VIDEAS: Supporting Answer-Set Program Development using Model-Driven Engineering Techniques*

Johannes Oetsch[1], Jörg Pührer[1], Martina Seidl[2,3], Hans Tompits[1], and Patrick Zwickl[4]

[1] Technische Universität Wien, Institut für Informationssysteme 184/3, Favoritenstraße 9–11, A-1040 Vienna, Austria
{oetsch,puehrer,tompits}@kr.tuwien.ac.at
[2] Johannes Kepler Universität Linz, Institut für Formale Modelle und Verifikation, Altenbergerstraße 69, A-4040 Linz, Austria
Martina.Seidl@jku.at
[3] Technische Universität Wien, Institut für Softwaretechnik, 188/3 Favoritenstraße 9–11, A-1040 Vienna, Austria
[4] FTW Telecommunications Research Center Vienna Donau-City-Straße 1, A-1220 Vienna, Austria
zwickl@ftw.at

Abstract. Recently, the techniques of *model-driven engineering* (MDE) have been proven valuable to manage the complexity of modern software systems during the software development process. In the area of answer-set programming (ASP), the focus is set so far on theoretical aspects, applications, and the development of efficient solvers, reducing the attention that is paid to the pragmatics of programming and assisting tools. To address this issue, we propose the MDE-based program development method VIDEAS by introducing explicit model-to-code mappings and code generation strategies ensuring compliant specification of facts and essential constraints. Its practical applicability is realised on the basis of a prototypical implementation.

1 Introduction

Answer-set programming (ASP) [1] offers a powerful framework for declarative problem solving based on principles of logic programming. The basic idea of ASP is to encode problems in terms of answer-set programs whose results (the "answer sets") represent the solutions of the original problems. This is in contrast to traditional logic-based knowledge representation languages like Prolog where the solutions are given by proofs.

* This research was partially supported by the Austrian Science Fund (FWF) under grant P21698 and the Vienna Science and Technology Fund (WWTF) under grant ICT10-018. The last author would like to acknowledge additional funding from FTW Telecommunications Research Center Vienna (Project U-0).

During the last decade, ASP experienced considerable attention in the academic artificial intelligence community, leading to successful applications in diverse areas like planning, diagnosis, symbolic model checking, decision support systems, and bioinformatics, among others, exploiting various sophisticated solvers like `DLV`[5] or `clasp`[6]. So far, ASP research mainly focused on (i) formal properties of the answer-set semantics, (ii) issues related to using it for knowledge representation and reasoning, and (iii) the development of efficient ASP solvers, being the trailblazers to the success in the academic world. However, ASP could not attract the same interest as other programming languages outside academia. This might be because comparably little attention was paid to the development of programming methods and tools, which are arguably indispensable to convince software engineers and programmers to take advantage of the benefits of answer-set programming.

In particular, no graphical modelling environment as it is available in traditional software engineering in terms of the *Unified Modeling Language* (UML) and in traditional data engineering in terms of the *entity relationship diagram* (ER diagram) [2] has been introduced for ASP so far. One explanation for the absence of such modelling tools may be seen in the fact that answer-set programs themselves are already defined at a high level of abstraction—in contrast to imperative programs—and may in turn be regarded as executable specification themselves. However, practice has shown that the development of answer-set programs is not always straightforward and that programs are, as all human-made artefacts, prone to errors. Consider, for example, the facts `airplan(boeing)` and `airplane(airbus)`. This small program excerpt already contains a mistake. A predicate name is misspelled, which might result in some unexpected program behaviour. Furthermore, most current ASP solvers do not support type checking. A notable exception is the `DLV`$^+$ system [3] that supports typing and concepts from object-oriented programming. If values of predicate arguments are expected to come from a specific domain only, specific constraints have to be included in the program. This requires additional programming effort and could even be a further source for programming errors.

To support answer-set programmers, we developed the tool VIDEAS [4], standing for "VIsual DEsign support for Answer-Set programming", which graphically supports the partial specification of answer-set programs. Due to the close relationship between answer-set programs and deductive databases, in VIDEAS, ER diagrams are used as a starting point for the development of answer-set programs. The constraints on the problem domain from an ER diagram are automatically translated to ASP itself. Having such constraints as part of a problem encoding can be compared to using assertions in *C* programs. To support the development of a fact base, VIDEAS automatically generates a program providing an input mask for correctly specifying the facts. To realise VIDEAS, we used well-established technologies from the active field of *model-driven engineering* (MDE) which provides tools for building the necessary graphical modelling editors as well as a code generator.

This paper is structured as follows. First, we review the status quo concerning the development support for answer-set programs in Section 2. Then, after some back-

[5] `http://www.dlvsystem.com`.
[6] `http://potassco.sourceforge.net`.

ground on ASP in Section 3, we present the basic ideas of the VIDEAS approach where ER diagrams serve as starting point for the program development and visualisation in Section 4. To this end, we introduce a mapping between ER diagrams and ASP in Section 5. Finally, we discuss the implementation in Section 6 and conclude with an outlook to future work in Section 7.

2 Related Work

For assisting the development of answer-set programs, much research effort is recently spent in debugging. For example, debugging concepts for inconsistent answer-sets [5] and model extensions [6] have been proposed. Likewise, debugging methodologies like recursive ASP debugging [7, 8], meta-programming techniques [9, 10], translational debugging reformulating programs in natural language [11], justification concepts for truth values [12], or stepping [13] have been developed. However, only limited effort is spent on directly supporting the construction phase of programs with few exceptions like the specification of services with preferences by Confalonieri et al. [14]. There are also efforts towards the realisation of *integrated development environments* (IDEs), like `APE` [15], `ASPIDE` [16], `SeaLion` [17], and `iGROM` [7], as well as for systems allowing the visualisation of answer sets, as done in the systems `ASPVIZ` [18], `IDPDraw`[8], and, most recently, the `Kara` [19] plugin of `SeaLion`.

