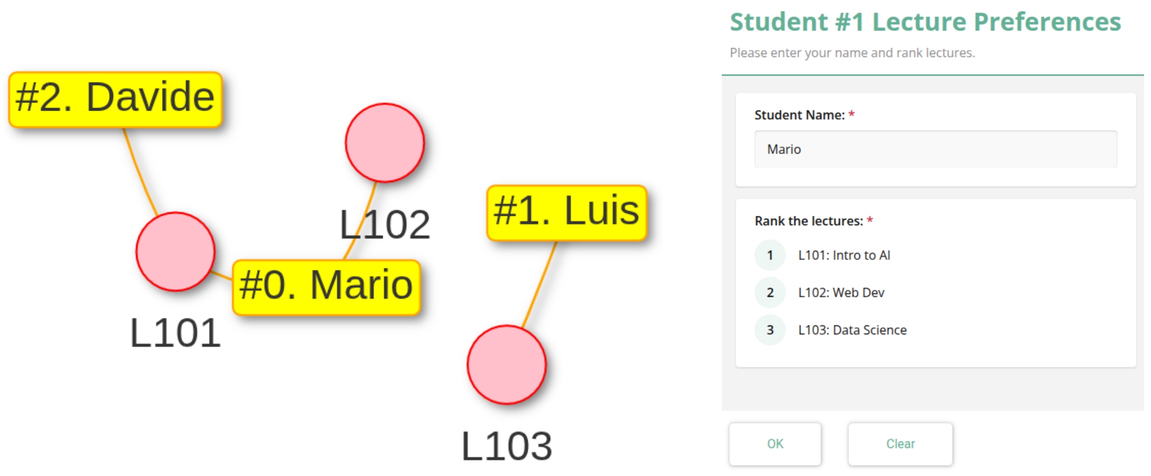
```
lecture("L101", "L101: Intro to AI").   capacity("L101", 2).
lecture("L102", "L102: Web Dev").       capacity("L102", 1).
lecture("L103", "L103: Data Science").  capacity("L103", 1).

student(1,"Mario").          student(2,"Luis").           student(3,"Davide").
preference(1,"L101",1).  preference(2,"L101",3).  preference(3,"L101",2).
preference(1,"L102",2).  preference(2,"L102",2).  preference(3,"L102",3).
preference(1,"L103",3).  preference(2,"L103",1).  preference(3,"L103",1).
```

the program has three optimal answer sets, among them `("Mario","L101")`, `("Mario","L102")`, `("Luis","L103")`, `("Davide","L101")`. ■

2.2. ASP Chef

An *operation* O is a function receiving as input a sequence of interpretations and producing as 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 $(\mathit{encode}, \mathit{Ingredients}, \mathit{decode})$, where $\mathit{Ingredients}$ 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 $[[\texttt{\_\_base64\_\_("}s\texttt{")}]]$, where $s = \mathit{Base64}(s_{in})$; this way, ASP Chef can deal with arbitrary content beyond the syntax of standard ASP. After that, the ingredients are applied one after another. Finally, if decode is true, every occurrence of $\texttt{\_\_base64\_\_(}s\texttt{)}$ is replaced with (the ASCII string associated with) $\mathit{Base64}^{-1}(s)$. Among the operations supported by ASP Chef there are *Encode*$\langle p, s\rangle$ to extend every interpretation as input with the atom $p\texttt{("}t\texttt{")}$, where $t = \mathit{Base64}(s)$; *Search Models*$\langle \Pi, n\rangle$ to replace every interpretation I as input with up to n answer sets of $\Pi \cup \{p(\bar{t}). \mid p(\bar{t}) \in I\}$; *Show*$\langle\Pi\rangle$ to replace every interpretation I as input with the projected answer set $\Pi \cup \{p(\bar{t}). \mid p(\bar{t}) \in I\}$ (where Π comprises only `#show` directives. *Optimize*$\langle \Pi, n\rangle$ to replace every interpretation I as input with up to n optimal answer sets of $\Pi \cup \{p(\bar{t}). \mid p(\bar{t}) \in I\}$.

Example 2. The problem from Example 1 can be addressed in ASP Chef by a recipe comprising a single *Optimize*$\langle\{r_1, r_2, r_3, r_4, r_5\}, 1\rangle$. Alternatively, a recipe separating computational and presentational aspects would comprise two ingredients, namely *Optimize*$\langle\{r_1, r_2, r_3, r_4\}, 1\rangle$ and *Show*$\langle\{r_5\}\rangle$. ■



Student #1 Lecture Preferences

Please enter your name and rank lectures.

Student Name: *

Mario

Rank the lectures: *

1 L101: Intro to AI

2 L102: Web Dev

3 L103: Data Science

Figure 1: Graphical representation of student-lecture assignment for the problem presented in Example 1 (left) and a SurveyJS form to rank lectures (right).

