Querying and Visualizing Digital Twins with ASP and ASP Chef: Preliminary Report

Mario Alviano, Paola Guarasci*

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

Abstract

This paper explores the integration of the Digital Twins Definition Language (DTDL) with ASP Chef, a web-based platform for Answer Set Programming (ASP). The goal of the research is to take advantage of the capabilities of ASP Chef to query, analyze, and visualize digital twins. As a first step in this direction, we introduce a method for mapping DTDL-defined digital twin models into ASP facts. This approach enables a seamless transition from high-level digital twin specifications to declarative reasoning and interactive visualization, offering a flexible framework for interpreting and exploring digital twin configurations.

Keywords

answer set programming, digital twins, visualization

1. Introduction

Digital twins are virtual representations of physical entities, such as sensors, rooms, vehicles, or more complex systems, designed to reflect their structure, behavior, and real-time state. These digital counterparts enable simulation, monitoring, and data-driven reasoning over physical environments. The modeling of digital twins requires a precise and expressive language to describe their properties, telemetry, relationships, and component structure. To this end, Microsoft has developed the Digital Twins Definition Language (DTDL), a domain-specific modeling language tailored to define digital twin interfaces and instances. DTDL is built upon JSON-LD (JavaScript Object Notation for Linked Data), a serialization format for linked data that extends standard JSON. JSON-LD was introduced as a way to integrate structured data into the evolving Semantic Web using familiar JSON syntax, while maintaining compatibility with RDF (Resource Description Framework). RDF defines a data model to represent relationships between entities on the Web. JSON-LD serves as a concrete serialization of RDF data and in fact can be simultaneously interpreted both as valid JSON and as an RDF document. This foundation allows DTDL to support semantic interoperability, reuse, and extensibility through the definition of domain-specific ontologies. Models in DTDL can inherit from one another, enabling abstraction and specialization, and can either define custom concepts or reuse industry-specific vocabularies. For instance, the RealEstateCore ontology for smart buildings (https://github.com/Azure/opendigitaltwins-building) and the EnergyGrid ontology for energy systems (https://github.com/Azure/opendigitaltwins-building) are both provided as reference models within the Azure Digital Twins ecosystem. Technically, a DTDL model is defined as a forest of DTDL elements—interfaces, properties, telemetry, commands, and relationships—organized under unique identifiers called DTMIs (Digital Twins Model Identifiers). These models can be composed, extended, and reused across different domains and applications (https://github.com/Azure/opendigitaltwins-dtdl/blob/master/DTDL/v4/DTDL.Specification.v4.md).

ASP Chef [1, 2] is a web-based platform designed to support interactive development, execution, and visualization of logic programs within the paradigm of Answer Set Programming (ASP). While many ASP environments focus primarily on the writing and debugging of code, ASP Chef adopts a higher-level perspective, enabling users to design and execute pipelines of operations, known as recipes,

CILC 2025: 40th Italian Conference on Computational Logic, June 25–27, 2025, Alghero, Italy

*Corresponding author.

mario.alviano@unical.it (M. Alviano); paola.guarasci@unical.it (P. Guarasci)
https://alviano.net/ (M. Alviano)
0000-0002-2052-2063 (M. Alviano); 0009-0005-8763-1431 (P. Guarasci)

that process and transform answer sets in a structured and modular fashion. Each recipe in ASP Chef is composed of a series of ingredients, where each ingredient encapsulates a specific operation: grounding a logic program, solving it to compute answer sets, filtering results based on user-defined conditions, applying queries, generating new interpretations, or rendering output in visual form. This modular design allows users to construct complex logic-based workflows in a step-by-step manner, facilitating experimentation, compositional reasoning, and reuse of processing logic across different applications.

What sets ASP Chef apart is its ability to manage interpretation streams (i.e., sequences of partial or complete answer sets) through these pipelines. Rather than returning a static set of results, recipes can transform intermediate results, chain operations, and ultimately produce outputs that are either textual (e.g., ASP facts) or graphical (e.g., network visualizations). The visual layer is powered by templating systems such as Mustache, integrated with JavaScript visualization libraries like *@vis.js/Network*, enabling interactive graphical representations of logical models. The user interface of ASP Chef is entirely browser-based and designed for accessibility and ease of use, lowering the barrier for both newcomers and experts. Users do not need to install any external solvers or tools locally; the platform integrates standard ASP solvers such as CLINGO and handles grounding, solving, and visualization client-side. This makes it particularly suitable for educational contexts, rapid prototyping, and data-rich domains where logical models must be both computed and interpreted by humans.

The objective of this work is to enable the analysis and visualization of digital twins defined using DTDL within the declarative and visual framework offered by ASP Chef. While DTDL provides a standardized, extensible way to model the structure and semantics of digital twins, defining interfaces, properties, relationships, and telemetry, its focus remains on representation rather than reasoning. By translating DTDL models into ASP facts, we can take advantage of the expressive power of ASP to perform complex queries, detect inconsistencies, infer implicit knowledge, and simulate behaviors over digital twin models. Integrating these two technologies allows for the creation of flexible reasoning pipelines where DTDL-defined twins serve as the data layer, and ASP Chef recipes provide the logic and visualization layers. In this approach, the declarative logic encoded in ASP can be used to filter or analyze digital twin instances. At the same time, the built-in support for templating and graph-based visualization in ASP Chef makes it possible to render digital twin networks in an interactive and human-friendly form. This integration thus bridges high-level semantic modeling with logic-based analysis and visual feedback, empowering users to inspect, reason over, and communicate digital twin configurations more effectively. From a practical standpoint, this integration is particularly relevant in contexts where explainability, rule-based control, or constraint checking over digital twins is essential. For example, in smart building scenarios, one may use DTDL to describe a the layout of a building, sensors, and control systems, and then use ASP to verify that sensor placement satisfies coverage constraints, or to detect redundant or missing components. In energy grids or manufacturing processes, logical rules may encode safety policies, energy efficiency conditions, or fault detection mechanisms that can be automatically applied to the representation of the digital twin.

