FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

On Supporting the Development of Answer-Set Programs using Model-driven Engineering Techniques

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieurin

im Rahmen des Studiums

European Master in Computational Logic

eingereicht von

Paula-Andra Busoniu
Matrikelnummer 1128272

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung: a.o. Univ.-Prof. Dr. Hans Tompits
Mitwirkung: Jörg Pührer

Wien, 20.07.2013

(Unterschrift Verfasserin) (Unterschrift Betreuung)

Erklärung zur Verfassung der Arbeit

Paula-Andra Busoniu
Lienfeldergasse 60C 16-17, 1160 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.

(Ort, Datum) (Unterschrift Verfasserin)

Acknowledgements

First of all, I would like to express my sincere and deepest gratitude to my advisor, a.o. Univ.-Prof. Dr. Hans Tompits, for his continuous guidance, patience, and support.

Furthermore, I am especially grateful to Jörg Pührer and Johannes Oetsch who have always provided meaningful inputs, feedback, and suggestions about this work.

I would also like to thank my parents, for their financial and moral support without which I would have never been able to finish this thesis.

Abstract

Answer-set programming (ASP) is an approach for declarative problem solving with roots in the areas of logic programming and knowledge representation. Due to its expressive power and the availability of efficient solvers, it has been successfully applied in several different fields such as knowledge representation and reasoning, constraint satisfaction problems, planning, diagnosis and semantic-web reasoning. Furthermore, in the last years, it was also exploited in the development of industrial applications.

As a result of its growing popularity, besides the theoretical issues and the implementation and improvement of ASP solvers, methodologies and engineering tools to assist the programmer during the development process have also become the focus of ASP research. Although it is widely viewed that answer-set programs are specifications of themselves, producing answer-set programs is not always straightforward. Another problem of answer-set programs is the traditional visualisation of answer sets in a textual manner, which makes the extraction of relevant information a tedious task, particularly for large interpretations. Graphical representations are easier to understand and well-established model-driven engineering techniques and technologies are especially helpful in guiding the ASP development process by graphical models, starting from modelling the problem domain and ending at the visualisation of problem solutions.

In this thesis, we address the issue of the usage of graphical models for the automation of ASP code generation and for the representation of answer sets. The proposed graphical modelling environment is a round-trip tool that provides the user with a graphical editor to represent the problem domain using a subset of the UML class diagram. From the model, a description of an ASP language signature (predicates, arities, types, and meaning of the argument terms) is automatically generated. The signature, together with the constraints of the domain, is expressed in Lana, a meta-language for annotating answer-set programs. Based on the signature in Lana, the programmer proceeds to develop ASP encodings.

After computing the answer sets of the answer-set program, the user is able to visualise the problem solutions by means of UML object diagrams. Violations of constraints in the solutions can be seen either by checking for certain error-indicating atoms in answer sets or can be visualised directly in the graphical representation of the solution.

Kurzfassung

Antwortmengen-Programmierung (engl., “answer-set programming” - ASP) ist ein Ansatz für deklaratives Problemlösen mit Wurzeln in den Bereichen der Logik-Programmierung und Wissensrepräsentation. Aufgrund seiner Ausdruckskraft und der Verfügbarkeit effizienter Solver wurde es erfolgreich in verschiedenen Bereichen eingesetzt, etwa in der Wissensrepräsentation und dem logischen Schließen, für Constraint Satisfaction Probleme, Planungs- und Diagnoseprobleme, sowie für das Semantische Web. Darüber hinaus wird es auch in industriellen Anwendungen verwendet.

Als Folge der wachsenden Popularität von ASP hat sich der Schwerpunkt der Forschung in diesem Gebiet von theoretischen Fragen sowie der Umsetzung und Verbesserung von Solver Technologie zu Fragen der Methodik und der Bereitstellung von Entwicklungswerkzeugen verschoben, um den Programmierer bei der Entwicklung zu unterstützen. Obwohl, allgemein betrachtet, Antwortmengen-Programme als Spezifikationen ihrer selbst angesehen werden können, ist ihre Erstellung nicht immer einfach. Ein weiteres Problem im Umgang mit ASP ist, daß Antwortmengen üblicherweise in Textform dargestellt werden, was die Extraktion relevanter Informationen, insbesondere für große Mengen, erschwert. Grafische Darstellungen sind leichter zu verstehen, und etablierte modellbasierte Entwicklungstechniken und -technologien sind besonders hilfreich für den ASP Entwicklungsprozess mittels grafischer Modelle, beginnend von der Modellierung des Problembereiches und endend mit der Visualisierung von Problemlösungen.

In dieser Masterarbeit geht es um den Einsatz von grafischen Modellen für die Automatisierung von ASP Code-Generierung und der Darstellung von Antwortmengen. Die vorgeschlagene grafische Modellierungsumgebung umfasst ein umfangreiches Tool mit einem grafischen Editor zur Darstellung eines Problems mittels eines Fragmentes des UML Klassendiagramms. Aus dem Modell wird eine Beschreibung einer ASP Sprachsignatur automatisch generiert. Die Signatur, zusammen mit den Einschränkungen der Domäne, werden in Lana, einer Annotationsprache für ASP, beschrieben. Basierend auf der Signatur in Lana kann der Programmierer den ASP Code entwickeln.

Nach der Berechnung der Antwortmengen eines Programms kann der Benutzer die Problemlösung mit Hilfe von UML Objekt-Diagrammen visualisieren. Die Verletzung von Regeln in den Lösungen können entweder durch Überprüfen bestimmter Atome in Antwortmengen oder direkt in der graphischen Darstellung der Lösung erkannt werden.

Contents

1 Introduction **1**
- 1.1 Motivation . . . 1
- 1.2 Related Work . . . 2
- 1.3 Overview of our Results . . . 2
- 1.4 Structure of the Thesis . . . 4

2 Preliminaries **5**
- 2.1 Answer-Set Programming . . . 5
  - 2.1.1 **clasp** and **gringo** . . . 8
  - 2.1.2 The **DLV** System . . . 10
  - 2.1.3 The Integrated Development Environment **SeaLion** for ASP . . . 12
  - 2.1.4 The Annotating Language **Lana** . . . 13
- 2.2 Model-Driven Engineering . . . 15
  - 2.2.1 Unified Modelling Language . . . 15
    - 2.2.1.1 Class Diagrams . . . 16
    - 2.2.1.2 Object Diagrams . . . 18

3 Problem Domain Modelling for ASP **21**
- 3.1 Relevant UML Class Diagram Features . . . 22
  - 3.1.1 Formal Description . . . 23
- 3.2 Translation of UML Class Diagrams to ASP . . . 28
  - 3.2.1 Preprocessing the UML Class Diagram . . . 28
  - 3.2.2 Mapping the UML Class Diagram to ASP . . . 34
  - 3.2.3 **DLV** Modifications . . . 50
- 3.3 Chapter Summary . . . 53

4 Visualising the Problem Solutions **55**
- 4.1 The Modified UML Object Diagram . . . 55
  - 4.1.1 Formal Description . . . 56
- 4.2 Mapping the Problem Solution to a UML Object Diagram . . . 60
  - 4.2.1 Collecting the Instances, Relationships, and Generalisations from an Answer Set . . . 61
  - 4.2.2 Adding Instances and Links . . . 68

4.3 Chapter Summary 75

5 Implementation **77**
5.1 Graphical Editors in Eclipse using the Graphical Modelling Framework 77
5.2 Architecture 78
5.3 UML Class Diagram Editor 80
5.4 The Automatic Translation of UML Class Diagrams to ASP 82
5.5 UML Object Diagram Editor 85
5.6 The Automatic Translation of Interpretations to UML Object Diagrams 88
5.7 Chapter Summary 89

6 Example **91**

7 Conclusions and Related Work **99**

Bibliography **101**

CHAPTER 1

Introduction

1.1 Motivation

Answer-set programming (ASP) is an approach for declarative problem solving with roots in the areas of logic programming and knowledge representation. Due to its expressive power and the availability of efficient solvers, it has been successfully applied in several different fields such as knowledge representation and reasoning [5], constraint satisfaction problems [3], planning [11, 28], diagnosis [4], and semantic-web reasoning [13]. Furthermore, in the last years, it was also exploited in the development of industrial applications [7].

As a result of its growing popularity, the focus of ASP research has currently switched from theoretical issues and the implementation and improvement of ASP solvers towards the development of methodologies and engineering tools to assist the programmer during the coding process. Although it is widely viewed that answer-set programs are specifications of themselves, producing answer-set programs is not always straightforward. Another problem of answer-set programs is that answer sets are usually represented in a textual manner, which makes the extraction of relevant information a tedious task, particularly for large interpretations. A more convenient form of representing answer sets is in a graphical fashion, for which different tools have already been developed in the ASP literature, like the `Kara` system [27]. The latter approach is part of `SeaLion` [33], an integrated development environment (IDE) developed in conjunction of an ongoing research project on methods and methodologies for developing answer-set programs [32]. `SeaLion` incorporates several features known from IDEs for other programming languages, like a debugging component or a documentation generator, ASPDoc, which allows the automatic generation of source-code documentation, in a fashion similar to JavaDoc, based on the Lana annotation language [10]. `SeaLion` and ASPDoc will be discussed in more detail in Sections 2.1.3 and 2.1.4, respectively.

In this thesis, we follow the graphical visualisation method but by using techniques and technologies from *model-driven engineering* [30] for supporting the development of answer-set programs. The main focus of model-driven engineering are *domain models*, at different levels of abstraction, with the purpose of increasing the efficiency by raising the level of abstraction,

and eliminating the errors by using automatic model transformations. In the approach proposed in this thesis, the model-driven engineering methodology is used to guide the ASP development process by graphical models, starting from modelling the problem domain and ending at the visualisation of problem solutions. More specifically, UML class and object diagrams[1] [17] are used in the automation of ASP code generation and the representation of answer sets.

Our goal is to adapt programming tools used in model-driven engineering which had an impact in other paradigms. We believe that the techniques and technologies employed by this approach would improve the quality of the development process and would also attract the less experienced users towards ASP. Furthermore, the use of graphical models and ASPDoc ensures an improved documentation, minimising at the same time the effort needed to realise it.

1.2 Related Work

The implementation of tools for easing the encoding process of answer-set programs has been investigated in recent years. Besides `SeaLion`, major progresses have been realised in terms of IDEs for core ASP languages such as `APE` [38], `ASPIDE` [16], and `iGROM` [25]. The features implemented in the IDEs include highlighting, autocompletion, annotations used to spot the errors, quick fixes, dynamic code templates, and debugging. In addition, `VisualASP` [15] and `ASPIDE` use graphical representations to express ASP concepts, such as dependency graphs.

The visualisation of solutions is also possible in different systems such as `ASPVIZ` [34], `IDPDraw` [24], and `Kara` [27] (the latter also supports the visual editing of interpretations).

Concerning the use of models to support the design of answer-set programs, the first step in this direction has been realised by `VIDEAS` [31]. However, the approach used in `VIDEAS` is different from the work presented in this thesis, starting with the fact that `VIDEAS` automatically generates ASP code, not only the signatures of the programs. Furthermore, instead of using UML class diagrams, `VIDEAS` uses ER diagrams [35] to describe the domain model. The two main advantages of using UML class diagrams are their concise graphical representation which was proven to be more comprehensible [9] and the possibility to integrate them with other UML models, such as UML object diagrams.

1.3 Overview of our Results

The results described in this thesis include, on the one hand, a theoretical framework describing the translation of UML class diagrams to ASP signatures and the representation of answer sets as UML object diagrams, and, on the other hand, a practical implementation, developed in Java. The final outcome is an Eclipse plugin integrated with `SeaLion`.

The overall work cycle our approach is depicted in Figure 1.1. A user, who does not require knowledge about ASP, is provided with a graphical modelling environment that allows to represent the problem domain using a subset of a the UML class diagram. From the model, a description of an ASP language signature (predicates, arities, types, and meaning of the argument terms) is automatically generated. The signature, together with the constraints of the domain,

[1] "UML" is short for *Unified Modelling Language*.

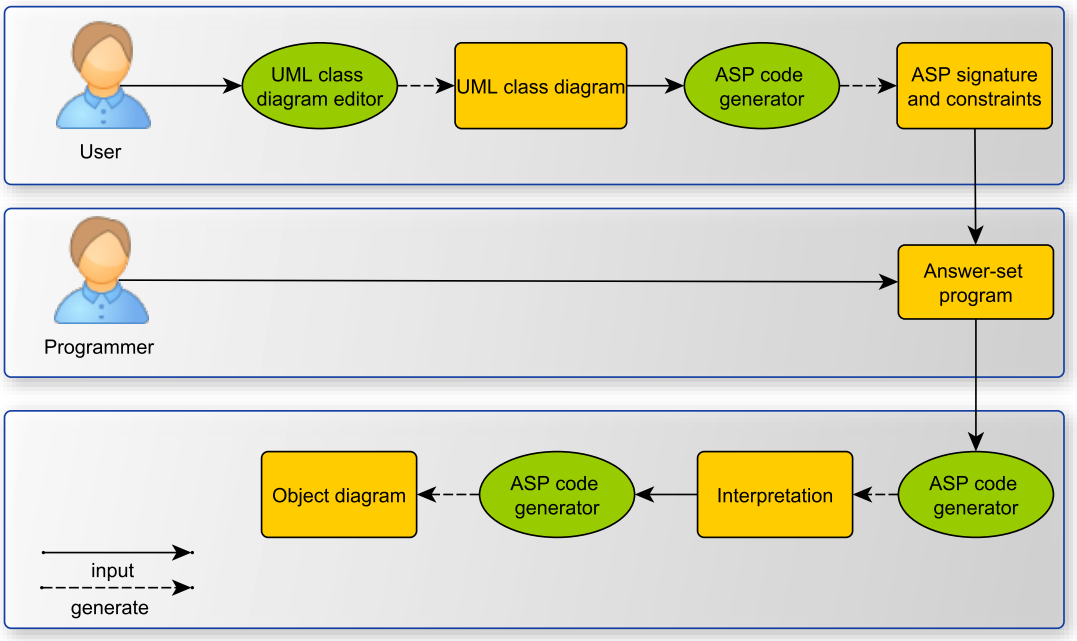

Figure 1.1: The workflow of the encoding process.

are expressed in Lana [10], a meta description language for ASP. The constraints (primary key constrains for instances, cardinality constraints for associations, and completeness or disjointness constraints for generalisations) are expressed as assertions. Based on the signature in Lana, the programmer proceeds to develop ASP encodings.

After computing the answer sets of the answer-set program, the user is able to visualise the problem solutions by means of UML object diagrams. Violation of constraints in the solutions can be seen either by checking for certain error-indicating atoms in answer sets or can be visualised directly in the graphical representation of the solution.

For the implementation of the two graphical editors (namely, the UML class diagram and UML object diagram editors), the technologies provided by the *Eclipse Modelling Framework* (EMF) [37] and the *Graphical Modelling Framework* (GMF) [22] are used. The graphical models created using GMF are stored in Ecore XML models and are accessible by EMF libraries, which are required for the creation and manipulation of the diagrams.

`SeaLion` offers support for the languages of `DLV` and `Gringo` to a large extent and the possibility to use external solvers. The resulting answer sets can be parsed by the IDE and displayed as expandable tree structures in a dedicated Eclipse view for interpretations. Starting from there, the user can invoke the program to choose the interpretation (or a fragment of the interpretation) he or she wants to visualise. For this step, the Ecore XML model representing the UML class diagram must be provided as a blueprint for the instance model.

The most important aspect of the system is the strong connection between the designing and the visualisation phases of the encoding process, allowing the user to visualise the solutions in

terms of the initial specifications of the problem.

In addition, both `Gringo` [18] and `DLV` [6] languages are supported and a certain amount of customisation in the code generation is allowed.

1.4 Structure of the Thesis

In Chapter 2, we present a concise background about answer-set programming and model-driven engineering. Afterwards, in Chapter 3, we offer a formal translation from UML class diagrams to ASP and in Chapter 4 we discuss the mapping of answer sets to UML object diagrams in the context of the initial UML class diagram. Chapter 5 gives an overview about the implementation, and we conclude in Chapter 7, in which we also discuss possible future developments.

CHAPTER 2

Preliminaries

2.1 Answer-Set Programming

Answer-set programming (ASP) is an approach for declarative problem solving with roots in the areas of logic programming and knowledge representation.

Definition 1. An *alphabet* (*for logic programs*) is a triple $\mathcal{A} = \langle \mathcal{P}, \mathcal{V}, \mathcal{C} \rangle$, where $\mathcal{P}$ is a finite non-empty set of *predicate symbols*, $\mathcal{V}$ is a set of *variables*, and $\mathcal{C}$ is a non-empty set of *constant symbols*. An *arity* $n \in \mathbb{N}$ is assigned to each predicate symbol.

By convention, the notation for the predicate p with arity n is p/n. In addition, predicate symbols are denoted by strings starting with a letter, variables are denoted by strings starting with a capital letter, and constants are denoted by numbers or strings starting with a lower case symbol.

Definition 2. Let $\mathcal{A} = \langle \mathcal{P}, \mathcal{V}, \mathcal{C} \rangle$ be an alphabet. The elements of $\mathcal{C} \cup \mathcal{V}$ are called *terms*. A *ground term* is a constant. An *atom* over $\mathcal{A}$ is a string of the form $p(t_1, \ldots, t_n)$, where $p/n \in \mathcal{P}$ and $\{t_1, \ldots, t_n\} \subseteq \mathcal{C} \cup \mathcal{V}$. A *negated atom over* $\mathcal{A}$ is an expression of form $\texttt{not}\ a$, where a is an atom over $\mathcal{A}$ and $\texttt{not}$ is called *default negation*. A *literal over* $\mathcal{A}$ is either an atom or a negated atom over $\mathcal{A}$.

Definition 3. The set of *ground atoms* over a set A of predicates and a set C of constants is given by

$$\mathcal{B}_{A,C} = \{p(c_1, \ldots, c_n) \mid p/n \in A, \{c_1, \ldots, c_n\} \subseteq C\}.$$

Definition 4. A (*disjunctive*) *rule over* $\mathcal{A}$ is an ordered pair of the form

$$h_1 \mid \ldots \mid h_k :- b_1, \ldots, b_n, \texttt{not}\ b_{n+1}, \ldots, \texttt{not}\ b_m, \tag{2.1}$$

where $h_1, \ldots, h_k, b_1, \ldots, b_n, b_{n+1}, \ldots, b_m$ are atoms over $\mathcal{A}$.

For a rule r of form (2.1), the following notation is introduced:

- $head(r) = \{h_1, \ldots, h_k\}$ is the *head* of r,
- $body(r) = \{b_1, \ldots, b_n, \texttt{not}\ b_{n+1}, \ldots, \texttt{not}\ b_m\}$ is the *body* of r,
- $body^+(r) = \{b_1, \ldots, b_n\}$ is the *positive body* of r, and
- $body^-(r) = \{b_{n+1}, \ldots, b_m\}$ is the *negative body* of r.

Definition 5. A rule r of form (2.1) is *non-disjunctive* if $k \leq 1$, *normal* if $k = 1$, *positive* if $body^-(r) = \emptyset$, a *fact* if $body(r) = \emptyset$, and *ground* if r contains no variable. Furthermore r is *safe* if every variable occurring in $head(r) \cup body^-(r)$ also occurs in $body^+(r)$.

For facts, we usually omit the symbol "$:-$".

Definition 6. A *disjunctive logic program*, or simply a *program*, is a finite set of safe rules.

Defining the programs only over safe rules guarantees that no additional constants come into play during the evaluation of a program.

Definition 7. A *non-disjunctive* (resp., *normal*, *positive*) program (over $\mathcal{A}$) is a program (over $\mathcal{A}$) for which every rule in the program is non-disjunctive (resp., normal, positive).

Definition 8. Let e be either a rule or a program. Then,

- $\mathcal{P}_e$ is the set of all predicate symbols occurring in e,
- $\mathcal{V}_e$ is the set of all variables occurring in e, and
- $\mathcal{C}_e$ is the set of all constants occurring in e.

Definition 9. An *interpretation I over $\mathcal{A}$* is a set of ground atoms over $\mathcal{A}$.

Definition 10. An interpretation I over $\mathcal{A}$ *satisfies* a ground atom a if $a \in I$. In this case, we write $I \models a$. If I does not satisfy a, we write $I \not\models a$. Furthermore, I satisfies $\texttt{not}\ a$, symbolically $I \models \texttt{not}\ a$, if $I \not\models a$. I satisfies a set of atoms if I satisfies every atom in the set.

Definition 11. An interpretation I over $\mathcal{A}$ is a *model* of a ground rule r if $I \models body^+(r)$ and $I \cap body^-(r) = \emptyset$ implies $head(r) \cap I \neq \emptyset$. In this case, we write $I \models r$.