Several operations in ASP Chef support expansion of *Mustache templates* [21]; among them, there are *Expand Mustache Queries*, *@vis.js/Network* (to visualize graphs), *Tabulator* (to arrange data in interactive tables), and *ApexCharts* (to produce different kinds of charts). A Mustache template comprises queries of the form `{{ Π }}`, where Π is an ASP program with `#show` directives—alternatively, `{{=` $p(\overline{t})$ `:` *conjunctive_query* `}}` for `{{ #show` $p(\overline{t})$ `:` *conjunctive_query*`. }}`. Intuitively, queries are expanded using one projected answer set of $\Pi \cup \{p(\overline{t}).\mid p(\overline{t}) \in I\}$, where I is the interpretation on which the template is applied on. Separators can be specified using the predicates `separator/1` (for tuples of terms), and `term_separator/1` (for terms within a tuple). The varadic predicate `show/*` extends a shown tuple of terms (its first argument) with additional arguments that enable repeating tuples as output and can be used as sorting keys (using predicate `sort/1`). Moreover, Mustache queries can use `@string_format(format, ...)` to format a string using the given format string and arguments, and floating-point numbers are supported with the format `real("NUMBER")`. Format strings can also be written as (multiline) *f-strings* of the form `{{f"..."}}`, using data interpolation `${`*expression*`:`*format*`}` to render *expression* according to the given *format*.

Example 3. Recipes from Example 2 can be further extended by including ingredients to visualize input and computed solution graphically. To this aim, the following Mustache template can be combined with *@vis.js/Network* to obtain the graph shown in Figure 1:

```
{
  data: {
    nodes: [
      {{= {{f"{ id: "S${S}", label: "#${S}. ${N}", group: "student" }"}} :
            student(S,N) }}
      {{= {{f"{ id: ${L}, label: ${L}, group: "lecture" }"}}: lecture(L,_) }}
    ],
    edges: [ {{= {{f"{ from: "S${S}", to: "${L}" }"}} : assigned(S,L) }} ]
  },
  options: {
    nodes: {
      shape: "dot",
      size: 30,
      font: { size: 32 },
      borderWidth: 2,
      shadow: true
    },
    edges: {
      width: 2,
      shadow: true
    },
    groups: {
      student: {
        shape: "box",
        color: {
          background: "yellow",
          border: "orange"
        }
      },
      lecture: {
        color: {
          background: "pink",
          border: "red"
        }
      }
    }
  }
}
```

Mustache queries define nodes and links in the graph starting from facts in the computed answer set. A recipe addressing the selection problem and producing the visualization shown in Figure 1 is available at https://asp-chef.alviano.net/s/ASPOCP2025/student-lecture. ■

2.3. SurveyJS

SurveyJS is a comprehensive open-source framework for form creation and management in web applications. It provides a set of modular JavaScript libraries designed to simplify the development, deployment, and analysis of interactive forms and surveys. The system consists of four primary components:

- *Form Library:* This is the core component relevant to our integration. The Form Library is a free and open-source library, released under the MIT license, that allows developers to define and render dynamic forms using a JSON-based configuration format. It can be embedded into any modern web application and supports the collection of user responses, either locally or via a backend service. The library accommodates both simple use cases (such as contact or feedback forms) and more sophisticated scenarios involving conditional logic, calculated fields, and dynamic form behavior based on user input.
- *Survey Creator:* A graphical user interface (GUI) for building surveys and forms through a no-code, drag-and-drop environment. It is intended to be accessible to non-technical users, enabling rapid form design without programming knowledge.
- *Dashboard:* An interactive analytics component that provides customizable visualizations (such as charts and tables) to support the inspection and analysis of survey responses.
- *PDF Generator:* A tool for exporting surveys to PDF format, including both blank (fillable) and completed forms.

In this work, we focus exclusively on the Form Library, which serves as the foundation for capturing user input in our ASP-based pipeline system. While not directly integrated into our system, the Survey Creator can be used as a convenient external tool to prototype and sketch forms before refining them in JSON format for use within ASP Chef.

The Form Library of SurveyJS offers a comprehensive and extensible set of components for building sophisticated web-based forms. Its architecture supports full customization of form elements (including titles, placeholder texts, tooltips, and validation messages) allowing developers to craft user interfaces that are both accessible and context-aware. This high degree of configurability ensures a seamless and guided user experience, which is particularly beneficial in domains requiring precision and structured input. Beyond basic input types such as single-line text (e.g., for name, email, or date), long text, ratings, and radio button groups, the Form Library includes a diverse range of advanced question elements, including Single-Select Dropdown, Multi-Select Dropdown (Tag Box), Image Picker, Multiple Textboxes, Ranking and Rank-by-Selection Widgets, and HTML Blocks for Custom Layouts or Explanatory Content. In addition to its rich component library, the Form Library supports a variety of advanced features designed to enhance user engagement and facilitate complex form logic, among them *side navigation*, *progress bar* and *input validation*. Furthermore, the Form Library provides intuitive mechanisms for defining conditional logic and dynamic behavior. For example, form designers can specify visibility rules—making certain questions appear only if specific conditions are met, or dynamically adjust the content of confirmation messages, such as a "Thank You" page, based on user responses.

Example 4. Below is a JSON object that SurveyJS interpret to produce the form shown in Figure 1.

```
{
  "title": "Student #1 Lecture Preferences",
  "description": "Please enter your name and rank lectures.",
  "pages": [
    {
      "name": "page1",
      "elements": [
        {
          "type": "text",
          "name": "name",
          "title": "Student Name:",
          "isRequired": true
        },
        {
          "type": "ranking",
          "name": "lecture_preferences",
          "title": "Rank the lectures:",
          "choices": [
            { "value": "L101", "text": "L101: Intro to AI" },
            { "value": "L102", "text": "L102: Web Dev" },
            { "value": "L103", "text": "L103: Data Science" },
          ],
          "isRequired": true
        }
      ],
    }
  ]
}
```

Note that the second element in `page1` is of type `ranking` with `choices` for the lectures of Example 1. ■

3. SurveyJS Operation in ASP Chef

The integration of SurveyJS into ASP Chef is encapsulated as a dedicated pipeline operation, named *SurveyJS*, which is designed to dynamically generate and render interactive forms during pipeline execution. This operation enables the use of form-based human input at specified points within a logic-based workflow.