2. Background

2.1. Digital Twins and the Digital Twins Definition Language (DTDL)

A digital twin is a digital abstraction of a physical entity, characterized by a structured data model and real-time data integration. It maintains a synchronized representation of the current state of a physical object, and can be used to analyze its historical behavior, and potential future evolution. Formally, a digital twin consists of four main parts:

1. A semantic model that defines the structure of the twin (e.g., properties, telemetry, commands, and relationships).
2. Bindings to live data sources that feed the model with real-time observations.
3. Behavioral logic for monitoring, diagnostics, or decision-making (e.g., rule-based systems or machine learning).

4. Interfaces for interaction, including APIs for querying or commanding the physical system.

Here we focus on the semantic model of digital twins.

The Digital Twins Definition Language (DTDL) is a modeling language designed to define the structure, behavior, and relationships of digital twins. It is built upon JSON-LD, enabling semantic interoperability and integration with linked data systems. DTDL v4 introduces several metamodel classes that serve as building blocks for modeling digital twins. The primary construct in DTDL are interfaces. An *Interface* encapsulates a set of elements (defined in the `contents` array) that describe the capabilities and structure of a digital twin. In other words, it defines a reusable model that specifies the data the twin can expose (through properties and telemetry), the actions it can perform (via commands), and its relationships with other digital twins. Each interface is uniquely identified by a Digital Twin Model Identifier (DTMI) and can include the following elements: properties defining readable and writable values representing the state of the twin; telemetries representing data emitted by the twin, typically used for monitoring or analytics; commands defining callable operations that the twin can perform; relationships describing links between the twin and other twins, establishing a graph of interconnected entities; components allowing composition by including other interfaces as part of the current interface, promoting modularity and reuse. Interfaces can also extend other interfaces (`extends` array), supporting inheritance and the creation of hierarchical models. A *Property* represents a value associated with the state of the twin. Properties can be simple (e.g., integers, strings) or complex (e.g., objects, arrays), according to their schema declaration. They may be marked as writable, indicating that their values can be updated. *Telemetry* elements define data that the twin emits, such as sensor readings or status updates. They specify the schema of the emitted data and are typically used for real-time monitoring. A *Command* specifies an operation that can be invoked on the twin. Commands can have defined request and response schemas, facilitating structured interactions with the twin. *Relationships* establish connections between twins, enabling the modeling of complex systems as interconnected graphs. Each relationship can specify a target interface, multiplicity, and can include properties to describe the nature of the connection (`properties` array). A *Component* allows an interface to include another interface, promoting modular design and reuse. Components are useful for modeling complex entities composed of simpler parts.

DTDL supports various schemas to define the data types used in properties, telemetry, and commands. Here we consider *primitive schemas* for basic data types such as boolean, integer, double, string, and dateTime, and *complex schemas* for structures like Object, Array, Map, and Enum. The *Object* schema is used to define records composed of multiple named fields (`fields` array), each associated with its own `name` and `schema`. The *Array* schema defines a homogeneous list of elements, where each element conforms to a single specified type (`elementSchema`). The *Map* schema allows the representation of key-value collections in which all keys are strings, and all values share a common schema; a map schema includes two nested objects, namely `mapKey` and `mapValue`, both defining a `name` and a `schema`. The *Enum* schema defines a controlled vocabulary of possible values for a property, telemetry, or command parameter; it consists of a `valueSchema` indicating the underlying data type (string, integer, or boolean), and an `enumValues` array (each value comprising `name` and `value`). Moreover, all constructs optionally include `comment`, `description` and `displayName` metadata.

Example 1. Here is an example taken from the DTDL documentation (https://github.com/Azure/opendigitaltwins-dtdl/blob/master/DTDL/v4/DTDL.v4.md):

```
{
  "@context": "dtmi:dtdl:context;4",
  "@id": "dtmi:com:example:thermostat;1",
  "@type": "Interface",
  "displayName": "thermostat",
  "contents": [
    {
      "@type": "Telemetry",
      "name": "temp",
      "schema": "double"
    },
    {
      "@type": "Property",
      "name": "setPointTemp",
      "writable": true,
      "schema": "double"
    }
  ]
}
```

The above digital twin defines an interface representing a device called `thermostat` (with DTMI `dtmi:com:example:thermostat;1`). The interface includes two elements in its `contents`: a *Telemetry* named `temp`, which is a read-only data stream representing the current temperature (using the `double` data type); a (writable) *Property* named `setPointTemp`, which allows users or systems to configure the desired target temperature (again, using the `double` data type). Overall, this digital twin model describes a simplified thermostat that reports current temperature via telemetry and allows control of the target temperature via a writable property. ■

2.2. Answer Set Programming (ASP)

In ASP, a (logic) program is composed of a finite set of rules that describe how certain facts, known as atoms, can be derived. Each rule consists of a head and a body. The head represents a possible conclusion, either a single atom or a choice among multiple atoms, while the body defines the conditions that must be satisfied for the head to hold. These conditions are expressed as a conjunction of literals, which may also include aggregates and arithmetic constraints. Depending on the form of the head, a rule may deterministically derive its conclusion or non-deterministically guess it as part of a choice. Formally, an ASP program Π gives rise to a set of answer sets (also known as stable models), which are interpretations that satisfy all the rules of Π while meeting a specific stability condition. This condition ensures that each model is both supported (i.e., every true atom is justified by the program) and minimal (i.e., no proper subset of the model also satisfies the program) [3]. Depending on the structure of the program, Π may yield zero, one, or multiple answer sets, each corresponding to a distinct solution of the problem encoded in the program.