To derive certain consistency constraints for ensuring data-model compliance with the program code, Sureshkumar et al. [15] use dependency graphs while Konczak et al. [20] use colored graphs.

Similar approaches have been introduced by Kehrer and Neumann [21] and by Amalfi and Provetti [22], where it is proposed to derive logic programs from *extended ER diagrams* (EER diagrams). The intention behind that work was not to support the development of ASP programs, but to use the inference mechanisms of logic programming to reason about the instances of the database.

`VisualASP` [23] and `ASPIDE` [16] offer an environment for the graphical specification of answer-set programs by providing an editor for directly visualising the ASP concepts. VIDEAS, in contrast, takes advantage of the abstraction power of the ER diagram and adopts the query by a diagram approach (cf. the survey article by Catarci et al. [24]) for program specification. On the other hand, a fact generator concept as given by Chirila et al. [25]—a language-independent generator of facts for logics—is transferred to ASP.

Overall, we are not aware of any solutions for introducing modelling techniques into the ASP context as we discuss in the next sections.

3 Preliminaries

In this section, we recapitulate the basics of ASP. For more details, we refer to the literature [1]. An *answer-set program*, Π, or *program* for short, is defined over an alphabet

[7] `http://igrom.sourceforge.net/`.

[8] `https://dtai.cs.kuleuven.be/krr/software/visualisation`.

$\mathcal{A}_{\mathsf{ASP}} = (P, C, V)$, where P, C, and V are finite and non-empty sets of *predicate symbols*, *constant symbols*, and *variables*, respectively. Each predicate symbol is assumed to have a unique natural number assigned, called its *arity*. A *term* is either a variable from V or a constant from C. An *atom* is an expression of the form $p(t_1, \ldots, t_n)$, where p is a predicate symbol of arity $n \geq 0$ from P, and $t_1, \ldots, t_n$ are terms. A *literal* is either an atom p or a negated atom $\neg p$. The symbol "$\neg$" is referred to as *strong negation*. Besides strong negation, there is a second, weaker form of negation, called *default negation*, written "not". Intuitively, $not\ p$ states that p cannot be proved, where p is a literal. That is, default negation corresponds to *negation as failure*.

A *rule*, r, is an expression of the form

$$h :- b_1, \ldots, b_k, not\ b_{k+1}, \ldots, not\ b_n,$$

where $h, b_1, \ldots, b_n$ are literals with $n \geq 0$. The informal meaning of rule r is that h is asserted whenever $b_1, \ldots, b_n$ are derivable whilst $b_{k+1}, \ldots, b_n$ are *not* derivable. The literal h is called the *head* of r and $b_1, \ldots, b_k, not\ b_{k+1}, \ldots, not\ b_n$ form the *body* of r. A rule is a *fact* if its body is empty, and a *constraint* if its head is empty. The intuitive reading of a constraint is that it is forbidden that all literals $b_1, \ldots, b_k$ are true but none of $b_{k+1}, \ldots, b_n$ are true. For facts, usually the symbol "$:-$" is omitted. We refer to rules which are neither constraints nor facts as *proper rules*. An atom, a literal, or a rule is called *ground* if it does not contain any variables. An *answer-set program* is a set of rules. We call a program *positive* if none of its rules contains a default-negated literal. In the following, the language of ASP is referred to as $\mathcal{L}_{\mathsf{ASP}}$.

The semantics of answer-set programs is defined in terms of consistent sets of literals representing three-valued interpretations (true, false, undefined). Programs without default negation are associated with their least satisfying consistent set of literals. If no such set can be found, the program is inconsistent. The semantics of programs containing default negation is defined by a reduction to the least model semantics via the *Gelfond-Lifschitz transformation*. For a formal definition of the semantics, we refer to the textbook by Baral [1].

Example 1 (Three-Colouring Problem). For a given graph consisting of a set of vertices and a set of edges, a three-colouring is an assignment from the vertices into three colours, say red, green, and blue, such that no two adjacent vertices are assigned the same colour. Deciding whether such a three-colouring exists is well-known to be NP-complete, yet colourings can be computed by a simple answer-set program as given below, containing just three rules and one constraint. The facts are necessary for the representation of the graph, i.e., for the *input*.

$$\begin{aligned}
&vtx(a).\ vtx(b).\ vtx(c).\ edge(a,b).\ edge(a,c).\ \ldots \\
clrd(V, red) &:- vtx(V), not\ clrd(V, green), not\ clrd(V, blue). \\
clrd(V, green) &:- vtx(V), not\ clrd(V, red), not\ clrd(V, blue). \\
clrd(V, blue) &:- vtx(V), not\ clrd(V, red), not\ clrd(V, green). \\
&:- edge(V, U), clrd(V, C), clrd(U, C).
\end{aligned}$$

The proper rules state that a node is assigned a certain colour if it is not assigned any of the other two colours. The constraint guarantees that two adjacent nodes never have the same colour. The colours are represented by constants red, $green$, and $blue$.

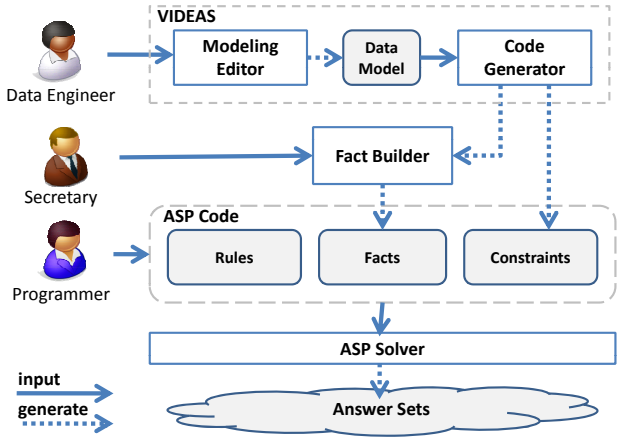

Fig. 1. The development process.