The survey operation accepts the following parameters:

- `predicate` (required): A unary predicate name whose extension contains a JSON-formatted string representing the SurveyJS configuration, as documented at https://surveyjs.io/form-library/documentation/overview. Each instance of `predicate/1` defines one survey form to be rendered, and may include a `data` property to pre-populate the form with existing data (e.g., for editing or continuing partially filled forms).

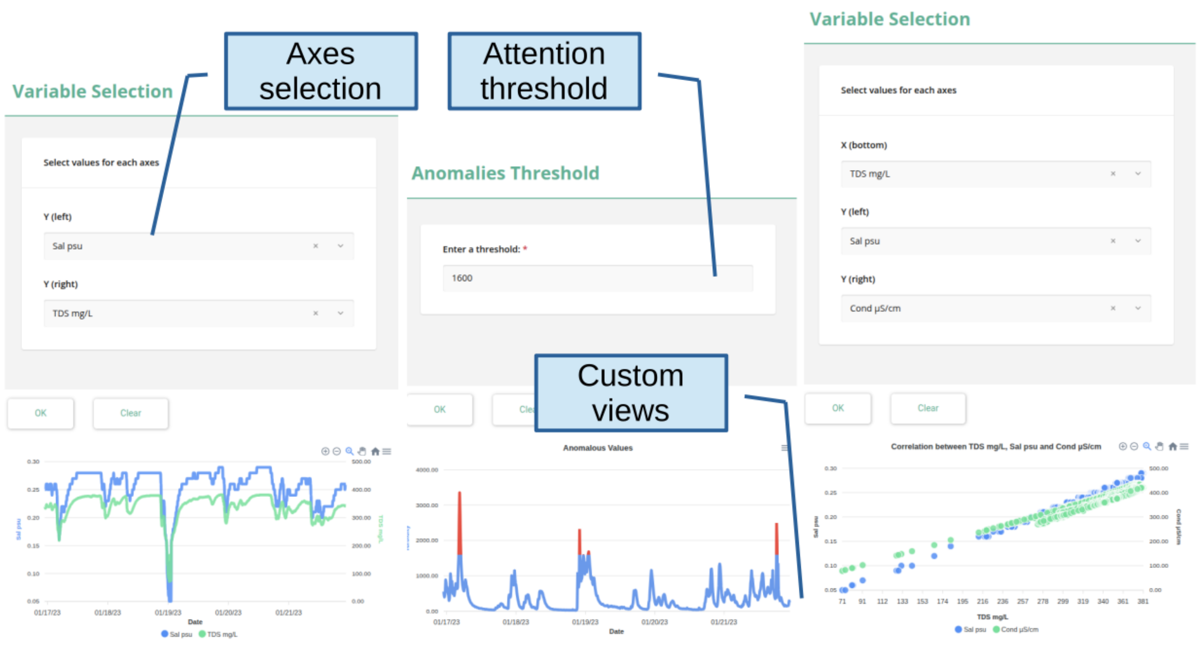
- `output_predicate` (optional): A unary predicate to which the collected form data is written, once the user submits the form by clicking the OK button. The resulting JSON object captures the values entered by the user and can be processed by downstream components in the pipeline.

The operation proceeds as follows:

1. *Form Configuration Resolution:* For each fact of the form `predicate(`*config*`)`, the operation extracts the JSON configuration string from the Base64-encoded string *config*, and parses it to produce a SurveyJS form. This configuration may contain Mustache placeholders to be resolved prior to rendering, based on the current ASP model.
2. *Form Pre-filling:* If the JSON configuration object includes a `data` property, its value is used to initialize the SurveyJS form with pre-filled values. This supports edit scenarios, workflow resumption, or data injection from earlier reasoning stages.
3. *User Interaction:* The SurveyJS form is rendered in the frontend. Users may complete the form and trigger submission by clicking the OK button. The system ensures that required fields are validated before submission proceeds.
4. *Data Capture:* Upon submission, the entered values are captured as a single JSON object and injected into the model as a new instance of `output_predicate/1`. Specifically, the JSON object is Base64-encoded so that is can be stored as a string compatible with the syntax accepted by ASP solvers. This object can then be consumed by subsequent operations, such as further ASP programs, Mustache templates, or data visualization steps.

Example 5. The JSON object from Example 4 can be revised as a Mustache template:

```
{
  "title": "Student #{{= ID : student(ID) }} Lecture Preferences",
  "description": "Please enter your name and rank lectures.",
  "pages": [
    {
      "name": "page1",
      "elements": [
        {
          "type": "text",
          "name": "name",
          "title": "Student Name:",
          "isRequired": true
        },
        {
          "type": "ranking",
          "name": "lecture_preferences",
          "title": "Rank the lectures:",
          "choices": [
            {{= {{f"{ "value": "${ID}", "text": "${Name}" },"}} :
            lecture(ID, Name) }}
          ],
          "isRequired": true
        }
      ],
    }
  ],
  data: {
    student_id: {{= ID : student(ID) }}
  }
}
```


Variable Selection

Select values for each axes

Y (left): Sal psu

Y (right): TDS mg/L

Anomalies Threshold

Enter a threshold: * 1600

Variable Selection

Select values for each axes

X (bottom): TDS mg/L

Y (left): Sal psu

Y (right): Cond µS/cm

Figure 2: Interactive customization of data visualization using SurveyJS forms in the Tech4You pipeline. Three forms illustrate different stages of user-driven configuration: (left) selection of two water quality parameters to compare over time; (center) definition of an attention threshold to highlight anomalies; (right) selection of two parameters to compare against a third control variable. The corresponding ApexCharts plots reflect these configurations in real time, enabling flexible, domain-informed exploration of environmental data.