The intuition behind an interpretation I is that I represents an assumption about what is true and what is false.

Definition 12 ([19]). The *reduct* of a ground program Π with respect to an interpretation I is the positive program

$$\Pi^I = \{head(r) :- body^+(r) \mid r \in \Pi, I \cap body^-(r) = \emptyset\}.$$

Definition 13. An interpretation I over $\mathcal{A}$ is a *stable model* (or *answer set*) of a disjunctive ground program Π if I is a minimal model of the reduct Π^I, that is, if

- $I \models \Pi$ and
- for each $J \subset I, J \not\models \Pi^I$.

Until now, we have only introduced a way to determine the answer sets of ground programs. In order to define the answer sets of a non-ground program, we need an additional step in which the program is transformed into a ground program.

Definition 14. The *Herbrand universe* of a program Π over $\mathcal{A} = \langle \mathcal{P}, \mathcal{V}, \mathcal{C} \rangle$ is the set

$$\mathcal{HU}(\Pi) = \begin{cases} \mathcal{C}_\Pi, & \text{if } \mathcal{C}_\Pi \neq \emptyset, \\ \{\mathrm{c}\}, & \text{otherwise, with } c \text{ an arbitrary constant in } \mathcal{C}. \end{cases}$$

Definition 15. The *Herbrand base* of a program Π is the set

$$\mathcal{HB}(\Pi) = \mathcal{B}_{\mathcal{P}_\Pi, \mathcal{HU}(\Pi)}.$$

Definition 16. A *substitution over* $\mathcal{A}$ is a partial function $\theta : \mathcal{V} \to \mathcal{C} \cup \mathcal{V}$ mapping variables to terms. Furthermore, if, for each $v \in \mathcal{V}$, $\theta(v) \in \mathcal{C}$, θ is called *grounding*.

Definition 17. Let e be an expression (i.e., an atom, set of atoms, rule, or program) over $\mathcal{A} = \langle \mathcal{P}, \mathcal{V}, \mathcal{C} \rangle$. Then, $e\theta$ is the expression resulting from e by replacing every variable $v \in \mathcal{V}$ in e by $\theta(v)$.

Definition 18. The grounding of a rule r over a set C of constants is the set of ground rules

$$\mathcal{G}_{r,C} = \{r\theta \mid \theta : \mathcal{V}_r \to C\}.$$

The grounding of a program Π over a set C of constants is the ground program

$$\mathcal{G}_{\Pi,C} = \bigcup_{r \in \Pi} \mathcal{G}_{r,C}.$$

Finally, the grounding of a program Π is the ground program

$$\mathcal{G}_\Pi = \mathcal{G}_{\Pi, \mathcal{HU}(\Pi)}.$$

Definition 19. An interpretation I of a program Π is an answer set of Π if I is an answer set of $\mathcal{G}_\Pi^I$.

Note that an answer set of a non-ground program Π is a subset of the Herbrand base of Π.

The set of all answer sets of a program Π is denoted by $AS(\Pi)$. If $AS(\Pi) = \emptyset$, then Π is *inconsistent*, otherwise Π is *consistent*.

By restricting the answer sets to the minimal models, the atoms that are not present in the head of any rule cannot be present in an answer set. Intuitively, every atom in an answer set needs a justification, i.e., an atom cannot be true if there is no rule deriving it. Therefore, all the facts of a program are present in all its answer sets. On the other hand, constraints are used to eliminate solution candidates which do not fulfil certain conditions.

Programs may have zero, one, or more answer sets, as illustrated next.

Example 1. Consider the programs Π_1, Π_2, and Π_3:

$$\begin{aligned}\Pi_1 &= \{p :- \texttt{not}\ p\},\\ \Pi_2 &= \{p \mid q\},\ \text{and}\\ \Pi_3 &= \{p \mid q,\\ &\qquad p :- q,\\ &\qquad q :- p\}.\end{aligned}$$

Their answer sets are given as follows:

$$\begin{aligned}AS(\Pi_1) &= \emptyset,\\ AS(\Pi_2) &= \{\{p\},\{q\}\},\ \text{and}\\ AS(\Pi_3) &= \{\{p,q\}\}.\end{aligned}$$

A system used for computing answer sets is an *ASP solver*. Although the syntax of the disjunctive rules is the same in any system, the solvers provide additional features, such as built-in arithmetic constants, which introduce new syntactic elements. The use and meaning of these new features differ from solver to solver. There exist many efficient solvers, but we will only discuss two of them, namely `clasp` [18] and `DLV` [6], focusing on the differences in the syntax of the additional elements used in the implementation.

2.1.1 `clasp` and `gringo`

We will not discuss all the specifics of the `clasp` and `gringo` system, but only the additional elements used later on—*comments* and *aggregates*. The `gringo` syntax allows two types of comments—line comments and block comments. A line comment starts with "%"; block comments are represented between "%*" and "*%".

Definition 20. An *aggregate function* (in `gringo`) is an operation over multisets of weighted literals. An *aggregate atom* has the following form:

$$l\ \texttt{op}\ [L_1 = w_1, \ldots, L_n = w_n]\ u,$$

where l and u are the *lower* and *upper bound*, respectively, *op* is an aggregate function, and $[L_1 = w_1, \ldots, L_n = w_n]$ is a multiset of weighted literals.

Intuitively, the aggregate predicate is evaluated to true if the result of `op` over the multiset of weights of the true literals is between the bounds l and u (inclusively). Among other aggregate functions, `gringo` supports the aggregate `#sum` which returns the sum of weights.

We are particularly interested in the aggregate function `#count`:

$$l\ \texttt{\#count}\{L_1, \ldots, L_n\}\ u.$$

Informally, it is defined as being equivalent with the aggregate `#sum` with all the weights over the literals set to 1 [18]. The aggregate counts the number of different literals in the multiset $\{L_1, \ldots, L_n\}$.

A more compact notation of this aggregate predicate is

$$l\ \{L_1, \ldots, L_n\}\ u.$$

We formally define the satisfaction of aggregate atoms under an interpretation next. For this purpose, the grounding of rules has to be reconsidered.

Definition 21. A variable X is *local* in an aggregate if it occurs in at least one literal in the aggregate but not in any other non-aggregate atom outside the brackets (but it can occur in any other aggregate inside the same rule). A variable is *global* with respect to a rule if it occurs in at least one non-aggregate atom.

The set of global variables of a rule r is denoted by $\mathcal{V}_r^{\mathrm{g}}$.

Literals in the aggregates can contain constants, local variables, and global variables. Furthermore, the limits must not be present. When the lower limit is not present, it is considered to be 0. When the upper limit is not present, only the lower limit is considered.

Definition 22. In presence of aggregates, the grounding of a rule r over a set C of constants is given by

$$\mathcal{G}_{r,C} = \{r\theta \mid \theta : \mathcal{V}_r^{\mathrm{g}} \to C \text{ and for every bound } l \text{ of an aggregate atom, } l\theta \in \mathbb{N}\}.$$

The grounding of a program contains only rules without global variables and the lower and upper bound of all the aggregate atoms are integers.

Definition 23. An aggregate atom

$$l\ \texttt{\#count}\{L_1 = w_1, \ldots, L_n = w_n\}\ u,$$

containing only local variables, is satisfied by an interpretation I over $\mathcal{A}$, symbolically

$$I \models l\ \texttt{\#count}\{L_1 = w_1, \ldots, L_n = w_n\}\ u,$$

if $l \leq \mid S \mid \leq u$, where S is the set defined as follows:

$$S = \bigcup_{i=1,\ldots,n} \{L_i\theta \mid \exists\theta : \mathcal{V} \to \mathcal{C}, I \models L_i\theta\}.$$

The following example shows the semantics of the aggregate predicate `#count` in `clasp`:

Example 2. Consider the program

$$\begin{aligned}
\Pi = \{&r_1 = p(a),\\
&r_2 = p(b),\\
&r_3 = p(c),\\
&r_4 = l(a,b),\\
&r_5 = l(a,c),\\
&r_6 = l(b,c),\\
&r_7 = int(0),\\
&r_8 = int(X) :- int(X), X < 10, N = X + 1,\\
&r_9 = g(X,N) :- p(X), N\ \{l(X,Y)\}\ N, int(N),\\
&r_{10} = no(N) :- N\ \{l(X,Y)\}\ N, int(N),\\
&r_{11} = sum(N) :- N\ \ \texttt{\#sum}\ \ [g(X,Y) = Y]\ N, int(N)\}.
\end{aligned}$$

Due to the presence of the rules r_7 and r_8, any answer set includes the ground atoms $int(0), \ldots, int(10)$. Rule r_9 ensures that a ground atom $g(X, N)\theta$, where θ is a grounding, is present in an answer set I if $p(X)\theta \in I$, and the cardinality of the set

$$\{l(X\theta, t) \mid \text{there is some } t \in \mathcal{C}_\Pi \text{ such that } l(X\theta, t) \in I\}$$

is $N\theta$. Rule r_{10} ensures that $no(n) \in I$ if the number of ground atoms corresponding to the atoms $l(X, Y)$ and $p(X)$ present in I is equal to n. Furthermore, rule r_{11} ensures that $sum(n) \in I$ if $n = \sum_{g(X,i)\in I} i$. The single answer set is given by

$$\begin{aligned} AS(\Pi) = \{\{&int(0), \ldots, int(10),\\ &p(a), p(b), p(c), l(a, b), l(a, c), l(b, c), g(a, 2), g(b, 1), g(c, 0), no(3), sum(3)\}\}. \end{aligned}$$

2.1.2 The DLV System

DLV [6] is a deductive database system, based on *disjunctive datalog*. Unlike gringo, DLV does not allow block comments. Another difference is that the aggregate function #count has a different syntax and, even more, a different semantics.

Definition 24. An *aggregate function* (in DLV) is of the form fS, where f is a function name among #count, #min, #max, #sum, and #times, and S is called *symbolic set* and has the syntax

$$\{V_1, \ldots, V_n : L_1, \ldots, L_m\},$$

where $V_1, \ldots, V_n$ are local variables and $L_1, \ldots, L_m$ are non-aggregate literals.

Literals in the aggregate constants can contain constants and local and global variables.

Definition 25. An *aggregate atom* (in DLV) is of the form

$$L_g \prec_1 fS \prec_2 R_g,$$

where $f(S)$ is an aggregate function, $\prec_1, \prec_2 \in \{=, <, <=, >, >=\}$, and L_g and R_g (called *left guard* and *right guard*, respectively) are terms.

In the context of aggregates, the safety condition for a rule has to be reconsidered.

Definition 26 ([14]). A rule r is *safe* if the following conditions hold:

1. Each global variable of r appears in a positive standard literal in $body^+(r)$.
2. Each local variable of r that appears in a symbolic set $\{Vars : Conj\}$ also appears in a positive literal in $Conj$.
3. Each guard of an aggregate atom of r is either a constant or a global variable.

In DLV, #count is an aggregate function, returns a number, and has the syntax

$$\texttt{\#count}\ S,$$

where S is a symbolic set.

The grounding of a rule in this context is defined the same as in gringo (see Definition 22).

Definition 27. The result of the aggregate function

$$\texttt{\#count}\{V_1, \ldots, V_n : L_1, \ldots, L_m\}$$

containing only local variables is defined over the interpretation I as the cardinality of the following set

$$\{\langle V_1\theta, \ldots, V_n\theta\rangle \mid \theta : \mathcal{V} \to \mathcal{C}, I \models \{L_1, \ldots, L_m\}\theta\}.$$

Another aggregate function needed in the encoding is #sum, which also returns a number and has the syntax

$$\texttt{\#sum}\ S,$$

where S is a symbolic set.

The aggregate function #sum returns the sum of the first local variable from the symbolic set.

Definition 28. The result of the aggregate function

$$\texttt{\#sum}\{V_1, \ldots, V_n : L_1, \ldots, L_m\}$$

containing only local variables is defined over the interpretation I as the sum of the elements in the set

$$\{v_1 | \langle v_1, \ldots, v_n\rangle \in SS\},$$

where

$$SS = \{\langle V_1\theta, \ldots, V_n\theta\rangle \mid \theta : \mathcal{V} \to \mathcal{C}, I \models \{L_1, \ldots, L_m\}\theta\}.$$

Example 3. Consider the following program:

$$\begin{aligned}\Pi = \{&p(a),\\ &p(b),\\ &p(c),\\ &l(a,b),\\ &l(a,c),\\ &l(b,c),\\ &g(X,Z) :- p(X), \texttt{\#count}\{Y : l(X,Y)\} = Z,\\ &no(Z) :- \texttt{\#count}\{X, Y : l(X,Y), p(X)\} = Z,\\ &sum(Z) :- \texttt{\#sum}\{Y, X : g(X,Y)\} = Z\}.\end{aligned}$$

Then, we have that

$$AS(\Pi) = \{\{p(a), p(b), p(c), l(a,b), l(a,c), l(b,c), g(a,2), g(b,1), g(c,0), no(3), sum(3)\}\}.$$

The difference in the semantics of the aggregate #count between clasp and DLV is underlined by the necessity of rules r_7 and r_8 in Example 2 in order to obtain the same answer set as in Example 3.

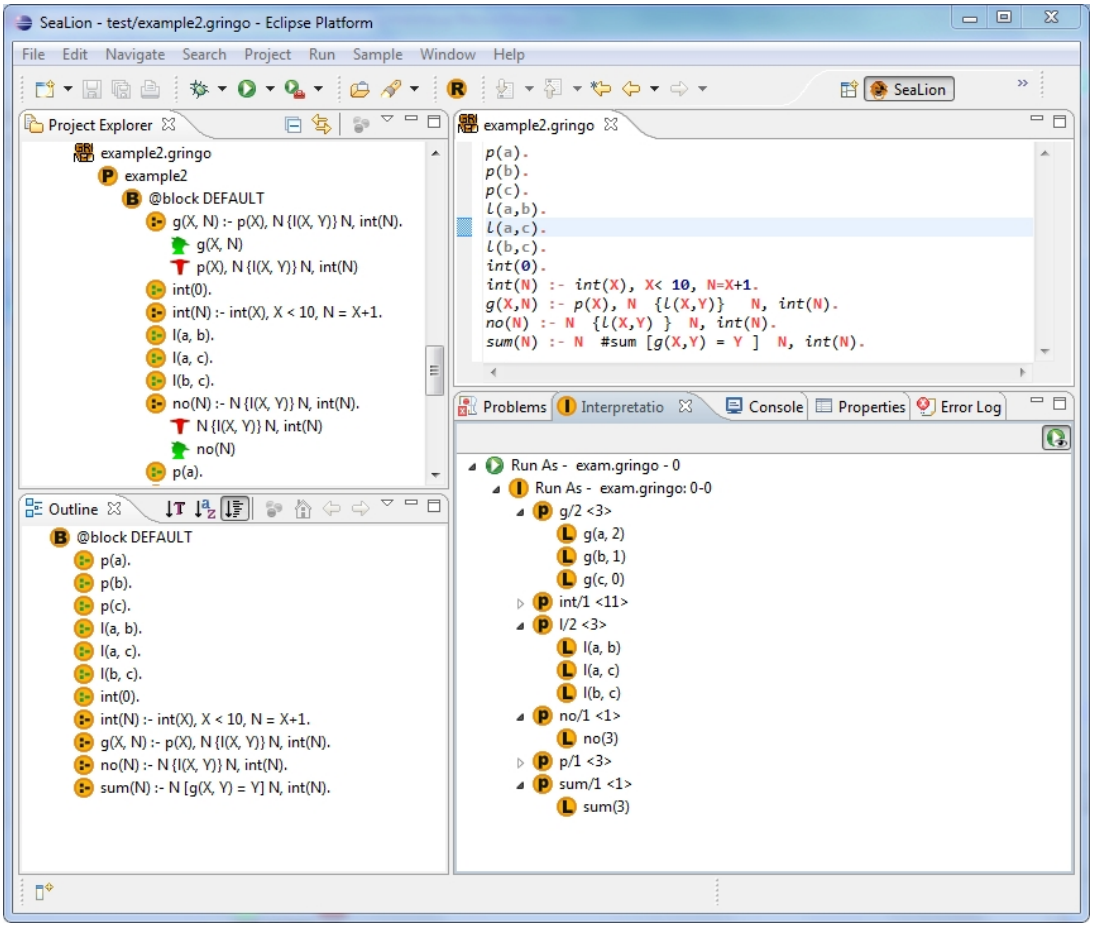

Figure 2.1: SeaLion.

2.1.3 The Integrated Development Environment SeaLion for ASP

SeaLion [33] is an integrated development environment for ASP built as a plugin on top of Eclipse (see Figure 2.1), providing an API framework that can be used in the development of new features.

It supports both DLV and gringo to a large extent and includes two source-code editors which offer syntax highlighting—one for gringo and one for DLV. The files with the extension ".lp", ".lparse",".gr", or ".gringo" are opened by default in the gringo editor, while the files with the extension ".dl" or ".dlv" are opened by default in the DLV editor. However, the user can choose any editor to open a file, independently of its extension. For solving an answer-set program, SeaLion allows the user to add external solvers.

The answer sets of a program can be visualised either in the Eclipse's console view or in an *interpretation view*, as expandable trees of depth 3. The root node of the tree is the interpretation itself and is marked as "I". The leaves of the tree are the ground atoms present in the answer set.

The atoms over the same predicate p are grouped together as children of the node marked with the predicate name, followed by its arity. Besides providing a better visualisation of answer sets, the interpretation view also represents a starting point in the development of answer-set tools which use interpretations. Furthermore, using the drag and drop feature, an interpretation can be represented in the file opened in the editor as a program containing only facts.

`SeaLion` supports Eclipse features such as Eclipse annotations in order to report problems. Moreover, refactoring of answer-set programs is supported—in particular, the uniform and safe renaming of ASP elements. Another feature implemented in `SeaLion` is the possibility to visualise answer-set programs in the Eclipse's Project Explorer as a tree representation, with predicates as nodes. Clicking on a node allows the user to visualise the corresponding source code in the file. Other features under development include debugging, autocompletion, and quick fixes for source-code problems.

2.1.4 The Annotating Language `Lana`

Lana (short for "Language for ANnotating Answer-set programs") [10] is an annotation language for structuring, documenting, and testing answer-set programs. Lana annotations are given in the comments of the program, which makes them invisible to ASP solvers. To distinguish them from other comments, an extra "`*`" is added after "`%`" at the beginning of a line.

Lana's two main tools are ASPDoc and ASPUnit, which are inspired by JavaDoc and JUnit, respectively. ASPDoc is a documentation tool, which takes annotated ASP code and produces HTML files as output. The ASPDoc generator can be accessed through the export menu of Eclipse. ASPUnit is an implementation of a unit-testing framework.

The keywords in Lana normally start with the symbol `@`. Grouping the rules in blocks is the central feature of Lana. Blocks are introduced using the keyword `@block` followed by an optional name and the opening bracket "{". The rules and all the other ASPDoc elements between "{" and "}" belong to this block. Blocks can be nested, but they must not overlap. This grouping has no semantic meaning; its purpose is solely to document the fact that some rules belong together. Among other features, Lana allows the user to specify the signature of the predicates in an answer-set program. The name of a predicate, together with its arity, is introduced by the keyword `@atom`. Furthermore, for testing purposes, the keywords `@input` and `@output` are used to specify that certain atoms represent input or output atoms for a block, respectively.

Further information about the arguments of the predicates and their domains are introduced by the keyword `@atom`. This may allow automatic verification of type violations.

ASPUnit allows the formulation of unit tests for single blocks. Certain keywords are used to introduce different assertions, such as pre-, post-, and general conditions. Every assertion belongs to a block and is formulated in ASP. An assertion contains one of the keywords `always` or `never` followed by a set of ground atoms. The assertion holds if all the ground atoms after `always` and the negation of the atoms after `never` are entailed by the rules in the assertion combined with a set of predicates.

A precondition is assumed to hold for the input predicates, a postcondition is assumed to hold for the output predicates, and a general assertion is assumed to hold for all predicates, be them input, output, or unspecified predicates.

Table 2.1: Overview of Lana Elements.