To specify the intended output of a program, ASP allows the use of output directives, in particular the `#show` directive. These directives restrict the visible part of the answer sets to specific atoms or terms. The general form of a show directive is:

`#show` $p(\bar{t})$ `:` $conjunctive\_query$ `.`

Here, p denotes an optional predicate symbol, $\bar{t}$ is a (possibly empty) sequence of terms, and $conjunctive\_query$ is a conjunction of literals serving as a condition for displaying instances of $p(\bar{t})$. During answer set computation, only those atoms matching the criteria specified by one or more `#show` directives are retained in the output, effectively projecting the full answer sets onto a user-defined view. This mechanism offers a flexible and declarative approach to controlling the output of a program, which is especially valuable in large or complex settings where only a selected subset of the derived information is pertinent for subsequent processing or human interpretation.

For a detailed specification of syntax and semantics, including `#show` and other directives, we refer to the ASP-Core-2 standard format [4], which defines a widely-adopted common ground for ASP languages and tools.

Example 2. Consider a scenario where we want to select a subset of thermostat models that jointly provide a required set of features (e.g., telemetry and writable properties) for integration into a smart building system. The following ASP program models this selection problem:

r_1: `1 <= {selected(T) : thermostat(T)} <= N :- limit(N).`
r_2: `:- required(F), #count{T : selected(T), provides(T,F)} = 0.`
r_3: `#show T : selected(T).`

Rule r_1 is a *choice rule* that selects between 1 and N thermostat models (as determined by the fact `limit(N)`). Rule r_2 is a *constraint* that enforces full coverage: every required feature F must be covered by at least one selected thermostat. The `#show` directive in r_3 ensures that only the selected thermostats appear in the output, effectively projecting the answer sets to the relevant subset. Suppose the following input facts define the thermostats, the features they support, and the desired feature set:

```
feature (temp).       feature (setPointTemp).      feature(humidity).
required(temp).       required(setPointTemp).      limit(2).
thermostat(t1).       provides(t1, temp).          provides(t1, humidity).
thermostat(t2).       provides(t2, temp).          provides(t2, setPointTemp).
thermostat(t3).       provides(t3, setPointTemp).
```

In this setting, `temp` represents telemetry (e.g., current temperature), `setPointTemp` is a writable property (i.e., a configurable target temperature), and `humidity` is an additional, optional telemetry signal. The program has the projected answer set `t1 t3`, indicating that selecting thermostats `t1` and `t3` ensures coverage of the required features (`temp` and `setPointTemp`) using at most two thermostats.

Actually, in this case there is also an answer comprising `t2` alone. The number of selected thermostats can be minimized by adding to the program the following *weak constraint*:

```
:~ selected(T). [1@1, T]
```

To simplify the presentation, we omit the use of weak constraints in the remainder of this paper. ■

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 $(\mathit{encode}, \mathit{Ingredients}, \mathit{decode})$, 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 [[`__base64__("`s`")`]], where $s = \mathit{Base64}(s_{in})$ (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) $\mathit{Base64}^{-1}(s)$. 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 $t = \mathit{Base64}(s)$; *Search Models*$\langle\Pi, n\rangle$ to replace every interpretation I in input with up to n answer sets of $\Pi \cup \{p(\overline{t}). \mid p(\overline{t}) \in I\}$; *Show*$\langle\Pi\rangle$ to replace every interpretation I in input with the projected answer set $\Pi \cup \{p(\overline{t}). \mid p(\overline{t}) \in I\}$ (where Π comprises only `#show` directives.

Example 3. The problem from Example 2 can be addressed in ASP Chef by a recipe comprising a single *Search Models*$\langle\{r_1, r_2, r_3\}, 1\rangle$. Alternatively, a recipe separating computational and presentational aspects would comprise two ingredients, namely *Search Models*$\langle\{r_1, r_2\}, 1\rangle$ and *Show*$\langle\{r_3\}\rangle$. ■

Several operations in ASP Chef support expansion of *Mustache templates* [5]; 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, `{{= `$p(\overline{t})$` : `$\mathit{conjunctive\_query}$` }}` for `{{ #show `$p(\overline{t})$` : `$\mathit{conjunctive\_query}$`. }}`. 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 in 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 `${`$\mathit{expression}$`:`format`}` to render $\mathit{expression}$ according to the given format.

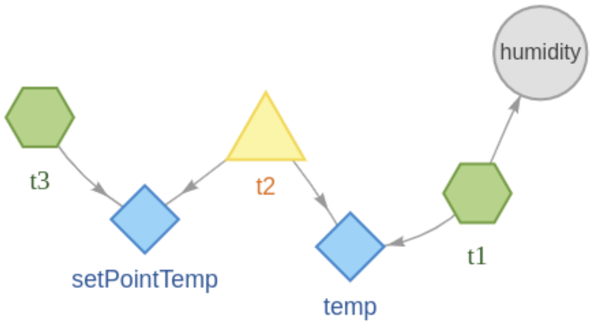


Figure 1: Graphical representation of input and selected thermostats for the problem presented in Example 2

Example 4. The recipes presented in Example 3 can be further enriched by adding specific ingredients for the graphical visualization of both the input data and the computed solutions. To this end, the Mustache template provided below can be employed in combination with the *@vis.js/Network* rendering engine to produce the network graph depicted in Figure 1:

```
{ data: {
    nodes: [
      {{= {{f"{ id: "${T}", label: "${T}",
                  group: "selected_thermostat" }"}} : thermostat(T), selected(T) }}
      {{= {{f"{ id: "${T}", label: "${T}",
                  group: "other_thermostat" }"}}: thermostat(T), not selected(T) }}
      {{= {{f"{ id: "${F}", label: "${F}",
                  group: "required_parameter" }"}} : parameter(F), required(F) }}
      {{= {{f"{ id: "${F}", label: "${F}",
                  group: "other_parameter" }"}} : parameter(F), not required(F) }} ],
    edges: [ {{= {{f"{ from: "${T}", to: "${F}" }"}} : provides(T,F) }} ],  },
  options: {
    nodes: { size: 28, font: {size: 20}, borderWidth: 2 },
    edges: { width: 1.5, color: {color: "#999", highlight: "#222", hover: "#444"},
             arrows: { to: { enabled: true, type: "arrow", scaleFactor: 0.7 } }, },
    groups: {
      selected_thermostat: { shape: "hexagon",
        color: { background: "#aed581", border: "#689f38" },
        font: { color: "#1b5e20", size: 22, face: "Georgia" } },
      other_thermostat: { shape: "triangle",
        color: { background: "#fff59d",  border: "#fdd835" },
        font: { color: "#ef6c00", size: 20 } },
      ⋮
```

