

aspguid: A Declarative GUI Specification Language for ASP Programs

BACHELOR'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Bachelor of Science

in

Software & Information Engineering

by

Niklas Natter

Registration Number 1425344

to the Faculty of Informatics

at the TU Wien

Advisor: O.Univ.Prof. Dipl.-Ing. Dr.techn. Thomas Eiter

Assistance: Univ.Ass. Dipl.-Ing. Dipl.-Ing. Dr.techn. Christoph Redl

Vienna, 22nd September, 2017

Niklas Natter

Thomas Eiter

Erklärung zur Verfassung der Arbeit

Niklas Natter
Eisenstadtplatz 2/8
1100 Wien

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 22. September 2017

Niklas Natter

Acknowledgements

First and foremost, I would like to thank my research supervisor Dr. Christoph Redl, who incessantly assisted me on every step throughout the journey of researching and writing this thesis. The contentious conversations during our regular meetings and the incredible fast responses to my unscheduled requests helped me a lot to keep the right direction and make progress. I am very thankful for the personal thoughts and suggestions for improvements. The accomplishment of completing this thesis would not have been possible without this dedicated supervision.

Furthermore, I wish to thank Prof. Dr. Thomas Eiter for the valuable feedback and the personal thoughts in the final phase of writing this thesis.

Finally, I would like to express my very profound gratitude to my parents Annette Natter and Otmar Natter and to my significant other Julia Moll, for their continual and irreplaceable emotional support during my time at the university. I am especially grateful for all the material and non-material assistance during the last years, which allowed me to completely focus my mind on my education. I am fully aware that this kind of support is not self-evident. Thank you.

Abstract

Answer Set Programming (ASP) is a fully declarative programming paradigm, rooted in logic programming and non-monotonic reasoning [10]. The simple syntax of ASP, paired with the high expressiveness of the language and the availability of efficient ASP solvers, enable an intuitive and efficient way for solving a wide range of computational problems.

Nevertheless, the ASP language is not a full general-purpose language [13]. Software applications typically provide graphical user interfaces, that allow users without prior programming experience to make use of the encoded core functionality. While the ASP language allows to realize this core functionality in a convenient manner in many cases, graphical user interfaces cannot be implemented in the ASP paradigm.

At the moment, this restriction of the ASP paradigm is bypassed by embedding ASP programs into traditional imperative applications, wherein graphical user interfaces are developed using existing imperative programming techniques. Unfortunately, this procedure results in a significant overhead for developers and leads to a notable amount of repetitive work when implementing multiple applications.

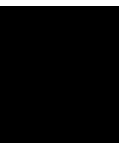
To address this issue, this thesis proposes a new approach, named `aspguid`, for implementing applications usable by users without prior logic programming experience based on annotated ASP programs. The proposed approach consists of a declarative language for defining graphical user interfaces for ASP programs and a compiler that translates ASP programs which are annotated with such definitions into procedural code, that realizes respective graphical user interface assisted applications.

In total, the `aspguid` approach provides a straightforward solution for developing ASP-dependent applications usable by non-experts, which significantly reduces the development overhead and minimizes the repetitive workload for developers, compared to current state of the art solutions.

Contents

1	Introduction	1
2	Answer Set Programming	5
2.1	Syntax	5
2.2	Semantics	6
2.3	Examples	6
3	Structure of ASP Encodings	9
3.1	Uniform Problem Encoding	9
3.2	Input of an ASP Application	10
3.3	Output of an ASP Application	11
4	aspguid Language	13
4.1	GUI-Definition Object	14
4.2	program_information Object	14
4.3	program_input Object	15
4.4	program_output Object	18
4.5	Value-Property Types	21
5	Implementation of the Compiler	27
5.1	aspguid Compiler	27
5.2	Compiled Java Application	29
6	Showcase & Assessment	33
6.1	Project Planning Problem	33
6.2	Project Planning Application	34
6.3	Project Planning ASP Encoding	37
6.4	Project Planning GUI Definition	37
6.5	Showcase Assessment	39
7	Conclusion	41
7.1	Related Work	42
7.2	Future Work	43

A	apsguid Language Specification	45
A.1	GUI-Definition Object	45
A.2	program_information Object	46
A.3	program_input Object	46
A.4	program_output Object	49
B	Showcase: Project Planning	53
B.1	Annotated Project Planning ASP Program	53
B.2	Example Project Planning Problem Instance	56
	Bibliography	57



Introduction

Answer Set Programming (ASP) is a fully declarative programming paradigm, rooted in logic programming and non-monotonic reasoning [23]. In ASP, computational problems are encoded as logic programs, whose answer sets correspond to the solutions of the computational problem [17]. Answer sets of such ASP programs are interpretations that consist of ground first-order literals and are calculated by ASP solvers.

The simple syntax of ASP, paired with the high expressiveness of the language and the availability of efficient ASP solvers, enable an intuitive and efficient way for solving a wide range of computational problems. The ASP paradigm has been successfully applied in various scientific areas, ranging from planning and decision making [3, 2, 15] to knowledge management [1, 14, 6, 21], and was recently used in several industry-level applications such as workforce management [28] and e-tourism planning [29].

Nevertheless, the ASP language is not a full general-purpose language [13]. While the ASP language allows to realize a wide range of complex computational problems with better readability and extensibility compared to imperative programming languages [13], aspects like graphical user interfaces cannot be implemented in the ASP paradigm. Due to this incapability, input data for ASP programs must be formalized in the ASP syntax and results of ASP programs are presented as raw answer sets only.

While these restrictions of ASP programs are negligible for the use in the scientific environment, they lead to a significant obstacle for users without prior logic programming experience. Especially the incapability of realizing intuitive user interfaces severely hinders the development of applications usable by non-experts in the ASP language and therefore decelerates the growth of the ASP community. We will refer to applications usable by users without programming experience as *end-user applications* from this point.

At the moment, end-user applications that make use of the advantages of the ASP paradigm are realized by embedding ASP programs into imperative applications. In these applications, the graphical user interface is developed using existing imperative programming techniques, while the core problem is encoded and solved in the ASP paradigm. On evaluation, the input data is passed from the imperative user interface to the embedded ASP program. At a latter point, the results are read from the answer sets and displayed in the user interface.

To simplify the process of embedding ASP programs into imperative applications, various techniques which support the integration of ASP with other programming languages were proposed recently [13, 27, 8, 30]. These approaches valuably assist developers by handling the evaluation of ASP programs, managing the required data representation format conversions and controlling the data flow between ASP programs and imperative components of an application.

Still, these techniques come with a notable overhead for developers when implementing ASP-dependent end-user applications. In addition to developing the ASP program and defining the data which is exchanged, a developer needs to implement the graphical user interface, including the input- and output-handling, on his own. This means that a developer must be able to employ not only the ASP language, but also an imperative programming language additionally. Furthermore, such applications are reusable only to a certain point and therefore these approaches lead to repetitive work when implementing multiple applications.

To overcome the stated issues and further reduce the overhead for developers, this thesis proposes a new approach, named *aspguid*, for implementing end-user applications based on annotated ASP programs. The proposed approach consists of a declarative language for defining graphical user interfaces for ASP programs and a compiler that translates ASP programs which are annotated with such definitions into procedural code, that realizes respective graphical user interface assisted applications.

The *aspguid* language, which enables the annotation of ASP programs with graphical user interface definitions, allows for the specification of versatile graphical user interfaces by providing general user interface elements along with different types of input-elements and output-elements. Furthermore, the language is designed in such a way that annotated ASP programs are still usable as ordinary ASP programs and that existing ASP programs can easily be extended with graphical user interface definitions.

The further presented *aspguid* compiler allows to translate an annotated ASP program into an executable Java application, which provides a respective graphical user interface that allows for the input of data in an intuitive and assisted way and displays the output data in a structured and formatted manner. Moreover, the compiler allows to generate the Java source code of such an application, which can be adjusted and extended to special requirements not supported by the *aspguid* language yet or which can be used as initial project setup for the implementation of more sophisticated features.

In total, the proposed `aspguid` approach provides a straightforward solution for implementing applications usable by users without prior logic programming experience based on annotated ASP programs. The approach significantly reduces the development overhead and minimizes the repetitive workload for developers, compared to the current state of the art solution of embedding ASP programs into manually developed imperative graphical user interfaces.

The remainder of this thesis is structured as follows:

- **Chapter 2** briefly recapitulates the syntax and the semantics of the ASP language and presents two example ASP programs.
- **Chapter 3** examines the structure of ASP encodings and analyzes the input and output of ASP programs.
- **Chapter 4** proposes and explains the `aspguid` language, which enables the annotation of ASP programs with graphical user interface definitions.
- **Chapter 5** introduces and analyzes the `aspguid` compiler, which allows to translate annotated ASP programs into graphical user interface assisted applications.
- **Chapter 6** presents and assesses a complete showcase application, which was implemented in the ASP language, annotated using the `aspguid` language and compiled with the `aspguid` compiler.
- **Chapter 7** summarizes the proposed techniques, discusses related work and outlines possible further work in this area.
- **Appendix A** provides a complete language specification for the proposed `aspguid` language.
- **Appendix B** presents the complete annotated ASP program of the showcase application and provides an example problem instance for the application.

Answer Set Programming

Answer Set Programming (ASP) is a fully declarative programming paradigm, proposed in the area of non-monotonic reasoning and logic-programming [23]. This chapter briefly recapitulates the basics of the ASP paradigm. For a more complete and formal in-depth insight into ASP, we refer to other literature [1, 23, 10].

2.1 Syntax

A *term* t is a variable or a constant, wherein variables start with an uppercase letter and constants start with a lowercase letter. An *atom* is an expression $p(t_1, \dots, t_n)$, where p is a *predicate* of arity n and t_1, \dots, t_n are terms. An atom is *ground*, if it do not contain any variables. A *literal* l is either a positive literal p or a negative literal $\text{not } p$, where p is an atom. A literal is *ground*, if the atom of the literal is ground.

A (*disjunctive*) *rule* r is of the form:

$$a_1 \vee \dots \vee a_n \leftarrow b_1, \dots, b_k, \text{not } b_{k+1}, \dots, \text{not } b_m$$

where $a_1, \dots, a_n, b_1, \dots, b_m$ are atoms.

The disjunction $a_1 \vee \dots \vee a_n$ is called *head* of the rule r . The conjunction $b_1, \dots, b_k, \text{not } b_{k+1}, \dots, \text{not } b_m$ is named *body* of the rule r , wherein b_1, \dots, b_k is named positive body and $\text{not } b_{k+1}, \dots, \text{not } b_m$ is called negative body.

If the body of a rule is empty, the rule is called a (possibly disjunctive) *fact* and the \leftarrow sign is usually omitted. If the head of a rule is empty, the rule is called a *constraint*.

A rule r is *ground*, if all all atoms are ground. An *ASP program* P is a finite set of rules.

A rule is *safe*, if each variable appearing in the rule also appears in at least one literal in the positive body of the rule r . An ASP program P is safe, if each of its rules is safe. From this point, we will only consider safe ASP programs.

2.2 Semantics

Herbrand Universe and Herbrand Base: Let P be an ASP program. The *Herbrand Universe*, denoted by U_P , is the set of all constants appearing in P . The *Herbrand Base*, denoted as B_P , is the set of all possible ground atoms, which can be constructed from the predicates appearing in P with the constants of U_P .

Ground Instantiation: Let r be a rule of an ASP program P . $Ground(r)$ denotes the set of rules which can be obtained by replacing each variable in r by constants in U_P in all possible ways. The ground instantiation of the program P is defined by the set $grnd(P) = \bigcup_{r \in P} Ground(r)$.

Interpretation and Model: An interpretation of an program P is a subset I of B_P . A ground positive literal a is true (resp., false) w.r.t. I if $a \in I$ (resp., $a \notin I$). A ground negative literal $not\ a$ is true (resp., false) w.r.t. I if $a \notin I$ (resp., $a \in I$).

A ground rule r is satisfied by an interpretation I , if at least one literal in the head of the rule is true w.r.t. I , or if at least one literal in the body of the rule is false w.r.t. I . An interpretation I is a *model* of an program P , if I satisfies all rules of the ground instantiation $grnd(P)$ of P .