4 Answer-Set Programming with VIDEAS

As we have seen in the previous section, answer-set programs consist of three parts: (i) the facts, representing the given knowledge, i.e., the database, (ii) the proper rules, which allow for the inference of new knowledge, and (iii) the constraints, asserting the integrity of the given as well as of the derived knowledge. The facts together with the constraints may be used to simulate the functionality offered by a relational database, whilst the deductive rules introduce reasoning facilities. At the moment, all of these three blocks involve code-based program development requiring specific ASP knowledge.

In *model-driven engineering* (MDE) [26], models serve as primary development artefacts from which code can be generated. Within the development process, models are more than mere documentation items as in traditional software engineering. Besides the fact that graphical visualisation is in general easier understandable for the human software engineer and programmer, models may be automatically transformed into executable code. Consequently, inconsistencies between the models and the code can be avoided. These advantages are used in VIDEAS to support the definition of facts and to generate constraints ensuring the data consistency of answer-set programs. In particular, ER diagrams are used as starting point for the program development.

In the VIDEAS framework, answer-set programs are constructed by three tasks: (i) modelling, (ii) building a fact base, and (iii) implementation of the program. The different tasks may be accomplished by people with different backgrounds. Specific knowledge about ASP is only required in the third step. Figure 1 gives an illustration of the overall development process with VIDEAS.

Modelling. In the first step, an ER diagram is specified using a graphical modelling editor that is part of the VIDEAS system (a screenshot of the editor is depicted in Fig. 3). The diagram describes entities and relations between entities of the problem

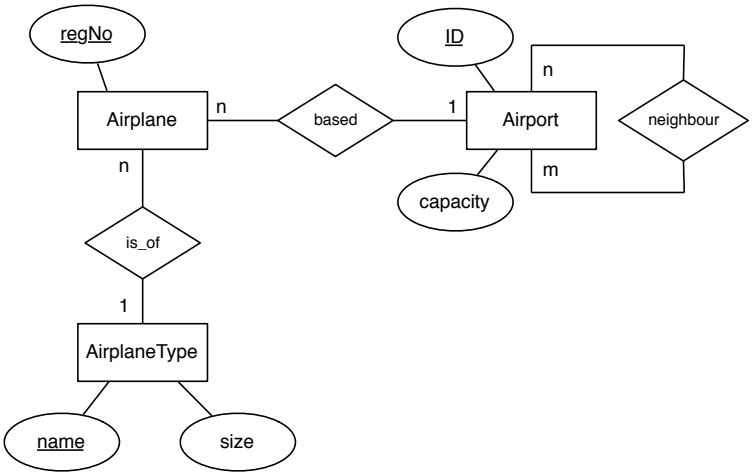

Fig. 2. Example of an ER diagram.

domain under consideration. From the ER diagram, several kinds of constraints can be derived automatically which may be integrated in the answer-set program to assure consistency of the program's answer sets with the data model and to ease debugging.

Building a fact base. After the modelling phase, the FactBuilder component allows to safely enter data by means of facts. The FactBuilder tool ensures that the entered data is consistent with the ER model. The resulting fact base may serve as an assertional knowledge base for the answer-set program. It is also possible to enter the data at a later point in time or to define multiple knowledge bases which increases the versatility of problem representations. Figure 5 gives an example exploiting the FactBuilder tool.

Implementation. Finally, the program under development has to be completed. That is, all properties of problem solutions beyond the constraints imposed by the ER diagram have to be formalised in ASP. VIDEAS does not impose any restrictions on answer-set programmers concerning the implementation step but rather provides assistance for some parts of the development process by offering modelling and visualisation techniques as well as the automated generation of constraint systems.

Before we describe how we realised the VIDEAS system, we discuss how the mapping between ER diagram and ASP is accomplished.

5 Mapping ER Diagrams to Answer-Set Program Code

An explicit mapping strategy for transforming ER diagrams to answer-set programs is required to allow an automated code generation. In this section, we introduce a formal description of ER diagrams which we map to ASP. The ER diagram shown in Fig. 2

serves as illustrating example which will be represented as an answer-set program. Note that we use only standard concepts of ASP as described in Section 3, but no solver-specific language extensions. Furthermore, for the sake of simplicity, we consider here only a subset of the language of ER diagrams—in particular, we do not consider arbitrary cardinality constraints.

Let $\mathcal{L}_{\mathsf{ER}}$ be the language of ER diagrams defined over an alphabet $\mathcal{A}_{\mathsf{ER}}=(R, A, D)$, where R denotes a set of *relations*, A denotes a set of *attributes*, and D is a non-empty set called *domain*. Each attribute $a \in A$ is assigned with its associated domain $d(a)$, which is a subset of D, i.e., $d(a) \subseteq D$. In the ER diagram of Fig. 2, `Airplane/1` and `based/2` are examples for relations, `ID` is an example for an attribute, and the numbers, referred to as `integer/1`, are elements of the domain.

In view of our purposes, we define an ER diagram $\Omega \in \mathcal{L}_{\mathsf{ER}}$ as a triple $(\mathsf{E}, \mathsf{R}, \mathsf{k})$, consisting of a set E of *entities*, a set R of *relationships*, and a function $\mathsf{k} : \mathsf{E} \to \mathbb{N}$ defining the primary key for each entity. In the following, we introduce these concepts iteratively and provide a mapping $\sigma_\alpha : \mathcal{L}_{\mathsf{ER}} \times \mathcal{I} \to \mathcal{L}_{\mathsf{ASP}}$ from the language $\mathcal{L}_{\mathsf{ER}}$ of ER diagrams, along with the set $\mathcal{I}$ of all instances of entities, to the language $\mathcal{L}_{\mathsf{ASP}}$ of answer-set programs, parametrised by a mapping α relating the respective alphabets, which will be formally introduced first.