It is worth noting that Mustache queries are employed to define the nodes and edges of the graph based on facts extracted from the computed answer set. A complete recipe addressing the selection problem, along with the visualization presented in Figure 1, is available at https://asp-chef.alviano.net/s/CILC2025/thermostat-selection. ■

3. Mapping DTDL Models to ASP Facts

To enable automated reasoning and visualization over digital twin models written in DTDL, we define a structured mapping from DTDL elements to ASP facts. This mapping captures both the syntactic structure and semantic annotations of DTDL interfaces and their contents, using a uniform and compositional set of predicates tailored for use within the ASP Chef platform.

3.1. Representing Device Structure and Sensor Data

Identifiers. Each DTDL interface is uniquely identified by a DTMI, a URI-like string that serves as a stable reference across all mappings. Elements (such as properties, telemetry, relationships, and commands) with no explicit ID declared within an owner construct (e.g., an interface) are identified using pairs of the form $(\mathit{OwnerId}, \mathit{Name})$, where *OwnerId* is the ID of the containing interface (e.g., the DTMI of the owner interface) and *Name* is the local name of the element within that interface. This tuple-based identifier scheme ensures uniqueness while preserving the hierarchical structure of DTDL. This convention also simplifies reasoning and querying: interface-scoped names avoid global namespace conflicts and enable consistent access to nested structures, which is particularly useful when dealing with multiple models or modular ontologies.

Metadata. Comments, descriptions and display names are uniformly represented using three predicates: `description/2`, `displayName/2`, and `comment/2`. These predicates are used to annotate both interfaces and their individual elements (e.g., properties, telemetry, relationships, commands). The first term in these predicates is the ID of the element, and the second term contains the metadata.

Interfaces. Each interface is represented by an instance of `interface/1`. Elements within the `contains` array of an interface (with ID) I are represented using the predicates `has_property/3`, `has_telemetry/3`, `has_command/3`, `has_relationship/3`, and `has_component/3`. Interfaces within the `extends` array of I are represented using the predicate `extends/2`.

Properties and Telemetries. Each property is represented by an instance of `property/1`. Properties marked as writable also occur in the mapping as instances of `writable/1`. Each telemetry is represented by an instance of `telemetry/1`. Telemetry elements in DTDL represent data emitted by the digital twin, such as sensor readings. The ASP encoding closely mirrors the property representation, with distinct predicates to differentiate their semantics:

Example 5. Let us consider the DTDL model defined in Example 1. Its mapping to ASP facts is the following:

```
interface(dtmi).
  displayName(dtmi, "Thermostat").
  has_telemetry(dtmi, "temp", (dtmi, "temp")).
  has_property(dtmi, "setPointTemp", (dtmi, "setPointTemp")).
telemetry((dtmi, "temp")).
  schema((dtmi, "temp"), "double").
property((dtmi, "setPointTemp")).
  schema((dtmi, "setPointTemp"), "double").
  writable((dtmi, "setPointTemp")).
```

where *dtmi* is `"dtmi:com:example:Thermostat;1"`. ■

3.2. Encoding Connectivity, Actions, and Data Types

Relationships. Each relationship is represented by an instance of `relationship/1`. If multiplicities are specified, they are represented by predicates `minMultiplicity/2` and `maxMultiplicity/2`. Similarly for the target interface, we use predicate `target/2`. As for properties, if a relationship is marked as writable, we add an instance of predicate `writable/1`. Each property within the `properties` array of a relationship (with ID) I is represented using the predicate `has_property/2`.

Commands and Components. Each command is represent by an instance of `command/1`. Request and response schemas of a command C are assigned IDs, say respectively *Req* and *Res*, and linked together with facts `command_request(`C, *Req*`)` and `command_response(`C, *Res*`)`. Nullable conditions are represented by instances of `nullable_command_request/1` and `nullable_command_response/1`. Each component is represented by an instance of `component/1`.

Schemas. The schema of each element is represented by an instance of `schema/2`. Primitive schemas are represented by a string (as defined in DTDL). Complex schemas are instead associated with instances of the following predicates: `object/1` for objects, and `has_field/3` for each named field with the `fields` array of an object; `array/2` for arrays, where the second term is the schema associated with elements in the array (`elementSchema`); `map/2` for maps, where the second term is the schema of `mapValue`; `enum/2` for enumerations, where the second term is the schema `valueSchema`, and `enum_value/3` for each named value within the `enumValues` array.

Example 6. Let us consider a relationship `connectedSensors` targeting up to 20 instances of the interface `SoilMoistureSensor` (here associated with DTMI `"dtmi:...:SoilMoistureSensor;1"`, where 1 is the version of the model):

```
{ "@type": "Relationship",
  "name": "connectedSensors",
  "target": "dtmi:...:SoilMoistureSensor;1",
  "maxMultiplicity": 20,
  "description": "Soil moisture sensors connected to this controller" }
```

It maps to the following facts:

```
has_relationship("dtmi:...","connectedSensors",("dtmi:...","connectedSensors")).
relationship(("dtmi:...","connectedSensors")).
maxMultiplicity(("dtmi:...","connectedSensors"),20).
target(("dtmi:...","connectedSensors"),"dtmi:...:SoilMoistureSensor;1").
description(("dtmi:...","connectedSensors"),
             "Soil moisture sensors connected to this controller").
```

Now, let us consider a property `operatingStatus` taking one value among `operational`, `maintenance`, `error`, and `disabled`:

```
{ "@type": "Property",
  "name": "operatingStatus",
  "schema": {
    "@type": "Enum",
    "valueSchema": "string",
    "enumValues": [
      { "name": "operational", "displayName": "Operational",
        "enumValue": "operational" },
      { "name": "maintenance", "displayName": "Under Maintenance",
        "enumValue": "maintenance" },
      { "name": "error", "displayName": "Error", "enumValue": "error" },
      { "name": "disabled", "displayName": "Disabled", "enumValue": "disabled" }
    ] } }
```