Gelfond-Lifschitz Reduct and Answer Set: Given a ground program P and an interpretation I , the Gelfond-Lifschitz reduct [16] of P w.r.t. I is the set P^I which is obtained from P by (i) deleting all rules in which at least one literal in the negative body of the rule is false w.r.t. I , (ii) deleting the negative body from the remaining rules.

A model I of P is an *answer set* (or stable model), if I is a minimal model (under subset inclusion) of P^I . For a general (possible non-ground) ASP program P , an interpretation I is an answer set, if it is an answer set of $grnd(P)$.

2.3 Examples

This section presents two basic computational problems and their respective ASP programs. These examples illustrate how the ASP language can be employed on concrete problems and are used for illustration purposes in later chapters.

We will use the ASP-Core-2 format [7], wherein the characters $:-$ represent the \leftarrow symbol, for presenting ASP source code from this point. Furthermore, the \hookrightarrow symbol will be used in source code listings to indicate line breaks caused by insufficient listing widths.

2.3.1 Graph Coloring

The 3-coloring problem is a well known NP-complete problem from the area of graph theory. Given an undirected graph which consists of an arbitrary number of nodes and edges, each node must be assigned to one of three colors. The colors have to be assigned in such a way, that every pair of adjacent nodes is assigned to distinct colors.


```

% nodes and edges of the graph                                1
node(a). node(b). node(c). node(d).                          2
edge(a,b). edge(a,c). edge(a,d). edge(b,c). edge(c,d).       3
                                                                4
% each node is assigned to a color                             5
node_color(N,red) v node_color(N,green) v node_color(N,blue) :- 6
    ⇐ node(N).
% two adjacent nodes cannot be assigned to the same color      7
:- node_color(X,C), node_color(Y,C), edge(X,Y).              8

```

Listing 2.1: ASP program for the 3-coloring problem

Listing 2.1 presents an ASP program for the 3-coloring problem, wherein the first three lines define the graph which is processed. The rule stated in line 6 assigns every node to a color and the constraint stated in line 8 prevents that adjacent nodes are assigned to the same color.

2.3.2 Graph Reachability

In graph theory, reachability refers to the ability to get from one node of a graph to another node by traversing one or multiple edges of the graph. A node n_1 is reachable from a node n_0 , if there exists a sequence of adjacent nodes, which starts with the node n_0 and ends with the node n_1 .

```

% nodes and edges of the graph                                1
node(a). node(b). node(c). node(d).                          2
edge(a,c). edge(c,d). edge(b,c).                             3
% starting node for the reachability problem                  4
start_node(a).                                                5
                                                                6
% every node which is connected to the starting node is reachable 7
reachable(X) :- start_node(Y), edge(Y,X).                     8
% every node which is connected to a reachable node is reachable 9
reachable(X) :- reachable(Y), edge(Y,X).                      10
                                                                11
count_reachable(C) :- #count{N : reachable(N)} = C.          12
all_reachable :- count_reachable(C), #count{N : node(N)} = C. 13

```

Listing 2.2: ASP program for the graph reachability problem

Listing 2.2 presents an ASP program for the graph reachability problem, wherein the first four lines define the graph which is processed and the start node, for which the set of reachable nodes is calculated. Reachable nodes for the start node are determined by the rules from line 7 to line 10. The last two lines of the program utilize aggregate functions to extract information about the number of reachable nodes.

Structure of ASP Encodings

ASP programs consist of a set of proper rules and facts. Since the head of (non-disjunctive) facts inside an ASP program must be true in every answer set, from a knowledge engineering perspective, the set of facts inside an ASP program can be seen as explicit knowledge. From this perspective, the remaining rules inside the program can be seen as implicit knowledge, which defines how new knowledge is derived from existing knowledge.

3.1 Uniform Problem Encoding

It is common to separate the general problem specification from the specific problem instance in an ASP program [18, 19]. This allows to use the general problem specification for solving arbitrary instances of the encoded problem. In this setting, a problem instance is encoded as explicit knowledge and the general problem specification is mainly represented as implicit knowledge.

In fact, both ASP programs presented in Section 2.3 are implemented as uniform problem encodings. For example, the general problem specification of the 3-coloring problem is encoded from line 5 to line 8 in the presented program. On the other hand, the specific problem instance is encoded from line 1 to line 3.

```
% nodes and edges of the graph                                1
node(a). node(b). node(c). node(d).                            2
edge(a,b). edge(a,c). edge(a,d). edge(b,c). edge(c,d).        3
```

Listing 3.1: Specific problem instance inside the 3-coloring program

```

% each node is assigned to a color                                5
node_color(N,red) v node_color(N,green) v node_color(N,blue) :- 6
    ⇔ node(N) .
% two adjacent nodes cannot be assigned to the same color        7
:- node_color(X,C) , node_color(Y,C) , edge(X,Y) .              8

```

Listing 3.2: General problem specification inside the 3-coloring program

3.2 Input of an ASP Application

When implementing an end-user application, it is necessary that the application can be used for varying input problem instances. Therefore it can be assumed that the ASP program for an end-user application is implemented as uniform problem encoding, wherein the problem instance represents the input of the application.

Usually, problem instances contain only disjunction-free facts. Also, as we are considering safe programs only, each fact must be variable-free. Therefore, we can define the input of an ASP program for an end-user application as a **set of variable-free atoms**.

3.2.1 Entity Perspective

According to Section 2.1, a variable-free atom is an expression $p(t_1, \dots, t_n)$, where p is a *predicate* of arity n and t_1, \dots, t_n are constants. From another perspective, the variable-free atom $p(t_1, \dots, t_n)$ can be seen as entity of the *type* p with the attribute values t_1, \dots, t_n . From this perspective, which we will refer to as entity perspective from this point, the input of an ASP program for an end-user application is defined as set of entities.

As a single predicate has a particular arity in a fixed ASP program, an entity of a specific type has a fixed number of attributes from the entity perspective. Thus, we can define a meaning for each attribute of a specific type.

```

% nodes and edges of the graph                                    1
node(a) . node(b) . node(c) . node(d) .                          2
edge(a,c) . edge(c,d) . edge(b,c) .                              3
% starting node for the reachability problem                     4
start_node(a) .                                                  5
                                                                6

```

Listing 3.3: Problem instance for the graph reachability problem

Listing 3.3 states a problem instance for the graph reachability problem. From the entity perspective, this problem instance consists of a set of entities of the types `node`, `edge` and `start_node`.

The `node` type has one attribute, which is the identifier of a node. The `edge` type consists of two attributes, which specify the start node and the end node of an edge. The `start_node` type has one attribute, which is the identifier of a node.

3.2.2 Input Characteristics

Typical problem instances for specific ASP programs must fulfill certain properties and therefore are restricted in specific ways. This section lists common restrictions for useful problem instances for ASP programs.

Restricted Entity Types: Typical problem instances for a specific ASP program contains only entities of certain types. Considering the graph reachability program, a useful problem instance contains only entities of the types `node`, `edge` and `start_node`.

Restricted Number of Entities: An ASP program may restrict the number of entities of particular types. Considering the graph reachability program, a useful problem instance contains at least one entity of the type `node` and exactly one entity of the type `start_node`.

Restricted Attribute Values: An ASP program may restrict the set of valid values for specific attributes of entities of particular types for useful problem instances. Considering the graph reachability program, the attribute of an `start_node` entity must refer to a value of the attribute of a `node` entity in a useful problem instance.

3.3 Output of an ASP Application

The result of an ASP program is defined as a set of answer sets, wherein a single answer set is a set of ground atoms. When implementing an end-user application, it is desirable to extract a subset from each answer set, which we will refer to as *solution set* and which contains only the atoms that represent a solution to the encoded problem. Thus, we can define the output of an ASP program for an end-user application as a **set of solution sets**, wherein each solution set is a **set of ground atoms**.

3.3.1 Entity Representation

From the entity perspective, the output of an ASP program of an end-user application is defined as set of solution sets, wherein each solution set is a set of entities and represents a single solution for the encoded problem.

```

{                                                                    1
node(a), node(b), node(c), node(d),                                2
edge(a,b), edge(a,c), edge(a,d), edge(b,c), edge(c,d),            3
node_color(a,green), node_color(b,red), node_color(c,blue),        4
  ⇔ node_color(d,red)
}                                                                    5
{                                                                    6
node(a), node(b), node(c), node(d),                                7
...                                                                8

```

Listing 3.4: Excerpt from the answer sets of the 3-coloring program

```
{
node_color(a,green), node_color(b,red), node_color(c,blue),
  ⇔ node_color(d,red)
}
{
node_color(a,red), node_color(b,blue), node_color(c,green),
  ⇔ node_color(d,blue)
...

```

1
2
3
4
5
6

Listing 3.5: Excerpt from the solution sets of the 3-coloring program

Listing 3.4 presents an excerpt from the answers sets of the 3-coloring program and Listing 3.5 points out an excerpt from the respective solution sets. A single solution set of the 3-coloring program consists of a set of entities of the type `node_color`.

Each `node_color` entity represents a color-assignment to a node and consists of two attributes, wherein the first attribute is the identifier of a node and the second attribute is the assigned color for this node.

aspguid Language

This chapter presents the aspguid language as an extension of the ASP language, which allows to annotate ASP programs with graphical user interface (GUI) definitions in a declarative manner. It was designed minding the characteristics of ASP encodings of end-user applications analyzed in Chapter 3.

When using the aspguid language, the GUI of an ASP program is specified by defining a `GUI-definition` object in a multi-line comment inside the program. Through this, annotated ASP programs are still usable as ordinary ASP programs and existing programs can easily be extended with GUI definitions.

The `GUI-definition` object of the aspguid language uses the JSON format and contains *object-properties*, *element-properties* and *value-properties*. *Object-properties* and *value-properties* consist of a defined name and a value of a respective type. *Element-properties* are used to declare GUI components and consist of an element-type, an identifier and a value of a type respective to the element-type. An identifier of an *element-property* starts with a lowercase letter and contains only alphanumeric characters and underscores. An element-type of an *element-property* starts with an @ symbol.

Furthermore, the aspguid language uses three custom string formats as types for *value-properties*. The `REPRESENTATION-TEMPLATE` type is used to define how the data of GUI components is displayed in the GUI and represented in an ASP program. The `SOURCE-SELECTOR` type defines a set of valid values which can be entered via a GUI component and the `CONDITION-STATEMENT` type is used to control the visibility of GUI components. The custom *value-property* types of the aspguid language are explained in detail in Section 4.5.

The remainder of this chapter presents the structure, the most important properties and the custom *value-property* types of the aspguid language. A full language specification, including all available properties of the `GUI-definition` object of the aspguid language, is presented in Appendix A.

4.1 GUI-Definition Object

The GUI of an ASP program is specified by annotating the ASP program with a GUI-definition object in a custom multi-line comment. An excerpt from an annotated ASP program is presented in Listing 4.1. The custom multi-line comment must not contain any other data or text beside the GUI-definition object.

```
%*::: 1
{ 2
  "program_information": {...}, 3
  "program_input": {...}, 4
  "program_output": {...} 5
} 6
:*% 7
8
% nodes and edges of the graph 9
node(a). node(b). node(c). node(d). 10
edge(a,b). edge(a,c). edge(a,d). edge(b,c). edge(c,d). 11
... 12
```

Listing 4.1: GUI-definition object placed inside a custom multi-line comment

The GUI-definition object of the aspguid language consists of three basic object-properties. The *program_information* object-property specifies general information about the annotated program. The *program_input* object-property declares the input-components of the GUI of the program and the *program_output* object-property declares the output-components of the GUI of the program.