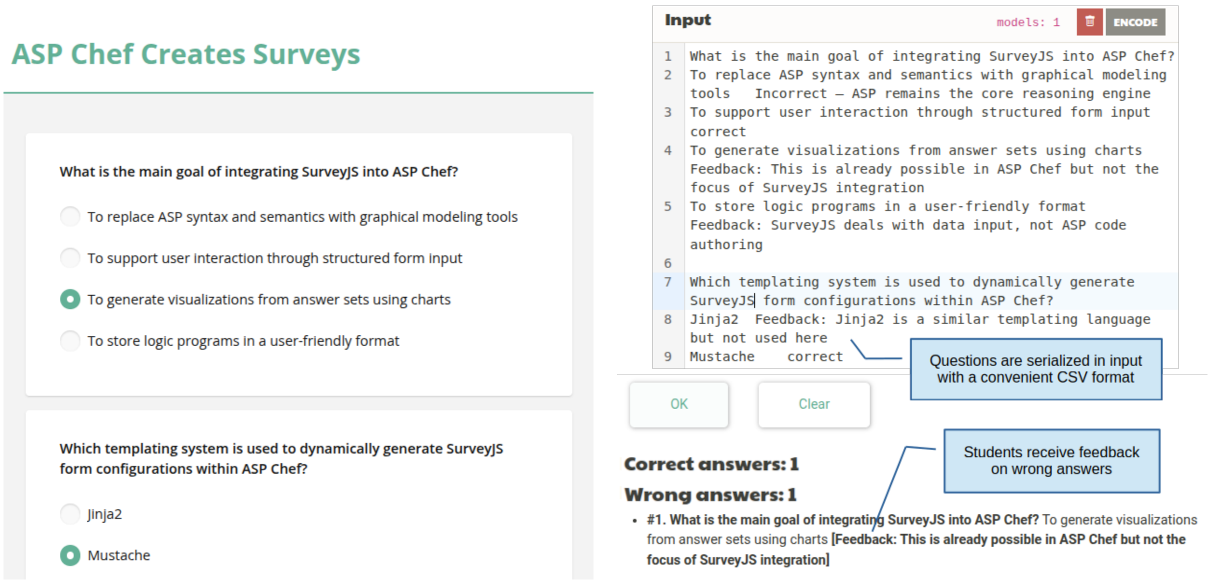
Combining the above template with `student(1)` and the facts representing lectures from Example 1 results into the JSON object from Example 4. We can therefore define a recipe (https://asp-chef.alviano.net/s/ASPOCP2025/student-lecture-2) including a form for setting the number of students, and then producing a form for each student using the above template. Alternatively, we can take further advantage of SurveyJS and produce a dynamic form (recipe hosted at https://asp-chef.alviano.net/s/ASPOCP2025/student-lecture-3). ■

4. Use Cases

4.1. Data Analysis in the Tech4You Project

One of the central goals of the Tech4You project is the development of intelligent tools for the analysis of environmental data, with a particular focus on real-time water quality monitoring. A suite of interactive visualizations was developed using ASP Chef to support domain experts in data exploration, anomaly detection, and hypothesis formulation [22]. Declarative ASP programs were employed to preprocess and filter incoming sensor data, as well as to encode and evaluate domain-specific conditions of interest (such as threshold violations, inter-parameter correlations, and logical consistency constraints). These logical results were rendered through integrated JavaScript libraries like vis.js, Tabulator, and ApexCharts, enabling rich and responsive visual outputs.

Initially, however, key configuration parameters (such as the choice of variables to visualize or the threshold values used to flag anomalies) were hard-coded directly in ASP facts. This design limited the ability of domain experts to adapt the analysis without modifying the underlying logic code. The integration of SurveyJS into ASP Chef has addressed this limitation by enabling the creation of dynamic, user-driven forms. Domain experts can now interact with a SurveyJS interface to select which sensor parameters (e.g., temperature, pH, turbidity) to visualize, and define threshold values for warnings or alerts. These inputs are captured as structured JSON objects through SurveyJS forms and are automatically injected into the ASP pipeline, without requiring any manual editing of ASP facts. The


ASP Chef Creates Surveys
Input
models: 1
ENCODE
OK
Clear
Questions are serialized in input with a convenient CSV format
Students receive feedback on wrong answers

What is the main goal of integrating SurveyJS into ASP Chef?

- ( ) To replace ASP syntax and semantics with graphical modeling tools
- ( ) To support user interaction through structured form input
- (●) To generate visualizations from answer sets using charts
- ( ) To store logic programs in a user-friendly format

Which templating system is used to dynamically generate SurveyJS form configurations within ASP Chef?