Element	Definition	Informal Description
block element	*block* | *atom* | *term* | *input signature* | *output signature* | *precondition* | *postcondition*	Lana elements related to blocks.
block	"@block" *name* "{" [*description*] {*block element*} [*ASP code*] "}"	Groups ASP rules into coherent parts.
atom	"@atom" *name* "("*termList*")" [*description*]	Defines a predicate; *termList* are the predicate's arguments.
term	"@term" *name* [*description*] [*type*]	Declares a term from some *atom* term list, its meaning, and type information.
type	"@from" *groundTerms* | "@with" *ruleBdy* | "@samerangeas" *term*	Type of a term is defined by a list of ground terms, the terms satisfying *ruleBdy*, or as the type of another term.
input signature	"@input" *inputPredicates* | "@requires" *inputPredicates*	Declares input predicates of a block as a list of name/arity pairs.
output signature	"@output" *outputPredicates* | "@defines" *outputPredicates*	Declares output predicates of a block as a list of name/arity pairs.
assertion	"@assert" *name* "{" [*description*] *assertspec* "}"	A logical condition for answer sets.
pre-condition	"@precon" *name* "{" [*description*] *assertSpec* "}"	A logical condition for the inputs of a block.
post-condition	"@postcon" *name* "{" [*description*] *assertspec* "}"	A logical condition for the answer sets of a block.
assertspec	("@always" | "@never") *atmList* [*embASPcode*]	The testmode for assertions, preconditions and postconditions; *embASPcode* is code within the Lana comment environment.

A summary of the elements in Lana is presented in Table 2.1.

In order to exemplify the Lana annotations, we extract an example as discussed by Oetsch, Pührer, and Tompits [33]:

```
%* @block maze {
%*  This is the main block of the maze generation program.
%*  @atom entrance(R,C) gives the position of the maze entrance
%*  @term R is a row index
%*    @with 0 < R, R < 20
%*  @term C is a column index
%*    @with 0 < R, R < 20
%*  ...
    empty(R,C) | wall(R,C) :- row(R),col(C).
    ...
%* }
```

2.2 Model-Driven Engineering

The main focus of *model-driven engineering* (MDE) [30] are *domain models*, with the purpose of increasing the efficiency by raising the level of abstraction and eliminating errors by using automatic model transformations.

Models are tools used for abstraction and may provide different perspectives of a system, facilitating its comprehension. They are written in a well-defined language and represent valuable assets, especially if they are synchronised between themselves and the code of the system. One of the main concerns in the MDE community is maintaining consistency between models and reducing the effort that is required for that. The models can be used in the designing phase, to *forward engineer* the system. Another use is to *reverse engineer* an existing system in order to better explain its functioning. Tools that provide the user with both functionalities are called *round-trip tools* [17].

Metamodels are used to describe the syntax specifications of the modelling language, which is called *domain-specific modelling language* (*DSML*). The models have to respect the syntax specified by their metamodels. The semantics of a DSML can also be specified in a metamodel, although that is usually not the case. UML is commonly used as a metamodel (it will be described in more detail in Section 2.2.1). The concrete model is designed using the syntax of the DSML and provides certain information about the application, like its structure and behaviour.

Model-transformation processes are used to perform different transformation steps starting from a high abstraction level and ending with the code generation. Every transformation step is composed of rules mapping the elements of one model to another. There are two different types of model transformations: *model-to-model* and *model-to-code*. Model-to-model transformations are used in order to refine the model and to provide additional details. Using the model transformations in order to generate code is a "correct-by-construction" approach, instead of the normal "construct-by-correction" approach [36].

Models represent valuable assets in the system verification as well. In order to verify the correctness of the system, three different MDE techniques are used: *model validation*, *model checking*, and *model-based testing*.

2.2.1 Unified Modelling Language

The *Unified Modelling Language* [17] is a general set of graphical notations and diagrams used to describe software systems, in particular those built using concepts from object-oriented programming. It helps to provide a level of abstraction and its main use is to document information. UML is an open standard, evolving under the control of the *Object Management Group* (*OMG*), which is an open consortium of companies. UML was created by combining the best concepts from data modelling, business modelling, object modelling, and component modelling, resulting in a versatile language which has become the standard for visualising, specifying, constructing, and documenting various software systems. Thus, UML is often used as a sketching tool, to allow team members to discuss issues that might arise during the implementation of the system. However, the use of UML can be extended to the point where the entire system is specified in UML diagrams, which then effectively act as blueprints for the implemented system. In this case, the focus of the diagram is on completeness rather than highlighting only important inform-

ation. Furthermore, with the help of various tool sets, UML can itself be used as a programming language, with developers drawing UML diagrams which are compiled directly to executable code. However, this last approach requires particularly sophisticated tools [17].

2.2.1.1 Class Diagrams

A *class diagram* is used to represent the structure of a system, with the types of objects in the system and the various relationships among them. A class diagram is constituted of the following elements:

Class. A class is an element which defines the type of a set of objects, together with their properties. A class is usually represented by a rectangle with three different compartments. The name of the class is represented in the first compartment, the attributes in the second, and the operations of the class in the third compartment.

Attribute. An attribute represents a property of a class. It is represented by a row in the class. The minimal requirements for an attribute are its name and type. Additionally, default values, multiplicities, and even visibility modifiers can be present. The multiplicity on attributes is used to indicate how many attributes with this name and type must be present in the object. The default multiplicity for an attribute is 1.

Association. The association represents a named relationship between two classes. It is represented as a line between classes, from the source class to the target class. Usually, the line has an open arrowhead pointing at the target class. The arrowhead implies the navigability concept, indicating that one class is accessible from another. Furthermore, a bidirectional association indicates that the objects involved in the association should be synchronised.

Multiplicities at both ends of the association are added to indicate how many objects participate in the association. The multiplicity is represented as a pair "$min..max$" of numbers meaning that the number of objects belonging to the respective class related through this association with another object must be between min and max (inclusively).

Aggregation. The aggregation is a particular type of association and represents a whole-part relationship. However, it is not very clear what is the exact meaning of the aggregation and there are many different interpretations. UML does not provide any semantics for the aggregation [17, p. 67].

The symbol of the aggregation is a line between the source class to the target class. In addition to the arrowhead pointing at the target class, an empty diamond is present at the source end.

Composition. The composition adds the ownership meaning to the aggregation. The ownership relationship implies that even though a class may be a component of many other classes, one instance of the class must have only one owner. For example, a microchip can be part of computers, cars, air planes, and other electronic devices, but a specific microchip (uniquely identified by its attributes values) can be part of one and only one device.

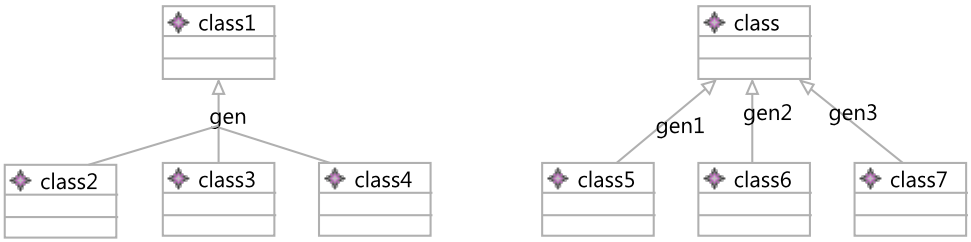

Figure 2.2: Single generalisation vs. multiple generalisations.

The graphical representation of composition is the same as for the aggregation, with the difference that the diamond is filled and the source multiplicity is not present, being considered 1..1.

Generalisation. The generalisation signifies that the properties of one class are inherited in another class. In object-oriented programming, this is implemented through the concept of inheritance. The generalisation is graphically represented by a line from a source class to a target class, with a triangle arrowhead at the target class. The source class is called *specialisation class*, while the target class is called *general class*.

Multiple classification. The main idea of multiple classification is that a class is described as a subtype of more classes, without defining exactly to which type it belongs. For example, any person is either a man or a woman, but the entire class "person" is neither of type "man" nor of type "woman". Another aspect of multiple classification is disjointness—one person cannot be both a man and a woman.

Generalisation set. The generalisation set was introduced in UML 2 in order to deal with multiple classifications. One aspect of the multiple classification that needs to be dealt with is the allowance of certain combinations. This element groups together a set of specialisation classes for a generalisation. The additional meaning captured by the generalisation set is disjointness—any object of the general class may be inherited in only one of the objects within that set. In Figure 2.2, the difference between single and multiple generalisations is displayed. The single generalisation represents a generalisation gen with the generalisation set $\{class2, class3, class4\}$ and the multiple generalisations represent three generalisations, $gen1$, $gen2$, and $gen3$, with their generalisation sets $\{class5\}$, $\{class6\}$, and $\{class7\}$, respectively. If the generalisation is complete, it implies that any object of the general class has to be an object of one of the classes within the generalisation set.

To illustrate the multiple classification concept, consider Figure 2.3, following Fowler [17, p. 77]. In this diagram,

$\{female, patient, nurse\}$, $\{male, psychotherapist\}$, $\{female, patient\}$, and $\{female, surgeon\}$

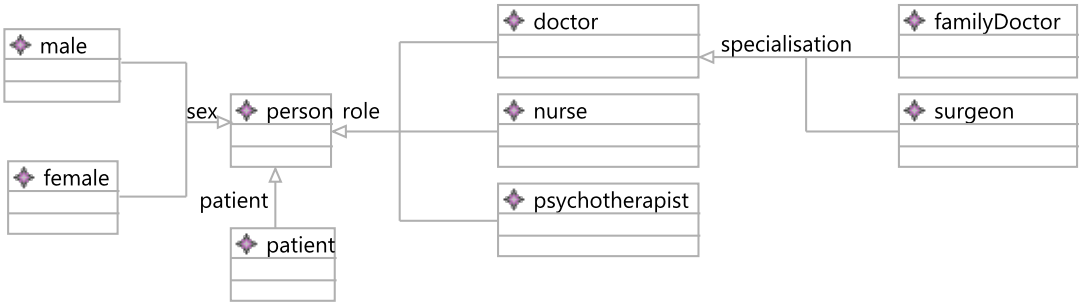

Figure 2.3: Generalisation example.

are some of the legal combinations of specialisation classes. The combination

$$\{patient, psychotherapist, nurse\}$$

is illegal because it contains two classes from the same generalisation set. Furthermore, the generalisation with the generalisation set $\{male, female\}$ should be complete because any person is either a male or a female, but the other two are not complete.

Association class. The association class allows the user to add a class to a relationship in order to define additional properties for it.

Qualified association. This concept is used as an association for every instance of an attribute.

Constraints. Diferent constrains such as completeness and exclusivity over classification are allowed.

Other elements of the UML class diagram. Other elements of the UML class diagram, such as *operation*, *interface*, *abstract class*, *derived attributes*, *access* and *visibility modifiers*, *dependency*, and *dynamic classification*, describe the behaviour or the implementation of the classes and are closely related to the object-oriented concept of methods.

2.2.1.2 Object Diagrams

Object diagrams, also called *instance diagrams*, are used to represent concrete sets of objects present in the system at a certain point in time. The same graphical elements from the class diagram are used, with the observation that classes are implemented as concrete objects in which all the attributes have a defined value. Furthermore, all the links (aggregations, associations, and compositions) are represented by lines between objects, without any multiplicities, considering the fact that in an object diagram they are concrete implementations of the relationships.

A UML object diagram representing patients and their doctors is displayed in Figure 2.4. The diagram depicts that Doctor Harry Jones treats patient Elaine Smith for insomnia, patient Oliver Walker for poliomyelitis, and patient Jack Taylor for hepatitis A. Sarah Brown is treated for pneumonia by Doctor Anna Wilson, who also treats Elaine Smith for insomnia.

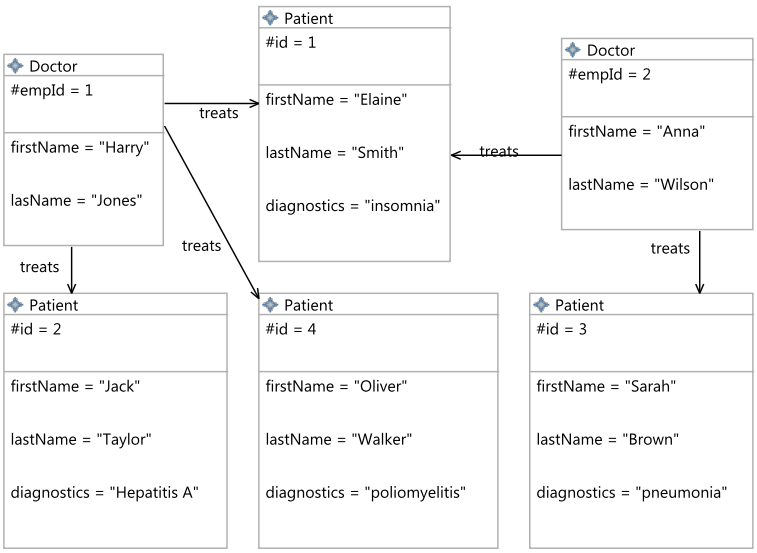

Figure 2.4: Example UML Object Diagram.

CHAPTER 3

Problem Domain Modelling for ASP

The two candidates taken into consideration to describe the domain model were the *enhanced entity relationship* (EER) diagram and the UML class diagram.

ASP is strongly related to relational databases. While the EER diagram is widely used in relational databases design, UML is a more general graphical modelling standard used for describing and designing software systems in an object-oriented style. Even if UML class diagrams are very successful in providing additional information regarding the system dynamics (operations, methods, behaviour), when it comes to data structure, it has the same expressive power as the EER diagram. Therefore, when we selected the UML class diagram, we had to take into consideration additional aspects, such as aesthetics.

The main two advantages of the UML class diagram over the EER diagram are its widespread acceptance as a standard and the possibility to integrate it with other UML models, such as the UML object diagram. Another important feature of UML is the use of XMI as standard input/output format. Furthermore, the more concise graphical representation of UML was proven to be more comprehensible [9].

However, the two approaches—object-oriented vs. relational database—are significantly different. Moreover, the capabilities of the UML class diagram to describe the behaviour of entities are meaningless for ASP. Thus, only a subset of the UML class diagram features are of interest in representing the problem domain. All the features of UML class diagrams describing the behaviour or object-oriented programming concepts which have no meaning in ASP are eliminated.

Nonetheless, there is a concept that UML class diagrams miss—the identifier concept in EER diagrams. The idea of a primary key is not required in the UML class diagram, since every object is uniquely identified by a generated surrogate key called *object identifier* (OID). David C. Hay [23] suggested a series of different modalities to deal with this. One proposal is to extend UML with the $\langle\langle ID \rangle\rangle$ stereotype. Another possibility is in the object-oriented manner—generate an OID for every entity instance. This is equivalent to adding in every class an extra attribute with the same use and constraints as a primary key. The third possibility is to make use of the

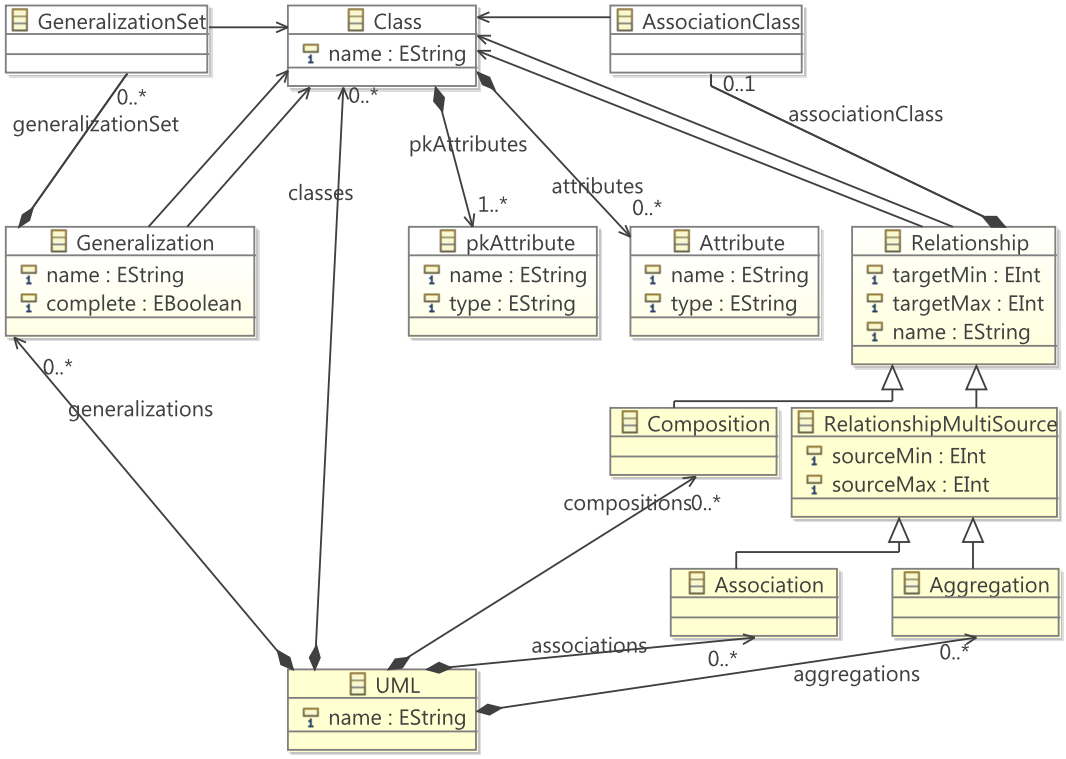

Figure 3.1: Metamodel of the UML class diagram.

"#" (octothorpe) symbol next to the attribute name[1] (see the UML class diagram in Figure 3.2 below). The last one (using the "#" symbol) is the solution we adopt here.

3.1 Relevant UML Class Diagram Features

The metamodel describing the relevant fragment of the UML class diagram is shown in Figure 3.1. The elements of the metamodel are described in the following.

UML class diagram. The UML class diagram is the top element containing all the other elements. No other element can exist outside an UML class diagram.

Class. The class is an element with two sets of attributes—primary key attributes and non-primary key attributes.

Attribute. The attribute is a property of a class and it may be either a primary key or a non-primary key attribute, depending on which set in the class it belongs to. Every attribute

[1] In UML, "#" is used to symbolise a protected attribute.

has a name and a type. No other properties are supported for attributes. The multiplicity on attributes is also not allowed, due to its implicit requirement to deal with arrays or lists in order to store the attributes. Nevertheless, this is not a major drawback, since these attributes can be represented through a one-to-many association (relationship) to a newly created class that describes the respective attribute.

Aggregation and Association. The aggregation and the association are relationships which allow multiplicity on the source class. They contain a name and the two multiplicities for the source and the target class. As discussed in Section 2.2.1, the graphical representation of the association in the UML class diagram captures the concept of navigability, which is represented through the direction of the arrow on the link (unidirectional or bidirectional). This is an object-oriented concept and has no meaning in either relational databases or ASP. Consequently, the directionality is only informative and all associations are represented as unidirectional associations.

Composition. The composition is a particular type of association which does not allow multiplicity on the source class, being by default $1..1$.

Association class. The association class is an element which links a class to a relationship. It has no properties and it is owned by the relationship, meaning that it cannot exist without a relationship.

Generalisation. The generalisation connects two classes, the source one being considered the specialisation class of the generalisation and the target one being the general class.

Generalisation set. The generalisation set is an element which links a generalisation to a class, allowing the user to create a generalisation set. They do not exist independently from a generalisation.

Qualified association is a feature that is excluded. This concept is used as an association for every instance of an attribute. But since the multiplicity on attributes is not allowed, this concept loses its meaning.

3.1.1 Formal Description

We need to formally describe a UML class diagram. In order to do so, we need to impose some constraints on the UML class diagrams. We assume that the names of the classes, associations, aggregations, generalisations, and association classes are unique in a UML class diagram. Furthermore, the character "_" does not occur in the name of any element of a diagram. In addition, if the upper bound in a multiplicity is -1, it means that there is no upper limit on the number of instances involved in that relationship.

However, an attribute can exist only inside a class and its name is unique only in the class it belongs to. In order to uniquely identify the attributes in the entire diagram, we alter their names by including the name of the class they belong to. Furthermore, we want to ensure that the alteration is bijective so that we can determine reversely the name of the attribute and the class it belongs to.

Definition 29. An *alphabet* (for UML class diagrams) is a finite or countably infinite set Σ. The elements of this set are called *symbols*.

Definition 30. For an alphabet Σ, the set of *strings over* Σ is the smallest set Σ^* satisfying the following conditions:

1. $\epsilon \in \Sigma^*$, where ϵ is the empty string, and
2. if $w \in \Sigma^*$ and $s \in \Sigma$, then $ws \in \Sigma^*$, where ws represents the concatenation of strings.

Definition 31. A *language* over Σ is a subset of Σ^*.

Definition 32. The *concatenation* of two languages L_1 and L_2 is the set

$$L_1 L_2 = \{xy | x \in L_1, y \in L_2\}.$$

Definition 33. The k *times concatenation* of a language L is denoted by L^k and is the set

$$L^k = \begin{cases} \{\epsilon\}, & \text{if } k = 0, \\ LL^{k-1}, & \text{otherwise.} \end{cases}$$

The following additional conventions are used:

- $L^* = \bigcup_{i \geq 0} L^i$,
- $L^+ = \bigcup_{i \geq 1} L^i$, and
- $[a_k - a_{k+n}] = \bigcup_{0 \leq i \leq n} \{a_{k+i}\}$, where $(a_0, a_1, \ldots)$ is a finite or infinite sequence of symbols, for $k \geq 0$ and $n \geq 1$.