4.2 program_information Object

The *program_information* object can contain four value-properties which specify general information about the annotated program. The information specified in this object is supposed to be displayed in the GUI of the program.

```
"program_information": { 1
  "name": "Graph 3-Coloring", 2
  "description": "3-Coloring is a well known NP-complete 3
    ⇨ problem from the area of graph theory. Given an
    ⇨ undirected graph containing an arbitrary number of nodes
    ⇨ and edges, each node must be assigned to one of...",
  "author": "Niklas Natter", 4
  "version": "0.1" 5
} 6
```

Listing 4.2: program_information object for the 3-coloring program

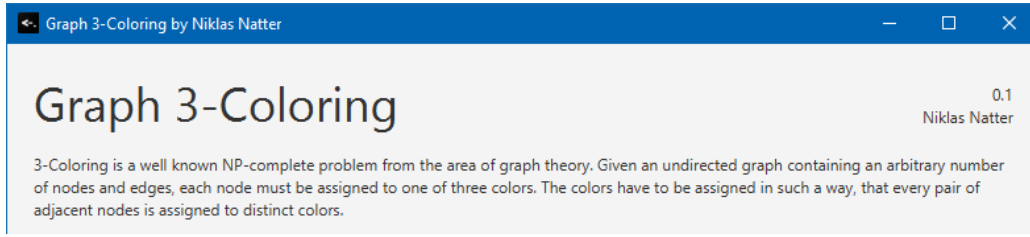


Figure 4.1: `program_information` GUI component for the 3-coloring program

Listing 4.2 presents a `program_information` object for the 3-coloring program. The value of the `name` value-property is displayed in the window title and in the GUI of the program. The values of the optional value-properties `description`, `author` and `version` are displayed in the GUI of the program as shown in Figure 4.1.

4.3 program_input Object

The `program_input` object contains an arbitrary number of element-properties, which declare the input-components of the GUI of the program. The `aspguid` language currently includes two element-types which can be used inside the `program_input` object. The `@entity_input` type enables the input of a set of entities of a particular entity-type and the `@value_input` type facilitates the input of an arbitrary number of values. We will refer to element-properties of these element-types as input-elements from this point.

```
"program_input": {                                1
  "@entity_input:node": {...},                    2
  "@entity_input:edge": {...},                    3
  "@value_input:start_node": {...}                 4
}                                                    5
```

Listing 4.3: Abbreviated `program_input` object for the reachability program

Listing 4.3 presents an abbreviated `program_input` object for the reachability program. The object contains two `@entity_input` input-elements which facilitate the input of the nodes and the edges of a graph. Furthermore, the presented `program_input` object contains a `@value_input` input-element that enables the input of the start node of the reachability problem.

In general, an input-element is defined in the `program_input` object by a property, which contains the type and the identifier of the input-element in the property-name and holds the input-element object as value. For example, line 2 of Listing 4.3 states an input-element of the type `@entity_input` with the identifier `node`.

The identifier of an input-element must be unique among all declared input-elements. The order of the input-components in the GUI of the annotated program is defined by the order of the input-elements inside the `program_input` object.

4.3.1 @entity_input Element-Type

```

"@entity_input:edge": {
  "title": "Edges",
  "description": "Edges of the graph. An edge connects a
    ↔ start-node to an end-node.",
  "gui_representation": "::from -> ::to",
  "atom_representation": "edge(::from,::to)",
  "@input_attribute:from": {
    "name": "from",
    "description": "Identifier of the start-node",
    "value_source": "$node[0]"
  },
  "@input_attribute:to": {
    "name": "to",
    "description": "Identifier of the end-node",
    "value_source": "$node[0]"
  }
}

```

Listing 4.4: @entity_input input-element for the input of edges of a graph

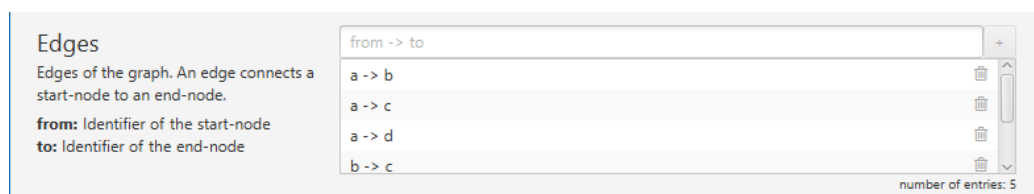


Figure 4.2: @entity_input input-component for the input of edges of a graph

Listing 4.4 presents an @entity_input element that declares an input-component for a set of entities, which represent the edges of a graph. Edge entities consist of two attributes, which specify the start node and the end node of an edge and which are defined by the @input_attribute element-properties from line 6 to line 15. The *gui_representation* value-property defines how an edge is displayed in the GUI and the *atom_representation* value-property specifies how an edge is represented as an atom. Furthermore, the *value_source* value-property of the @input_attribute elements define the set of valid values for start nodes and end nodes of edge entities.

The values of the *title* value-property and the *description* value-property of the presented @entity_input element, along with the values of the *name* value-property and the *description* value-property of the @input_attribute elements, are used to explain the edge input-component to the user of the application and are displayed in the GUI of the program as shown in Figure 4.2.

In general, an element of the type @entity_input declares an input-component, which enables the input of a set of entities of a particular entity-type. The attributes of this entity-type are defined by an arbitrary number of element-properties of the element-type @input_attribute, which we will refer to as attribute-elements in this section.

The *gui_representation* value-property of an `@entity_input` input-element defines, in which format an entity can be entered by a user and in which format a single entity is displayed in the GUI. The *atom_representation* value-property of an `@entity_input` input-element defines how a single entity is represented as an atom. The values of these properties are of the type `REPRESENTATION-TEMPLATE` and can contain placeholders for the attribute-elements of the `@entity_input` input-element.

The *input_count_min* value-property and the *input_count_max* value-property of an `@entity_input` input-element define the number of entities, which can be entered via the input-component. This allows to restrict the number of entities of a particular type (see Section 3.2.2).

@input_attribute Element-Type

An `@input_attribute` element defines an attribute of the entity-type of an input-component declared by an `@entity_input` input-element.

The set of valid values for such an attribute can be restricted by the *value_source* value-property of the `@input_attribute` element. The value of this property is of the type `SOURCE-SELECTOR`.

4.3.2 @value_input Element-Type

```

"@value_input:start_node": {                                1
  "title": "Start Node",                                    2
  "gui_representation": "Identifier of the node, of which the set 3
  ↪ of reachable nodes is computed: ::id",
  "@input_value:id": {                                      4
    "value_source": "$node[0]",                             5
    "atom_representation": "start_node(::id)"               6
  }                                                          7
}                                                            8

```

Listing 4.5: `@value_input` input-element for the reachability program

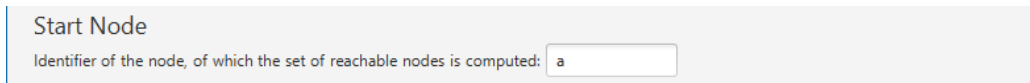


Figure 4.3: `@value_input` input-component for the reachability program

Listing 4.5 presents a `@value_input` element that declares an input-component for one value, which represents the start node of the reachability problem. The start node value is defined by the `@input_value` element-property from line 4 to line 7. The *atom_representation* value-property of the `@input_value` element specifies how the start node is represented as an atom and the *value_source* value-property defines the set of valid start node values.

The values of the *title* value-property and the *gui_representation* value-property of the presented `@value_input` element are used to explain the meaning of the start node input-component to the user of the application and are displayed in the GUI of the program as shown in Figure 4.3.

In general, an element of the type `@value_input` declares an input-component, which enables the input of an arbitrary number of values. The values which can be entered via the input-component are defined by an arbitrary number of element-properties of the element-type *@input_value*. We will refer to element-properties of this type as value-elements in this section.

The *gui_representation* value-property of an `@value_input` input-element defines the content of the input-component in the GUI. The value of this property is of the type REPRESENTATION-TEMPLATE and can contain placeholders for the value-elements of the `@value_input` input-element.

@input_value Element-Type

An `@input_value` element defines a value which can be entered via an input-component declared by a *@value_input* input-element. Formally, a value is an entity of a particular type that consists of exactly one attribute. An `@input_value` element allows to input exactly one entity of this type via the respective input-component and therefore restricts the number of entities for this type (see Section 3.2.2).

The *atom_representation* value-property of an `@input_value` element defines how the value is represented as an atom. The value of this property is of the type REPRESENTATION-TEMPLATE and can contain a placeholder for the `@input_value` element.

The set of valid values for an `@input_value` element can be restricted with the *value_source* value-property. The value of this property is of the type SOURCE-SELECTOR.

4.4 program_output Object

The `program_output` object contains an arbitrary number of element-properties, which declare the output-components of the GUI of the program. The `aspguid` language currently includes two element-property types which can be used inside the `program_output` object. The *@entity_output* type enables the output of a set of entities of a particular entity-type and the *@value_output* type facilitates the output of an arbitrary number of values. We will refer to element-properties of these types as output-elements from this point.

```
"program_output": {                                1
  "@value_output:count_reachable": {...},          2
  "@entity_output:reachable_nodes": {...}          3
}                                                    4
```

Listing 4.6: Abbreviated `program_output` object for the reachability program

Listing 4.6 presents an abbreviated `program_output` object for the reachability program. The object contains a `@value_output` output-element which facilitates the output of the number of reachable nodes and an `@entity_output` output-element that enables the output of the set of reachable nodes.

In general, an output-element is defined inside the `program_output` object by a property, which contains the type and the identifier of the output-element in the property-name and holds the output-element object as value. For example, line 2 of Listing 4.6 states an output-element of the type `@value_output` with the identifier `count_reachable`.

The identifier of an output-element must be unique among all declared output-elements. The order of the output-components in the GUI of the annotated program is defined by the order of the output-elements inside the `program_output` object.

4.4.1 @entity_output Element-Type

```

"@entity_output:color_assignment": {                                1
  "title": "Color Assignments",                                     2
  "description": "The computed color assignments according to the  3
    ↪ 3-coloring problem.",
  "atom_representation": "node_color(::node_id, ::color)",         4
  "gui_representation": "::node_id is assigned to ::color",       5
  "@output_attribute:node_id": {},                                  6
  "@output_attribute:color": {}                                    7
}                                                                    8

```

Listing 4.7: @entity_output output-element for the 3-coloring program



Figure 4.4: @entity_output output-component for the 3-coloring program

Listing 4.7 presents an `@entity_output` element that declares an output-component for a set of entities, which represent the color assignments of the 3-coloring problem. Assignment entities consist of two attributes, which specify the node identifier and the assigned color of a color assignment and which are defined by the `@output_attribute` element-properties on line 6 and line 7. The `gui_representation` value-property defines how a color assignment is displayed in the GUI and the `atom_representation` value-property specifies how a color assignment is represented in an answer set.

The values of the `title` value-property and the `description` value-property of the presented `@entity_output` element are used to explain the color assignment output-component to the user of the application and are displayed in the GUI of the program as shown in Figure 4.4.

In general, an element of the type `@entity_output` declares an output-component, which enables the output of a set of entities of a particular entity-type. The attributes of this entity-type are defined by an arbitrary number of element-properties of the element-type `@output_attribute`, which we will refer to as attribute-elements in this section.

The *atom_representation* value-property of an `@entity_output` output-element defines how a single entity is represented as an atom in an answer set. The *gui_representation* value-property of an `@entity_output` output-element defines, in which format a single entity is displayed in the GUI. The values of these properties are of the type REPRESENTATION-TEMPLATE and can contain placeholders for the attribute-elements of the `@entity_output` output-element.

The *output_condition* value-property of an `@entity_output` output-element can be used to control the visibility of the respective output-component in the GUI. The value of this property is of the type CONDITION-STATEMENT.

@output_attribute Element-Type

An `@output_attribute` element defines an attribute of the entity-type of an output-component declared by an `@entity_output` output-element.

4.4.2 @value_output Element-Type

```

"@value_output:count_reachable": {                                1
  "title": "Number of reachable nodes",                            2
  "gui_representation": "There are ::count nodes reachable from  3
    ↔ the start node.",
  "output_condition": "?display_count",                            4
  "@input_value:count": {                                         5
    "atom_representation": "count_reachable(::count)."           6
  }                                                                7
}                                                                    8

```

Listing 4.8: `@value_output` output-element for the reachability program

Figure 4.5: `@value_output` output-component for the reachability program

Listing 4.8 presents a `@value_output` element that declares an output-component for one value, which represents the number of reachable nodes of the reachability problem. The value is defined by the `@output_value` element-property from line 5 to line 7.

The *atom_representation* value-property of the `@output_value` element specifies how the number of reachable nodes is represented in an answer set. Furthermore, the *output_condition* value-property of the `@value_output` element defines that the output-component is only visible in the GUI, if the displayed answer set contains the atom *display_count*.