- ( ) Jinja2
- (●) Mustache

```
What is the main goal of integrating SurveyJS into ASP Chef?
To replace ASP syntax and semantics with graphical modeling tools   Incorrect – ASP remains the core reasoning engine
To support user interaction through structured form input   correct
To generate visualizations from answer sets using charts   Feedback: This is already possible in ASP Chef but not the focus of SurveyJS integration
To store logic programs in a user-friendly format   Feedback: SurveyJS deals with data input, not ASP code authoring

Which templating system is used to dynamically generate SurveyJS form configurations within ASP Chef?
Jinja2   Feedback: Jinja2 is a similar templating language but not used here
Mustache   correct
```

Correct answers: 1

Wrong answers: 1

- **#1. What is the main goal of integrating SurveyJS into ASP Chef?** To generate visualizations from answer sets using charts **[Feedback: This is already possible in ASP Chef but not the focus of SurveyJS integration]**

Figure 3: Rendered SurveyJS interface used to quiz readers about the paper. The form dynamically adapts based on the CSV as input, and feedback is computed using declarative ASP rules.

resulting configuration data is interpreted at runtime, allowing the logic program to adapt its reasoning and visualization logic accordingly. Examples taken from https://asp-chef.alviano.net/s/ASPOCP2025/t4y are shown in Figure 2.

4.2. Self-Evaluation Quizzes

One of the key applications of the SurveyJS integration in ASP Chef is the design and deployment of self-evaluation quizzes within digital learning environments. These quizzes provide students with a structured and interactive opportunity to assess their understanding of course material, while enabling instructors to automate feedback and track learning progress through declarative reasoning. Each quiz is defined as a SurveyJS form, where questions may range from single- and multiple-choice items to open-ended responses, dropdowns, or ranking tasks. The form structure is authored using a structured JSON format, optionally generated or customized via Mustache templates, which allows for rapid prototyping and flexible reuse across multiple course modules. The evaluation logic is expressed entirely through ASP rules, separating presentation from reasoning. SurveyJS is responsible for capturing raw responses, while the ASP Chef pipeline evaluates them using logical inference. This architecture enables a clean and modular design: form logic is decoupled from grading semantics, which can involve detecting misconceptions, scoring partial correctness, or suggesting remediation based on answer patterns. Upon form submission, the answers of the student are captured as a JSON object and passed into the pipeline. These responses are matched against expected answers, and the pipeline computes personalized feedback using inference rules. For example, if a student selects an incorrect answer associated with a known conceptual error, the system may recommend reviewing a specific lecture segment or provide an explanatory hint.

To demonstrate the capabilities of the SurveyJS integration in ASP Chef, we designed a self-contained recipe that allows readers to test their understanding of the paper itself. The recipe uses a series of multiple-choice questions, defined in a plain-text format, which are rendered into an interactive survey form. Answers are evaluated using ASP rules, and detailed feedback is presented based on the selected responses. The quiz is hosted at https://asp-chef.alviano.net/s/ASPOCP2025/quiz, and a screenshot is shown in Figure 3.

Input Format. Each question is given as a single line of text followed by four answer options, and each option includes a TAB-separated feedback annotation or the keyword `correct`. An extract is given in the Input panel shown in Figure 3.

Recipe Overview. This recipe is composed of the following main steps:

1. The plain-text quiz, formatted using TAB-separated values, is parsed using *Parse CSV*.
2. Facts of the form question(q_{id}, *question*) and answer(q_{id}, i, *answer*, *feedback*) are obtained using *Search Models*, *Index*, *Show* and other ASP Chef operations.
3. Questions and answers are transformed into SurveyJS JSON using the Mustache template

```
{
  title: "ASP Chef Creates Surveys",
  pages: [
    {
      elements: [
        {{= {{f"
          {
            type: "radiogroup",
            name: "${Q}",
            title: ${@string_double_quote(Text)},
            choices: [
              {{
                #show show(@string_double_quote(A),I) : answer(${Q},I,A,_).
                #show sort(2).
              }}],
          },
        "}} : question(Q, Text) }}
      ]
    }
  ]
}
```

4. The interactive quiz is shown using SurveyJS, and the answers provided by the student are stored in `__output__/1`.
5. Javascript is used to rewrite the JSON output into facts of the form __output__(q_{id}, *answer*), and the ASP rules

```
correct(Q) :- question(Q,_), __output__(Q,T), answer(Q,_,T,"correct").
wrong(Q) :- question(Q,_),__output__(Q,T),not answer(Q,_,T,"correct").
```

   mark the provided answers.
6. Markdown is used to generate a summary with score and targeted feedback:

```
#### Correct answers: {{= C : C = #count{Q : correct(Q)} }}

#### Wrong answers: {{= C : C = #count{Q : wrong(Q)} }}
{{
  #show show(
             {{f"- __#${Q}. ${T}__
                 {{= @string_concat(T', " __[", Feedback, "]__") :
                      __output__(${Q},T'), answer(Q,_,T',Feedback)
                 }}
             "}},
             Q
            ) : wrong(Q), question(Q,T).
  #show sort(2).
}}
```

5. Related Work

ASP Chef [23] is a lightweight, web-based environment designed to facilitate the exploration and transformation of answer sets through interactive pipelines. In its earlier iterations, ASP Chef emphasized the visual representation of interpretations [24], enabling users to inspect models via graphical views (using libraries such as @vis.js/Network for graph-like structures) and tabular displays via Tabulator for record-oriented data [21]. These features allowed users to reason about complex ASP outputs without diving into textual representations of answer sets, a valuable step toward making ASP more accessible to domain experts unfamiliar with its syntax. The present work builds on these capabilities by introducing SurveyJS-based form interaction within ASP Chef, which adds a new mode of visualization: structured data in the form of interactive web forms. In contrast to previous approaches of output representation, SurveyJS forms allow users not only to review results but also to parameterize and influence subsequent ASP computations directly through guided input. This advancement significantly enhances the support provided by the platform for interactive, human-in-the-loop workflows.