Definition 1 (Symbol Mapping). *Given a language $\mathcal{L}_{\mathsf{ER}}$ over an alphabet $\mathcal{A}_{\mathsf{ER}} = (R, A, D)$ and a language $\mathcal{L}_{\mathsf{ASP}}$ over an alphabet $\mathcal{A}_{\mathsf{ASP}} = (P, C, V)$, a* symbol mapping *is an injective function $\alpha : R \cup A \cup D \to P \cup C \cup V$ such that* (*i*) *for any $r \in R$, $\alpha(r) \in P$, where r and $\alpha(r)$ have the same arity,* (*ii*) *for any $a \in A$, $\alpha(a) \in V$, and* (*iii*) *for any $d \in D$, $\alpha(d) \in C$.*

Hence, according to this definition, in an answer-set program, relations are represented by predicate symbols, attributes are represented by variables, and elements of the domain are represented by constants. Furthermore, a symbol mapping is stipulated to be injective, which means that for all elements x, y of its domain, we have that $x \neq y$ implies $\alpha(x) \neq \alpha(x)$. From the definition of a function, this implies in turn that $x \neq y$ holds precisely in case $\alpha(x) \neq \alpha(y)$ holds. Thus, all terms are uniquely identifiable.

Definition 2 (Entity). *Let $\mathcal{A}_{\mathsf{ER}} = (R, A, D)$ be an alphabet. An* entity *over $\mathcal{A}_{\mathsf{ER}}$ is an expression of the form $e(a_1, \ldots, a_n)$ with $e \in R$ and $a_1, \ldots, a_n \in A$ such that, for every attribute a_i and a_j, $1 \leq i, j \leq n$, it holds that $a_i \neq a_j$ iff $i \neq j$. An* instance *of an entity $e(a_1, \ldots, a_n)$ is an expression of the form $e(w_1, \ldots, w_n)$, where $w_i \in d(a_i)$, for every $1 \leq i \leq n$. The term w_i is called a* value *of attribute a_i.*

An entity is hence characterised by its attributes which in turn range over a given domain. In an ER diagram, an entity is visualised by a rectangle containing the name of the entity and the attributes are represented as ellipses which are attached to the entity they belong to. In Fig. 2, `Airport(ID, capacity)` and `Airplane(regNo)` are examples of entities. Instances are not part of an ER diagram, but they can be seen as actual entries of a database which is realised according to the schema defined by the ER diagram. For instance, `Airport(''vienna'', 500)` would be an example for an instance in the ER diagram of Fig. 2.

The entities as introduced above will be used in VIDEAS for the specification of the FactBuilder which supports the creation of instances which are included in the answer-set program in terms of facts. The entities will thus be mapped to ASP predicates whilst their instances are mapped to facts.

Definition 3 (Mapping of Entities). *Let $\mathcal{A}_{\mathsf{ER}} = (R, A, D)$ be an alphabet and α a symbol mapping defined over $\mathcal{A}_{\mathsf{ER}}$. Let furthermore $\mathcal{E}$ be the set of all entities over $\mathcal{A}_{\mathsf{ER}}$. An* entity mapping *relative to α is a function $\epsilon_\alpha : \mathcal{E} \rightarrow \mathcal{L}_{\mathsf{ASP}}$ such that*

$$\epsilon_\alpha(e(a_1, \ldots, a_n)) = \alpha(e)(\alpha(a_1), \ldots, \alpha(a_n)),$$

for every entity $e(a_1, \ldots, a_n) \in \mathcal{E}$.

Definition 4 (Mapping of Instances of Entities). *Let $\mathcal{A}_{\mathsf{ER}} = (R, A, D)$ be an alphabet and α a symbol mapping defined over $\mathcal{A}_{\mathsf{ER}}$. Let furthermore $\mathcal{I}$ be the set of all instances of entities over $\mathcal{A}_{\mathsf{ER}}$. An* instance mapping *relative to α is a function $\iota_\alpha : \mathcal{I} \rightarrow \mathcal{L}_{\mathsf{ASP}}$ such that, for every instance $e(w_1, \ldots w_n) \in \mathcal{I}$,*

$$\iota_\alpha(e(w_1, \ldots, w_n)) = \alpha(e)(\alpha(w_1), \ldots, \alpha(w_n)).$$

According to this definition, the entity instance `Airport(``vienna'', 500)` is mapped to the fact $airport(vienna, 500)$.

Since we have no type system available in general, which asserts that the attributes are correctly instantiated, we therefore have to simulate this in ASP, as described in what follows. This is of particular importance if facts are not only created automatically but if they are also added manually.

Definition 5 (Mapping Domain Constraints). *Let $\mathcal{A}_{\mathsf{ER}} = (R, A, D)$ be an alphabet and α a symbol mapping defined over $\mathcal{A}_{\mathsf{ER}}$. Let furthermore $\mathcal{E}$ be the set of all entities over $\mathcal{A}_{\mathsf{ER}}$. A* domain constraint mapping *relative to α is a function $\delta_\alpha : \mathcal{E} \rightarrow \mathcal{L}_{\mathsf{ASP}}$ such that, for every entity $e(a_1, \ldots, a_n) \in \mathcal{E}$,*

$$\begin{aligned}
&\delta_\alpha(e(a_1, \ldots, a_n)) = \\
&\quad \textstyle\bigcup_{1 \leq i \leq n} \{ domainCheck_e_a_i(\alpha(a_i)) \mathbin{:-} \alpha(e)(_, \ldots, \alpha(a_i), \ldots, _), \\
&\qquad\qquad\qquad\qquad\qquad\qquad\quad domain_a_i(\alpha(a_i)); \\
&\qquad\qquad\qquad\qquad\qquad\quad \mathbin{:-} not\ domainCheck_e_a_i(\alpha(a_i)), \\
&\qquad\qquad\qquad\qquad\qquad\qquad\quad \alpha(e)(_, \ldots, \alpha(a_i), \ldots, _) \},
\end{aligned}$$

where $domainCheck\_e\_a_i$ is a new predicate symbol used for asserting that instances of the entity e have valid values for attribute a_i and $domain\_a_i$ describes that $\alpha(a_i)$ is in $\{\alpha(w_j) | w_j \in d(a_i)\}$.