The values of the *title* value-property and the *gui_representation* value-property of the presented `@value_output` element are used to explain the meaning of the number of reachable nodes output-component to the user of the application and are displayed in the GUI of the program as shown in Figure 4.5.

In general, an element of the type `@value_output` declares an output-component, which enables the output of an arbitrary number of values. The values which are output via this output-component are defined by an arbitrary number of element-properties of the element-type `@output_value`. We will refer to element-properties of this type as value-elements in this section.

The *gui_representation* value-property of a `@value_output` output-element defines the content of the output-component in the GUI. The value of this property is of the type REPRESENTATION-TEMPLATE and can contain placeholders for the value-elements of the `@value_output` output-element.

The *output_condition* value-property of an `@value_output` output-element can be used to control the visibility of the respective output-component in the GUI. The value of this property is of the type CONDITION-STATEMENT.

@output_value Element-Type

An `@output_value` defines a value which is output via the output-component declared by a `@value_output` output-element. Formally, a value is an entity of a particular type that consists of exactly one attribute.

The *atom_representation* value-property of an `@output_value` element defines how the value is represented as an atom in an answer set. If the value of this property corresponds to more than one atom in an answer set, the alphabetically first atom is used. The value of the *atom_representation* value-property is of the type REPRESENTATION-TEMPLATE and can contain a placeholder for the `@output_value` element.

4.5 Value-Property Types

The aspguid language uses three custom string formats as types for value-properties of the GUI-definition object, namely the REPRESENTATION-TEMPLATE type, the SOURCE-SELECTOR type and the CONDITION-STATEMENT type. This section explains the purpose and the format of these types.

4.5.1 Representation Template Type

The REPRESENTATION-TEMPLATE type is used for two purposes in the aspguid language. On the one hand, *gui_representation* properties use the REPRESENTATION-TEMPLATE type to define in which format the data of input-components and output-components is displayed in the GUI of an annotated program. On the other hand, *atom_representation* properties use the REPRESENTATION-TEMPLATE type to define how the data of input-components and output-components is represented in an ASP program or in an answer set of an ASP program.

Values of the REPRESENTATION-TEMPLATE type are strings which can contain placeholders referring to element-properties of the GUI-definition object. A placeholder inside a string of the type REPRESENTATION-TEMPLATE consist of the identifier of the respective element-property prepended with two colon symbols `::`.

```
"@entity_output:color_assignment": {                               1
  "atom_representation": "node_color(::node_id,::color)",         2
  "gui_representation": "::node_id is assigned to ::color",       3
  "@output_attribute:node_id": {...},                             4
  "@output_attribute:color": {...},                               5
  ...                                                             6
}                                                                    7
```

Listing 4.9: REPRESENTATION-TEMPLATE type inside an @entity_output element

Listing 4.9 presents an example, wherein the *atom_representation* property of the @entity_output output-element declares, that the values of the *node_id* attribute are extracted from the first argument of the *node_color*-atoms in an answer set. Considering an answer set containing the atom *node_color*(n_1 ,red), the *gui_representation* property of the @entity_output output-element defines, that this atom leads to the output of *n_1 is assigned to red* in the respective output-component.

```
"@value_input:start_node": {                                       1
  "gui_representation": "Identifier of the node, of which the set  2
  ↪ of reachable nodes is computed: ::node_id",
  "@input_value:node_id": {                                       3
    "atom_representation": "start_node(::node_id)"               4
  },                                                             5
  ...                                                             6
}                                                                    7
```

Listing 4.10: REPRESENTATION-TEMPLATE type inside a @value_input element

Listing 4.10 presents an example, wherein the *gui_representation* property of the @value_input input-element defines the content of the respective input-component. In this case, the input-component declared by the @value_input input-element replaces the placeholder `::node_id` with an editable input field.

Considering the input of the value *alpha* into this input field by the user, the *atom_representation* property of the @input_value input-element defines, that this input value is represented as the atom *start_node(alpha)*.

Default Representation Template

If the *gui_representation* property or the *atom_representation* property of an input-element or output-element is not specified, the aspguid language uses a default representation template as fallback value.

The default representation template for an input-element or an output-element contains the identifier of the element and includes all possible placeholders for the element. The order of the placeholders in the default representation template is defined by the order of the respective elements in the GUI-definition object.

An @entity_input (@entity_output) element with the identifier *edge*, which contains two @input_attribute (@output_attribute) elements with the identifiers *from* and *to*, has the default representation template *edge(::from,::to)*.

A @value_input (@value_output) element with the identifier *start_node*, which contains one @input_value (@output_value) element with the identifier *node_id*, has the default representation template *start_node(::node_id)*.

An @input_value (@output_value) element with the identifier *node_id*, has the default representation template *node_id(::node_id)*.

It can be seen that the format of the default representation template is very similar to the values of the *atom_representation* properties presented in previous examples. In fact, the usage of default representation templates with appropriate element identifiers allows to omit explicitly stated *atom_representation* properties in most cases.

4.5.2 Source Selector Type

The SOURCE-SELECTOR type is used to define a value source which contains a set of values. The *value_source* property of an @input_argument element or an @input_value element uses the SOURCE-SELECTOR type for defining the set of valid values for the element. This allows to restrict the set of valid values for an attribute of an entity (see Section 3.2.2) and enables to validate dependencies between entities of different types in the GUI of an annotated program.

Values of the SOURCE-SELECTOR type are strings that consist of a *value source definition* prepended with a \$ symbol. The aspguid language currently supports two types of value source definitions: the ATOM-ARGUMENT VALUE SOURCE type and the TYPE VALUE SOURCE type.

Atom-Argument Value Source

An ATOM-ARGUMENT VALUE SOURCE definition consist of a *predicate name* and an *argument index*, which is stated within square brackets. For example, the string *\$node[0]* refers to an ATOM-ARGUMENT VALUE SOURCE with the predicate name *node* and the argument index *0*.

An ATOM-ARGUMENT VALUE SOURCE contains all values, which are stated as argument on the respective argument-index in an atom with the respective predicate name. An ATOM-ARGUMENT VALUE SOURCE extracts matching values from atoms which are stated in the ASP program and atoms which are generated by the *atom_representation* property of input-elements.

```
"@input_value:node_id": {                                1
  "value_source": "$node[0]",                             2
  ...                                                     3
}                                                         4
```

Listing 4.11: SOURCE-SELECTOR type inside a @value_input element

```
node(a). node(b). node(c). node(d).                    1
edge(a,c). edge(c,d). edge(b,c).                       2
...                                                     3
```

Listing 4.12: Excerpt from an ASP program

Listing 4.11 presents an example, wherein the *value_source* property of the @input_value input-element defines the set of valid values for the element with an ATOM-ARGUMENT VALUE SOURCE. Considering the excerpt from an ASP program stated in Listing 4.11, the set of valid values for the @input_value input-element would contain the values *a*, *b*, *c* and *d*.

```
"@entity_input:node": {                                1
  "@input_argument:name": {...},                         2
  "atom_representation": "node(:name)."                  3
  ...                                                     4
}                                                         5
```

Listing 4.13: Excerpt from a GUI-definition object

Considering a GUI-definition object contains the @input_value input-element presented in Listing 4.11 and the @entity_input input-element presented in Listing 4.13. In this setting, the set of valid values for the @input_value element would contain all node-names which were entered via the input-component declared by the @entity_input element.

Type Value Source

A TYPE VALUE SOURCE definition consist of a *type-name* prepended with a # symbol. For example, the string *\$#int* refers to a TYPE VALUE SOURCE with the type-name *int*.

A TYPE VALUE SOURCE contains all values of the respective type. The aspguid language currently supports only the type-name *int*, but further types, such as alphanumeric strings defined by regular expressions, can be easily added. A TYPE VALUE SOURCE with the type-name *int* contains all integer values.

```

"@input_value:max_steps": {                                1
  "value_source": "$#int",                                  2
  ...                                                         3
}                                                             4

```

Listing 4.14: SOURCE-SELECTOR type inside a @value_input element

Listing 4.14 presents an example, wherein the *value_source* property of the @input_value input-element defines the set of valid values for the element with an TYPE VALUE SOURCE. In this setting, the set of valid values for the @input_value element would contain all integer values.

4.5.3 Condition Statement Type

The CONDITION-STATEMENT type is used to define a condition based on the presence of a particular atom in an answer set. The *output_condition* property of an output-element uses the CONDITION-STATEMENT type for defining a condition for the visibility of the respective output-component. When specifying the *output_condition* property of an output-element, the the output-component declared by the output-element is only visible in the GUI of the annotated program, if the condition is fulfilled.

Values of the CONDITION-STATEMENT type are strings that consist of a ground atom prepended with a ? symbol. A condition is fulfilled for an answer set, if the respective atom is contained in the answer set.

```

"@value_output:count_reachable": {                          1
  "title": "Number of reachable nodes",                      2
  "output_condition": "?display_count",                      3
  ...                                                         4
}                                                             5

```

Listing 4.15: CONDITION-STATEMENT type inside a @value_output element

Listing 4.15 presents an example, wherein the *output_condition* property of the @value_output output-element specifies, that the output-component of the output-element is displayed in the GUI of the annotated program only, if the currently displayed answer set contains the atom *display_count*.

Implementation of the Compiler

This chapter introduces the `aspguid` compiler for the `aspguid` language presented in Chapter 4. In the following, the functionality and the basic architecture of the `aspguid` compiler is described. Furthermore, the functionality and the architecture of applications compiled with the `aspguid` compiler is explained.

5.1 `aspguid` Compiler

The `aspguid` compiler implements the `aspguid` language and allows to **generate** the Java application source code for an annotated ASP program, to **compile** an annotated ASP program into an executable `.jar` application and to directly **execute** the application for an annotated ASP program for testing purposes.

The source code of the `aspguid` compiler and an executable `.jar` version of the `aspguid` compiler is available on GitHub¹. The compiler requires an installed JDK² of version 8 or above. When using OpenJDK³, additionally the OpenJFX⁴ package is required. The `.jar` version of the `aspguid` compiler is executed with the command:

```
java -jar aspguidc.jar {-g|-c|-e} input_file
```

The `input_file` argument is a path to an annotated ASP program. The option `-g` stand for *generate*, the option `-c` stands for *compile* and the option `-e` stands for *execute*. Generated files are stored in the parent directory of the annotated ASP program.

¹<https://github.com/nnatter/aspguid-compiler>

²<http://www.oracle.com/technetwork/java/javase/downloads/index.html>

³<http://openjdk.java.net/>

⁴<http://openjdk.java.net/projects/openjfx/>

When compiling an annotated ASP program into an executable .jar application, the resulting .jar file represents a standalone application which can be executed on any system with an installed JRE⁵. An application compiled by the `aspguid` compiler provides a GUI respective to the GUI definition of an annotated ASP program and allows to evaluate the underlying ASP program via this GUI. A more detailed description of the functionality and the architecture of such an application is presented in Section 5.2.

When generating the Java application source code for an annotated ASP program, the source code of the executable application is generated into a new directory. The generated Java source code conforms with the standardized *Maven*⁶ *directory layout* and provides an *Ant*⁷ *build file*, which allows to compile the source code into an executable .jar application or to execute the respective application directly.

The ability to generate the Java application source code allows to adjust an application compiled by the `aspguid` compiler to requirements which are not supported by the `aspguid` language yet. Also, the generated source code can be used as an initial project setup for the implementation of an application with more sophisticated features. For example, the generated source code could be extended with the functionality, to connect the application to a database or to graphically visualize the output of the ASP program.

5.1.1 Architecture

The `aspguid` compiler is implemented in *Java 8.0* and builds upon a central COMPILER CLASS, which uses three basic service components: the PARSING SERVICE, the GENERATION SERVICE and the COMPILATION SERVICE. Furthermore, the compiler heavily uses custom exceptions for error handling and utilizes the *Java logging facilities* for managing the output of the compiler. A complete implementation documentation in the javadoc-format⁸ is available on GitHub⁹.