In parallel, several other systems have explored the integration of interactivity within ASP ecosystems. The Clinguin system [16] is a notable example that introduces graphical user interface construction natively within ASP logic programs. Developers describe the interface as sets of ASP facts, which are then interpreted and rendered dynamically, allowing continuous feedback loops between the user and the solver. The tight coupling of logic and UI definitions implemented by Clinguin enables fully reactive logic-driven interfaces without the need for external frontend logic, demonstrating a powerful synergy between ASP and UI design paradigms. The IDP system [25], part of the broader efforts of the Knowledge Representation community toward declarative problem-solving tools, also includes interactive visualization and model manipulation features [12]. IDP supports model expansion, and its visual interfaces allow users to inspect and interact with models under constraints, often leveraging graphical components or data tables to aid exploration. While IDP is more rooted in classical logic and model expansion frameworks, its goals align closely with making declarative content more navigable and user-friendly. Beyond these, there are other initiatives exploring similar intersections. Systems like Kara [11] (a tool for visualizing answer sets via annotations in ASP) and SeaLion [26] (an IDE with visual debugging for ASP) aim to reduce the barrier between declarative logic and its application environments by providing visual or semi-visual tooling.

Compared to these efforts, our approach in ASP Chef emphasizes modularity, web-native integration, and ease of configuration, rather than aiming to develop a fully reactive or general-purpose GUI system. The integration of SurveyJS leverages widely adopted web technologies (such as JSON, HTML5, and JavaScript) allowing forms to be defined, rendered, and processed with minimal overhead and broad compatibility. This choice makes ASP Chef more accessible and easier to extend than toolkits that rely on custom or logic-embedded interface specifications. A key enabler of flexibility in our integration is the use of Mustache templates [21], which allow SurveyJS form definitions to be dynamically constructed from ASP interpretations. This mechanism supports a decoupled design pattern, in which interface elements are not statically bound to logic programs, but rather generated as reactive content based on intermediate reasoning results. Related work by Haverinen et al. [27] proposes an architecture in which structured data, stored in a software repository, acts as the foundation for subsequent ASP-based compliance checks. Like ASP Chef, their approach utilizes JSON to record and structure domain knowledge, which is then interpreted by ASP rules. However, while their goal is centered around DevSecOps compliance, in ASP Chef the use of JSON (particularly in combination with Mustache templating) enables dynamic generation of forms and visualization elements based on logic program outputs. The role of JSON in both systems emphasizes its suitability as an intermediate representation format between user-facing interfaces and logic-driven reasoning backends.

However, ASP Chef deliberately avoids control structures such as loops or recursion in its pipeline architecture. As a result, it is not intended to support the development of fully-fledged interactive applications or live graphical interfaces in the same manner as systems like Clinguin or IDP. Instead, the integration of SurveyJS is conceived primarily as a way to provide parameter tuning and contextual input within a broader logic pipeline (e.g., selecting thresholds, toggling analysis options, or specifying visualization preferences) without modifying the underlying ASP code. From a technical perspective, the integration of user-specified parameters through SurveyJS forms can be viewed as analogous to the use of external atoms in ASP (cf. [28]). Both mechanisms enable data injection into logic programs from external sources. However, while external atoms are evaluated lazily and interleave with the propagation loop of the solver, in ASP Chef the form-based interaction occurs at the granularity of pipeline steps, where form results are serialized into JSON and re-fed into the next ingredient. While it is technically feasible to simulate more interactive behavior (such as maintaining application state on a local server and using polling or long-polling mechanisms to trigger periodic re-execution of ASP Chef recipes) we have not pursued this direction in practice. Such an approach, though theoretically viable, would likely be computationally inefficient, as it would require repeated re-execution of substantial portions of the recipe in response to each polling request. This highlights a key distinction in design philosophy: ASP Chef is optimized for lightweight, stateless, and compositional workflows, rather than continuous interaction or event-driven computation. In this context, the SurveyJS integration represents a pragmatic and effective means of bridging declarative inference with targeted user input, while remaining aligned with the lightweight and modular design principles of ASP Chef.

6. Conclusion

In this work, we presented the integration of SurveyJS, a modern open-source library for creating dynamic web forms, into ASP Chef, a web-based platform for constructing and executing ASP-based pipelines. This integration enriches ASP Chef with a new dimension of interactivity, allowing users to inject structured data into logic workflows and visualize ASP outputs in the form of interactive forms—bridging declarative reasoning and user-driven input. The design emphasizes modularity, web compatibility, and declarative configurability, achieved through the use of Mustache templates to generate JSON-based form configurations dynamically from ASP interpretations. This enables forms to be constructed and pre-filled based on reasoning results, while user responses are seamlessly captured and fed back into the pipeline for further analysis or transformation.

We demonstrated the effectiveness of this integration in two concrete domains. In the Tech4You project, SurveyJS forms are used to support data analysis and visualization for water quality monitoring, enabling domain experts to interactively configure diagnostic parameters and explore ASP-derived insights. In the educational domain, we employed SurveyJS to build self-evaluation quizzes that allow students to assess their understanding, with ASP rules providing personalized, logic-based feedback.