It maps to the following facts:

```
has_property("dtmi:...","operatingStatus",("dtmi:...","operatingStatus")).
property(("dtmi:...","operatingStatus")).
schema(("dtmi:...","operatingStatus"),("dtmi:...","operatingStatus")).
enum(("dtmi:...",operatingStatus),"string").
enum_value(("dtmi:...",operatingStatus),"operational","operational").
enum_value(("dtmi:...",operatingStatus),"maintenance","maintenance").
enum_value(("dtmi:...",operatingStatus),"error","error").
enum_value(("dtmi:...",operatingStatus),"disabled","disabled").
```

■

4. Querying and Visualizing Digital Twins

To enable advanced reasoning and interactive exploration of digital twins within the ASP Chef framework, we have extended the system with a custom operation, *@DTDL/Parse*. This operation is specifically designed to ingest digital twin models defined in DTDL (v4) and convert them into an ASP representation suitable for querying, optimization, and visualization.

4.1. The @DTDL/Parse Operation

The *@DTDL/Parse* operation is parameterized by two inputs: the name of a predicate used to carry the DTDL model(s), and a prefix to be prepended to all generated ASP facts, allowing for modular integration and namespace control within larger ASP programs. The operation works by Base64-decoding each term of the specified predicate, which is assumed to contain a serialized DTDL model in JSON format. It then applies the syntactic and semantic transformation detailed in Section 3, mapping DTDL constructs (such as interfaces, properties, relationships, and telemetry) into a corresponding set of ASP facts. This transformation enables the use of standard ASP reasoning techniques on data originally expressed in DTDL, providing a powerful bridge between model-based system design and logic-based reasoning.

Example 7. Let us consider again the DTDL model from Example 1. The ASP facts in Example 5 can be obtained by the recipe hosted at https://asp-chef.alviano.net/s/CILC2025/thermostat-parse. The recipe Base64-encodes its input (which contains the DTDL model) in order to be processed by *@DTDL/Parse*$\langle$`__base64__`$, \epsilon\rangle$ (where ϵ is the empty string). The result can be inspected with an *Output Encoded Content* ingredient, and it is ready to be further processed by other operations. ■

4.2. Integration with ASP Chef Operations

Once the DTDL input has been parsed and converted into ASP facts, the result is re-encoded in Base64 using the same predicate name. This design choice ensures compatibility with the broader ASP Chef pipeline, allowing the transformed data to be passed seamlessly to subsequent operations, including:

- *Search Models*, for filtering and exploring answer sets; and
- *Optimize*, for computing optimal configurations of the digital twin.

This modular and declarative approach promotes reusability and composability of recipes, particularly in contexts where digital twin models must be queried, filtered, or incrementally refined.

Example 8 (Continuing Example 7)**.** The recipe can be extended with a *Search Models* ingredient using the following program:

```
thermostat(T) :- interface(I), displayName(I,T).
provides(T,F) :- interface(I), displayName(I,T), has_property(I,F,_).
provides(T,F) :- interface(I), displayName(I,T), has_telemetry(I,F,_).
```

If the input comprises the three thermostats of Example 2, we can subsequently address the selection program by adapting the recipe from Example 3. A complete recipe is available at https://asp-chef.alviano.net/s/CILC2025/thermostat-search-models. ■

4.3. Visualizing Digital Twins with @vis.js/Network

A key advantage of integrating DTDL with ASP Chef lies in the ability to render digital twin models graphically using the *@vis.js/Network* operation. The ASP facts produced by *@DTDL/Parse* can be consumed by Mustache templates to extract relevant entities (such as interfaces, properties, and relationships) and translate them into a format suitable for visual display. For example, nodes in

```
node(interface,I,N) :- interface(I),     displayName(I,N).
node(interface,I,I) :- interface(I), not displayName(I,_).

node(property,P,N)  :- property(P),     displayName(P,N).
node(property,P,S)  :- property(P), not displayName(P,_), schema(P,S),
                       primitive_schema(S).
node(property,P,"") :- property(P), not displayName(P,_), schema(P,S),
                       not primitive_schema(S).

node(telemetry,T,N)  :- telemetry(T),     displayName(T,N).
node(telemetry,T,S)  :- telemetry(T), not displayName(T,_), schema(T,S),
                        primitive_schema(S).
node(telemetry,T,"") :- telemetry(T), not displayName(T,_), schema(T,S),
                        not primitive_schema(S).

node(object,O,N) :- object(O),     displayName(O,N).
node(object,O,O) :- object(O), not displayName(O,_).
node(property,(F,N),S) :- has_field(F,N,S), primitive_schema(S).

node(enum,E,N)  :- enum(E,S), displayName(E,N).
node(enum,E,S)  :- enum(E,S), not displayName(E,_),     primitive_schema(S).
node(enum,E,"") :- enum(E,S), not displayName(E,_), not primitive_schema(S).
node(enum,(E,N),V) :- enum_value(E,N,V).

link(O,V,N) :- has_property(O,N,V).
link(O,V,N) :- has_telemetry(O,N,V).

link(ID,S,"<<schema>>") :- schema(ID,S), not primitive_schema(S).

link(F,(F,N),N) :- has_field(F,_,S),     displayName(F,N),     primitive_schema(S).
link(F,S,N)     :- has_field(F,_,S),     displayName(F,N), not primitive_schema(S).
link(F,(F,N),N) :- has_field(F,N,S), not displayName(F,_),     primitive_schema(S).
link(F,S,N)     :- has_field(F,N,S), not displayName(F,_), not primitive_schema(S).

link(E,(E,N),N) :- enum_value(E,N,V).
```

Figure 2: ASP program (snippets) producing a labeled graph representation of a DTDL model

the resulting network can represent entities, while edges encode connections such as ownership and hierarchies. This visualization enables users to quickly grasp the structure of a digital twin, including its internal organization and interdependencies, without having to inspect raw data manually. A concrete strategy is publicly available at https://asp-chef.alviano.net/s/CILC2025/dtdl-vis. The recipe combines the ASP representation of the digital twin models in input with the program reported in Figure 2 to obtain a labeled graph that can be easily coupled with the Mustache template reported in Figure 3 to produce the *@vis.js/Network* shown in Figure 4. A larger example is provided in the next section.