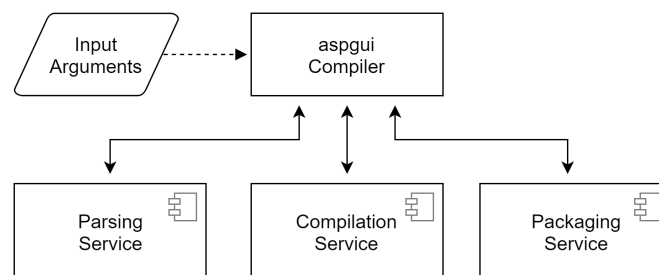


Figure 5.1: Basic architecture of the `aspguid` compiler implementation

⁵<http://www.oracle.com/technetwork/java/javase/downloads/index.html>

⁶<https://maven.apache.org/>

⁷<http://ant.apache.org/>

⁸<http://www.oracle.com/technetwork/java/javase/documentation/javadoc-137458.html>

⁹<https://nnatter.github.io/aspguid-compiler/docs/>

The PARSING SERVICE is responsible for extracting the GUI-definition object from the annotated ASP program, validating the content of the extracted GUI-definition object and storing the information from the GUI-definition object into an internal GUI-definition data structure. The service uses the *Gson*¹⁰ library to process the properties of the GUI-definition object.

The COMPILATION SERVICE of the compiler implements the functionality for generating the Java application source code for a given GUI-definition data structure. The Java source code is generated from code templates, which are stored in the resources folder of the compiler. The code templates are accessed with the *Guava*¹¹ library and processed with the *jtwig*¹² library.

The PACKAGING SERVICE is responsible for compiling and packaging the generated Java application source code into an executable .jar application. The generated source code is compiled by executing the respective target of the generated Ant build file. The target of the build file is executed with the *Ant Java library*.

5.2 Compiled Java Application

We will refer to applications compiled by the aspguid compiler as aspguid applications from this point. An aspguid application represents a standalone application which provides a GUI respective to the GUI definition of an annotated ASP program and allows to evaluate the underlying ASP program via this GUI.

The source code and an executable .jar version of a project planning aspguid application is available on GitHub¹³. A problem specific description of this example application is presented in Chapter 6. As an aspguid application is implemented in the Java language, an installed JRE of version 8 or above is required for the execution of such an application. Also, an aspguid application requires a DLV ASP solver instance [22], which must be executable through the command `dlv`.

The GUI of an aspguid application is split up in three areas, similar to the GUI-definition object of an annotated ASP program: the *information area* which displays general information about the application, the *input area* and the *output area*.

The *input area* of the GUI of an aspguid application contains the input-components, which are declared by the input-elements inside a GUI-definition object. The input-components assist the input of data by **validating the user input** in real time and **displaying contextual suggestions** based on the current user input. The data which is entered via this components is passed to the ASP solver in an appropriate format on the evaluation of the underlying ASP program.

¹⁰<https://github.com/google/gson>

¹¹<https://github.com/google/guava>

¹²<http://jtwig.org/>

¹³<https://github.com/nnatter/aspguid-project-planning>

Furthermore, the *input area* provides buttons for **evaluating the underlying ASP program** and **loading and storing problem instances**. Problem instances are stored in text files which contain the data of the input-components in the standardized ASP format. This allows to use problem instance files as input for an ASP solver independently from the aspguid application.

The *output area* of the GUI of an aspguid application is shown only after the evaluation of the underlying ASP program. It contains the output-components, which are declared by the output-elements inside a GUI-definition object. The data which is output via this components is extracted from the currently displayed answer set of the ASP program. If the evaluation of the ASP program leads to more than one answer set, the *output area* provides buttons to **change the currently displayed answer set**. Furthermore, if the ASP solver signals an error on evaluation of the ASP program, the *output area* of the GUI displays the respective error message.

5.2.1 Architecture

An aspguid application is implemented in *Java 8.0* and uses the *JavaFX*¹⁴ framework for the implementation of the GUI. This section briefly explains the basic architecture of such an application. A complete implementation documentation for an example aspguid application is available on GitHub¹⁵.

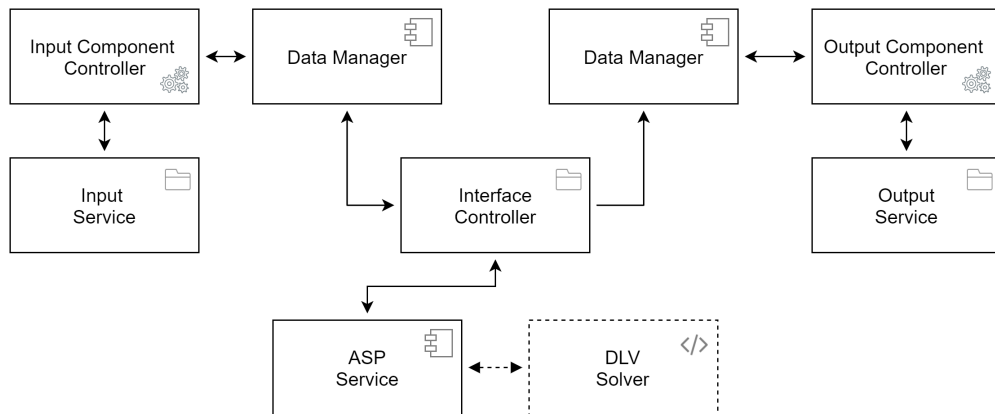


Figure 5.2: Basic architecture of an aspguid application

The source code of an aspguid application contains an INPUT COMPONENT CONTROLLER class for each input-element, which is stated in the GUI-definition object of the annotated ASP program. An INPUT-COMPONENT CONTROLLER is responsible for rendering the input-component in the GUI of the application.

¹⁴<https://docs.oracle.com/javafx/>

¹⁵<https://nnatter.github.io/aspguid-project-planning/docs/>

An INPUT COMPONENT CONTROLLER uses service components which are stored in the INPUT SERVICE PACKAGE to validate the entered data and compute suggestions for the current user input. Each INPUT COMPONENT CONTROLLER is assigned to a DATA MANAGER COMPONENT, which is responsible for managing the data which was entered via the respective input-component.

Additionally, the source code of an aspguid application contains an OUTPUT COMPONENT CONTROLLER class for each output-element, which is stated in the GUI-definition object of the annotated ASP program. An OUTPUT COMPONENT CONTROLLER is responsible for rendering the output-component in the GUI of the application.

An OUTPUT COMPONENT CONTROLLER uses service components which are stored in the OUTPUT SERVICE PACKAGE to manage the visibility of the output-component. Each OUTPUT COMPONENT CONTROLLER is assigned to a DATA MANAGER COMPONENT, which is responsible for managing the data which is displayed in the component.

The INTERFACE CONTROLLER CLASSES of an aspguid application are responsible for rendering and managing control elements which are displayed in the GUI. For example, the buttons for loading and storing problem instances and the button for evaluating the underlying ASP program are managed by these classes.

Furthermore, the INTERFACE CONTROLLER CLASSES are responsible for collecting the data which is managed by the DATA MANAGER COMPONENTS of the INPUT-COMPONENT CONTROLLERS before the ASP program is evaluated. After the evaluation of the ASP program, the INTERFACE CONTROLLER CLASSES are liable for setting the data of the currently displayed answer set to the DATA MANAGER COMPONENTS of the OUTPUT COMPONENT CONTROLLERS.

The ASP SERVICE implements the functionality for evaluating the underlying ASP program with the *DLV solver* and parsing the output of the *DLV solver*, which is executed in a sub-process of the aspguid application. The ASP SERVICE is designed in such a way, that it can be easily extended to support other ASP solvers.

Showcase & Assessment

This chapter presents an `aspguid` application for the project planning problem and assesses the `aspguid` approach proposed in this thesis based on this application.

The presented application was implemented in the ASP language, annotated with the `aspguid` language and compiled with the `aspguid` compiler. The source code of the presented application, along with an executable `.jar` version, an example problem instance, a full implementation documentation and the annotated ASP program and is available on GitHub¹. Additionally, the annotated ASP program and an example problem instance is presented in Appendix B.

6.1 Project Planning Problem

A project consists of one or more tasks, a set of dependencies between tasks and a number of assigned employees. Each task of a project has a name, a duration and occupies a defined number of employees. For a given project instance, a solution of the project planning problem is a project plan.

A project plan consists of a start time and end time for each task. In a valid project plan, start times and end times are assigned to the tasks in such a way, that the number of occupied employees does not exceed the number of project employees at any point during the project plan. Furthermore, the start time of a task which depends on another task must be greater or equal to the end time of the other task.

The project planning problem represents a constrained variation of the partial order planning problem [4] that allows to present multiple features of the `aspguid` language. Furthermore, the project planning problem is motivated by the well-known job shop scheduling problem [25], which is applicable to a broad range of practice-relevant challenges such as railway scheduling [24], sustainable manufacturing [11] and air traffic control [5].

¹<https://github.com/nnatter/aspguid-project-planning>

6.2 Project Planning Application

The `aspguid` application for the project planning problem provides a clear and intuitive GUI for defining project instances and displaying respective project plans, which are generated by the underlying ASP program. An executable `.jar` version of the presented application is available on GitHub².

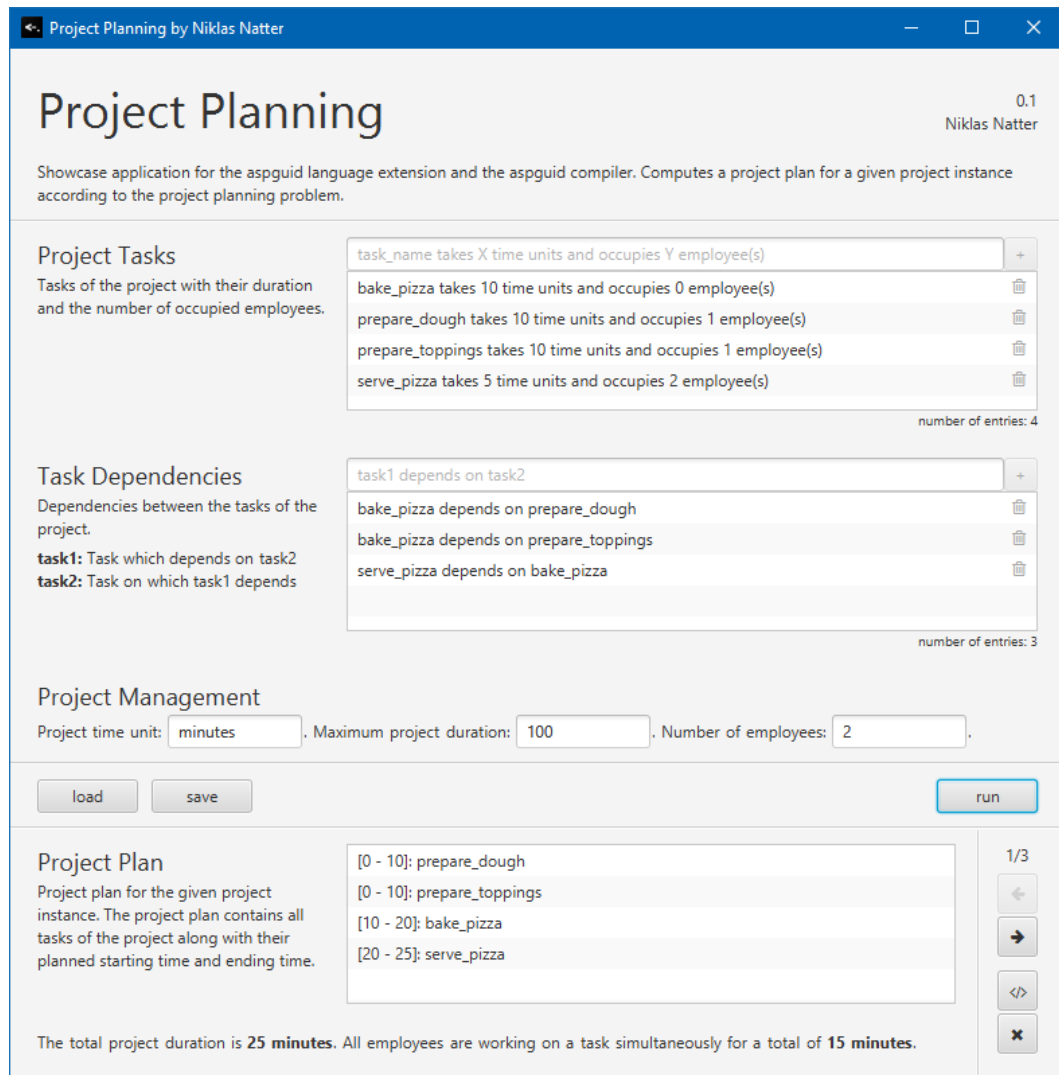


Figure 6.1: GUI of the `aspguid` application for the project planning problem

²<https://nnatter.github.io/aspguid-project-planning/artifact/project-planning.asp.jar>

The *input area* of the presented application provides concise input-components for defining tasks of a project and dependencies between these tasks, using **natural language sentences**. Project related information such as the maximum project duration and the number of assigned employees can be defined in a separate input-component.