Definition 34 ([8]). A *regular expression* over an alphabet Σ is an explicit formula describing a language over Σ and is defined recursively as follows:

1. Every element $a \in \Sigma$ is a regular expression describing the language $\{a\}$.
2. If r_1 and r_2 are regular expressions over Σ describing the languages L_1 and L_2, respectively, then
   - $r_1 r_2$ is a regular expression describing the language $L_1 L_2$,
   - $r_1 | r_2$ is a regular expression describing the language $L_1 \cup L_2$,
   - r_1^k is a regular expression describing the language L_1^k,
   - r_1^* is a regular expression describing the language L_1^*,
   - r_1^+ is a regular expression describing the language L_1^+.
3. Regular expressions over Σ are only the formulas that can be produced by the above rules.

Example 4. Assuming the usual alphabetical ordering for letters, the following are regular expressions:

$$[a-b]^* = \{\epsilon, a, b, aa, ab, ba, bb, \dots\};$$
$$[a-b]^+ = \{a, b, aa, ab, ba, bb, \dots\}.$$

Definition 35. A *UML* class diagram is a tuple

$$\alpha = \langle umlName, Assoc, Aggreg, Comp, Gen, Classes, Attr\rangle,$$

where $umlName \in$ [a-z]$^+$[a-zA-Z0-9]* is the *name* of the UML class diagram, $Assoc$ is a set of *associations*, $Aggreg$ a set of *aggregations*, $Comp$ a set of *compositions*, Gen a set of *generalisations*, $Classes$ a set of *classes*, and $Attr$ a set of *attributes* in the diagram, defined as follows:

1. An attribute in the class $className$ is a pair

   $$(className'\_attrName, attrType),$$

   where

   $$attrName, attrType \in \text{[a-zA-Z]}^+\text{[a-zA-Z0-9]}^*,$$

   representing the name and the type of the attribute of α, respectively, and $'$ transforms a string starting with a lower-case letter into a string starting with the corresponding upper-case one. Assuming that $'$ applied to a letter transforms the letter into its equivalent upper case (i.e., a$'$ = A, ..., z$'$ = Z), the function $'$ over a string in [a-zA-Z]$^+$[a-zA-Z0-9]* is thus defined as follows:

   $$s' = \begin{cases} u'v, & \text{if } s = uv, u \in \{\text{a}, \dots, \text{z}\}, \text{ and } v \in \text{[a-zA-Z0-9]}^*, \\ s, & \text{otherwise.} \end{cases}$$

2. A class is a tuple

   $$\langle className, A_{className}, k_{className}\rangle,$$

   where

   $$className \in \text{[a-z]}^+\text{[a-zA-Z0-9]}^*,$$

   representing the name of the class of α,

   $$A_{className} = (classNameA_1, \dots, classNameA_{n_{className}})$$

   is a sequence containing names of attributes from $Attr$, and

   $$1 \leq k_{className} \leq n_{className}$$

   specifies that the attributes with the names in the set

   $$\{classNameA_1, \dots, classNameA_{k_{className}}\} \subseteq A_{className}$$

   represent the primary key attributes of the class.

3. An association or an aggregation is a tuple

$$\langle assocName, srcCls, tgtCls, minSrc, maxSrc, minTgt, maxTgt, assocCls\rangle,$$

where $assocName \in$ [a-z]$^+$[a-zA-Z0-9]*, representing the name of the relationship of α, $srcCls$ and $tgtCls$ are names of classes from the set $Classes$, representing the source and target class of the relationship, respectively,

$$minSrc, minTgt \in \mathbb{N}, \text{and } maxSrc, maxTgt \in \mathbb{N} \cup \{-1\},$$

representing the multiplicities, and $assocCls$ is either $null$, if the relationship has no association class, or the name of a class from the set $Classes$ representing the association class, if it exists.

4. A composition is a tuple

$$\langle compName, srcCls, tgtCls, minTgt, maxTgt, assocCls\rangle,$$

where $compName \in$ [a-z]$^+$[a-zA-Z0-9]*, representing the name of the composition of α, $srcCls$ and $tgtCls$ are names of classes from the set $Classes$, representing the source and target class of the composition, respectively,

$$minTgt \in \mathbb{N} \text{ and } maxTgt \in \mathbb{N} \cup \{-1\},$$

representing the multiplicities, and $assocCls$ is either $null$, if the composition has no association class, or the name of a class from the set $Classes$ representing the association class, if it exists.

5. A generalisation (including its generalisation set) is a tuple

$$\langle genName, generalCls, genSet, complete\rangle,$$

where

$$genName \in \text{[a-z]}^+\text{[a-zA-Z0-9]}^*,$$

representing the name of the generalisation, $generalCls$ is the name of a class from the set $Classes$, representing the general class, $genSet$ is a set of names of classes from the set $Classes$, representing the generalisation set containing the specialisation classes of the generalisation, and $complete$ is a boolean variable specifying whether the generalisation is complete.

As a convention, from this point on, we will refer to the elements of α (i.e., attributes, classes, relationships, or generalisations) through their names (the first element in the pair or tuple that represents them). This is possible due to the fact that the names of the attributes are unique in the class and the names of all other elements are unique in the diagram.

Another convention is in the notation of a class. If we refer to a class through its name c, we use A_c to denote the set of attributes in the class, k_c to denote the number of primary key attributes, and $cA_1, \ldots, cA_{n_c}$ to denote the ordered elements of A_c.

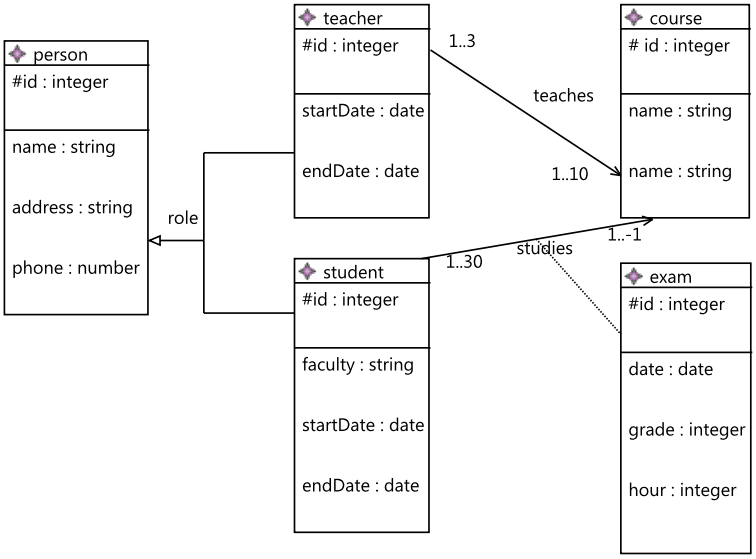

Figure 3.2: A UML class diagram.

Example 5. The UML class diagram in Figure 3.2 is formally represented by the tuple

$$\langle \text{uml}, Assoc, \emptyset, \emptyset, Gen, Classes, Attr \rangle,$$

where

$$\begin{aligned}
Attr = \{&(\text{Person_id, integer}), (\text{Person_name, string}), (\text{Person_address, string}),\\
&(\text{Person_phone, integer}), (\text{Teacher_id, integer}), (\text{Teacher_startDate, date}),\\
&(\text{Teacher_endDate, date}), (\text{Student_id, integer}), (\text{Student_faculty, string}),\\
&(\text{Student_startDate, date}), (\text{Student_endDate, date}), (\text{Course_id, integer}),\\
&(\text{Course_name, string}), (\text{Exam_id, integer}), (\text{Exam_date, date}),\\
&(\text{Exam_grade, integer})\},\\
Classes = \{&\langle \text{person}(\text{Person_id, Person_name, Person_address, Person_phone}), 1\rangle,\\
&\langle \text{teacher}, (\text{Teacher_id, Teacher_startDate, Teacher_endDate}), 1\rangle,\\
&\langle \text{student}, (\text{Student_id, Student_faculty, Student_startDate, Student_endDate}), 1\rangle,\\
&\langle \text{course}, (\text{Course_id, Course_name}), 1\rangle,\\
&\langle \text{exam}, (\text{Exam_id, Exam_date}), 1\rangle\},\\
Assoc = \{&\langle \text{teaches, teacher, course, 1, 3, 1, 10,} null\rangle,\\
&\langle \text{studies, student, course, 1, 30, 1, -1, exam}\rangle\}, \text{ and}\\
Gen = \{&\langle \text{role, \{student, teacher\}, false}\rangle\}.
\end{aligned}$$

3.2 Translation of UML Class Diagrams to ASP

The idea is to map every class to a predicate. For relationships and generalisations, we adapt some concepts from relational databases [26], such as foreign keys.

Definition 36. A *foreign key* is a set of attributes in a class matching the primary key attributes in another class.

In relational databases, the classes are mapped to tables and the relationships are mapped using foreign keys to cross-reference tables. The easiest way to handle generalisations is to add all the attributes of the general class to the specialisation classes. Another way is to have predicates for the specialisation classes, as well as for the general class and to reference the general class in all the specialisation classes through foreign keys. Adding all the attributes of the general class to the specialisation classes can lead to predicates with very large arities. Therefore, adding only the foreign key attributes is the preferred solution.

The generalisations, associations, and association classes require an additional preprocessing step in which the representation of the UML class diagram is altered by introducing foreign keys. This step is described in detail in Section 3.2.1.

3.2.1 Preprocessing the UML Class Diagram

In this section we describe how the classes are altered to fit our needs.

Definition 37. The *concatenation of two sequences* is defined as

$$(a_1, \ldots, a_n)(b_1, \ldots, b_m) = (a_1, \ldots, a_n, b_1, \ldots, b_m).$$

For every generalisation in the UML class diagram, the primary key attributes of the general class are added as foreign key attributes in the specialisation classes of the generalisation.

The relationships in a UML class diagram are handled differently, according to their multiplicities. For this purpose, we differentiate between three types of relationships.

One-to-one relationship. A *one-to-one relationship* is a relationship with the multiplicities $1..1$ at its both ends.

One-to-many (or many-to-one) relationship. A *one-to-many* (or *many-to-one*) *relationship* is a relationship with the multiplicity $1..1$ at one end of the relationship and a different multiplicity at the other end.

Many-to-many relationship. A *many-to-many relationship* is a relationship with both multiplicities different than $1..1$.

For a one-to-many relationship, it suffices to add the foreign key attributes of the mandatory (one end) class to the many-end class. For a one-to-one relationship, we prefer to add the foreign key attributes of the source class to the target class.

Definition 38. A *one-to-one* (*one-to-many*, *many-to-one*) *relationship* has one of the following forms:

$$\begin{aligned}
\langle r, c, c_1, 1, 1, 1, 1, ac\rangle &\in Assoc \cup Aggreg && \text{(one-to-one)},\\
\langle r, c_1, c, \min, \max, 1, 1, ac\rangle &\in Assoc \cup Aggreg, && min \neq 1 \text{ or } max \neq 1 \quad \text{(many-to-one)},\\
\langle r, c, c_1, 1, 1, \min, \max, ac\rangle &\in Assoc \cup Aggreg, && min \neq 1 \text{ or } max \neq 1 \quad \text{(one-to-many)},\\
\langle r, c_1, c, 1, 1, ac\rangle &\in Comp && \text{(one-to-one), or}\\
\langle r, c_1, c, \min, \max, ac\rangle &\in Comp, && min \neq 1 \text{ or } max \neq 1 \quad \text{(one-to-many)}.
\end{aligned}$$

Many-to-many relationships are represented in ASP through a new predicate. In the preprocessing step they require the introduction of a new class. The association class for a certain relationship is modified by adding the foreign keys of the two classes involved in the relationship.

Definition 39. The *preprocessed UML class diagram* corresponding to the UML class diagram

$$\alpha = \langle umlName, Assoc, Aggreg, Comp, Gen, Classes, Attr\rangle$$

is the tuple

$$\alpha^* = \langle umlName, Assoc, Aggreg, Comp, Gen, Classes^*, Attr^*\rangle,$$

where $Attr^*$ and $Classes^*$ include all the attributes in $Attr$ and the classes in $Classes$. Some classes in $Classes^*$ are modified and additional attributes and classes are included in the sets $Attr^*$ and $Classes^*$ in the preprocessing step as follows:

1. Every generalisation $\langle g, gc, gS, compl\rangle \in Gen$, for

   $$\langle gc, (gcA_1, \ldots, gcA_{n_{gc}}), k_{gc}\rangle \in Classes,$$

   is preprocessed by adding the foreign key $(gcA_1\_g, \ldots, gcA_{k_{gc}}\_g)$, referencing the general class, to all the specialisation classes in the generalisation set. Every specialisation class $c_i \in gS$, with $\langle c_i, A_{c_i}, k_{c_i}\rangle \in Classes^*$, becomes

   $$\langle c_i, A_{c_i}(gcA_1\_g, \ldots, gcA_{k_{gc}}\_g), k_{c_i}\rangle.$$

   If an attribute $aN \in \{gcA_1, \ldots, gcA_{k_{gc}}\}$, with $(aN, aT) \in Attr$, then $(aN\_g, aT) \in Attr^*$.

2. Every one-to-one or one-to-many relationship r with one of the forms in Definition 38 is preprocessed by adding the foreign key of the class $\langle c, A_c, k_c\rangle \in Classes$ to the class $\langle c_1, A_{c_1}, k_{c_1}\rangle \in Classes^*$, which becomes

   $$\langle c_1, A_{c_1}(cA_1\_type\_r, \ldots, cA_{k_c}\_type\_r), k_{c_1}\rangle,$$

   where

   $$type = \begin{cases} \text{src}, & \text{if } c \text{ is the source class of } r,\\ \text{tgt}, & \text{if } c \text{ is the target class of } r. \end{cases}$$

   If an attribute $aN \in \{cA_1, \ldots, cA_{k_c}\}$, with $(aN, aT) \in Attr$, then $(aN\_type\_r, aT) \in Attr^*$.

3. Every many-to-many relationship

$$\langle r, s, t, minS, maxS, minT, maxT, ac\rangle \in Assoc \cup Aggreg,$$

where $minS$ and $maxS$, and $minT$ and $maxT$, respectively, are not simultaneously 1, is preprocessed by introducing a new class r with $k_s + k_t$ attributes representing the foreign keys of the classes s and t. The newly added class r is the tuple

$$\langle r, (sA_1\text{\_src\_}r, \ldots, sA_{k_s}\text{\_src\_}r, tA_1\text{\_tgt\_}r, \ldots, tA_{k_t}\text{\_tgt\_}r), k_s + k_t\rangle.$$

If $aN \in \{sA_1, \ldots, sA_{k_s}\}$, with $(aN, aT) \in Attr$, then $(aN\text{\_src\_}r, aT) \in Attr^*$. If $aN \in \{tA_1, \ldots, tA_{k_t}\}$ and $(aN, aT) \in Attr$, then $(aN\text{\_tgt\_}r, aT) \in Attr^*$.

4. Every association class of a relationship

$$\langle r, s, t, minS, maxS, minT, maxT, ac\rangle \in Assoc \cup Aggreg \text{ or}$$

$$\langle r, s, t, minT, maxT, ac\rangle \in Comp,$$

where $ac \neq null$, $\langle s, A_s, k_s\rangle \in Classes$, and $\langle t, A_t, k_t\rangle \in Classes$, is preprocessed by adding the foreign keys of the classes involved in the relationship. The class

$$\langle ac, A_{ac}, k_{ac}\rangle \in Classes^*$$

becomes

$$\langle ac, A_{ac}(sA_1\text{\_src\_}r, \ldots, sA_{k_s}\text{\_src\_}r, tA_1\text{\_tgt\_}r, \ldots, tA_{k_t}\text{\_tgt\_}r), k_{ac}\rangle.$$

If $aN \in \{sA_1, \ldots, sA_{k_s}\}$ and $(aN, aT) \in Attr$, then $(aN\text{\_src\_}r, aT) \in Attr^*$. Furthermore, if $aN \in \{tA_1, \ldots, tA_{k_t}\}$ and $(aN, aT) \in Attr$, then $(aN\text{\_tgt\_}r, aT) \in Attr^*$.

Moreover, only the attributes and classes in the initial diagram or introduced in the above preprocessing step are included in the preprocessed diagram.

One can note that the primary key attributes of the classes in the initial UML class diagram are not modified. Furthermore, although the above are the default transformations, the user has the possibility to choose a different representation for relationships which may require different transformations. The user may force a one-to-one (one-to-many, many-to-one) relationship to be handled like a many-to-many relationship, or he or she may choose to represent a relationship solely through its association class (if it exists).

The preprocessed UML class diagram (Definition 39) has certain properties which will be very useful in the translations.

Theorem 1. Any foreign key attribute cls'_a_suf in a preprocessed UML class diagram, where $cls \in$ [a-z]$^+$[a-zA-Z0-9]* and $a \in$ [a-zA-Z]$^+$[a-zA-Z0-9]*, must have a corresponding primary key attribute cls'_a in the class cls.

Proof. Trivially, according to Definition 39. □

Theorem 2. The name of any attribute $\langle attrN, attrT \rangle \in Attr^*$ in a preprocessed UML class diagram has one of the following forms:

- cls'_a, where $cls \in$ [a-z]$^+$[a-zA-Z0-9]* and $a \in$ [a-zA-Z]$^+$[a-zA-Z0-9]*,
- cls'_a_gen, where $cls, gen \in$ [a-z]$^+$[a-zA-Z0-9]* and $a \in$ [a-zA-Z]$^+$[a-zA-Z0-9]*,
- $cls'_a_$src$_rel$, where $cls, rel \in$ [a-z]$^+$[a-zA-Z0-9]* and $a \in$ [a-zA-Z]$^+$[a-zA-Z0-9]*, or
- $cls_a_$tgt$_rel$, where $cls', rel \in$ [a-z]$^+$[a-zA-Z0-9]* and $a \in$ [a-zA-Z]$^+$[a-zA-Z0-9]*.

Proof. According to Definition 39, $Attr^*$ contains only the attributes in the initial UML class diagram and the foreign key attributes introduced in the preprocessing step. The names of the attributes in $Attr$ have the form cls'_a, where $cls \in$ [a-z]$^+$[a-zA-Z0-9]* and

$$a \in \text{[a-zA-Z]}^+\text{[a-zA-Z0-9]}^*$$

(Part 1 of Definition 35). The names of the foreign key attributes introduced in Definition 39 are either cls'_a_gen, where $cls, gen \in$ [a-z]$^+$[a-zA-Z0-9]* and $a \in$ [a-zA-Z]$^+$[a-zA-Z0-9]* (introduced for a generalisation, according to Part 1 of Definition 39) or $cls'_a_$src$_rel$ or $cls_a_$tgt$_rel$, where $cls', rel \in$ [a-z]$^+$[a-zA-Z0-9]* and $a \in$ [a-zA-Z]$^+$[a-zA-Z0-9]* (introduced for a relationship or an association class, according to Parts 2, 3, or 4 of Definition 39). □

Theorem 3. An attribute cls'_a in a preprocessed UML class diagram, where

$$cls \in \text{[a-z]}^+\text{[a-zA-Z0-9]}^*$$

and

$$a \in \text{[a-zA-Z]}^+\text{[a-zA-Z0-9]}^*$$

belongs only to the class cls.

Proof. According to Theorem 2, the names of all the foreign key attributes contain at least two different "_" symbols. Therefore, an attribute in a preprocessed UML class diagram with the name cls'_a, where $cls \in$ [a-z]$^+$[a-zA-Z0-9]* and $a \in$ [a-zA-Z]$^+$[a-zA-Z0-9]*, must be an attribute present in the initial UML class diagram. According to Definition 56, the name of an attribute in the UML class diagram is cls'_a, where cls is the name of the class it belongs to and thus it exists in only one class. □

Theorem 4. An attribute cls'_a_gen in a preprocessed UML class diagram, where $cls, gen \in$ [a-z]$^+$[a-zA-Z0-9]* and $a \in$ [a-zA-Z]$^+$[a-zA-Z0-9]*, cannot exist without the generalisation gen whose general class is cls.

Proof. The name of any attribute in the initial UML class diagram contains exactly two "_" characters. Therefore, an attribute with the name cls'_a_gen is an attribute introduced in the preprocessing step. The only foreign key attributes that contain three "_" characters are the ones introduced for the generalisations. According to Part 1 of Definition 39, for any foreign key attribute cls'_a_gen introduced in the preprocessing step, cls is the general class in the generalisation gen. □

Theorem 5. The foreign key attributes referencing the general class of a generalisation must be contained in and only in the specialisation classes in its generalisation set.