5. Use Case: Digital Twin Modeling of a Vineyard

This section illustrates a use case involving the modeling of a vineyard as a system of interconnected digital twins. DTDL is used to define the digital counterparts of key physical components involved in viticulture. The goal is to demonstrate how a complex system composed of sensors, controllers, and monitoring devices can be translated into ASP facts for reasoning and visualization purposes.

```
{ data: {
    nodes: [
      {{= {{f"{ id: '${I}', label: '${N}', group: '${T}' }"}} : node(T,I,N) }} ],
    edges: [
      {{= {{f"{ from: '${F}', to: '${T}', label: '${L}', arrows: 'to' }"}}
        : link(F,T,L) }} ] },
  options: {
    nodes: { shape: "dot", size:20, font:{size:16}, borderWidth:2, shadow: true },
    edges: { width: 2, shadow: true },
    groups: {
      interface: { font: { size: 24 }, size: 30,
        color: { background: lightgreen, border: green } },
      property: { color: { background: yellow, border: orange } },
      telemetry: { color: { background: pink, border: red } },
      object: { shape: "hexagon", color: {background:"#d1c4e9",border: "#512da8"}},
      enum: { shape: "diamond", color: { background:"#fff9c4", border:"#fbc02d" }}
    } } }
```

Figure 3: ASP Chef Mustache template (snippets) to configure *@vis.js/Network* for visualizing DTDL models

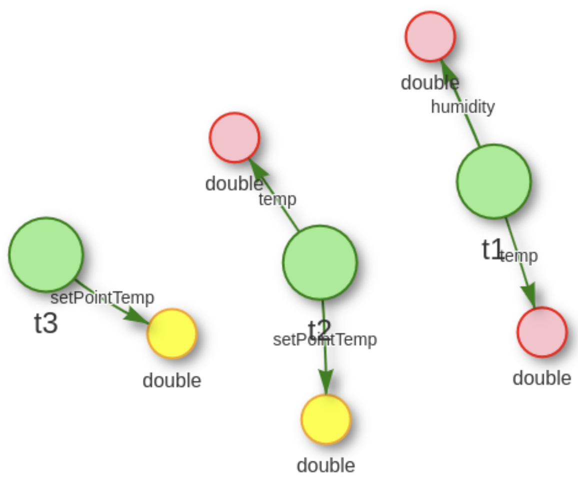


Figure 4: Visualization of the three thermostat DTDL models from Example 2 with *@vis.js/Network*

5.1. DTDL Interfaces for the Vineyard System

The system is composed of five main DTDL interfaces:

- **Vineyard Property:** represents the entire vineyard estate and acts as the central entry point of the model.
- **Pest Trap:** defines smart traps used for pest monitoring.
- **Soil Moisture Sensor:** describes environmental sensors installed in vineyard plots.
- **Irrigation Controller:** models smart irrigation systems used to manage water distribution.
- **Grape Monitor:** represents devices used to monitor grape development and maturity.

These interfaces define properties, telemetry data, commands, and relationships. For example, the **Vineyard Property** model includes descriptive fields such as name, owner, totalArea, and spatial data (location), as well as relationships to vineyard plots, weather stations, and wineries.

5.2. From DTDL to ASP: A Vineyard Interface

To enable reasoning over digital twin data in ASP Chef, each DTDL model is mapped to a collection of ASP facts. This translation makes it possible to analyze the model declaratively, verifying constraints and inferring system-level properties.

5.3. Modeling System Relationships and Constraints

A key part of the vineyard system lies in the relationships among components. The DTDL models define cardinality constraints and logical links between devices:

- **Vineyard Property** may contain up to 100 **Vineyard Plot** instances.
- **Irrigation Controller** can manage up to 10 plots.
- **Soil Moisture Sensor** and **Pest Trap** devices are installed within individual plots.
- **Grape Monitor** monitors a single plot and provides detailed telemetry.

These relationships are mapped into ASP facts using constructs like:

```
has_relationship("dtmi:...:VineyardProperty;1", "hasVineyardPlots",
                 "dtmi:...:VineyardPlot;1").
maxMultiplicity(("dtmi:...:VineyardProperty;1","hasVineyardPlots"), 100).
target(("dtmi:...:VineyardProperty;1","hasVineyardPlots"),
        "dtmi:...:VineyardPlot;1").
```

This structure supports the application of inference rules for validating instance data and ensuring conformance with the intended digital twin architecture.

5.4. Reasoning and Simulation

The resulting ASP knowledge base allows for reasoning tasks such as:

- Verifying that each **Vineyard Plot** is assigned a moisture sensor.
- Ensuring that no **Irrigation Controller** exceeds its control limit.
- Identifying unmonitored plots or plots with conflicting sensor assignments.
- Simulating scheduling decisions based on sensor telemetry (e.g., when to irrigate or harvest).