Furthermore, the input-components of the application assist the input of data by **validating the user input** in real time and **displaying contextual suggestions** based on the current user input.

Figure 6.2: Input-component which is used to define tasks

Figure 6.3: Suggestions displayed by the dependency input-component

Figure 6.4: Validation in the project management input-component

In addition to the input-components, the *input area* of the project planning application provides buttons for **storing** the current project instance, **loading** an existing project instance and **evaluating** the underlying ASP program.

After the evaluation of the ASP program, the output-components in the *output area* of the application display information about the generated project plans. If there are multiple possible plans, buttons on the right side of the *output area* allow to change the currently displayed plan. Furthermore, the *output area* of the application provides a button to display the **atoms of the answer set** of the currently displayed plan.

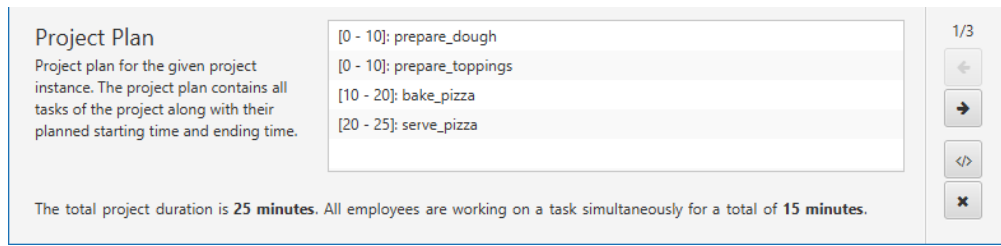


Figure 6.5: *Output area* of the project planning application

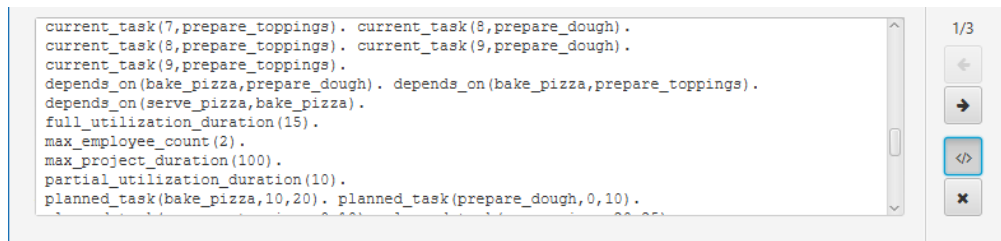


Figure 6.6: Atoms of the answer set of the currently displayed project plan

Generated project plans are displayed in an **intuitive format**, which contains the task names along with the planned start time and end time. Furthermore, the total project duration and the duration of full employee utilization is displayed. If an employee is not assigned to a task for over 75% of the project, a **warning** is displayed.

If the underlying ASP program is unable to generate valid project plans or the evaluation of the ASP program leads to a solver error, a **respective message** is displayed in the *output area* of the application.

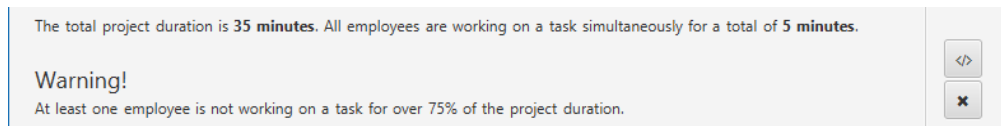


Figure 6.7: Warning on low employee utilization



Figure 6.8: Solver error message in the *output area*

6.3 Project Planning ASP Encoding

This section shortly examines the input and the output of the ASP program for the project planning problem. The complete annotated ASP program is available on GitHub³ and also presented in Appendix B.

An input problem instance for the project planning program consists of a set of entities of the types `task`, `depends_on`, `max_employee_count`, `max_project_duration` and `time_unit`.

The `task` type consists of three attributes which specify the name, the number of occupied employees and the duration of a task. The `depends_on` type represents dependencies between tasks and consists of two attributes which specify the respective tasks. The `max_employee_count` type has one attribute which is the number of assigned employees of the project. The `max_project_duration` type has one attribute which is the maximum duration of the project and the `time_unit` type has one attribute which specifies the time unit of the project.

A solution set of the project planning problem consists of a set of entities of the types `planned_task`, `project_duration`, `full_utilization_duration` and `utilization_warning`.

The `planned_task` type represents entries of the project plan and consists of three attributes which specify the task name, the start time and the end time. The `project_duration` type has one attribute which is the total duration of the project plan. The `full_utilization_duration` has one attribute which is the duration of full employee utilization during the project plan. The `utilization_warning` type has no attributes. If an entity of this type is present in an answer set, at least one employee is not assigned to a task for over 75% of the duration of the project plan.

6.4 Project Planning GUI Definition

The `GUI-definition` object of the annotated ASP program for the project planning problem contains 3 input-elements and 3 output-elements besides the straightforward program-information object. This section briefly explains selected parts of this `GUI-definition` object. The complete annotated ASP program is available on GitHub⁴ and also presented in Appendix B.

Listing 6.1 presents an excerpt from the `@entity_input` input-element which declares the task dependency input-component. It can be seen that the `gui_representation` property of the `@entity_input` element is used to enable the input of dependencies between tasks in natural language sentences.

³<https://raw.githubusercontent.com/nnatter/aspguid-project-planning/master/asp/project-planning.asp>

⁴<https://raw.githubusercontent.com/nnatter/aspguid-project-planning/master/asp/project-planning.asp>

```
@entity_input:depends_on": {
  "title": "Task Dependencies",
  "description": "Dependencies between the tasks of the
    ↪ project.",
  "gui_representation": "::task1 depends on ::task2",
  "@input_attribute:task1": {
    "description": "Task which depends on task2",
    "value_source": "$task[0]"
  },
  ...
}
```

Listing 6.1: Declaration of the task dependency input-component

Furthermore, the *value_source* property of the `@input_attribute` elements is used to ensure that entered dependencies refer to existing tasks of the project. The presented `@entity_input` element uses the default representation template as value for its *atom_representation* property.

```
@value_input:project_management": {
  "title": "Project Management",
  "gui_representation": "Project time unit: ::time_unit. Maximum
    ↪ project duration: ::max_project_duration...",
  "@input_value:time_unit": {
    "name": "time unit",
    "default_value": "days",
    "value_source": "$available_time_unit[0]"
  },
  ...
}
```

Listing 6.2: Declaration of the project information input-component

Listing 6.2 presents an excerpt from the `@value_input` input-element which declares the input-component for project management related information. The input-element uses a single `@input_value` element for each project management related input value.

The *value_source* property of the time unit `@input_value` element ensures that only time units stated in the ASP program can be used as value of the element. Together with the suggestion functionality of the input-component in the GUI, this leads to a user experience that is comparable to a drop-down menu.

Listing 6.3 presents an excerpt from the `@value_output` output-element which declares the project duration output-component. The *title* property of this `@value_output` element is not defined intentionally, as the output-component is displayed below the project plan component in the GUI of the application.


```

"@value_output:time_information": {                                1
  "gui_representation": "The total project duration is           2
  ↪ ::project_duration ::time_unit1. ... a total of
  ↪ ::full_utilization_duration ::time_unit2.",
  "@output_value:time_unit1": {                                    3
    "atom_representation": "time_unit (::time_unit1) "           4
  },                                                              5
  "@output_value:time_unit2": {                                    6
    "atom_representation": "time_unit (::time_unit2) "           7
  },                                                              8
  ...                                                            9
}                                                                  10

```

Listing 6.3: Declaration of the project duration output-component

For a better user experience, the `@value_output` element uses an `@output_value` element to extract the project time unit from the displayed answer set. As the time unit is displayed two times, but a representation template can contain each placeholder only a single time, the time unit is extracted two times by separate `@output_value` elements.

```

"@value_output:utilization_warning": {                             1
  "title": "Warning!",                                           2
  "gui_representation": "At least one employee is not...",      3
  "output_condition": "?bad_utilization"                         4
}                                                                  5

```

Listing 6.4: Declaration of the utilization warning output-component

Listing 6.4 presents an excerpt from the `@value_output` output-element which declares the employee utilization warning output-component. As the output-component displays a static text, the `@value_output` element does not contain any `@output_value` element.

The *output_condition* property of the `@value_output` element is used to display the warning only if the currently displayed answer set is tagged with a particular atom.

6.5 Showcase Assessment

The presented `aspguid` application for project planning problem shows that the `aspguid` language enables the declaration of manifold GUI definitions. In combination with the `aspguid` compiler, the language can be utilized to implement intuitive end-user applications based on annotated ASP programs.

The showcase application demonstrates that the custom property types of the `aspguid` language allow the definition of flexible GUIs and valuably extend the expressiveness of the language. Furthermore, the presented showcase application hints that the functionality of the input-elements and output-elements included in the proposed `aspguid` language can be utilized to define GUI components for a broad range of use cases.

Compared to a raw ASP program for the project planning problem, the presented `aspguid` application significantly simplifies the usage of the application by providing an intuitive GUI for the input of data and the presentation of results. Therefore, the application requires notably less user knowledge about the ASP ecosystem and can be operated by users that are inexperienced in the area of logic programming.

The presented application also reveals starting points for future work on the `aspguid` language and the `aspguid` compiler. Even though the fallback values for optional properties of the GUI-definition object allow to abbreviate the GUI-definition of an ASP program, the JSON format for GUI definitions leads to lengthy annotation sections which are prone to syntax errors. The usage of another format might reduce this flaws.

Furthermore, the input-elements and output-elements included in the current version of the `aspguid` language do not allow the usage of content like images or graphical visualizations. Here, additional types for input-elements and output-elements that allow the usage of multimedia content, could be designed.

A more comprehensive outline of possible improvements regarding to the `aspguid` language and the `aspguid` compiler is presented in Section 7.2.

Conclusion

This thesis proposed and assessed a new approach, named `aspguid`, for implementing applications usable by users without prior logic programming experience based on annotated ASP programs. The proposed approach consists of a declarative language for defining graphical user interfaces for ASP programs and a compiler that translates ASP programs which are annotated with such definitions into procedural code, that realizes respective graphical user interface assisted applications.

The `aspguid` language allows for the specification of versatile graphical user interfaces by providing general user interface elements along with different types of input-elements and output-elements. The presented `aspguid` compiler allows to translate an annotated ASP program into executable Java source code, which can be adjusted and extended by a Java developer to more sophisticated requirements. However, as the `aspguid` language is independent of a specific programming language, compilers that translate GUI definitions into source code for other programming languages could be developed in future work.

An assessment of the proposed techniques based on a showcase application is promising and hints that the `aspguid` approach can be utilized to implement flexible end-user applications for a broad range of use cases. Furthermore, the assessment suggests that applications implemented with the `aspguid` techniques can notably reduce the required user knowledge about the ASP ecosystem compared to raw ASP programs, by proving an intuitive and assisted way for the input of data and displaying the output data in a structured and formatted manner.

In total, the `aspguid` approach provides a straightforward solution for developing ASP-dependent applications usable by users without prior logic programming experience, which significantly reduces the development overhead and minimizes the repetitive workload for developers, compared to current state of the art solutions.

7.1 Related Work

To the best of our knowledge, there is no directly comparable work concerning developing applications with graphical user interfaces in the ASP paradigm or extending the ASP language with a standardized format for graphical user interface definitions. Still, there are related areas, namely integrated development environments in the ASP paradigm, the integration of ASP with other programming languages and the visualization of answer sets of ASP programs.