Proof. According to Part 1 of Definition 39, the foreign key attributes introduced for a generalisation are introduced in all the specialisation classes in its generalisation set. Furthermore, the only attributes that contain exactly three "_" characters in their names are the foreign key attributes introduced for generalisations. Thus, the foreign key attributes referencing the general class of a generalisation must be contained in and only in the specialisation classes in its generalisation set. □

Theorem 6. An attribute $cls'\_a\_type\_rel$ in a preprocessed UML class diagram, for $cls, rel \in$ [a-z]$^+$[a-zA-Z0-9]*, $a \in$ [a-zA-Z]$^+$[a-zA-Z0-9]*, and $type \in \{\text{src}, \text{tgt}\}$, cannot exist without the relationship rel whose source class is cls, if $type =$ src, or whose target class is cls, if $type =$ tgt.

Proof. The name of the attribute $cls'\_a\_type\_rel$ in the preprocessed UML class diagram contains four "_" characters. According to Part 1 of Definition 35, all the attributes in a UML class diagram contain exactly two "_" characters. Therefore, the attribute $cls'\_a\_type\_rel$ is introduced in Parts 2, 3, or 4 of Definition 39.. However, in all these preprocessing steps, the relationship rel must exist and if $type =$ src (resp., $type =$ src), the source (resp., the target) class is cls. □

Theorem 7. The association class of a relationship in a preprocessed UML class diagram contains the foreign key attributes of the two connected classes.

Proof. Trivially, according to the preprocessing step 4 of Definition 39. □

Theorem 8. If a relationship r is represented by a new predicate, the new class r introduced in the preprocessing step contains only the foreign key attributes of the linked classes, which also represent the primary key attributes of r. Furthermore, the foreign key attributes of the linked classes are present only in the class r and in the association class of r, if one exists.

Proof. A relationship r represented by a new predicate is either a many-to-many relationship or a relationship forced to be handled as a many-to-many relationship. Both cases require a preprocessing step as described in Part 3 of Definition 39. If the relationship r has an association class, the additional preprocessing step 4 of Definition 39 is performed.

Every foreign key attribute introduced for a relationship r has a unique name which includes both the name of the relationship and the name of the class it references (due to the names of the attributes in a UML class diagram in Definition 56 and the prefix $\_type\_r$). Therefore, according to the remarks above and the fact that any relationship can be handled either as a one-to-one relationship or a many-to-many relationship, but never as both, we can conclude that if a relationship r is represented by a new predicate, the new class r introduced in the preprocessing step contains only the foreign key attributes of the linked classes, which also represent the primary key attributes of r. Furthermore, the foreign key attributes of the linked classes are present only in the class r and in the association class of r, if one exists. □

Theorem 9. If a relationship r is represented by adding the foreign key attributes of the target to the source class, in a preprocessed UML class diagram, the foreign key attributes referencing the target class are contained in and only in the source class and the association class, if one exists. Furthermore, the foreign key attributes referencing the source class are contained only in the association class, if one exists.

Proof. A relationship r represented by adding the foreign key attributes of the target (source) class to the source (target) class is a one-to-one relationship handled in the preprocessing step 2 of Definition 39. If the relationship r has an association class, the additional preprocessing step 4 of Definition 39 is performed.

In the preprocessing step 2 of Definition 39, only the foreign key attributes of the target class are added to the source class. The foreign key attributes of the source class are added only in the association class, if one exists, according to the preprocessing step 4 of Definition 39.

Therefore, according to the remarks in Theorem 8 and above, we can conclude that if a relationship r is represented by adding the foreign key attributes of the target to the source class, in a preprocessed UML class diagram the foreign key attributes referencing the target class are contained in and only in the source class and the association class, if one exists. Furthermore, the foreign key referencing the source class are contained only in the association class, if one exists. □

Theorem 10. If a relationship r is represented by adding the foreign key attributes of the source to the target class, in a preprocessed UML class diagram the foreign key attributes referencing the source class are contained in and only in the target class and the association class, if one exists. Furthermore, the foreign key referencing the target class are contained only in the association class, if one exists.

Proof. Proof exactly like the proof of Theorem 9. □

Example 6. The preprocessed UML class diagram corresponding to the UML class diagram in Figure 3.2 is the tuple

$$\langle \text{uml}, Assoc, \emptyset, \emptyset, Gen, Classes^*, Attr^* \rangle,$$

where $Assoc$ and Gen are the sets defined in Example 5,

$$\begin{aligned} Attr^* = \{&\text{(Person\_id, integer), (Person\_name, string), (Person\_address, string),} \\ &\text{(Person\_phone, integer), (Teacher\_id, integer), (Teacher\_startDate, date),} \\ &\text{(Teacher\_endDate, date), (Student\_id, integer), (Student\_faculty, string),} \\ &\text{(Student\_startDate, date), (Student\_endDate, date), (Course\_id, integer),} \\ &\text{(Course\_name, string), (Exam\_id, integer), (Exam\_date, date),} \\ &\text{(Exam\_grade, integer), (Person\_id\_role, integer),} \\ &\text{(Student\_id\_src\_studies, integer), (Course\_id\_tgt\_studies, integer),} \\ &\text{(Teacher\_id\_src\_teaches, integer), (Course\_id\_tgt\_teaches, integer)}\}, \text{ and} \end{aligned}$$

$$
\begin{aligned}
Classes^* = \{&\langle \text{person}, (\text{Person\_id}, \text{Person\_name}, \text{Person\_address}, \text{Person\_phone}), 1\rangle,\\
&\langle \text{teacher}, (\text{Teacher\_id}, \text{Teacher\_startDate}, \text{Teacher\_endDate},\\
&\quad \text{Person\_id\_role}), 1\rangle,\\
&\langle \text{student}, (\text{Student\_id}, \text{Student\_faculty}, \text{Student\_startDate}, \text{Student\_endDate},\\
&\quad \text{Person\_id\_role}), 1\rangle,\\
&\langle \text{course}, (\text{Course\_id}, \text{Course\_name}), 1\rangle,\\
&\langle \text{exam}, (\text{Exam\_id}, \text{Exam\_date}, \text{Student\_id\_src\_studies},\\
&\quad \text{Course\_id\_tgt\_studies}), 1\rangle,\\
&\langle \text{studies}, (\text{Student\_id\_src\_studies}, \text{Course\_id\_tgt\_studies}), 1\rangle,\\
&\langle \text{teaches}, (\text{Teacher\_id\_src\_teaches}, \text{Course\_id\_tgt\_teaches}), 1\rangle.\}
\end{aligned}
$$

3.2.2 Mapping the UML Class Diagram to ASP

After performing all the possible transformations mentioned in Section 3.2.1, mapping the elements in the UML class diagram to terms and atoms in ASP as well as the generation of constraints are straightforward. The classes are mapped to predicates, the attributes are mapped to terms, and the relationships and generalisations play a role only in the generation of constraints.

For the encoding of the ASP translation, the `gringo` syntax is used in the following. The necessary modifications for `DLV` syntax are shown in Section 3.2.3.

Let the preprocessed UML class diagram we want to translate to ASP be

$$\alpha = \langle umlName, Assoc, Aggreg, Comp, Gen, Classes, Attr\rangle.$$

The preprocessed UML class diagram α is defined according to Definition 39. Therefore, the names of the attributes (namely $className'\_attrName$) begin with an upper case letter and the names of all other elements begin with a lower case letter.

There are two types of attributes when it comes to mapping them. The attributes

$$(className'\_attrName, attrType) \in Attr,$$

where "_" is neither in $attrName$ nor in $className$, are attributes belonging only to the class $className$. The attributes $\langle className'\_attrN\_end, attrT\rangle \in Attr$ are attributes added as part of a foreign key in the preprocessing step, where end includes the name of an aggregation, association, composition, or generalisation.

In what follows, we use Lana statements for describing the different mappings.

Definition 40. An attribute

$$(className'\_attrName, attrType) \in Attr,$$

where "_" is neither in $attrName$ nor in $className$, is mapped to the term $className'\_attrName$ with the domain defined by the predicate $attrType$ with arity 1. An attribute

$$\langle className'\_attrN\_end, attrT\rangle \in Attr,$$

where "_" is neither in $className$ nor in $attrName$, is mapped to the term $className'\_attrN\_end$ with the same domain as the term $className'\_attrN$.

The corresponding mappings in Lana are:

```
%**
 @term className'_attrName
 attrName of className
 @with attrType(#V)
*%

%**
 @term className'_attrName_end
 className'_attrName_end foreign key attribute
     referencing className
 @samerangeas className'_attrName
*%
```

Example 7. The attributes "(Person_id, integer)" and "(Person_id_role, integer)" from the pre-processed UML class diagram in Figure 3.2 are mapped to the following terms:

```
@term Person_id
id of person
@with integer(#V)

@term Person_id_role
id foreign key attribute referencing person
@samerangeas Person_id
```

There are two different approaches to mapping the classes in a UML class diagram. One option is to map the class to a single predicate over all the terms corresponding to the attributes. The other option is to map the class to multiple so-called *partitioning predicates* with smaller arities. These predicates are specified by the user and must respect the unique-names constraint.

Definition 41. A *partitioning predicate* for a class $\langle c, A_c, k_c \rangle \in Classes$ is a predicate $pred$ with arity n_{pred}, described by the atom $pred(predA_1, \ldots, predA_{n_{pred}})$, such that

$$\{cA_1, \ldots, cA_{k_c}\} = \{predA_1, \ldots, predA_{n_{pred}}\}.$$

Definition 42. A class $\langle c, A_c, k_c \rangle \in Classes$ can be mapped

1. to the predicate c with arity n_c, or
2. to a set of partitioning predicates

$$S_c = \{p_{c,1}(p_{c,1}A_1, \ldots, p_{c,1}A_{n_{p_{c,1}}}), \ldots, p_{c,m}(p_{c,m}A_1, \ldots, p_{c,m}A_{n_{p_{c,m}}})\},$$

such that

$$\bigcup_{p(pA_1,\ldots,pA_{n_p}) \in S_c} A_p = A_c.$$

The corresponding mappings in Lana are:

```
%**
 @atom  c(cA_1,...,cA_{n_c})
 cA_1,...,cA_{k_c} uniquely identify c
*%
%**
 @atom  p_{c,1}(p_{c,1}A_1,...,p_{c,1}A_{n_{p_{c,1}}})
 ...
 @atom  p_{c,m}(p_{c,m}A_1,...,p_{c,m}A_{n_{p_{c,m}}})
*%
```

Example 8. The class

$$\langle \text{teacher}, (\text{Teacher\_id}, \text{Teacher\_startDate}, \text{Teacher\_endDate}, \text{Person\_id\_role}), 1\rangle$$

from the preprocessed UML class diagram in Figure 3.2 is mapped by default to only one predicate and the following atom is defined:

```
@atom teacher(Teacher_id,Teacher_startDate,Teacher_endDate,
   Person_id_role)
Teacher_id uniquely identifies teacher
```

If the class

$$\langle \text{person}, \{\text{Person\_id}, \text{Person\_name}, \text{Person\_address}, \text{Person\_phone}\}, 1\rangle$$

from the preprocessed UML class diagram in Figure 3.2 is mapped to the set of partitioning predicates

$$\begin{aligned} S_{\text{person}} = \quad & \{\text{namePerson(Person\_id, Person\_name)} , \\ & \text{addressPerson(Person\_id, Person\_address)}, \\ & \text{phonePerson(Person\_id, Person\_phone)}\}, \end{aligned}$$

the following atoms are defined:

```
@atom namePerson(Person_id,Person_name)
contains only the name of the person

@atom addressPerson(Person_id,Person_address)
contains only the address of the person

@atom phonePerson(Person_id,Person_phone)
contains only the phone of the person
```

Mapping a class $\langle c, A_c, k_c\rangle \in \mathit{Classes}$ to a single predicate is equivalent to mapping it to the set of partitioning predicates $S_c = \{c(cA_1, \ldots, cA_{n_c})\}$. With this observation in mind, only the second mapping, i.e., given by Definition 42, will be considered in the generation of constraints.

The set of partitioning predicates describing a class c is denoted by S_c.

Due to the fact that the diagram α was altered as described in Section 3.2.1, the relationships (aggregations, associations, compositions) and generalisations are absorbed in classes and therefore no additional element mapping is necessary. However, they are involved in the constraints generation process. All the constraints are represented by assertions and require additional encodings.

To make the encodings readable, we define certain encodings and we assume that when a set of rules is represented in ASP code through its name, the name is replaced by all the rules in the set.

Definition 43. For a class $\langle c, A_c, k_c\rangle \in \mathit{Classes}$ and its partitioning predicates set S_c, we define the following encodings:

$$
\begin{aligned}
P^{\text{attr}}_{c,cA_i} &= \{\texttt{attr\_}c\texttt{\_}cA_i(cA_1,\ldots,cA_{k_c},cA_i) :- p(pA_1,\ldots,pA_{n_p}) \mid \\
&\qquad p(pA_1,\ldots,pA_{n_p}) \in S_c, cA_i \in \{pA_1,\ldots,pA_{n_p}\} \setminus \{cA_1,\ldots,cA_{k_c}\}\},\\
P^{\text{attr}}_{c} &= \bigcup_{cA_i, k_c < i \leq n_c} P^{\text{attr}}_{c,cA_i},\\
P^{\text{PK}}_{c} &= \{c\texttt{\_pk}(cA_1,\ldots,cA_{k_c}) :- p(pA_1,\ldots,pA_{n_p}) \mid p(pA_1,\ldots,pA_{n_p}) \in S_c\},\\
P^{\text{complete}}_{c} &= P^{\text{attr}}_{c} \cup \{c\texttt{\_complete}(cA_1,\ldots,cA_{n_c}) :- \\
&\qquad \texttt{attr\_}c\texttt{\_}cA_{k_c+1}(cA_1,\ldots,cA_{k_c},cA_{k_c+1}),\ldots,\\
&\qquad \texttt{attr\_}c\texttt{\_}cA_{n_c}(cA_1,\ldots,cA_{k_c},cA_{n_c}).|n_c > k_c\}\cup\\
&\qquad \{c\texttt{\_complete}(cA_1,\ldots,cA_{n_c}) \texttt{:-} p(pA_1,\ldots,pA_{n_p}) \mid \\
&\qquad p(pA_1,\ldots,pA_{n_p}) \in S_c, k_c = n_c = n_p\},\\
P^{\text{PKcomplete}}_{c} &= P^{\text{complete}}_{c} \cup \\
&\qquad \{c\texttt{\_complete\_pk}(cA_1,\ldots,cA_{k_c}) :- c\texttt{\_complete}(cA_1,\ldots,cA_{n_c})\}.
\end{aligned}
$$

Definition 44. For a class $\langle c, A_c, k_c\rangle \in \mathit{Classes}$, we define the function

$$\beta_c : [\text{a-zA-Z0-9}]^* \times [\text{a-zA-Z0-9}]^* \to [\text{a-zA-Z0-9}]^*,$$

$$\beta_c(var, No) = \begin{cases} var, & \text{if } var \in (cA_1,\ldots,cA_{k_c}),\\ varNo, & \text{if } var \notin (cA_1,\ldots,cA_{k_c}).\end{cases}$$

Moreover,

$$\beta_c(pred(term_1,\ldots,term_m), No) = pred(\beta_c(term_1, No),\ldots,\beta_c(term_m, No)).$$

The primary key constraint represents the fact that an instance is uniquely identified by its primary key. This constraint is violated when there exist two different instances with the same primary key. The primary key constraint also ensures that the foreign keys, introduced in the preprocessing step, reference at most one instance.

Definition 45. The *primary key constraint violation* for an attribute cA_i in the class $\langle c, A_c, k_c\rangle \in \mathit{Classes}$ has the encoding

$$C^{\text{PK}}_{c,cA_i} = P^{\text{attr}}_{c,cA_i} \cup \{\texttt{pkViolation\_}c\texttt{\_}cA_i :- \beta_c(\texttt{attr\_}c\texttt{\_}cA_i(cA_1,\ldots,cA_{k_c},cA_i),1), \beta_c(\texttt{attr\_}c\texttt{\_}cA_i(cA_1,\ldots,cA_{k_c},cA_i),2), \beta_c(cA_i,1) \neq \beta_c(cA_i,2)\}.$$

The corresponding Lana assertion is:

```
%**
 @assert pkViolation_c_cA_i {
  cA_i does not violate the primary key constraint
 @never pkViolation_c_cA_i
 C^PK_{c,cA_i}
}
*%
```

Example 9. The primary key violation for the attribute "Course_name" in the class "course" of the UML class diagram in Figure 3.2, if it is mapped to only one default predicate, is:

```
@assert pkViolation_course_Course_name {
Course_name does not violate the primary key constraint
@never pkViolation_course_Course_name
pkViolation_course_Course_name :-
   attr_course_Course_name(Course_id,Course_name1),
   attr_course_Course_name(Course_id,Course_name2),
   Course_name1!=Course_name2.
attr_course_Course_name(Course_id,Course_name) :-
   course(Course_id,Course_name).
}
```

The integrity constraints concern the partitioning predicates. This constraint is violated when there exists an instance (with its primary key attributes defined) for which some of the attribute values are not defined.

Definition 46. The *integrity constraint violation* for a partitioning predicate $p(pA_1,\ldots,pA_{n_p})$ of the class $\langle c, A_c, k_c\rangle \in$ *Classes* has the following encoding:

$$C^{\text{IV}}_{c,p} = P^{\text{PKcomplete}}_{c} \cup \{\texttt{integrityViolation\_}p\texttt{\_}c :- p(pA_1,\ldots,pA_{n_p}), \texttt{not } c\texttt{\_complete\_pk}(cA_1,\ldots,cA_{k_c})\}.$$

The corresponding Lana assertion is:

```
%**
 @assert integrityViolation_p_c {
 all the attributes of c must be defined
 @never integrityViolation_p_c
 C^IV_{c,p}
}
*%
```

Example 10. If the class

$$\langle \text{person}, (\text{Person\_id}, \text{Person\_name}, \text{Person\_address}, \text{Person\_phone}), 1 \rangle$$

from the preprocessed UML class diagram in Figure 3.2 is mapped to the set of partitioning predicates

$$S_{\text{person}} = \{\text{namePerson(Person\_id,Person\_name)}, \text{addressPerson(Person\_id,Person\_address)}, \text{phonePerson(Person\_id,Person\_phone)}\},$$

the integrity constraint violation for the partitioning predicate "namePerson" has the following encoding:

```
@assert integrityViolation_namePerson_person {
all the attributes of person must be defined
@never integrityViolation_namePerson_person
integrityViolation_namePerson_person :-
   namePerson(Person_id,Person_name),
   not person_complete_pk(Person_id).
person_complete_pk(Person_id) :-
   person_complete(Person_id, Person_name,
      Person_address, Person_phone).
person_complete(Person_id, Person_name,
      Person_address, Person_phone) :-
   attr_Person_name(Person_id, Person_name),
   attr_Person_address(Person_id, Person_address),
   attr_Person_phone(Person_id, Person_phone).
attr_person_Person_name(Person_id,Person_name) :-
   namePerson(Person_id,Person_name).
attr_person_Person_address(Person_id,Person_address) :-
   addressPerson(Person_id,Person_address).
attr_person_Person_phone(Person_id,Person_phone) :-
   phonePerson(Person_id,Person_phone).
}
```

The foreign key constraints represent the fact that all the referenced instances must exist. The foreign key referencing the class $\langle c, A_c, k_c \rangle \in \mathit{Classes}$ in the class $\langle c_1, A_{c_1}, k_{c_1} \rangle \in \mathit{Classes}$ is introduced for a UML class element (namely, aggregation, association, composition, or generalisation) in the preprocessing step. Considering the name of the element to be $elem$, according to the modifications defined in Section 3.2.1, the foreign key has the form

$$\{cA_1\_end, \ldots, cA_{k_c}\_end\},$$

where $end = elem$ if $elem$ is a generalisation, $end =$ src_$elem$ if c is the source class in the relationship $elem$, or $end =$ tgt_$elem$ if c is the target class in the relationship $elem$. Therefore, in order to bind the two predicates in an ASP rule through the foreign key, it suffices to add the

suffix $\_end$ as defined before to all attributes in c. The predicates $c(cA_1\_end, \ldots, cA_{n_c}\_end)$ and $c_1(c_1A_1, \ldots, c_1A_{n_{c_1}})$ share the variables

$$\{cA_1\_end, \ldots, cA_{k_c}\_end\},$$

which represent exactly the foreign key attributes.