Note that the final program needs to contain facts for $domain\_a_i$. Furthermore, predicate $domain\_a_i$ may be substituted by predefined predicates like $integer$ for certain attributes a_i. In the case of Example 1, for instance, it is defined by an explicit enumeration of the possible colour values.

Definition 6 (Primary Key). *Let $\Omega = (\mathsf{E}, \mathsf{R}, \mathsf{k})$ be an ER diagram. Then, the* primary key *of an entity $e(a_1, \ldots, a_n) \in \mathsf{E}$ is the k-tuple $pk(e(a_1, \ldots, a_n)) = (a_1, \ldots, a_k)$, where $k = \mathsf{k}(e(a_1, \ldots, a_n))$ for the function $\mathsf{k} : \mathsf{E} \rightarrow \mathbb{N}$.*

Intuitively, the aim of a primary key is to uniquely identify instances of an entity. That is, a primary key $pk(e(a_1, \ldots, a_n)) = (a_1, \ldots, a_k)$ should entail that for every set S of instances of the entity $e(a_1, \ldots, a_n)$, if $s = e(w_1^s, \ldots, w_k^s, w_{k+1}^s, \ldots, w_n^s)$ and $t = e(w_1^t, \ldots, w_k^t, w_{k+1}^t, \ldots, w_n^t)$ are two instances in S, then $(w_1^s, \ldots, w_k^s) = (w_1^t, \ldots, w_k^t)$ implies $s = t$. In the graphical representation of ER diagrams, primary key attributes are underlined. In the translation to ASP, we want to have rules asserting that no two entities with the same primary key exist which are different in the other attributes.

Definition 7 (Mapping Primary Keys.). *Let $\mathcal{A}_{\mathsf{ER}} = (R, A, D)$ be an alphabet and α a symbol mapping defined over $\mathcal{A}_{\mathsf{ER}}$. Let furthermore $\mathcal{E}$ be the set of all entities over $\mathcal{A}_{\mathsf{ER}}$. A* primary key mapping *relative to α is a function $\kappa_\alpha : \mathcal{E} \to \mathcal{L}_{\mathsf{ASP}}$ such that, for every entity $e(a_1, \ldots, a_k, \ldots, a_n)$ with primary key $(a_1, \ldots, a_k)$,*

$$\begin{aligned}
&\kappa_\alpha(e(a_1, \ldots, a_k, \ldots, a_n)) = \\
&\quad \textstyle\bigcup_{k+1 \leq i \leq n} \{ pkViolation_e_a_i(\alpha(a_1), \ldots, \alpha(a_k)) :- \epsilon_\alpha(e), \\
&\qquad\qquad \alpha(e)(\alpha(a_1), \ldots, \alpha(a_k), v_{k+1}, \ldots, v_i, \ldots, v_n), \alpha(a_i) \neq v_i; \\
&\qquad\quad :- pkViolation_e_a_i(v_1, \ldots, v_k), \alpha(e)(v_1, \ldots, v_k, _, \ldots, _) \},
\end{aligned}$$

where v_i are new ASP variables.

Having the mapping of primary keys at hand, we are now able to establish the mapping of relationships. As the following definition shows, a relationship is—like an entity—defined by the means of relations.

Definition 8 (Relationship). *Let $e_1, \ldots, e_n$ be entities over $\mathcal{A}_{\mathsf{ER}} = (R, A, D)$ with $pk(e_i) = (a_1^i, \ldots, a_{k_i}^i)$, for $1 \leq i \leq n$. Then, a* relationship *between $e_1, \ldots, e_n$ is given by an expression of form $r(a_1^1, \ldots, a_{k_1}^1, \ldots, a_1^n, \ldots, a_{k_n}^n)$, where $r \in R$. Relationships are instantiated the same way as entities.*

For example, in Fig. 2, `is_of` and `based` are relationships. Relationships are relations like entities, hence the instance mapping for relationships to $\mathcal{L}_{\mathsf{ASP}}$ is given by ι_α. However, we have to assert that the entities referred to in a relationship are actually existing which is achieved by the following rules. Note that cardinality restrictions are expressed accordingly.

Definition 9 (Mapping Key Reference Constraints). *Let $\mathcal{A}_{\mathsf{ER}} = (R, A, D)$ be an alphabet and α a symbol mapping defined over $\mathcal{A}_{\mathsf{ER}}$. Let furthermore $\mathcal{R}$ be the set of relationships over $\mathcal{A}_{\mathsf{ER}}$. A* key reference constraint mapping *relative to α is a function $\gamma_\alpha : \mathcal{R} \to \mathcal{L}_{\mathsf{ASP}}$ such that, for every relationship $r(a_1^1, \ldots, a_{k_1}^1, \ldots, a_1^n, \ldots, a_{k_n}^n)$,*

$$\begin{aligned}
&\gamma_\alpha(r(a_1^1, \ldots, a_{k_1}^1, \ldots, a_1^n, \ldots, a_{k_n}^n)) = \\
&\quad \bigcup_{1 \leq i \leq n} \{ refKeyCheck_r_e_i(\alpha(a_1^i), \ldots, \alpha(a_{k_i}^i)) :- \alpha(e_i)(\alpha(a_1^i), \ldots, \alpha(a_{k_i}^i), _, \ldots, _), \\
&\qquad\qquad \alpha(r)(_, \ldots, _, \alpha(a_1^i), \ldots, \alpha(a_{k_i}^i), _, \ldots, _); \\
&\qquad\quad :- not\ refKeyCheck_r_e_i(\alpha(a_1^i), \ldots, \alpha(a_{k_i}^i)), \\
&\qquad\qquad \alpha(r)(_, \ldots, _, \alpha(a_1^i), \ldots, \alpha(a_{k_i}^i), _, \ldots, _) \},
\end{aligned}$$

where e_i is an entity with $pk(e_i) = (a_1^i, \ldots, a_{k_i}^i)$ and r is a relationship between entities $e_1, \ldots, e_n$.