For example, relationships specifying as target an interface model not provided in input can be easily identified with the following rule:

```
node(undefined_interface,T,T) :- target(_,T), not interface(T).
```

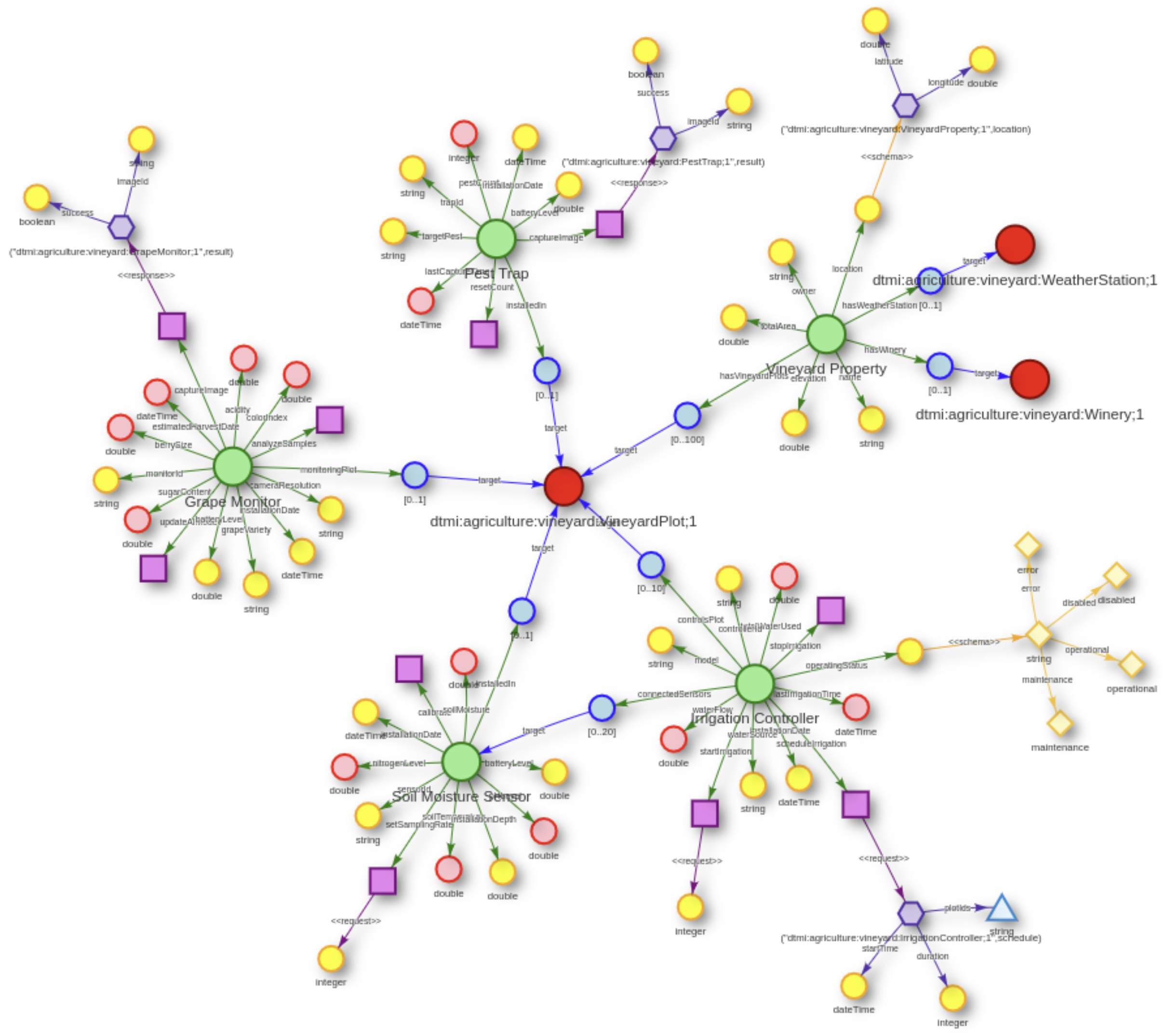
The obtained instance of `node/3` can in turn be coupled with a Mustache template extending Figure 3 with the group

```
undefined_interface: { font: { size: 24 }, size: 30,
  color: { background: red, border: darkred } },
```

to highlight in red any undefined interface, as shown in Figure 5. A complete recipe is available at https://asp-chef.alviano.net/s/CILC2025/vineyard.

6. Related Work

Digital twins have gained significant attention in recent years as a paradigm for modeling, monitoring, and controlling cyber-physical systems. DTDL, developed by Microsoft, has emerged as a standard for describing digital twin interfaces in a structured and extensible way. It is widely used in industrial IoT settings, particularly within the Azure Digital Twins platform. While DTDL focuses on the semantic modeling of digital entities, it does not offer built-in mechanisms for logical reasoning or constraint checking. In contrast, ASP provides a formalism well-suited for representing complex knowledge and deriving conclusions under non-monotonic reasoning. ASP has been successfully applied in configuration [6, 7, 8], diagnosis [9, 10, 11], and planning [12, 13, 14], but its integration with digital twin technologies remains underexplored.



dtmi:agriculture:vineyard:VineyardPlot;1
dtmi:agriculture:vineyard:WeatherStation;1
dtmi:agriculture:vineyard:Winery;1
Vineyard Property
Pest Trap
Grape Monitor
Soil Moisture Sensor
Irrigation Controller
("dtmi:agriculture:vineyard:GrapeMonitor;1",result)
("dtmi:agriculture:vineyard:PestTrap;1",result)
("dtmi:agriculture:vineyard:VineyardProperty;1",location)
("dtmi:agriculture:vineyard:IrrigationController;1",schedule)
<<response>>
<<schema>>
<<request>>
target
[0..1]
[0..100]
[0..10]
[0..20]
string
double
integer
boolean
dateTime
error
disabled
operational
maintenance

Figure 5: DTDL models of the vineyard use case, with undefined interfaces highlighted in red

On the visualization side, tools such as vis.js have been used extensively for interactive graph visualization. ASP Chef [1] introduced its first visualization functionality in [2] through the addition of the *Graph* ingredient, a component designed to generate graph-based visualizations as side effects of logic program executions. This feature enabled users to embed visual directives within ASP programs themselves, producing interactive graphical representations of answer sets. The approach follows the tradition of tools like ASPViz[15], IDPD3[16], and Kara [17], which link answer set semantics to visual metaphors. Building upon this foundation, recent versions of ASP Chef have taken a different route to enhance expressiveness and flexibility: rather than embedding visuals directly in ASP rules, they take advantage of Mustache templates to configure external JavaScript visualization libraries by querying answer sets. This model-driven approach allows ASP facts and their relationships to drive the generation of structured visual outputs. Notably, ASP Chef now supports integration with several high-quality frontend libraries, among them *@vis.js/Network* for interactive network diagrams. These capabilities allow users to construct rich, interactive visualizations tailored to the structure of ASP outputs, making the tool especially useful for teaching, demonstration, and domain-specific analytics. The use of templating decouples the visualization logic from ASP reasoning, promoting reusability and composability of recipes. Our work extends ASP Chef with the capability to ingest DTDL models, enabling a novel interplay between standard digital twin representations and rule-based reasoning.