Definition 47. The *foreign key constraint violation* for a class

$$\langle c, A_c, k_c\rangle \in \mathit{Classes}$$

referenced by the class $\langle c_1, A_{c_1}, k_{c_1}\rangle \in \mathit{Classes}$ as a result of preprocessing the element $elem$, has the following encoding:

$$\begin{aligned} C^{\mathrm{FK}}_{elem,c,c_1} = {} & P^{\text{complete}}_{c_1} \cup P^{\text{PKcomplete}}_{c} \cup \\ & \{\texttt{fkViolation\_}c_1\texttt{\_}c\_end :- \ c_1\texttt{\_complete}(c_1A_1, \ldots, c_1A_{n_{c_1}}), \\ & \quad \texttt{not } \ c\texttt{\_complete\_pk}(cA_1\_end, \ldots, cA_{k_c}\_end)\}, \end{aligned}$$

where $elem$, c, and c_1 belong to one of the following categories:

1. $\langle elem, c, c_1, 1, 1, min, max, elemClass\rangle \in Assoc \cup Aggreg$ and $elem$ is represented by adding the foreign key of c to c_1. In this case, $end =$ src_$elem$.

2. $\langle elem, c_1, c, min, max, 1, 1, elemClass\rangle \in Assoc \cup Aggreg$ and $elem$ is represented by adding the foreign key of c to c_1. In this case, $end =$ tgt_$elem$.

3. $\langle c_1, c, target, min_1, max_1, min_2, max_2, elemClass\rangle \in Assoc \cup Aggreg$ and c_1 is represented by adding the class c_1 for which the foreign keys of c and $target$ represent the primary key. In this case, $end =$ src_c_1.

4. $\langle c_1, source, c, min, max, min_2, max_2, elemClass\rangle \in Assoc \cup Aggreg$ and c_1 is represented by adding the class c_1 for which the foreign keys of c and $source$ represent the primary key. In this case, $end =$ tgt_c_1.

5. $\langle elem, c, target, min_1, max_1, min_2, max_2, c_1\rangle \in Assoc \cup Aggreg$ and $elem$ is represented solely by adding the foreign keys of c and $target$ to c_1. In this case, $end =$ src_$elem$.

6. $\langle elem, source, c, min, max, min_2, max_2, c_1\rangle \in Assoc \cup Aggreg$ and $elem$ is represented solely by adding the foreign keys of c and $source$ to c_1. In this case, $end =$ tgt_$elem$.

7. $\langle elem, c, c_1, min, max, assocCls\rangle \in Comp$ and $elem$ is represented by adding the foreign key of c to c_1. In this case, $end =$ src_$elem$.

8. $\langle c_1, c, target, min, max, elemClass\rangle \in Comp$ and c_1 is represented by adding the class c_1 for which the foreign keys of c and $target$ represent the primary key. In this case, $end =$ src_c_1.

9. $\langle c_1, source, c, min, max, elemClass\rangle \in Comp$ and c_1 is represented by adding the class c_1 for which the foreign keys of c and $source$ represent the primary key. In this case, $end = \text{tgt\_}c_1$.

10. $\langle elem, c, target, min, max, c_1\rangle \in Comp$ and $elem$ is represented solely by adding the foreign keys of c and $target$ to c_1. In this case, $end = \text{src\_}elem$.

11. $\langle elem, source, c, min, max, c_1\rangle \in Comp$ and $elem$ is represented solely by adding the foreign keys of c and $source$ to c_1. In this case, $end = \text{tgt\_}elem$.

12. $\langle elem, c, gS, complete\rangle \in Gen$ and $c_1 \in gS$. In this case, $end = elem$.

The corresponding Lana assertion is:

%**
@assert fkViolation_c_1_c_end {
c_1 references c
through the foreign key cA_1_$end, \ldots, cA_{k_c}$_end
no reference of a non existent c in c_1
@never fkViolation_c_1_c_end
$C^{\text{FK}}_{elem,c,c_1}$
}
*%

Example 11. The foreign key constraint violation for the class "course" and the association "teaches" in the UML class diagram in Figure 3.2 has the following encoding:

```
@assert fkViolation_teaches_course_tgt_teaches {
teaches references course
through the foreign key Course_id
no reference of a non existent course in teaches
@never fkViolation_teaches_course_tgt_teaches
fkViolation_teaches_course_tgt_teaches :-
   teaches_complete(Teacher_id_src_teaches,
      Course_id_tgt_teaches),
   not course_pk(Course_id_tgt_teaches).
course_pk(Course_id) :- course(Course_id,Course_name).
teaches_complete(Teacher_id_src_teaches,
      Course_id_tgt_teaches) :-
   teaches(Teacher_id_src_teaches,Course_id_tgt_teaches).
}
```

Definition 48. For relationships, the following functions are defined:

$$inv(type) = \begin{cases} \text{src}, & \text{if } type = \text{tgt}, \\ \text{tgt}, & \text{if } type = \text{src}. \end{cases}$$

$$\beta(var, type, elem) = \begin{cases} var, & \text{if } type = \text{“”} \text{ or } var \\ & \quad \text{is of form } nameVar\_inv(type)\_elem, \\ nameVar\_type\_elem, & \text{otherwise.} \end{cases}$$

Moreover,

$$\beta(pred(t_1, \ldots, t_m), type, elem) = pred(\beta(t_1, type, elem), \ldots, \beta(t_m, type, elem)).$$

Definition 49. We define the following encodings for a relationship $elem$, represented by the class c and having an association class $assocCls$ different than c:

$$\begin{aligned} P^{\text{FK}}_{elem} = {} & P^{\text{complete}}_{assocCls} \cup P^{\text{complete}}_{c} \cup \\ & \{elem\_\texttt{fk}(c_1A_1\_\text{src}\_elem, \ldots, c_1A_{k_{c_1}}\_\text{src}\_elem, \\ & \qquad c_2A_1\_\text{tgt}\_elem, \ldots, c_2A_{k_{c_2}}\_\text{tgt}\_elem) :- \\ & \quad assocCls\_\texttt{complete}(assocClsA_1, \ldots, assocClsA_{n_{assocCls}}), \\ & \quad elem\_\texttt{fk}(c_1A_1\_\text{src}\_elem, \ldots, c_1A_{k_{c_1}}\_\text{src}\_elem, \\ & \qquad c_2A_1\_\text{tgt}\_elem, \ldots, c_2A_{k_{c_2}}\_\text{tgt}\_elem) :- \\ & \quad \beta(c\_\texttt{complete}(cA_1, \ldots, cA_{n_c}), type, elem)\}, \end{aligned}$$

where $elem$, c, c_1, c_2 belong to one of the following categories:

1. $\langle elem, c_1, c_2, min, max, 1, 1, assocCls\rangle \in Assoc \cup Aggreg$ and $elem$ is represented by adding the foreign key of c_2 to c_1 and $c = c_1$. In this case, $type =$ src.

2. $\langle elem, c_1, c_2, 1, 1, min, max, assocCls\rangle \in Assoc \cup Aggreg$ and $elem$ is represented by adding the foreign key of c_1 to c_2 and $c = c_2$. In this case, $type =$ tgt.

3. $\langle elem, c_1, c_2, min_1, max_1, min_2, max_2, assocCls\rangle \in Assoc \cup Aggreg$ and $elem$ is represented by adding the class $elem$ for which the foreign keys of c_1 and c_2 represent the primary key. In this case, $c = elem$ and $type =$“”.

4. $\langle elem, c_1, c_2, min, max, assocCls\rangle \in Comp$ and $elem$ is represented by adding the foreign key of c_1 to c_2 and $c = c_2$. In this case, $type =$ tgt.

5. $\langle elem, c_1, c_2, min, max, assocCls\rangle \in Comp$ and $elem$ is represented by adding the class $elem$ for which the foreign keys of c_1 and c_2 represent the primary key. In this case, $c = elem$ and $type =$“”.

Definition 50. The *multiplicity constraint violations* for a relationship $elem$ have the following encodings:

$$C^{\texttt{multiplicity}}_{elem,type} = \begin{cases} P^{\text{complete}}_{c_1} \cup P^{\text{FK}}_{elem} \cup \\ \{\texttt{multiplicityViolation\_}type\_elem :- \\ \quad c\_\texttt{complete}(cA_1\_inv(type)\_elem, \ldots, cA_{n_c}\_inv(type)\_elem), \\ \quad \texttt{not } \ min_1\{elem\_\texttt{fk}(sA_1\_\text{src}\_elem, \ldots, sA_{k_s}\_\text{src}\_elem, \\ \qquad tA_1\_\text{tgt}\_elem, \ldots, tA_{k_t}\_\text{tgt}\_elem)\} max_1\}, \\ \quad \text{if } elem \text{ has an association class and } elem \\ \quad \text{is not represented solely by the association class,} \\ \\ P^{\text{complete}}_{c_1} \cup P^{\text{complete}}_{c} \cup \\ \{\texttt{multiplicityViolation\_}type\_elem :- \\ \quad c\_\texttt{complete}(cA_1\_inv(type)\_elem, \ldots, cA_{n_c}\_inv(type)\_elem), \\ \quad \texttt{not } \ min_1\{c_1\_\texttt{complete}(c_1A_1, \ldots, c_1A_{n_{c_1}})\} max_1\}, \\ \quad \text{otherwise,} \end{cases}$$

where $elem$, c, c_1, s, and t belong to one of these categories:

1. $\langle elem, c, c_1, 1, 1, min_1, max_1, assocCls\rangle \in Assoc \cup Aggreg$ and $elem$ is represented by adding the foreign key of c to c_1. In this case, $type = \text{tgt}$, $s = c$, and $t = c_1$.

2. $\langle elem, c_1, c, min_1, max_1, 1, 1, assocCls\rangle \in Assoc \cup Aggreg$ and $elem$ is represented by adding the foreign key of c to c_1. In this case, $type = \text{src}$, $s = c_1$, and $t = c$.

3. $\langle c_1, srcCls, c, min_1, max_1, min, max, assocCls\rangle \in Assoc \cup Aggreg$ and c_1 is represented by adding the class c_1 for which the foreign keys of c and $srcCls$ represent the primary key. In this case, $elem = c_1$, $type = \text{src}$, $s = srcCls$, and $t = c$.

4. $\langle c_1, c, tgtCls, min, max, min_1, max_1, assocCls\rangle \in Assoc \cup Aggreg$ and c_1 is represented by adding the class c_1 for which the foreign keys of c and $tgtCls$ represent the primary key. In this case, $elem = c_1$, $type = \text{tgt}$, $s = c$, and $t = tgtCls$.

5. $\langle elem, c, tgtCls, min, max, min_1, max_1, c_1\rangle \in Assoc \cup Aggreg$ and $elem$ is represented solely by adding the foreign keys of c and $tgtCls$ to c_1. In this case, $type = \text{tgt}$, $s = c$, and $t = tgtCls$.

6. $\langle elem, srcCls, c, min_1, max_1, min, max, c_1\rangle \in Assoc \cup Aggreg$ and $elem$ is represented solely by adding the foreign keys of c and $srcCls$ to c_1. In this case, $type = \text{src}$, $s = srcCls$, and $t = c$.

7. $\langle elem, c, c_1, min_1, max_1, assocCls\rangle \in Comp$ and $elem$ is represented by adding the foreign key of c to c_1. In this case, $type = \text{tgt}$, $s = c$, and $t = c_1$.

8. $\langle c_1, c, tgtCls, min_1, max_1, assocCls\rangle \in Comp$ and c_1 is represented by adding the class c_1 for which the foreign keys of c and $tgtCls$ represent the primary key. In this case, $elem = c_1$, $type = \text{tgt}$, $s = c$, and $t = tgtCls$.

9. $\langle c_1, srcCls, c, min, max, assocCls\rangle \in Comp$ and c_1 is represented by adding the class c_1 for which the foreign keys of c and $srcCls$ represent the primary key. In this case, $elem = c_1$, $type =$ src, $min_1 = max_1 = 1$, $s = srcCls$, and $t = c$.

10. $\langle elem, c, tgtCls, min_1, max_1, c_1\rangle \in Comp$ and $elem$ is represented solely by adding the foreign keys of c and $tgtCls$ to c_1. In this case, $type =$ tgt, $s = c$, and $t = tgtCls$.

11. $\langle elem, srcCls, c, min, max, c_1\rangle \in Comp$ and $elem$ is represented solely by adding the foreign keys of c and $srcCls$ to c_1. In this case, $type =$ src, $min_1 = max_1 = 1$, $s = srcCls$, and $t = c$.

The corresponding Lana assertion is:

%**
@assert multiplicityViolation_$type_elem$ {
min_1<=no(c_1 referencing c)<=max_1
@never multiplicityViolation_$type_elem$
$C^{\texttt{multiplicity}}_{elem,type}$
}
*%

Example 12. The multiplicity constraint violation for the target of the association "studies" in the UML class diagram in Figure 3.2 has the following encoding:

```
@assert multiplicityViolation_tgt_studies {
1<= no(studies referencing course)
@never multiplicityViolation_tgt_studies
multiplicityViolation_tgt_studies :-
   student_complete(Student_id_src_studies,
     Student_faculty_src_studies,
     Student_startDate_src_studies,
     Student_endDate_src_studies, Person_id_role_src_studies),
   not 1 {studies_fk(Student_id_src_studies,
     Course_id_tgt_studies)}.
studies_fk(Student_id_src_studies,Course_id_tgt_studies) :-
   studies_complete(Student_id_src_studies_tgt_studies,
     Course_id_tgt_studies_tgt_studies).
studies_fk(Student_id_src_studies,Course_id_tgt_studies) :-
   exam_complete(Exam_id, Exam_date, Exam_grade,
     Student_id_src_studies, Course_id_tgt_studies).
student_complete(Student_id, Student_faculty,
     Student_startDate, Student_endDate, Person_id_role) :-
   student(Student_id,Student_faculty,Student_startDate,
     Student_endDate,Person_id_role).
studies_complete(Student_id_src_studies,
      Course_id_tgt_studies) :-
```

```
    studies(Student_id_src_studies,Course_id_tgt_studies).
}
```

The association class implies a one-to-one connection with the relationship it represents. Therefore, one must verify the existence of the relationship and additional constraints must be defined.

The relationship reference constraint for an association class $assocCls$ of a relationship $elem$ represents the fact that no association instance must exist without its corresponding relationship. The relationship reference constraint must verify not only the existence of the two instances, but also the existence of the relationship.

Definition 51. The *relationship reference constraint violation* for the relationship $elem$ has the following encoding:

$$C_{elem}^{\text{RR}} = P_{assocCls}^{\text{complete}} \cup P_{c}^{\text{complete}} \cup \\ \{\texttt{relationshipReferenceViolation\_}assocCls\_elem :- \\ assocCls\texttt{\_complete}(assocClsA_1, \ldots, assocClsA_{n_{assocCls}}), \\ \texttt{not 1 } \{\beta_c(c\texttt{\_complete}(cA_1, \ldots, cA_{n_c}), type, elem)\} \texttt{ 1}\},$$

where $elem$ belongs to one of these categories:

1. $\langle elem, c, target, min, max, 1, 1, assocCls\rangle \in Assoc \cup Aggreg$ and $elem$ is represented by adding the foreign key of $target$ to c. In this case, $type =$ src.

2. $\langle elem, source, c, 1, 1, min, max, assocCls\rangle \in Assoc \cup Aggreg$ and $elem$ is represented by adding the foreign key of $source$ to c. In this case, $type =$ tgt.

3. $\langle elem, source, target, min_1, max_1, min_2, max_2, assocCls\rangle \in Assoc \cup Aggreg$ and $elem$ is represented by adding the class $elem$ for which the foreign keys of $source$ and $target$ represent the primary key. In this case, $c = elem$ and $type =$ "".

4. $\langle elem, source, c, min, max, assocCls\rangle \in Comp$ and $elem$ is represented by adding the foreign key of $source$ to c. In this case, $type =$ tgt.

5. $\langle elem, source, target, min, max, assocCls\rangle \in Comp$ and $elem$ is represented by adding the class $elem$ for which the foreign keys of $source$ and $target$ represent the primary key. In this case, $c = elem$ and $type =$ "".

The corresponding Lana assertion is:

```
%**
 @assert relationshipReferenceViolation_assocCls_elem {
 for every association instance assocCls
     must exist a relationship elem
 @never relationshipReferenceViolation_assocCls_elem
 C^RR_elem
}
*%
```

Example 13. The relationship reference constraint violation for the association class "exam" of the relationship "studies" in the UML class diagram in Figure 3.2 has the following encoding:

```
@assert relationshipReferenceViolation_exam_studies {
no reference of a non existent relationship studies in exam
@never relationshipReferenceViolation_exam_studies
relationshipReferenceViolation_exam_studies :-
   exam_complete(Exam_id, Exam_date, Exam_grade,
      Student_id_src_studies, Course_id_tgt_studies),
   not 1 {studies_complete(Student_id_src_studies,
      Course_id_tgt_studies)} 1.
exam_complete(Exam_id, Exam_date, Exam_grade,
      Student_id_src_studies, Course_id_tgt_studies) :-
   exam(Exam_id,Exam_date,Exam_grade,Student_id_src_studies,
      Course_id_tgt_studies).
studies_complete(Student_id_src_studies,
     Course_id_tgt_studies) :-
   studies(Student_id_src_studies,Course_id_tgt_studies).
}
```

The association instance constraint for a relationship $elem$ and its association class $assocCls$ represents the fact that every relationship must have its corresponding association instance.

Definition 52. The *association instance constraint violation* for a relationship $elem$ has the following encoding:

$$C^{\mathtt{AI}}_{elem} = P^{\text{complete}}_{assocCls} \cup P^{\text{complete}}_{c} \cup$$
$$\{\mathtt{associationInstanceViolation\_}elem\_assocCls :-$$
$$\beta_c(c\mathtt{\_complete}(cA_1, \ldots, cA_{n_c}), type, elem),$$
$$\mathtt{not\ 1\ }\{assocCls\mathtt{\_complete}(assocClsA_1, \ldots, assocClsA_{n_{assocCls}})\}\ \mathtt{1}\},$$

where $elem$ is defined in a similar fashion to Definition 51.

The corresponding Lana assertion is:

```
%**
 @assert associationInstanceViolation_elem_assocCls {
 every relationship elem
     must have an association instance assocCls
 @never associationInstanceViolation_elem_assocCls
 C^AI_elem
}
*%
```

Example 14. The association instance constraint violation for the relationship "studies" and its association class "exam" in the UML class diagram in Figure 3.2 has the following encoding:

```
@assert associationInstanceViolation_studies_exam {
every relationship studies must have an association
   instance exam
@never associationInstanceViolation_studies_exam
associationInstanceViolation_studies_exam :-
   studies_complete(Student_id_src_studies,
     Course_id_tgt_studies),
   not 1 {exam_complete(Exam_id, Exam_date, Exam_grade,
      Student_id_src_studies, Course_id_tgt_studies)} 1.
studies_complete(Student_id_src_studies,
     Course_id_tgt_studies) :-
   studies(Student_id_src_studies,Course_id_tgt_studies).
exam_complete(Exam_id, Exam_date, Exam_grade,
      Student_id_src_studies, Course_id_tgt_studies) :-
   exam(Exam_id,Exam_date,Exam_grade,Student_id_src_studies,
      Course_id_tgt_studies).
}
```

The generalisation constraint represents the fact that there cannot be two different generalisation instances for the same specialisation instance.

Definition 53. The *generalisation constraint violation* for the generalisation class

$$\langle g, gc, gS, compl \rangle \in Gen,$$

$\langle g, A_g, k_g \rangle \in Classes$, and a class $sp \in gS$ has the following encoding:

$$\begin{aligned} C_{g,sp}^{\texttt{generalisation}} = {} & P_{gc}^{\text{complete}} \cup P_{sp}^{\text{complete}} \cup \\ & \{\texttt{generalisationViolation\_}g\texttt{\_}sp :- \\ & \quad gc\texttt{\_complete}(gcA_1\texttt{\_}g, \ldots, gcA_{n_{gc}}\texttt{\_}g), \\ & \quad \texttt{2 } \{sp\texttt{\_complete}(spA_1, \ldots, spA_{n_{sp}})\}g\}. \end{aligned}$$

The corresponding Lana assertion is:

```
%**
 @assert generalisationViolation_g_sp {
 no two different sp specialisations of gc
 @never generalisationViolation_g_sp
 C^generalisation_g,sp
}
*%
```

Example 15. The generalisation constraint violation for the specialisation "teacher" of the generalisation "role" in the UML class diagram in Figure 3.2 has the following encoding:

```
@assert generalisationViolation_role_student {
```

```
no two different student specialisations of person
@never generalisationViolation_role_student
generalisationViolation_role_student :-
   person_complete(Person_id_role, Person_name_role,
      Person_address_role, Person_phone_role),
   2 {student_complete(Student_id, Student_faculty,
      Student_startDate, Student_endDate, Person_id_role)}.
person_complete(Person_id, Person_name, Person_address,
     Person_phone) :-
   person(Person_id,Person_name,Person_address,Person_phone).
student_complete(Student_id, Student_faculty,
      Student_startDate, Student_endDate, Person_id_role) :-
   student(Student_id,Student_faculty,Student_startDate,
      Student_endDate,Person_id_role).
}
```

A disjointness constraint represents the fact that there cannot be two different specialisation instances from the generalisation set for a generalisation.