7.1.1 Integrated Development Environments

The area of integrated development environments in the ASP paradigm includes the tools ASPIDE [12] and SeaLion [26]. These solutions support software engineers during the development of ASP programs by providing features such as syntax highlighting, code autocompletion and debugging support. These tools valuably expand the ASP ecosystem and simplify the development of ASP programs, but they cannot be utilized to realize GUIs for ASP programs.

7.1.2 Integration with Other Programming Languages

The techniques proposed in the area of integrating ASP with other programming languages aim to support the data exchange between ASP programs and components implemented in other programming languages. The py-aspio library [27] provides a language independent way for integrating ASP programs with object-oriented programming languages. Febraro et al. [13] proposed a Java specific technique called JASP for similar purposes. Approaches like EmbASP [8] and asp4j¹ support the mapping of data between Java and ASP. Furthermore, techniques like PyASP² or Tweety [30] assist the execution of ASP solvers from imperative code.

Although these techniques can be used to manually develop imperative GUIs for embedded ASP programs, they come with a notable overhead compared to the automated GUI generation capability of the the `aspguid` approach.

7.1.3 Visualization of Answer Sets

The field of visualizing data from answer sets of ASP programs includes the techniques ASPVIZ [9] and Kara [20]. The proposed approaches allow to generate graphical visualizations of interpretations of ASP programs and use the ASP language itself for defining the content of these visualizations. These techniques are used to graphically visualize results of ASP programs for debugging purposes. Although approaches from this field also can be utilized to visualize results of ASP programs in an intuitive manner, end-user applications that make use of this visualizations must be implemented manually.

¹<https://github.com/hbeck/asp4j>

²<https://pypi.python.org/pypi/pyasp>

7.2 Future Work

The `aspguid` approach provides a straightforward solution for implementing end-user applications based on ASP programs. Still, the `aspguid` language and the `aspguid` compiler could be extended and improved in various concerns in future work.

7.2.1 Language Improvements

The usage of the JSON format for the *GUI-definition object* of the `aspguid` language allows for an intuitive declaration of GUI definitions, but also leads to lengthy annotation sections inside ASP programs that are prone to syntax errors. The usage of another format for declaring GUI definitions might reduce this flaws.

The `aspguid` language currently includes two types of input-elements and two types of output-elements. The design of additional types would valuably extend the diversity and the expressiveness of the language. For example, an output type which integrates one of the visualization techniques mentioned in Section 7.1 could be designed. This would allow intuitive graphical visualizations of the output of the ASP program inside the GUI of an `aspguid` application.

Also, the types of input-elements and output-elements which are already included in the `aspguid` language could be further improved. For example, the *@entity_input* type and the *@entity_output* type could be extended to optionally support the sorting and grouping of entities by a particular attribute. This would further increase the flexibility of these element types.

In the current version of the `aspguid` approach, input- and output-components are aligned in a vertical list in the GUI of an annotated ASP program. This leads to ineffective application layouts for complex use cases. To overcome this issue, the `aspguid` language could be extended to support layout strategies, which define how components are aligned in the GUI of an program.

7.2.2 Technical Improvements

The *atom-argument value source*, which is used to define valid values for input-components, currently extracts values only from atoms which are stated in the ASP program or which are generated by an input-component of the application. This could be extended in such a way that an *atom-argument value source* also extracts values from atoms, which can be derived by rules inside the ASP program.

The proposed `aspguid` compiler and `aspguid` applications are implemented in Java 8.0 and therefore depend on an installed JRE and JDK of an appropriate version. The usage of a system native programming language would reduce the dependencies and improve the performance of these applications.

7.2.3 Integration with Other Tools

Applications compiled by the `aspguid` compiler currently invoke the DLV solver for computing the answer sets of an ASP program. Hereby, the compiler could be extended to support other solvers for a better flexibility.

`aspguid` applications already support the storing and loading of problem instances contained in a text file. This mechanism could be extended to allow the storing and loading of problem instances contained in other file formats like `.csv` files or excel spreadsheets.

Finally, the integration of the `aspguid` approach into existing integrated development environments, such as `ASPIDE` [12] and `SeaLion` [26], would lower the barriers for developers and further simplify the implementation of graphical user interface assisted applications in the ASP ecosystem.

apsguid Language Specification

A.1 GUI-Definition Object

Property Name	Type	Informal Description
program_information	object	Declares general information which is rendered in the GUI of the annotated program (specified in A.2).
program_input	object	Declares the input-components of the GUI of the annotated program (specified in A.3).
program_output	object	Declares the output-components of the GUI of the annotated program (specified in A.8).

Table A.1: Properties of the GUI-definition object

A.2 program_information Object

Property Name	Type	Informal Description
name	string	Name of the annotated program.
description (optional)	string	Description of the annotated program. If not specified, no description is displayed in the GUI.
author (optional)	string	Author of the annotated program. If not specified, no author is displayed in the GUI.
version (optional)	string	Version of the annotated program. If not specified, no version is displayed in the GUI.

Table A.2: Properties of the program_information object

A.3 program_input Object

Property Name	Type	Informal Description
@entity_input:id1	element	Input-element of the type @entity_input with the identifier id1. Declares an input-component in the GUI (specified in A.4).
@value_input:id2	element	Input-element of the type @value_input with the identifier id2. Declares an input-component in the GUI (specified in A.6).

Table A.3: Properties of the program_input object

A.3.1 @entity_input Element-Type

Property Name	Type	Informal Description
title (optional)	string	Title of the input-component. If not specified, no title is displayed in the GUI.
description (optional)	string	Description of the input-component. If not specified, no description is displayed in the GUI.
input_count_min (optional)	integer	Defines the minimum number of entities which must be entered via the input-component. If not defined, 0 is used.
input_count_max (optional)	integer	Defines the maximum number of entities which can be entered via the input-component. If not defined, the number of entities is not limited.
gui_representation (optional)	representation template	Defines in which format an entity can be entered and in which format a single entity is displayed in the GUI. If not specified, the default representation template is used (see 4.5.1).
code_representation (optional)	representation template	Defines how a single entity is represented as an atom. If not specified, the default representation template is used (see 4.5.1).
@input_attribute:id	element	Element of the type @input_attribute with the identifier id. Defines an attribute of the entity-type of the input-component (specified in A.5).

Table A.4: Properties of an @entity_input input-element

@input_attribute Element-Type

Property Name	Type	Informal Description
name (optional)	string	Name of the attribute. If not specified, the identifier of the element is used.
description (optional)	string	Description of the attribute. If not specified, no description is displayed in the GUI.
value_source (optional)	source selector	Restricts the set of valid values for the attribute. If not specified, every value is considered as valid (see 4.5.2).

Table A.5: Properties of an @input_attribute element

A.3.2 @value_input Element-Type

Property Name	Type	Informal Description
title (optional)	string	Title of the input-component. If not specified, no title is displayed in the GUI.
gui_representation (optional)	representation template	Defines the content of the input-component in the GUI. If not specified, the default representation template is used (see 4.5.1).
@input_value:id	element	Element of the type @input_value with the identifier id. Defines a value which can be entered via the input-component (specified in A.7).

Table A.6: Properties of a @value_input input-element

@input_value Element-Type

Property Name	Type	Informal Description
name (optional)	string	Name of the value which is displayed in the GUI. If not specified, the identifier of the element is used.
default_value (optional)	string	Default value of the element. If not specified, the value of the element is empty by default.
value_source (optional)	source selector	Restricts the set of valid values for the element. If not specified, every value is considered as valid (see 4.5.2).
code_representation (optional)	representation template	Defines how the value of the element is represented as an atom. If not specified, the default representation template is used (see 4.5.1).

Table A.7: Properties of an @input_value element

A.4 program_output Object

Property Name	Type	Informal Description
@entity_output:id1	element	Output-element of the type @entity_output with the identifier id1. Declares an output-component in the GUI (specified in A.9).
@value_output:id2	element	Output-element of the type @value_output with the identifier id2. Declares an output-component in the GUI (specified in A.11).

Table A.8: Properties of the program_output object

A.4.1 @entity_output Element-Type

Property Name	Type	Informal Description
title (optional)	string	Title of the output-component. If not specified, no title is displayed in the GUI.
description (optional)	string	Description of the output-component. If not specified, no description is displayed in the GUI.
output_condition (optional)	condition statement	Defines a condition for the visibility of the output-component. If not specified, the output-component is visible by default (see 4.5.3).
code_representation (optional)	representation template	Defines how a single entity is represented as an atom in an answer set. If not specified, the default representation template is used (see 4.5.1).
gui_representation (optional)	representation template	Defines in which format a single entity is displayed in the GUI. If not specified, the default representation template is used (see 4.5.1).
@output_attribute:id	element	Element of the type @output_attribute with the identifier id. Defines an attribute of the entity-type of the output-component (specified in A.10).

Table A.9: Properties of an @entity_output output-element

@output_attribute Element-Type

Property Name	Type	Informal Description
name (optional)	string	Name of the attribute. If not specified, the identifier of the element is used.
description (optional)	string	Description of the attribute. If not specified, no description is displayed in the GUI.

Table A.10: Properties of an @output_attribute element

A.4.2 @value_output Element-Type

Property Name	Type	Informal Description
title (optional)	string	Title of the output-component. If not specified, no title is displayed in the GUI.
output_condition (optional)	condition statement	Defines a condition for the visibility of the output-component. If not specified, the output-component is visible by default (see 4.5.3).
gui_representation (optional)	representation template	Defines the content of the output-component in the GUI. If not specified, the default representation template is used (see 4.5.1).
@output_value:id	element	@output_value element with the identifier id. Defines a value which is output via the output-component (specified in A.12).

Table A.11: Properties of a @value_output output-element

@output_value Element-Type

Property Name	Type	Informal Description
name (optional)	string	Name of the value which is displayed in the GUI. If not specified, the identifier of the element is used.
code_representation (optional)	representation template	Defines how the value of the element is represented as an atom in an answer set. If not specified, the default representation template is used (see 4.5.1).

Table A.12: Properties of an @output_value element

Showcase: Project Planning

B.1 Annotated Project Planning ASP Program

```

%*::: 1
{ 2
  "program_information": { 3
    "name": "Project Planning", 4
    "description": "Showcase application for the aspguid language 5
      ↳ extension and the aspguid compiler. Computes a project
      ↳ plan for a given project instance according to the
      ↳ project planning problem.",
    "author": "Niklas Natter", 6
    "version": "0.1" 7
  }, 8
  "program_input": { 9
    "@entity_input:task": { 10
      "title": "Project Tasks", 11
      "description": "Tasks of the project with their duration 12
        ↳ and the number of occupied employees.",
      "gui_representation": "::task_name takes ::duration time 13
        ↳ units and occupies ::employee_count employee(s)",
      "input_count_min": 1, 14
      "@input_attribute:task_name": {}, 15
      "@input_attribute:employee_count": { 16
        "name": "Y", 17
        "value_source": "$#int" 18
      }, 19
      "@input_attribute:duration": { 20
        "name": "X", 21
        "value_source": "$#int" 22
      } 23
    }, 24
    "@entity_input:depends_on": { 25

```

```

        "title": "Task Dependencies",
        "description": "Dependencies between the tasks of the
        ↪ project.",
        "gui_representation": "::task1 depends on ::task2",
        "@input_attribute:task1": {
            "description": "Task which depends on task2",
            "value_source": "$task[0]"
        },
        "@input_attribute:task2": {
            "description": "Task on which task1 depends",
            "value_source": "$task[0]"
        }
    },
    "@value_input:project_management": {
        "title": "Project Management",
        "gui_representation": "Project time unit: ::time_unit.
        ↪ Maximum project duration: ::max_project_duration.
        ↪ Number of employees: ::max_employee_count.",
        "@input_value:time_unit": {
            "name": "time unit",
            "default_value": "days",
            "value_source": "$available_time_unit[0]"
        },
        "@input_value:max_project_duration": {
            "name": "number",
            "value_source": "$#int"
        },
        "@input_value:max_employee_count": {
            "name": "number",
            "value_source": "$#int"
        }
    },
    "program_output": {
        "@entity_output:planned_task": {
            "title": "Project Plan",
            "description": "Project plan for the given project instance.
            ↪ The project plan contains all tasks of the project
            ↪ along with their planned starting time and ending
            ↪ time.",
            "gui_representation": "[::start - ::end]: ::task_name",
            "@output_attribute:task_name": {},
            "@output_attribute:start": {},
            "@output_attribute:end": {}
        },
        "@value_output:time_information": {
            "gui_representation": "The total project duration is
            ↪ ::project_duration ::time_unit1. All employees are
            ↪ working on a task simultaneously for a total of
            ↪ ::full_utilization_duration ::time_unit2.",
            "@output_value:project_duration": {},
            "@output_value:full_utilization_duration": {},
            "@output_value:time_unit1": {
                "atom_representation": "time_unit(::time_unit1)"
            }
        }
    },