Now, we have introduced all components for defining a mapping from ER diagrams to answer-set programs which is summarised by the following definition:

Definition 10. *Let $\mathcal{A}_{\mathsf{ER}}$ be an alphabet and α a symbol mapping over $\mathcal{A}_{\mathsf{ER}}$. Furthermore, let $\mathcal{I}$ be the set of all instances of entities over $\mathcal{A}_{\mathsf{ER}}$. Then, $\sigma_\alpha : \mathcal{L}_{\mathsf{ER}} \times \mathcal{I} \to \mathcal{L}_{\mathsf{ASP}}$ is the partial function mapping an ER diagram $\Omega = (\mathsf{E}, \mathsf{R}, \mathsf{k})$ over alphabet $\mathcal{A}_{\mathsf{ER}}$ and a set $\mathsf{I} \subseteq \mathcal{I}$ of instances into a program over language $\mathcal{L}_{\mathsf{ASP}}$ given by*

$$\sigma_\alpha(\Omega, \mathsf{I}) = \bigcup_{i \in \mathsf{I}} \{\iota_\alpha(i)\} \cup \bigcup_{e \in \mathsf{E}} \kappa_\alpha(e) \cup \bigcup_{r \in \mathsf{R}} \gamma_\alpha(r) \cup \bigcup_{e \in \mathsf{E}} \delta_\alpha(e),$$

providing I contains only instances of entities in E, otherwise $\sigma_\alpha(\Omega, \mathsf{I})$ is undefined.

With the function σ_α at hand, we are able to translate the information given by an ER diagram along with instances provided by the FactBuilder to a corresponding answer-set program. Note that the full program developed during the modelling phase may need to make use of entities in terms of their translation under ϵ_α. In the following section, we discuss how σ_α is actually implemented in VIDEAS.

6 Implementation

6.1 General Aspects

With VIDEAS, we provide a framework which supports the development of answer-set programs by providing an editor for creating ER diagrams from which ASP code may be derived as discussed in the previous section. In particular, constraints are automatically created. To establish the database, i.e., for entering the instances of the ER diagram, manual user input is required. Based on the constraints specified in the ER diagram, the FactBuilder tool is automatically generated which provides an input mask for specifying of the actual data. It asserts that the constraints are obeyed by the user.

VIDEAS has been implemented on top of the Eclipse platform[9]; the implementation is available at

`https://subversion.assembla.com/svn/videas/.`

In particular, technologies provided by the Eclipse Modeling Framework (EMF)[10] and the Graphical Modeling Framework (GMF)[11] projects have been used. The meta-model representing the ER diagram modelling language has been created using the Ecore modelling language which is specified within the EMF project. Based on this Ecore model (an implementation for a subset of the Meta Object Facility[12]), a graphical editor has been created using GMF. The models created by the graphical model editor are stored as Ecore XML, models being accessible by EMF libraries, which is a prerequisite for the code generation. Visually, n-ary relationships, for arbitrary n, can be

[9] `https://www.eclipse.org`.
[10] `http://www.eclipse.org/modeling/emf/`.
[11] `http://www.eclipse.org/modeling/gmf/`.
[12] `http://www.omg.org/mof/`.

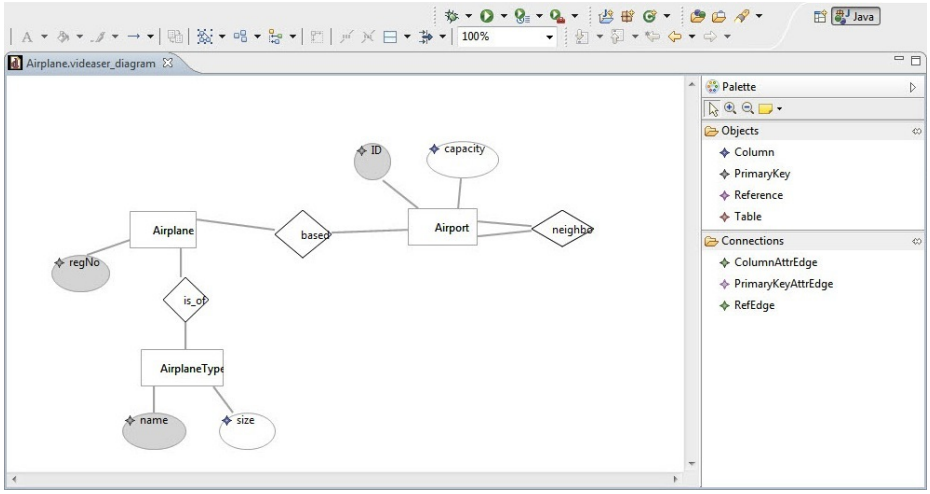

Fig. 3. Screenshot of the ER editor.

constructed, whereas the code generator will only transform binary and unary relationships at present.

The analysis of the constraints and the code generation are implemented in Java using Eclipse EMF (for deserialising models). This reduces the effort for generating recurrent code blocks, i.e., literals based on the same entity type or even instance are represented as *literal template*. The template information is stored in form of objects in hash tables, which contain ordered argument sequences (and their properties) for entities. These sequences already respect the transformation strategy of entities and relationships, as discussed in Section 5.

In particular, the code generator processes the models from the graphical editor. Again, this model is formulated in Ecore. The code generation itself can be grouped into three subsequent activities:

1. The model is analysed in order to compute and store the used literal template.
2. Type and primary key constraints are generated (cf. Fig. 4 for an example).
3. Input forms are prompted which enable a developer to fill in values that are used for inserting the data—the FactBuilder of VIDEAS (cf. Fig. 5).

The FactBuilder component also implements features like the automated look-up of values from a known domain. Finally, the constructed facts and constraints may be serialised to an answer-set program code file. In the VIDEAS prototype, all constraints and facts are serialised when the user quits the program.