Definition 54. The *disjointness constraint violation* for a generalisation $\langle g, gc, gS, compl\rangle \in Gen$, where $gS = \{sp_1, \ldots, sp_{lg}\}$ and $\langle gc, A_{gc}, k_{gc}\rangle \in Classes$, has the following encoding:

$$\begin{aligned} C_g^{\texttt{disjointness}} = {} & P_{gc}^{\text{complete}} \cup P_{sp_1}^{\text{complete}} \cup \cdots \cup P_{sp_{lg}}^{\text{complete}} \cup \\ & \{\texttt{disjointnessViolation\_}g :- \\ & \quad gc\texttt{\_complete}(gcA_1\texttt{\_}g, \ldots, gcA_{n_{gc}}\texttt{\_}g), \\ & \quad 2\ \{sp_1\texttt{\_complete}(sp_1A_1, \ldots, sp_1A_{n_{sp_1}}), \ldots, \\ & \quad\quad sp_{lg}\texttt{\_complete}(sp_{lg}A_1, \ldots, sp_{lg}A_{n_{sp_{lg}}})\}\}. \end{aligned}$$

The corresponding Lana assertion is:

```
%**
 @assert disjointnessViolation_g {
 no specialisation of gc as more than one g
 @never disjointnessViolation_g
 C_g^disjointness
}
*%
```

Example 16. The disjointness constraint violation for the generalisation "role" in the UML class diagram in Figure 3.2 has the following encoding:

```
@assert disjointnessViolation_role {
no specialisation of person as more than one role
@never disjointnessViolation_role
disjointnessViolation_role :-
```

```
   person_complete(Person_id_role, Person_name_role,
      Person_address_role, Person_phone_role),
   2 {student_complete(Student_id, Student_faculty,
         Student_startDate, Student_endDate, Person_id_role),
      teacher_complete(Teacher_id, Teacher_startDate,
         Teacher_endDate, Person_id_role)}.
person_complete(Person_id, Person_name, Person_address,
      Person_phone) :-
   person(Person_id,Person_name,Person_address,Person_phone).
student_complete(Student_id, Student_faculty,
      Student_startDate, Student_endDate, Person_id_role) :-
   student(Student_id,Student_faculty,Student_startDate,
      Student_endDate,Person_id_role).
teacher_complete(Teacher_id, Teacher_startDate,
      Teacher_endDate, Person_id_role) :-
   teacher(Teacher_id,Teacher_startDate,Teacher_endDate,
      Person_id_role).
}
```

If the generalisation set is complete, an additional completeness constraint is required. A completeness constraint represents the fact that for every general instance there exists exactly one specialisation instance from the generalisation set.

Definition 55. The *completeness constraint violation* for a generalisation $\langle g, gc, gS, compl\rangle \in Gen$, where $gS = \{sp_1, \ldots, sp_{lg}\}$ and $\langle gc, A_{gc}, k_{gc}\rangle \in Classes$, has the following encoding:

$$
\begin{aligned}
C_g^{\texttt{completeness}} = {} & P_{gc}^{\text{complete}} \cup P_{sp_1}^{\text{complete}} \cup \cdots \cup P_{sp_{lg}}^{\text{complete}} \cup \\
& \{\texttt{completenessViolation\_}g \texttt{ :-} \\
& \quad gc\texttt{\_complete}(gcA_1\texttt{\_}g, \ldots, gcA_{n_{gc}}\texttt{\_}g), \\
& \quad \texttt{1 }\{sp_1\texttt{\_complete}(sp_1A_1, \ldots, sp_1A_{n_{sp_1}}), \ldots, \\
& \qquad sp_{lg}\texttt{\_complete}(sp_{lg}A_1, \ldots, sp_{lg}A_{n_{sp_{lg}}})\}\texttt{1}\}.
\end{aligned}
$$

The corresponding Lana assertion is:

```
%**
 @assert completenessViolation_g {
 every gc must have one g specialisation
 @never completenessViolation_g
 C_g^completeness
}
*%
```

The completeness constraint covers the disjointness constraint and therefore the disjointness constraint is not required any more.

Definition 56. The UML Class Diagram

$$\langle umlName, Assoc, Aggreg, Comp, Gen, Classes, Attr\rangle$$

is mapped to the block $umlName$ that includes all the Lana elements corresponding to the UML Class Diagram, mapped as illustrated above.

The corresponding Lana assertion is as follows:

```
%**
 @block umlName {
 encoding of the UML diagram umlName
*%
ASP code
%**
}
*%
```

3.2.3 DLV Modifications

Concerning the translation described above, there are two differences between DLV and gringo one needs to consider.

The first one is the lack of block comments. This requires splitting the block comments into line comments by adding "%*" at the beginning of every Lana generated line.

The second difference involves the aggregate predicate #count (cf. Section 2.1.2) and requires additional encodings, especially when the multiset contains more elements (in the case of disjointness and completeness constraints for generalisation).

In the above gringo translation, most of the constraints containing the aggregate predicate are of the following form:

$$\begin{aligned} constraintName :- \; & pred(predA_1, \ldots, predA_{n_{pred}}), \\ & \texttt{not} \;\; min\{pred1(pred1A_1, \ldots, pred1A_{n_{pred1}})\}max \end{aligned}$$

with

$$\{predA_1, \ldots, predA_{n_{pred}}\} \cap \{pred1A_1, \ldots, pred1A_{n_{pred1}}\} \neq \emptyset.$$

Let $freeVars = \{pred1A_1, \ldots, pred1A_{n_{pred1}}\} \setminus \{predA_1, \ldots, predA_{n_{pred}}\}$. There are two cases to be considered: $freeVars = \emptyset$ and $freeVars \neq \emptyset$.

If $freeVars = \emptyset$ and considering $\{predA_1, \ldots, predA_{n_{pred}}\}$ constant, then

$$\#\{pred1A_1, \ldots, pred1A_{n_{pred1}}\} \in \{0, 1\}.$$

Therefore, min and max have to be taken into consideration:

- If $0 \leq min < max$ and $1 \leq max$, then the constraint is never violated, so there is no need to add any assertion.

- If $min > max$ or $1 < min \leq max$, then the constraint is always violated, because it cannot be the case that $min \leq \#\{\{pred1A_1, \ldots, pred1A_{n_{pred1}}\}\} \leq max$. Therefore, the constraint is a fact and has the form:

$$constraintName.$$

- If $min = max = 0$, then $\#\{pred1A_1, \ldots, pred1A_{n_{pred1}}\} = 0$. In this case, the constraint is violated if $pred(predA_1, \ldots, predA_{n_{pred}})$ and

$$pred1(pred1A_1, \ldots, pred1A_{n_{pred1}})$$

exist at the same time:

$$\begin{aligned} constraintName :- \; & pred(predA_1, \ldots, predA_{n_{pred}}), \\ & pred1(pred1A_1, \ldots, pred1A_{n_{pred1}}). \end{aligned}$$

- If $min \leq max$ and $min = 1$, then $\#\{pred1(pred1A_1, \ldots, pred1A_{n_{pred1}})\} = 1$. In this case, the constraint is violated if whenever $pred(predA_1, \ldots, predA_{n_{pred}})$ exists, there is no $pred1(pred1A_1, \ldots, pred1A_{n_{pred1}})$. Since $\{pred1A_1, \ldots, pred1A_{n_{pred1}}\} \subseteq \{predA_1, \ldots, predA_{n_{pred}}\}$, the following constraint is safe:

$$\begin{aligned} constraintName :- \; & pred(predA_1, \ldots, predA_{n_{pred}}), \\ & \texttt{not} \;\; pred1(pred1A_1, \ldots, pred1A_{n_{pred1}}). \end{aligned}$$

Otherwise, if $freeVars \neq \emptyset$, the constraint has the following form:

$$\begin{aligned} constraintName :- \; & pred(predA_1, \ldots, predA_{n_{pred}}), \\ & \texttt{not} \;\; min <= \texttt{\#count}\{pred1(pred1A_1, \ldots, pred1A_{n_{pred1}})\} <= max. \end{aligned}$$

Definition 57. Consider a generalisation $\langle g, gc, gS, compl \rangle \in Gen$ such that $\langle g, A_g, k_g \rangle \in Classes$ and $\langle sp_i, A_{sp_i}, k_{sp_i} \rangle \in Classes$, for all $sp_i \in gS$. Then, the DLV modification of $C^{\texttt{generalisation}}_{g,sp_i}$ is

$$\begin{aligned} C^{\texttt{generalisation}}_{g,sp_i} = \; & P^{\text{complete}}_{gc} \cup P^{\text{complete}}_{sp_i} \cup \\ & \{\texttt{generalisationViolation\_}g\texttt{\_}sp :- \\ & \quad gc\texttt{\_complete}(gcA_1\texttt{\_}g, \ldots, gcA_{n_{gc}}\texttt{\_}g), \\ & \quad \texttt{\#count}\{freeVars_i : sp_i\texttt{\_complete}(sp_iA_1, \ldots, sp_iA_{n_{sp_i}})\}\texttt{>=2}\}. \end{aligned}$$

Example 17. The generalisation constraint violation for the specialisation "teacher" of the generalisation "role" in the UML class diagram in Figure 3.2 has the following encoding in DLV:

```
%* @assert generalisationViolation_role_teacher {
%* no two different teacher specialisations of person
%* @never generalisationViolation_role_teacher
%* generalisationViolation_role_teacher :-
```

```
    person_complete(Person_id_role, Person_name_role,
        Person_address_role, Person_phone_role),
    #count{Teacher_id, Teacher_startDate, Teacher_endDate :
        teacher_complete(Teacher_id, Teacher_startDate,
        Teacher_endDate, Person_id_role)} >= 2.
%* person_complete(Person_id, Person_name, Person_address,
     Person_phone) :-
   person(Person_id,Person_name,Person_address,Person_phone).
%* teacher_complete(Teacher_id, Teacher_startDate,
     Teacher_endDate, Person_id_role) :-
   teacher(Teacher_id,Teacher_startDate,Teacher_endDate,
     Person_id_role).
%*}
```

For the disjointness and completeness constraints of a generalisation, because the multiset may contain more than one element, the aggregate function `#sum` is used. Additional predicates counting the number of specialisation instances are introduced for every class in the generalisation set.

For a generalisation $\langle g, gc, gS, compl\rangle \in Gen$, for $\langle g, A_g, k_g\rangle \in Classes$, the foreign key referencing gc in every specialisation class $sp \in gS$ is $(gcA_1\_g, \ldots, gcA_{k_{gc}}\_g)$. For the class $sp \in gS$, $freeVars_{sp} = \{spA_1, \ldots, spA_{n_{sp}}\} \setminus \{gcA_1\_g, \ldots, gcA_{k_{gc}}\_g\}$.

Definition 58. The following encoding is used for counting the number of specialisation instances of sp:

$$\begin{aligned} P^{\texttt{count}}_{g,sp} = \{&\texttt{count}(NoS, gcA_1\_g, \ldots, gcA_{k_{gc}}\_g, sp) :- \\ &gc\_\texttt{complete}(gcA_1\_g, \ldots, gcA_{n_{gc}}\_g), \\ &NoS = \#\texttt{count}\{freeVars_{sp} : \\ &\quad sp\_\texttt{complete}(spA_1, \ldots, spA_{n_{sp}})\}\}. \end{aligned}$$

According to the preprocessing step described in Section 3.2.1 and Definition 39, it can never be the case that $freeVars_{sp} = \emptyset$ and therefore we do not have to consider this case separately.

Definition 59. The DLV modification of the encoding of $C_g^{\texttt{disjointness}}$ is:

$$\begin{aligned} C_g^{\texttt{disjointness}} = &P^{\text{complete}}_{gc} \cup P^{\texttt{count}}_{g,sp_1} \cup \cdots \cup P^{\texttt{count}}_{g,sp_{lg}} \cup \\ &\{\texttt{disjointnessViolation}\_g :- \\ &\quad gc\_\texttt{complete}(gcA_1\_g, \ldots, gcA_{n_{gc}}\_g), \\ &\quad \#\texttt{sum}\{NoS, S : \texttt{count}(NoS, gcA_1\_g, \ldots, gcA_{k_{gc}}\_g, S)\} >= 2\}. \end{aligned}$$

Definition 60. The DLV modification of the encoding of $C_g^{\texttt{completeness}}$ is:

$$\begin{aligned} C_g^{\texttt{completeness}} = &P^{\text{complete}}_{gc} \cup P^{\texttt{count}}_{g,sp_1} \cup \cdots \cup P^{\texttt{count}}_{g,sp_{lg}} \cup \\ &\{\texttt{completenessViolation}\_g :- \\ &\quad gc\_\texttt{complete}(gcA_1\_g, \ldots, gcA_{n_{gc}}\_g), \\ &\quad \texttt{not } \#\texttt{sum}\{NoS, S : \texttt{count}(NoS, gcA_1\_g, \ldots, gcA_{k_{gc}}\_g, S)\} = 1\}. \end{aligned}$$

3.3 Chapter Summary

Due to their object oriented approach, only a fragment of the features of UML class diagrams is relevant for modelling the problem domain of an ASP program. The features describing the behaviour of classes or object oriented programming concepts are excluded. However, there is one concept that is not present in UML class diagram—the primary key. In order to handle this inconvenience, the "#" symbol placed in front of an attribute is used to symbolise that the attribute is part of the primary key of the class.

The predicates, arities, types, and meaning of their argument terms are extracted from the model and described in `Lana`. The approach used in the translation is inspired from relational databases—the attributes are mapped to terms, the classes are mapped to atoms, and the associations and generalisations are mapped by introducing foreign keys.

Although a default translation is provided, the user may decide towards a different approach on certain aspects, such as the representation of relationships and the use of multiple so-called *partitioning predicates* to represent a class.

The constraints implied by the graphical models are expressed as assertions. The translation constraints involve the primary keys, the foreign keys, the cardinality of relationships, the referential integrity between the relationships and their association instances, and the disjointness and the completeness of the generalisation.

Both `gringo` and `DLV` syntaxes are supported.

CHAPTER 4

Visualising the Problem Solutions

4.1 The Modified UML Object Diagram

The goal of the solution diagrams is to visualise a snapshot of the objects, as well as the errors in the solution. Therefore, in addition to the usual elements, i.e., instances (objects), generalisations, associations, aggregations, compositions, and association instances, new elements need to be introduced in order to represent the errors. An instance should be uniquely identified by its primary key. Although the specialised instances in the object diagrams normally absorb the attributes of their general instance, representing the generalisation as two instances linked by the generalisation symbol is preferred in order to allow the visualisation of errors such as disjointness or completeness constraint violations.

The additional elements and the errors they help to visualise are presented in the following:

Primary Key Violation Instance. All the instances with the same primary key are represented as one element. When there are more instances with the same primary key, they are merged into one single element, the primary key violation instance.

Primary Key Violation Attribute. In the primary key violation instance, some non-primary key attributes have multiple values. The primary key violation attribute is a compact way to represent an attribute with multiple values.

Empty Instance. An empty instance is an instance with no attribute values. This is not really an instance, but a representation of the class signature. The empty instance is necessary to represent completeness generalisation, association instance, or multiplicity constraint violations, where an instance of a certain type should exist but it does not.

Broken Reference Instance. A broken reference instance is an instance with some attribute values undefined. Incompletely defined instances are the result of either foreign key or integrity constraint violations. The broken reference instances may also contain primary key violation attributes.

Broken Association Instance. A broken association instance is an element representing an association instance or a relationship reference constraint violation—either there exists a relationship without an instance associated to it or there exists an instance associated to more than one relationship.

Source Multiplicity Violation Relationship. The source relationship is the element representing a relationship (aggregation, association, composition) which violates its source multiplicity. This element requires additional information about the multiplicity of the relationship—the multiplicity and the number of relationships starting in the source instance of the relationship.

Target Multiplicity Violation Relationship. The target multiplicity violation relationship is the element representing a relationship (aggregation, association, composition) which violates its target multiplicity. This element requires additional information about the multiplicity of the relationship—the multiplicity and the number of relationships ending in the target instance of the relationship.

Conjunctive Multiplicity Violation Relationship. The conjunctive multiplicity violation relationship is the element representing a relationship (aggregation, association, composition) which violates the source and target multiplicities at the same time. This element requires additional information about the multiplicities of the relationship—the multiplicities and the number of relationships starting and ending in the source and the target instance of the relationship.

Broken Generalisation. This element represents the foreign key constraint violation for a generalisation—i.e, an instance represents the specialisation of multiple general instances.

Disjointness Violation Generalisation. This element represents the disjointness constraint violation. In practice, it is represented as a normal generalisation, but it allows (requires) a generalisation set.

Completeness Violation Generalisation. This element represents the completeness constraint violation—when a generalisation is complete and there is a general instance without its corresponding specialisation instance. The completeness generalisation representation includes a generalisation set with empty instances.

4.1.1 Formal Description

Definition 61. The *UML object diagram* corresponding to the UML class diagram

$$\langle umlName, Assoc, Aggreg, Comp, Gen, Classes, Attr \rangle$$

is a tuple

$$\langle umlName, Instances, EmptyInst, PkViolInst, BrokenRefInst,$$
$$Aggregations, TgtMultViolAggreg, SrcMultViolAggreg, ConjMultViolAggreg,$$
$$Associations, TgtMultViolAssoc, SrcMultViolAssoc, ConjMultViolAssoc,$$
$$Compositions, TgtMultViolComp, SrcMultViolComp, ConjMultViolComp,$$
$$AssociationInstances, BrokenAssocInstances, Generalisation,$$
$$BrokenGen, DisjViolGen, ComplViolGen\rangle,$$

where

- $umlName$ is the name of the UML diagram,
- $Instances$ is the set of correct instances,
- $EmptyInst$ is the set of empty instances,
- $PkViolInst$ is the set of primary key violation instances,
- $BrokenRefInst$ is the set of primary broken instances,
- $Aggregations$ is the set of aggregations,
- $TgtMultViolAggreg$ is the set of target multiplicity violation aggregations,
- $SrcMultViolAggreg$ is the set of source multiplicity violation aggregations,
- $ConjMultViolAggreg$ is the set of conjunctive multiplicity violation aggregations,
- $Associations$ is the set of associations,
- $TgtMultViolAssoc$ is the set of target multiplicity violation associations,
- $SrcMultViolAssoc$ is the set of source multiplicity violation associations,
- $ConjMultViolAssoc$ is the set of conjunctive multiplicity violation associations,
- $Compositions$ is the set of compositions,
- $TgtMultViolComp$ is the set of target multiplicity violation compositions,
- $SrcMultViolComp$ is the set of source multiplicity violation compositions,
- $ConjMultViolComp$ is the set of conjunctive multiplicity violation compositions,
- $AssociationInstances$ is the set of association instances,
- $BrokenAssocInstances$ is the set of broken association instances,
- $Generalisation$ is the set of generalisations,

- $BrokenGen$ is the set of broken generalisations,
- $DisjViolGen$ is the set of disjointness violation generalisations, and
- $ComplViolGen$ is the set of completeness violation generalisations,

and the elements of the diagram are defined as follows:

1. An *attribute* is a pair $(attr, val)$, where $(attr, attrT) \in Classes$; $attr$ and val are the name and the value of the attribute, respectively.
2. An *empty attribute* is an attribute with no value and is represented only by its name, $attr$, where $(attr, attrT) \in Classes$.
3. A *primary key violation attribute* is a pair $(attr, \{val_1, \ldots, val_m\})$, where
$$(attr, attrT) \in Classes$$
and $attr$ and $\{val_1, \ldots, val_m\}$ are the name and the values of the attribute, respectively.
4. An *instance* for a class $\langle c, A_c, k_c\rangle \in Classes$ is a tuple
$$\langle c, pkAttrs, nonPkAttrs\rangle,$$
where $pkAttrs$ and $nonPkAttrs$ are sets of attributes. The following conditions must be fulfilled:
   - $|pkAttrs| = k_c$ and
   - $|A_c| = |pkAttrs \cup nonPkAttrs|$,

   where $|S|$ is the cardinality of a set S.
5. An *empty instance* for a class $\langle c, A_c, k_c\rangle \in Classes$ is a tuple
$$\langle c, (cA_1, \ldots, cA_{k_c}), \{cA_{k_c+1}, \ldots, cA_{n_c}\}\rangle,$$
where all the attributes are empty attributes and
$$A_c = \{cA_1, \ldots, cA_{k_c}, cA_{k_c+1}, \ldots, cA_{n_c}\}.$$
6. A *primary key violation instance* for a class $\langle c, A_c, k_c\rangle \in Classes$ is a tuple
$$\langle c, pkAttrs, nonPkAttrs, pkViolAttribs\rangle,$$
where the sets $pkAttrs$ and $nonPkAttrs$ contain attributes and $pkViolAttribs$ contains primary key violation attributes. The following conditions must be fulfilled:
   - $pkViolAttribs \neq \emptyset$,
   - $|pkAttrs| = k_c$, and

- $|A_c| = |pkAttrs \cup nonPkAttrs \cup pkViolAttribs|$.