```



```

    "@output_value:time_unit2": {
        "atom_representation": "time_unit(:time_unit2)"
    },
    "@value_output:utilization_warning": {
        "title": "Warning!",
        "gui_representation": "At least one employee is not working
        ↪ on a task for over 75% of the project duration.",
        "output_condition": "?bad_utilization"
    }
}
}
::*%

#maxint=200.

% define available time units for GUI
available_time_unit(seconds).
available_time_unit(minutes).
available_time_unit(hours).
available_time_unit(days).
available_time_unit(weeks).
available_time_unit(months).

% zero is a possible start time for a task, also every end of a
    ↪ planned task is a possible start time
possible_start_time(0).
possible_start_time(Time) :- planned_task(_,_,Time).

% plan every task
planned_task(Name,Start,End) v -planned_task(Name,Start,End) :-
    ↪ task(Name,_,Duration), possible_start_time(Start), End =
    ↪ Start + Duration.

% each task must be planned
task_is_assigned(Name) :- task(Name,_,_), planned_task(Name,_,_).
:- task(Name,_,_), not task_is_assigned(Name).

% a task must not be assigned to more than one start time
:- task(Name,_,_), planned_task(Name,Start1,_),
    ↪ planned_task(Name,Start2,_), Start1 <> Start2.

% a task which depend on an other tasks must not start, before
    ↪ the other tasks is completed
:- depends_on(Task1,Task2), planned_task(Task1,Start,_),
    ↪ planned_task(Task2,_,End), End > Start.

% the planned project duration must not exceed the maximum
    ↪ duration of the project
project_duration(Duration) :- #max{End : planned_task(_,_,End)} =
    ↪ Duration, #int(Duration).
:- project_duration(Duration), max_project_duration(MaxDuration),
    ↪ Duration > MaxDuration.

```

```

% time points during the project
time_point(X) :- project_duration(Duration), #int(X), X >= 0, X <
    Duration.

% the number of employees assigned to a task on a specific time
    point must not exceed the number of employees of the project
current_task(Time,Name) :- time_point(Time),
    planned_task(Name,Start,End), Start <= Time, End > Time.
current_employee_count(Time,Count) :- time_point(Time),
    #sum{EmployeeCount,Name : task(Name,EmployeeCount,_),
    current_task(Time,Name)} = Count, #int(Count).
:- current_employee_count(_,EmployeeCount),
    max_employee_count(MaxEmployeeCount), EmployeeCount >
    MaxEmployeeCount.

% calculate duration of full staff utilization throughout the
    project
full_utilization_duration(Duration) :-
    max_employee_count(MaxEmployeeCount), #count{T :
    current_employee_count(T,C), C = MaxEmployeeCount} =
    Duration, #int(Duration).
partial_utilization_duration(Duration) :-
    full_utilization_duration(FullUtilization),
    project_duration(ProjectDuration), Duration =
    ProjectDuration - FullUtilization.

% tag plan with bad_utilization, if the full staff utilization
    duration is below 25% of the project duration
bad_utilization :- full_utilization_duration(FullUtilization),
    partial_utilization_duration(PartialUtilization),
    TempFullUtilization = FullUtilization * 3,
    TempFullUtilization < PartialUtilization.

```

Listing B.1: Annotated ASP program for the project planning problem

B.2 Example Project Planning Problem Instance

```

task(bake_pizza,0,10). task(prepare_dough,1,10).
    task(prepare_toppings,1,10). task(serve_pizza,2,5).
depends_on(bake_pizza,prepare_dough).
    depends_on(bake_pizza,prepare_toppings).
    depends_on(serve_pizza,bake_pizza).
max_employee_count(2). max_project_duration(100).
    time_unit(minutes).

```

Listing B.2: Example problem instance for the project planning program

Bibliography

- [1] Chitta Baral. *Knowledge Representation, Reasoning and Declarative Problem Solving*. Cambridge University Press, New York, NY, USA, 1st edition, 2010. ISBN 0521147751, 9780521147750.
- [2] Chitta Baral and Michael Gelfond. *Reasoning Agents in Dynamic Domains*, pages 257–279. Springer US, Boston, MA, 2000. ISBN 978-1-4615-1567-8.
- [3] Victor A. Bardadym. *Computer-aided school and university timetabling: The new wave*, pages 22–45. Springer Berlin Heidelberg, Berlin, Heidelberg, 1996. ISBN 978-3-540-70682-3.
- [4] Anthony Barrett and Daniel S. Weld. Partial-order planning: Evaluating possible efficiency gains. *Artificial Intelligence*, 67(1):71 – 112, 1994.
- [5] Lucio Bianco, Paolo Dell’Olmo, and Stefano Giordani. Scheduling models for air traffic control in terminal areas. *Journal of Scheduling*, 9(3):223–253, Jun 2006.
- [6] Loreto Bravo and Leopoldo Bertossi. Logic programs for consistently querying data integration systems. In *Proceedings of the 18th International Joint Conference on Artificial Intelligence, IJCAI’03*, pages 10–15, San Francisco, CA, USA, 2003. Morgan Kaufmann Publishers Inc.
- [7] Francesco Calimeri, Wolfgang Faber, Martin Gebser, Giovambattista Ianni, Roland Kaminski, Thomas Krennwallner, Nicola Leone, Francesco Ricca, and Torsten Schaub. ASP-Core-2: Input language format. *ASP Standardization Working Group, Tech. Rep*, 2012.
- [8] Francesco Calimeri, Davide Fuscà, Stefano Germano, Simona Perri, and Jessica Zangari. Embedding asp in mobile systems: discussion and preliminary implementations. In *Proceedings of the Eighth Workshop on Answer Set Programming and Other Computing Paradigms (ASPOCP 2015), workshop of the 31st International Conference on Logic Programming (ICLP 2015)*, 2015.
- [9] Owen Cliffe, Marina De Vos, Martin Brain, and Julian Padget. *ASPVIZ: Declarative Visualisation and Animation Using Answer Set Programming*, pages 724–728. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008. ISBN 978-3-540-89982-2.

- [10] Thomas Eiter, Giovambattista Ianni, and Thomas Krennwallner. *Answer Set Programming: A Primer*, pages 40–110. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009. ISBN 978-3-642-03754-2.
- [11] Kan Fang, Nelson Uhan, Fu Zhao, and John W. Sutherland. *A New Shop Scheduling Approach in Support of Sustainable Manufacturing*, pages 305–310. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011. ISBN 978-3-642-19692-8.
- [12] Onofrio Febbraro, Kristian Reale, and Francesco Ricca. *ASPIDE: Integrated Development Environment for Answer Set Programming*, pages 317–330. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011. ISBN 978-3-642-20895-9.
- [13] Onofrio Febbraro, Giovanni Grasso, Nicola Leone, and Francesco Ricca. JASP: A framework for integrating answer set programming with Java. In *Proceedings of the Thirteenth International Conference on Principles of Knowledge Representation and Reasoning*, KR’12, pages 541–551. AAAI Press, 2012. ISBN 978-1-57735-560-1.
- [14] Enrico Franconi, Antonio Laureti Palma, Nicola Leone, Simona Perri, and Francesco Scarcello. *Census Data Repair: A Challenging Application of Disjunctive Logic Programming*, pages 561–578. Springer Berlin Heidelberg, Berlin, Heidelberg, 2001. ISBN 978-3-540-45653-7.
- [15] Alfredo Garro, Luigi Palopoli, and Francesco Ricca. Exploiting agents in e-learning and skills management context. *AI Commun.*, 19(2):137–154, January 2006.
- [16] Michael Gelfond and Vladimir Lifschitz. Classical negation in logic programs and disjunctive databases. *New Generation Computing*, 9(3):365–385, 1991.
- [17] Giovanni Grasso, Nicola Leone, and Francesco Ricca. *Answer Set Programming: Language, Applications and Development Tools*, pages 19–34. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013. ISBN 978-3-642-39666-3.
- [18] Tomi Janhunnen, Ilkka Niemelä, Johannes Oetsch, Jörg Pührer, and Hans Tompits. On testing answer-set programs. In *Proceedings of the 2010 Conference on ECAI 2010: 19th European Conference on Artificial Intelligence*, pages 951–956, Amsterdam, The Netherlands, The Netherlands, 2010. IOS Press. ISBN 978-1-60750-605-8.
- [19] Benjamin Kiesl, Peter Schüller, and Hans Tompits. On structural analysis of non-ground answer-set programs. In *ICLP*, 2015.
- [20] Christian Kloimüllner, Johannes Oetsch, Jörg Pührer, and Hans Tompits. *Kara: A System for Visualising and Visual Editing of Interpretations for Answer-Set Programs*, pages 325–344. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013. ISBN 978-3-642-41524-1.
- [21] Nicola Leone, Gianluigi Greco, Giovambattista Ianni, Vincenzino Lio, Giorgio Terracina, Thomas Eiter, Wolfgang Faber, Michael Fink, Georg Gottlob, Riccardo

- Rosati, Domenico Lembo, Maurizio Lenzerini, Marco Ruzzi, Edyta Kalka, Bartosz Nowicki, and Witold Staniszkis. The infomix system for advanced integration of incomplete and inconsistent data. In *Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data*, SIGMOD '05, pages 915–917, New York, NY, USA, 2005. ACM. ISBN 1-59593-060-4.
- [22] Nicola Leone, Gerald Pfeifer, Wolfgang Faber, Thomas Eiter, Georg Gottlob, Simona Perri, and Francesco Scarcello. The dl原因v system for knowledge representation and reasoning. *ACM Trans. Comput. Logic*, 7(3):499–562, July 2006.
- [23] Vladimir Lifschitz. Answer set planning (abstract). In *Proceedings of the 5th International Conference on Logic Programming and Nonmonotonic Reasoning*, LPNMR '99, pages 373–374, London, UK, UK, 1999. Springer-Verlag. ISBN 3-540-66749-0.
- [24] Shi Qiang Liu and Erhan Kozan. Scheduling trains as a blocking parallel-machine job shop scheduling problem. *Computers & Operations Research*, 36(10):2840 – 2852, 2009.
- [25] Alan S. Manne. On the job-shop scheduling problem. *Operations Research*, 8(2):219–223, 1960.
- [26] Johannes Oetsch, Jörg Pührer, and Hans Tompits. *The SeaLion has Landed: An IDE for Answer-Set Programming—Preliminary Report*, pages 305–324. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013. ISBN 978-3-642-41524-1.
- [27] Jakob Rath and Christoph Redl. *Integrating Answer Set Programming with Object-Oriented Languages*, pages 50–67. Springer International Publishing, Cham, 2017. ISBN 978-3-319-51676-9.
- [28] F. Ricca, G. Grasso, M. Alviano, M. Manna, V. Lio, S. Iiritano, and N. Leone. Team-building with answer set programming in the gioia-tauro seaport. *Theory Pract. Log. Program.*, 12(3):361–381, May 2012.
- [29] Francesco Ricca, Antonella Dimasi, Giovanni Grasso, Salvatore Maria Ielpa, Salvatore Iiritano, Marco Manna, and Nicola Leone. A logic-based system for e-tourism. *Fundam. Inf.*, 105(1-2):35–55, January 2010.
- [30] Matthias Thimm. Tweety: A comprehensive collection of java libraries for logical aspects of artificial intelligence and knowledge representation. In *Proceedings of the Fourteenth International Conference on Principles of Knowledge Representation and Reasoning*, KR'14, pages 528–537. AAAI Press, 2014. ISBN 1-57735-657-8, 978-1-57735-657-8.