In the following, we first discuss how constraints are derived from the ER diagram and transferred to ASP code. Then, we give a short overview on the generation of the FactBuilder.

```
%%%%%% PRIMARY KEY CONSTRAINT

pkViolation_airport_capacity(ID) :- airport(ID, Capacity),
                                    airport(ID, Capacity2),
                                    Capacity != Capacity2.
:- pkViolation_airport_capacity(ID), airport(ID,_).

%%%%%% REFERENCE TYPE CONSTRAINT

refKeyCheck_isOf_airplaneType(TypeName) :-
                              airplaneType(TypeName,_),
                              airplane(_,TypeName,_).

:- not refKeyCheck_isOf_airplaneType(TypeName),
                              airplane(_,TypName,_).

%%%%%% DOMAIN CONSTRAINT

domainCheck_airport_capacity(Capacity) :- airport(_, Capacity),
                                          integer(Capacity).
:- not domainCheck_airport_capacity(Capacity),
   airport(_, Capacity).
```

Fig. 4. Excerpt of constraints generated from an ER diagram.

6.2 Generation of Constraints

VIDEAS derives multiple types of constraints like primary key constraints and domain constraints from the ER diagram which are expressed in terms of rules in the resulting answer-set program, as formally discussed in the previous section.

Primary Key Constraints. For the generation of primary key constraints, the primary keys are retrieved from the literal template and the mapping from the previous section is applied. Figure 4 shows an example of a primary key constraint in an answer-set program. The first rule evaluates to true if there exist two instances of the entity `airport` which have the same values in the key attributes but which differ in their other attributes. Irrelevant attributes are marked by the symbol "_" in this rule. Using the same variables in the two body literals for the key attributes asserts that the values of the key are the same. The constraint asserts that no answer sets are allowed where the primary key constraint is violated.

Type Constraints. The type consistency for attributes can be validated by defining an artificial intermediary type avoiding unintentional specifications of values. For this purpose, two rules are generated for each attribute. The first rule ensures the correctness of the attribute by validating whether the attribute has the expected type. The second rule eliminates answer sets comprising attributes violating this condition. In Fig. 4, a

```
:add airplane
  regNr: 1
  airplaneType.name: Boeing737
  airport.ID: ap1

 % RESULTING FACT
 airplane(1,Boeing737,ap1).
```

Fig. 5. An example for the FactBuilder component.

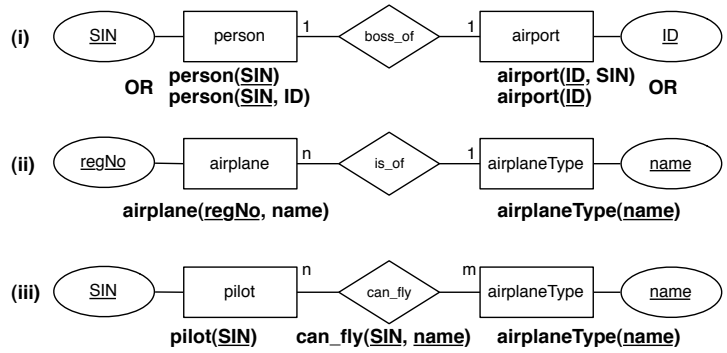

Fig. 6. The cardinalities of relationships of ER diagrams.

generated type constraint for the attribute `capacity` of the entity `airport` is given. Each `airport` requires this attribute as being of the type `integer`. For relationships, the key constraints are defined accordingly.

6.3 Generation of Entity Instances

For supporting the data input, for each ER diagram, the FactBuilder tool is automatically generated. The FactBuilder is a command-line tool which checks that the constraints specified in the ER diagram are obeyed when entity instances as well as relationship instances are entered. For example, in Fig. 5, an instance of an airplane is entered. Note that the $1 : n$ relationship to the airplane type is already taken into account.

Wildcards (denoted by "?" as input value for an argument) assist the developer in resolving primary key values of referred entities. Each wildcard opens a dialogue allowing the ad-hoc specification of referred entity instances. The primary keys of the newly specified instance is automatically merged with the previously entered values forming the arguments of the created fact. Circular dependencies can be resolved manually. We aim for extending the VIDEAS tool for allowing the automated resolution of references.

In fact, describing an instance of the entity `airplane`, the key value to the referenced airplane type is directly included in the arguments of airplane. This is an optimisation of the mapping of one-to-one and one-to-many relationships. In one-to-one

relationships, both entities (and their instances) are symmetrically dependent on each other, e.g., a `person` can reference an `airport` (an argument in `person` representing the primary key of an `airport`), but an `airport` can also reference a `person` (cf. Fig. 6). In order to avoid redundancies, a double-sided linking is avoided, leading to two potential solution variants. In VIDEAS, one entity is randomly picked to which a reference to the primary key attributes of the other entity is added. For the mapping of one-to-many relationships, the primary key value of the entity with cardinality one is added to every entity instance of the other entity. Note that giving one variant preference over the other does not result in any loss of information concerning navigability due to the inference mechanisms of ASP. In contrast, for many-to-many relationships, two instances are linked with the help of a third additional predicate. This corresponds exactly to the mapping described in the previous section.

7 Conclusion and Future Work

VIDEAS exploits techniques standard in the discipline of model-driven engineering (MDE). It focuses on supporting the answer-set programmer during the development phase by establishing visual program construction methods and automated code generators. The distinguishing feature of MDE is that models are first-class citizens in the engineering process rather than mere documentation artefacts.