In summary, while prior works have addressed model-based design, digital twin representation, or ASP reasoning in isolation, this paper contributes a novel (yet preliminary) integration that combines semantic modeling (via DTDL), logical reasoning (via ASP), and interactive visualization (via ASP Chef), offering a complete pipeline for digital twin analysis and validation.

7. Conclusion

This paper presents a preliminary report on our ongoing effort to extend ASP Chef with support for the Digital Twins Definition Language (DTDL). Our goal is to enable reasoning and visualization capabilities over digital twin models by integrating a widely adopted semantic modeling language with the declarative power of Answer Set Programming. To this end, we defined a mapping from DTDL constructs (such as interfaces, properties, telemetry, commands, and relationships) to ASP facts, capturing the structure and semantics of digital twin models in a logic-programmable format. This mapping has been implemented as a new operation within ASP Chef, which processes Base64-encoded DTDL models and emits a corresponding set of ASP facts. The new functionality integrates seamlessly with other ASP Chef operations. It supports model querying, constraint validation, and the generation of interactive visualizations through templated configurations of front-end libraries such as *@vis.js/Network*. The approach is composable and declarative, aligning well with the recipe-based workflow typical of ASP Chef.

A current limitation of our work lies in the absence of support for the serialization of digital twin instances, that is, concrete entities conforming to DTDL models, along with their runtime data (e.g., sensor readings, command states). Addressing this aspect will be the focus of future development. Our goal is to enable full digital twin lifecycle support, from model ingestion to instance management and data-driven reasoning.

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] M. Alviano, D. Cirimele, L. A. Rodriguez Reiners, Introducing ASP recipes and ASP Chef, in: ICLP Workshops, volume 3437 of *CEUR Workshop Proceedings*, CEUR-WS.org, 2023.

[2] 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.

[3] 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.

[4] 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.

[5] 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.

[6] C. Dodaro, P. Gasteiger, N. Leone, B. Musitsch, F. Ricca, K. Schekotihin, Combining answer set programming and domain heuristics for solving hard industrial problems (application paper), Theory Pract. Log. Program. 16 (2016) 653–669. URL: https://doi.org/10.1017/S1471068416000284. doi:10.1017/S1471068416000284.

[7] E. Gençay, P. Schüller, E. Erdem, Applications of non-monotonic reasoning to automotive product configuration using answer set programming, J. Intell. Manuf. 30 (2019) 1407–1422. URL: https://doi.org/10.1007/s10845-017-1333-3. doi:10.1007/S10845-017-1333-3.

[8] V. Myllärniemi, J. Tiihonen, M. Raatikainen, A. Felfernig, Using answer set programming for feature model representation and configuration, in: A. Felfernig, C. Forza, A. Haag (Eds.), Proceedings of the 16th International Configuration Workshop, Novi Sad, Serbia, September 25-26, 2014, volume 1220 of *CEUR Workshop Proceedings*, CEUR-WS.org, 2014, pp. 1–8. URL: https://ceur-ws.org/Vol-1220/01_confws2014_submission_14.pdf.

[9] F. Wotawa, D. Kaufmann, Model-based reasoning using answer set programming, Appl. Intell. 52 (2022) 16993–17011. URL: https://doi.org/10.1007/s10489-022-03272-2. doi:10.1007/S10489-022-03272-2.

[10] F. Wotawa, On the use of answer set programming for model-based diagnosis, in: H. Fujita, P. Fournier-Viger, M. Ali, J. Sasaki (Eds.), Trends in Artificial Intelligence Theory and Applications. Artificial Intelligence Practices - 33rd International Conference on Industrial, Engineering and Other Applications of Applied Intelligent Systems, IEA/AIE 2020, Kitakyushu, Japan, September 22-25, 2020, Proceedings, volume 12144 of *Lecture Notes in Computer Science*, Springer, 2020, pp. 518–529. URL: https://doi.org/10.1007/978-3-030-55789-8_45. doi:10.1007/978-3-030-55789-8_45.

[11] A. A. Falkner, G. Friedrich, K. Schekotihin, R. Taupe, E. C. Teppan, Industrial applications of answer set programming, Künstliche Intell. 32 (2018) 165–176. URL: https://doi.org/10.1007/s13218-018-0548-6. doi:10.1007/S13218-018-0548-6.

[12] V. Lifschitz, Answer set programming and plan generation, Artif. Intell. 138 (2002) 39–54. URL: https://doi.org/10.1016/S0004-3702(02)00186-8. doi:10.1016/S0004-3702(02)00186-8.

[13] V. Nguyen, V. L. Stylianos, T. C. Son, W. Yeoh, Explainable planning using answer set programming, in: D. Calvanese, E. Erdem, M. Thielscher (Eds.), Proceedings of the 17th International Conference on Principles of Knowledge Representation and Reasoning, KR 2020, Rhodes, Greece, September 12-18, 2020, 2020, pp. 662–666. URL: https://doi.org/10.24963/kr.2020/66. doi:10.24963/KR.2020/66.

[14] T. C. Son, E. Pontelli, M. Balduccini, T. Schaub, Answer set planning: A survey, CoRR abs/2202.05793 (2022). URL: https://arxiv.org/abs/2202.05793. arXiv:2202.05793.

[15] 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.

[16] R. Lapauw, I. Dasseville, M. Denecker, Visualising interactive inferences with IDPD3, CoRR abs/1511.00928 (2015).

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