As a direction for future work, we foresee the extension of this integration to support visualization and modification of multiple records within the same form-based interface. This could be achieved by generalizing the current single-record survey operation or by introducing new operations explicitly designed to manage and iterate over collections of records associated with a shared form template. Such capabilities would significantly broaden the expressive power of form-based interaction in ASP Chef, enabling use cases such as batch editing, side-by-side comparison of interpretations, or iterative refinement of datasets within declarative workflows.

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 projects CAL.HUB.RIA (CUP H53C22000800006) and RADIOAMICA (CUP H53C22000650006); by the Italian Ministry of Enterprises and Made in Italy under project STROKE 5.0 (CUP B29J23000430005); under PN RIC project ASVIN "Assistente Virtuale Intelligente di Negozio" (CUP

B29J24000200005); and by the LAIA lab (part of the SILA labs). Mario Alviano is member of Gruppo Nazionale Calcolo Scientifico-Istituto Nazionale di Alta Matematica (GNCS-INdAM).

Declaration on Generative AI

During the preparation of this work, the authors used ChatGPT-4o for grammar and spelling check. After using this tool, the authors reviewed and edited the content as needed and take full responsibility for the publication's content.

References

[1] G. Brewka, T. Eiter, M. Truszczynski, Answer Set Programming at a glance, Commun. ACM 54 (2011) 92–103. doi:10.1145/2043174.2043195.

[2] E. Erdem, M. Gelfond, N. Leone, Applications of Answer Set Programming, AI Mag. 37 (2016) 53–68.

[3] V. Lifschitz, Answer Set Programming, Springer, 2019.

[4] R. Kaminski, J. Romero, T. Schaub, P. Wanko, How to Build Your Own ASP-based System?!, Theory Pract. Log. Program. 23 (2023) 299–361.

[5] M. Alviano, C. Dodaro, S. Fiorentino, A. Previti, F. Ricca, ASP and subset minimality: Enumeration, cautious reasoning and MUSes, Artif. Intell. 320 (2023) 103931.

[6] P. Cappanera, M. Gavanelli, M. Nonato, M. Roma, Logic-Based Benders Decomposition in Answer Set Programming for Chronic Outpatients Scheduling, TPLP 23 (2023) 848–864. doi:10.1017/S147106842300025X.

[7] M. Cardellini, P. D. Nardi, C. Dodaro, G. Galatà, A. Giardini, M. Maratea, I. Porro, Solving Rehabilitation Scheduling Problems via a Two-Phase ASP Approach, TPLP 24 (2024) 344–367. doi:10.1017/S1471068423000030.

[8] F. Wotawa, On the Use of Answer Set Programming for Model-Based Diagnosis, in: H. Fujita, P. Fournier-Viger, M. Ali, J. Sasaki (Eds.), IEA/AIE 2020, Kitakyushu, Japan, September 22-25, 2020, Proceedings, volume 12144 of *LNCS*, Springer, 2020, pp. 518–529. doi:10.1007/978-3-030-55789-8_45.

[9] R. Taupe, G. Friedrich, K. Schekotihin, A. Weinzierl, Solving Configuration Problems with ASP and Declarative Domain Specific Heuristics, in: M. Aldanondo, A. A. Falkner, A. Felfernig, M. Stettinger (Eds.), Proceedings of the 23rd International Configuration Workshop (CWS/ConfWS 2021), Vienna, Austria, 16-17 September, 2021, volume 2945 of *CEUR Workshop Proceedings*, CEUR-WS.org, 2021, pp. 13–20. URL: https://ceur-ws.org/Vol-2945/21-RT-ConfWS21_paper_4.pdf.

[10] O. Cliffe, M. D. Vos, M. Brain, J. A. Padget, ASPVIZ: Declarative Visualisation and Animation Using Answer Set Programming, in: ICLP, volume 5366 of *Lecture Notes in Computer Science*, Springer, 2008, pp. 724–728.

[11] C. Kloimüllner, J. Oetsch, J. Pührer, H. Tompits, Kara: A System for Visualising and Visual Editing of Interpretations for Answer-Set Programs, in: H. Tompits, S. Abreu, J. Oetsch, J. Pührer, D. Seipel, M. Umeda, A. Wolf (Eds.), Applications of Declarative Programming and Knowledge Management - 19th International Conference, INAP 2011, and 25th Workshop on Logic Programming, WLP 2011, Vienna, Austria, September 28-30, 2011, Revised Selected Papers, volume 7773 of *Lecture Notes in Computer Science*, Springer, 2011, pp. 325–344. URL: https://doi.org/10.1007/978-3-642-41524-1_20.

[12] R. Lapauw, I. Dasseville, M. Denecker, Visualising interactive inferences with IDPD3, CoRR abs/1511.00928 (2015). URL: http://arxiv.org/abs/1511.00928. arXiv:1511.00928.

[13] L. Bourneuf, An Answer Set Programming Environment for High-Level Specification and Visualization of FCA, in: S. O. Kuznetsov, A. Napoli, S. Rudolph (Eds.), FCA4AI 2018, Stockholm, Sweden, July 13, 2018, volume 2149 of *CEUR Workshop Proceedings*, CEUR-WS.org, 2018, pp. 9–20. URL: https://ceur-ws.org/Vol-2149/paper2.pdf.

[14] S. Hahn, O. Sabuncu, T. Schaub, T. Stolzmann, *Clingraph*: A System for ASP-based Visualization, Theory Pract. Log. Program. 24 (2024) 533–559. URL: https://doi.org/10.1017/s147106842400005x. doi:10.1017/S147106842400005X.