In VIDEAS the programmers are encouraged to construct their programs as data models with the help of ER diagrams before implementing a program. The benefit of an explicit model is that the developer's attention can be drawn to design decisions, e.g., naming strategies and relevant relationships, rather than to code specifics. Such models even serve as basis for the automated generation of constraints, which target common pitfalls of entity interrelations which are difficult to be addressed by hand—e.g., an argument value referencing to a wrong entity. Another advantage is the ability for automatically generating input masks for the consistent definition of a fact base for an answer-set program. Implicit constraints represented by the ER model, e.g., cardinalities, can be automatically ported into the language of answer-set programs. As a consequence, all these VIDEAS contributions jointly target to bypass common pitfalls, to improve the quality of the program code (i.e., explicit designing of literals), and to reduce efforts for constructing the program basis (i.e., assistance by a code generator). The generated code—consisting of facts and constraints—does not expose any restriction on the further development process. Arbitrary rules can be used for complementing the returned code base, and each generated rule can be textually modified or post-processed by traditional techniques.

For future work, we intend to consider additional model concepts like inheritance relationship as well as the support for other modelling languages, e.g., subsets of the UML class diagram. The latter may in particular be a beneficial candidate for answer-set programs, as the language-inherent extension mechanism of UML profiles may be used to adapt the UML class diagram to our specific purposes. We also plan to extend the VIDEAS framework to visualise potential inconsistencies between answer-sets of a program and the data model directly at the level of the underlying ER diagram. Another focus for further work is intended to be set around advanced language features such

as relationships involving an arbitrary number of entities. Finally, the full potential of VIDEAS is exploited if it is included in an integrated development environment like ASPIDE [16] or SeaLion [17].

References

1. Baral, C.: Knowledge Representation, Reasoning, and Declarative Problem Solving. Cambridge University Press, Cambridge, England (2003)
2. Chen, P.: The entity-relationship model—Toward a unified view of data. ACM Transactions on Database Systems **1**(1) (1976) 9–36
3. Ricca, F., Leone, N.: Disjunctive logic programming with types and objects: The DLV$^+$ system. Journal of Applied Logic **5**(3) (2007) 545–573
4. Oetsch, J., Pührer, J., Seidl, M., Tompits, H., Zwickl, P.: VIDEAS: A development tool for answer-set programs based on model-driven engineering technology. In: Proc. LPNMR'11. Volume 6645 of LNCS, Springer (2011) 382–387
5. Syrjänen, T.: Debugging inconsistent answer set programs. In: Proc. NMR'06. (2006) 77–83
6. Wittocx, J., Vlaeminck, H., Denecker, M.: Debugging for model expansion. In: Proc. ICLP'09. Volume 5649 of LNCS, Springer (2009) 296–311
7. Brain, M., De Vos, M.: Debugging logic programs under the answer-set semantics. In: Proc. ASP'05. Volume 142 of CEUR Workshop Proc., CEUR-WS.org (2005) 141–152
8. Brain, M., Gebser, M., Pührer, J., Schaub, T., Tompits, H., Woltran, S.: Debugging ASP programs by means of ASP. In: Proc. LPNMR'07. Volume 4483 of LNCS, Springer (2007) 31–43
9. Gebser, M., Pührer, J., Schaub, T., Tompits, H.: A meta-programming technique for debugging answer-set programs. In: Proc. AAAI'08, AAAI Press (2008) 448–453
10. Oetsch, J., Pührer, J., Tompits, H.: Catching the Ouroboros: On debugging non-ground answer-set programs. Theory and Practice of Logic Programming **10**(4-6) (2010) 513–529
11. Mikitiuk, A., Moseley, E., Truszczynski, M.: Towards debugging of answer-set programs in the language PSpb. In: Proc. ICAI'07, CSREA Press (2007) 635–640
12. Pontelli, E., Son, T.C., El-Khatib, O.: Justifications for logic programs under answer set semantics. Theory and Practice of Logic Programming **9**(1) (2009) 1–56
13. Johannes Oetsch, Jörg Pührer, H.T.: Stepping through an answer-set program. In: Proc. LPNMR'11. Volume 6645 of LNCS, Springer (2011) 134–147
14. Confalonieri, R., Nieves, J.C., Vázquez-Salceda, J.: A preference meta-model for logic programs with possibilistic ordered disjunction. In: Proc. SEA'09. (2009) 19–33
15. Sureshkumar, A., De Vos, M., Brain, M., Fitch, J.: APE: An AnsProlog* environment. In: Proc. SEA'07. (2007) 71–85
16. Febbraro, O., Reale, K., Ricca, F.: ASPIDE: Integrated development environment for answer set programming. In: Proc. LPNMR'11. Volume 6645 of LNCS, Springer (2011) 317–330
17. Oetsch, J., Pührer, J., Tompits, H.: The SeaLion has landed: An IDE for answer-set programming—Preliminary report (2011). Submitted draft.
18. Cliffe, O., De Vos, M., Brain, M., Padget, J.A.: ASPVIZ: Declarative visualisation and animation using answer set programming. In: Proc. ICLP'08. Volume 5366 of LNCS, (2008) 724–728
19. Kloimüllner, C., Oetsch, J., Pührer, J., Tompits, H.: Kara: A system for visualising and visual editing of interpretations (2011). Submitted draft.
20. Konczak, K., Linke, T., Schaub, T.: Graphs and colorings for answer set programming. Theory and Practice of Logic Programming **6**(1-2) (2006) 61–106

21. Kehrer, N., Neumann, G.: An EER prototyping environment and its implementation in a datalog language. In: Proc. ER'92. Volume 645 of LNCS, Springer (1992) 243–261
22. Amalfi, M., Provetti, A.: From extended entity-relationship schemata to illustrative instances. In: Proc. LID'08. (2008)
23. Febbraro, O., Reale, K., Ricca, F.: A visual interface for drawing ASP programs. In: Proc. CILC'10. (2010)
24. Catarci, T., Costabile, M.F., Levialdi, S., Batini, C.: Visual query systems for databases: A survey. J. Visual Languages and Computing **8**(2) (1997) 215–260
25. Chirila, C.B., Jebelean, C., Slavici, T., Cretu, V.: Generating logic representations for programs in a language independent fashion. WSEAS Transactions on Computers **9**(10) (2010) 1201–1211
26. Schmidt, D.C.: Guest editor's introduction: Model-driven engineering. IEEE Computer **39**(2) (2006) 25–31