[15] A. Bertagnon, M. Gavanelli, ASPECT: Answer Set rePresentation as vEctor graphiCs in laTex, J. Log. Comput. 34 (2024) 1580–1607. URL: https://doi.org/10.1093/logcom/exae042. doi:10.1093/LOGCOM/EXAE042.

[16] A. Beiser, S. Hahn, T. Schaub, ASP-driven User-interaction with Clinguin, in: P. Cabalar, F. Fabiano, M. Gebser, G. Gupta, T. Swift (Eds.), Proceedings 40th International Conference on Logic Programming, ICLP 2024, University of Texas at Dallas, Dallas Texas, USA, October 14-17 2024, volume 416 of *EPTCS*, 2024, pp. 215–228. URL: https://doi.org/10.4204/EPTCS.416.19. doi:10.4204/EPTCS.416.19.

[17] M. Gebser, R. Kaminski, B. Kaufmann, T. Schaub, Multi-shot ASP solving with Clingo, Theory Pract. Log. Program. 19 (2019) 27–82. doi:10.1017/S1471068418000054.

[18] B. D. Cat, B. Bogaerts, M. Bruynooghe, M. Denecker, Predicate Logic as a Modelling Language: The IDP System, CoRR abs/1401.6312 (2014). URL: http://arxiv.org/abs/1401.6312. arXiv:1401.6312.

[19] M. Gelfond, V. Lifschitz, Logic programs with classical negation, in: D. Warren, P. Szeredi (Eds.), Logic Programming: Proc. of the Seventh International Conference, 1990, pp. 579–597.

[20] F. Calimeri, W. Faber, M. Gebser, G. Ianni, R. Kaminski, T. Krennwallner, N. Leone, M. Maratea, F. Ricca, T. Schaub, ASP-Core-2 Input Language Format, Theory Pract. Log. Program. 20 (2020) 294–309. URL: https://doi.org/10.1017/S1471068419000450. doi:10.1017/S1471068419000450.

[21] M. Alviano, W. Faber, L. A. Rodriguez Reiners, ASP Chef grows Mustache to look better, 2025. URL: https://arxiv.org/abs/2505.24537. arXiv:2505.24537.

[22] M. Alviano, L. A. Rodriguez Reiners, ASP Chef for Water Waste Monitoring, in: L. Pulina, L. Pandolfo (Eds.), Proceedings of the 40th Italian Conference on Computational Logic, Alghero, Italy, June 25-27, 2025, CEUR Workshop Proceedings, CEUR-WS.org, 2025.

[23] M. Alviano, D. Cirimele, L. A. Rodriguez Reiners, Introducing ASP recipes and ASP Chef, in: J. Arias, S. Batsakis, W. Faber, G. Gupta, F. Pacenza, E. Papadakis, L. Robaldo, K. Rückschloß, E. Salazar, Z. G. Saribatur, I. Tachmazidis, F. Weitkämper, A. Z. Wyner (Eds.), Proceedings of the International Conference on Logic Programming 2023 Workshops co-located with the 39th International Conference on Logic Programming (ICLP 2023), London, United Kingdom, July 9th and 10th, 2023, volume 3437 of *CEUR Workshop Proceedings*, CEUR-WS.org, 2023. URL: https://ceur-ws.org/Vol-3437/paper4ASPOCP.pdf.

[24] M. Alviano, L. A. Rodriguez Reiners, ASP Chef: Draw and Expand, in: P. Marquis, M. Ortiz, M. Pagnucco (Eds.), Proceedings of the 21st International Conference on Principles of Knowledge Representation and Reasoning, KR 2024, Hanoi, Vietnam. November 2-8, 2024, 2024. URL: https://doi.org/10.24963/kr.2024/68. doi:10.24963/KR.2024/68.

[25] B. Bogaerts, J. Jansen, B. D. Cat, G. Janssens, M. Bruynooghe, M. Denecker, Bootstrapping Inference in the IDP Knowledge Base System, New Gener. Comput. 34 (2016) 193–220. URL: https://doi.org/10.1007/s00354-016-0301-3. doi:10.1007/S00354-016-0301-3.

[26] P. Busoniu, J. Oetsch, J. Pührer, P. Skocovsky, H. Tompits, SeaLion: An eclipse-based IDE for answer-set programming with advanced debugging support, Theory Pract. Log. Program. 13 (2013) 657–673. URL: https://doi.org/10.1017/S1471068413000410. doi:10.1017/S1471068413000410.

[27] H. Haverinen, T. Janhunen, T. Päivärinta, S. Lempinen, S. Kaartinen, S. Merilä, Automating Cybersecurity Compliance in DevSecOps with Open Information Model for Security as Code, in: Proceedings of the 4th Eclipse Security, AI, Architecture and Modelling Conference on Data Space, eSAAM 2024, Mainz, Germany, 22 October 2024, ACM, 2024, pp. 93–102. URL: https://doi.org/10.1145/3685651.3686700. doi:10.1145/3685651.3686700.

[28] T. Eiter, T. Kaminski, C. Redl, A. Weinzierl, Exploiting Partial Assignments for Efficient Evaluation of Answer Set Programs with External Source Access, J. Artif. Intell. Res. 62 (2018) 665–727. URL: https://doi.org/10.1613/jair.1.11221. doi:10.1613/JAIR.1.11221.