7. A *broken reference instance* for a class $\langle c, A_c, k_c \rangle \in Classes$ is a tuple

$$\langle c, pkAttrs, nonPkAttrs \rangle,$$

where $pkAttrs$ and $nonPkAttrs$ contain attributes and empty attributes; $nonPkAttrs$ may also contain primary key violation attributes. The following conditions must be fulfilled:

- there is some $attr \in pkAttrs \cup nonPkAttrs$, where $attr$ is an empty attribute,
- $|pkAttrs| = k_c$, and
- $|A_c| = |pkAttrs \cup nonPkAttrs|$.

8. A *relationship* (*aggregation*, *association*, *composition*) is a tuple

$$\langle relName, sourceInstance, targetInstance \rangle,$$

where $sourceInstance$ and $targetInstance$ represent instances (i.e., instances, empty instances, or broken reference instances).

9. A *source multiplicity violation relationship* (*aggregation*, *association*, *composition*) is a tuple

$$\langle relName, sourceInstance, targetInstance, noSourceInstances, minSrc, maxSrc \rangle,$$

where $sourceInstance$ and $targetInstance$ represent instances (i.e., instances, empty instances, or broken reference instances) and $noSourceInstances > maxSrc$ (if $maxSrc \neq -1$) or $noSourceInstances < minSrc$.

10. A *target multiplicity violation relationship* (*aggregation*, *association*, *composition*) is a tuple

$$\langle relName, sourceInstance, targetInstance, noTargetInstances, minTgt, maxTgt \rangle,$$

where $sourceInstance$ and $targetInstance$ represent instances (i.e., instances, empty instances, or broken reference instances) and $noTargetInstances > maxTgt$ (if $maxTgt \neq -1$) or $noTargetInstances < minTgt$.

11. A *conjunctive multiplicity violation relationship* (*aggregation*, *association*, *composition*) is a tuple

$$\langle relName, sourceInstance, targetInstance, noSourceInstances, \\ minSrc, maxSrc, noTargetInstances, minTgt, maxTgt \rangle,$$

where $sourceInstance$ and $targetInstance$ represent instances (i.e., instances, empty instances, or broken reference instances), $noSourceInstances > maxSrc$ (if $maxSrc \neq -1$) or $noSourceInstances < minSrc$, and $noTargetInstances > maxTgt$ (if $maxTgt \neq -1$), or $noTargetInstances < minTgt$.

12. An *association instance* is a tuple

$$(relationship, instance),$$

where $relationship$ represents a relationship and $instance$ represents the instance associated with it.

13. A *broken association instance* is defined exactly as the association instance, by the tuple

$$(relationship, instance).$$

14. A *generalisation* is a tuple

$$\langle genName, generalInstance, specificInstance \rangle,$$

where $generalInstance$ and $specificInstance$ represent instances (i.e., instances, empty instances or broken reference instances).

15. A *broken generalisation* is defined exactly as the generalisation, by the tuple

$$\langle genName, generalInstance, specificInstance \rangle.$$

16. A *disjointness violation generalisation* is a tuple

$$\langle genName, generalInstance, genSet \rangle,$$

where $generalInstance$ represents an instance of any kind and $genSet$ represents a set of specialisation instances. The following conditions must be fulfilled:

- $|genSet| > 1$ and
- there is no empty instance $instance \in genSet$.

17. A *completeness violation generalisation* is a tuple

$$\langle genName, generalInstance, genSet \rangle,$$

where $generalInstance$ represents an instance of any kind and $genSet$ represents a set of specialisation instances. The following conditions must be fulfilled:

- $|genSet| \geq 1$ and
- every $instance \in genSet$ is an empty instance.

4.2 Mapping the Problem Solution to a UML Object Diagram

The UML class diagram used to model the problem domain, together with the additional preprocessing information (the representation of the relationships and the partitioning predicates) are required for the representation of an answer set as UML object diagram. The UML class diagram is preprocessed according to the provided information, as it was shown in Section 3.2.1.

4.2.1 Collecting the Instances, Relationships, and Generalisations from an Answer Set

Only the predicates present in the partitioning predicate set of one of the classes in the UML diagram are considered in the translation.

In this step, the focus is not on classifying the elements of the diagram according to their errors, but collecting all the information in the answer set in the most compact manner possible. Therefore, the most general representation of the instances, relationships, and generalisations is used.

Let the UML class diagram be represented as in Definition 35 by a tuple

$$\langle umlName, Assoc, Aggreg, Comp, Gen, Classes, Attr\rangle,$$

and let S_c be the partitioning predicate set of the class $\langle c, A_c, k_c\rangle \in Classes$.

Definition 62. A *partial instance* is a set of the form

$$\{(pred\,A_1, pred\,Val_1), \ldots, (pred\,A_{n_{pred}}, pred\,Val_{n_{pred}})\}$$

for which there exists a literal

$$pred(pred\,Val_1, \ldots, pred\,Val_{n_{pred}})$$

in the interpretation and a partitioning predicate

$$pred(pred\,A_1, \ldots, pred\,A_{n_{pred}}) \in S_c$$

for $\langle c, A_c, k_c\rangle \in Classes$.

A partial instance is a direct representation of a literal in the interpretation, according to the information in the preprocessed UML class diagram. The partial instances correspond to the preprocessed classes so they contain also the foreign key attributes introduced for relationships or generalisations.

It is not mandatory that all the attributes of the class are defined in the partial instances. However, since the primary key attributes are present in all the partitioning predicates, all the primary key attributes must be defined.

Let P_c be the set of all partial instances of the class c.

An instance is identified by its primary key attributes and all the partial instances with the same primary key represent particular information about the same instance. The partial instances with the same primary key describing instances with no primary key violation attributes should have the same values for all the attributes.

The instances of the class c are reconstructed by merging all the possible partial instances according to Algorithm 4.2. After applying this algorithm, only the largest partial instances (with respect to subset inclusion) are present in the partial instance sets of the classes.

We need a generic representation which must include all the information contained in the answer set regarding an instance identified by its unique primary key.

input : two partial instances p_1 and p_2
output: true if the two partial instances can be merged

```
1 canMerge(p1,p2)
2 foreach (a, b) ∈ p1 do
3     if (a, x) ∈ p2, x ≠ b then   // the partial instances can be merged
      only if the defined attributes have the same values
4         return false ;
5     end
6     return true ;
7 end
```

Algorithm 4.1: Algorithm `canMerge`.

input : the set of partial instances P_c
output: the set of all possible merged partial instances P_c

```
1  merge(Pc)
2  foreach p1 ∈ Pc do
3      merged ← false;
4      foreach p2 ∈ Pc, p2 ≠ p1 do
5          if canMerge(p1, p2) then
6              merged ← true;
7              Pc ← Pc ∪ {p1 ∪ p2} ;
8          end
9      end
10     if merged then        // remove the partial instances that are
       included in others
11         Pc ← Pc \ {p1} ;
12     end
13 end
```

Algorithm 4.2: Algorithm `merge`.

Definition 63. A *generic instance* of a class $\langle c, (cA_1, \ldots, cA_{n_c}), k_c\rangle \in \mathit{Classes}$ has the following structure:

$$\langle c, pkAttrs, nonPkAttrs, pkViolAttribs\rangle,$$

where:

- $pkAttrs$ and $nonPkAttrs$ represent sets of attributes and have the form

$$\{(attr_1, val_1), \ldots, (attr_n, val_n)\},$$

- $\{attr|(attr, val) \in pkAttrs\} \subseteq \{cA_1, \ldots, cA_{k_c}\}$,

- $pkViolAttribs$ is the set of attributes with multiple values that violate the primary key constraint and has the form

$$\{(attr_1, \{val_{1,1}, \ldots, val_{1,m}\}), \ldots, (attr_n, \{val_{n,1}, \ldots, val_{n,m}\})\},$$

- $\{attr | (attr, val) \in nonPkAttrs \cup pkViolAttribs\} \subseteq \{cA_{k_c+1}, \ldots, cA_{n_c}\}$,

- for each attribute $(attr, val) \in nonPkAttrs$, there is no set $values$ such that $(attr, values) \in pkViolAttribs$, and

- for each attribute $(attr, values) \in pkViolAttribs$, there is no val such that $(attr, val) \in nonPkAttrs$.

While the partial instances correspond to the preprocessed classes, the generic instances correspond to the classes in the initial UML class diagram, thus contain only the proper attributes of the classes.

Let P_c be the set of all partial instances of the class c. The instances of c are extracted from the partial instances according to Algorithm 4.4. The partial instances which do have the same values for their primary key attributes, but different values for some of the non-primary key attributes, are combined in the same generic instance, using the primary key violation attributes.

As with the case of partial instances, it is not mandatory that all the attributes of the class are defined in the generic instances. However, considering the fact that they are extracted from partial instances, their primary key attributes must be defined.

All the relationships (aggregations, associations, compositions) are graphically represented as links between two instances. Ideally, beside one single association instance, a relationship does not need to include anything else. However, due to the possible errors that can be present, more association instances for the same relationship may exist.

Definition 64. A *generic relationship* for a relationship

$$\langle relName, srcCls, tgtCls, minSrc, maxSrc, minTgt, maxTgt, assocCls \rangle \in Assoc \cup Aggreg$$

or

$$\langle relName, srcCls, tgtCls, minTgt, maxTgt, assocCls \rangle \in Comp$$

is a tuple

$$\langle relName, pkSource, pkTarget, pkAssocInstances \rangle,$$

where:

- $pkSource$ and $pkTarget$ are sets containing primary key attributes of the source and the target instance, respectively, and have the form

$$\{(attr_1, val_1), \ldots, (attr_n, val_n)\},$$

input : the set of instances $Inst$, the name of the class c, and the attributes $pkAttrs$ and $nonPkAttrs$

output: the set of instances $Inst$ including the new instance

```
1  extractInstance(className, pkAttrs, nonPkAttrs, Inst)
2  if ⟨className, pkAttrs, nonPkAttrs_1, pkViolAttribs⟩ ∈ Inst then    // the classes
   are identified only by the primary key
3      foreach (attr, val) ∈ nonPkAttrs do
4          if (attr, val_1) ∈ nonPkAttrs_1, val_1 ≠ val then    // multiple values for
           the attribute attr
5              pkViolAttribs ← pkViolAttribs ∪ {(attr, {val, val_1})} ;
6              nonPkAttrs_1 ← nonPkAttrs_1 \ {(attr, val_1)} ;
7          end
8          else if (attr, val_1) ∈ pkViolAttribs, val ∉ val_1 then    // multiple values
           for the attribute attr
9              pkViolAttribs ← pkViolAttribs ∪ {(attr, val_1 ∪ {val})} ;
10         end
11     end
12 end
13 else                                                    // add a new instance
14     Inst ←Inst ∪{⟨className, pkAttrs, nonPkAttrs, ∅⟩} ;
15
```

Algorithm 4.3: Algorithm `extractInstance`.

- $pkAssocInstances$ is the set containing the primary keys of all the association instances linked to the relationship and has the form

$$\{\{(attr_{1,1}, val_{1,1}), \ldots, (attr_{1,n_1}, val_{1,n_1})\}, \ldots, \\ \{(attr_{k,1}, val_{k,1}), \ldots, (attr_{k,n_k}, val_{k,n_k})\}\},$$

- if $\langle srcCls, (srcClsA_1, \ldots, srcClsA_{n_{srcCls}}), k_{srcCls}\rangle \in Classes$, then $\{attr|(attr, val) \in pkSource\} \subseteq \{srcClsA_1, \ldots, srcClsA_{k_{srcCls}}\}$,

- if $\langle tgtCls, (tgtClsA_1, \ldots, tgtClsA_{n_{tgtCls}}), k_{tgtCls}\rangle \in Classes$, then

$$\{attr|(attr, val) \in pkTarget\} \subseteq \{tgtClsA_1, \ldots, tgtClsA_{k_{tgtCls}}\},$$

and

- if $\langle assocCls, (assocClsA_1, \ldots, assocClsA_{n_{assocCls}}), k_{assocCls}\rangle \in Classes$, then for all $pkAssoc \in pkAssocInstances$,

$$\{attr|(attr, val) \in pkAssoc\} \subseteq \{assocClsA_1, \ldots, assocClsA_{k_{assocCls}}\}.$$

input : the set of partial instances P_c for every class c in $Classes\_$
output: the set of instances $Instances$

```
1 Inst ← ∅ ;
2 foreach ⟨c, (cA_1, ..., cA_n), k⟩ ∈ Classes do
3     foreach p ∈ P_c do
4         pkAttrs ← {(attr, val)|(attr, val) ∈ p, attr ∈ {cA_1, ..., cA_k}} ;
5         nonPkAttrs ← {(attr, val)|(attr, val) ∈ p, attr ∈ {cA_{k+1}, ..., cA_n}} ;
6         extractInstance(c, pkAttrs, nonPkAttrs, Inst) ;
7     end
8 end
```

Algorithm 4.4: Algorithm `extractInstances`.

The relationships are represented in the ASP encoding by introducing foreign keys which may be split in several partitioning predicates. Therefore, it may be the case that not all the primary key attributes for the involved instances are defined.

Using the preprocessed UML class diagram, the additional information describing the representation of the relationships and the partial instances, and the generic relationships, are extracted in the set $Relationships$ by gathering the foreign key attributes from the partial instances, as described in Algorithm 4.5.

All the association instances and the relationships contained in the association instances must also be extracted. They are extracted as described in Algorithm 4.6. The association instances which violate the relationship reference constraint are added to the set $RRViolAssocInstances$, represented only through their primary key attributes. The other association instances are represented in the relationship.

A generalisation is represented as a link between two instances. However, it may be the case that there are more specialisation instances for the same general instance. Therefore, the generalisation representation includes its generalisation set.

Definition 65. A *generic generalisation* corresponding to a UML generalisation

$$\langle genName, generalCls, genSet, complete\rangle \in Gen$$

is a tuple

$$\langle genName, pkGeneral, genSet\rangle,$$

where:

- $pkGeneral$ is a set containing primary key attributes of the general instance and has the form

$$\{(attr_1, val_1), \ldots, (attr_n, val_n)\},$$

input : the relationship information, the set of $Classes$ and the set $Relationships$
output: the set $Relationships$ containing the new relationships $relName$

if minSrc $= 1,$ maxSrc $= 1,$ relName *is represented by adding the foreign key of* srcCls *to* tgtCls **then**
  foreach p $\in$ P_{tgtCls} **do**
    pkS $\leftarrow \{(\text{srcCls\_attr}, \text{val}) | (\text{srcCls\_attr\_src\_relName}, \text{val}) \in \text{p}\}$;
    pkT $\leftarrow \{(\text{attr}, \text{val}) | \langle \text{tgtCls}, (\text{tgtClsA}_1, \ldots, \text{tgtClsA}_{n_{tgtCls}}), \text{k}_{tgtCls} \rangle \in$ Classes_, attr $\in \{\text{tgtClsA}_1, \ldots, \text{tgtClsA}_{k_{tgtCls}}\}, (\text{attr}, \text{val}) \in \text{p}\}$;
    Relationships $\leftarrow$ Relationships $\cup \{\langle \text{relName}, \text{pkS}, \text{pkT}, \emptyset \rangle\}$;
  end
end
else if minTgt $= 1,$ maxTgt $= 1,$ relName *is represented by adding the foreign key of* tgtCls *to* srcCls **then**
  foreach p $\in$ P_{srcCls} **do**
    pkT $\leftarrow \{(\text{tgtCls\_attr}, \text{val}) | (\text{tgtCls\_attr\_tgt\_relName}, \text{val}) \in \text{p}\}$;
    pkS $\leftarrow \{(\text{attr}, \text{val}) | \langle \text{srcCls}, (\text{srcClsA}_1, \ldots, \text{srcClsA}_{n_{srcCls}}), \text{k}_{srcCls} \rangle \in$ Classes_, attr $\in \{\text{srcClsA}_1, \ldots, \text{srcClsA}_{k_{srcCls}}\}, (\text{attr}, \text{val}) \in \text{p}\}$;
    Relationships $\leftarrow$ Relationships $\cup \{\langle \text{relName}, \text{pkS}, \text{pkT}, \emptyset \rangle\}$;
  end
end
else if relName *is represented by a new class* relName **then**
  foreach p $\in$ $\mathsf{P}_{relName}$ **do**
    pkS $\leftarrow \{(\text{srcCls\_attr}, \text{val}) | (\text{srcCls\_attr\_src\_relName}, \text{val}) \in \text{p}\}$;
    pkT $\leftarrow \{(\text{tgtCls\_attr}, \text{val}) | (\text{tgtCls\_attr\_tgt\_relName}, \text{val}) \in \text{p}\}$;
    Relationships $\leftarrow$ Relationships $\cup \{\langle \text{relName}, \text{pkS}, \text{pkT}, \emptyset \rangle\}$;
  end
end

Algorithm 4.5: Algorithm `extractRelationships`.

- $genSet$ is the set containing the primary keys of all the specialisation instances of the general instance and has the form

$$\{(sp_1, \{(attr_{1,1}, val_{1,1}), \ldots, (attr_{1,n_1}, val_{1,n_1})\}), \ldots \\ (sp_m, \{(attr_{m,1}, val_{m,1}), \ldots, (attr_{m,n_m}, val_{m,n_m})\})\},$$

- if $\langle generalCls, (generalClsA_1, \ldots, generalClsA_{n_{generalCls}}), k_{generalCls} \rangle \in Classes$, then

$$\{attr | (attr, val) \in pkGeneral\} \subseteq \{generalClsA_1, \ldots, generalClsA_{k_{generalCls}}\},$$

  and

- if $(specificCls, pkSpecific) \in genSet$ and

$$\langle specificCls, (specificClsA_1, \ldots, specificClsA_{n_{specificCls}}), k_{specificCls} \rangle \in Classes$$

input : the relationship information $\langle relName, srcCls, tgtCls, minSrc, maxSrc, minTgt, maxTgt, assocCls\rangle$, the set of $Classes$ and the set $Relationships$

output: the set $Relationships$ containing the new relationships $relName$ and the set $RRViolAssocInstances$

```
1  if assocCls ≠ null then
2      foreach p ∈ P_assocCls do
3          pkS ← {(srcCls_attr, val)|(srcCls_attr_src_relName, val) ∈ p} ;
4          pkT ← {(tgtCls_attr, val)|(tgtCls_attr_tgt_relName, val) ∈ p} ;
5          pkAssoc ←
           {(attr, val)|⟨assocCls, ( assocClsA_1, ..., assocClsA_{n_assocCls} ), k_assocCls⟩ ∈
           Classes, attr ∈ {assocClsA_1, ..., assocClsA_{k_assocCls}}, (attr, val) ∈ p} ;
6          if ⟨relName, pkS, pkT, pkA⟩ ∈ Relationships then
7              pkA ← pkA ∪ {pkAssoc} ;
8          end
9          else if relName is represented solely by its association class assocCls then
10             Relationships ← Relationships ∪ {⟨relName, pkS, pkT, {pkAssoc}⟩} ;
11         end
12         else    // the relationship does not exist, so the assoc
           instance is added to RRViolAssocInstances
13             Relationships ← Relationships ∪ {⟨relName, pkS, pkT, {pkAssoc}⟩} ;
14             RRViolAssocInstances ← RRViolAssocInstances ∪ {pkAssoc} ;
15
16     end
17 end
```

Algorithm 4.6: Algorithm `extractAssociationInstances`.

then

$$\{attr|(attr, val) \in pkSpecific\} = \{specificClsA_1, \ldots, specificClsA_{k_{specificCls}}\}.$$

In the same way as the relationships, the generalisations are represented in the ASP encoding by introducing foreign keys which may be split in several partitioning predicates. It may be therefore the case that not all the primary key attributes for the involved instances are defined. However, the generalisations are represented by introducing the foreign key of the general class to the specific class and thus all the primary key attributes of the specialisation instances, if they exist, are defined.

Using the preprocessed UML class diagram and the partial instances, the generic generalisations are extracted in the set $Generalisations$, by gathering the foreign key attributes from the partial instances, as described in Algorithm 4.7.

input : the general information $\langle genName, generalCls, genSet, complete\rangle$, the set of $Classes$ and the set of generalisations $Generalisations$
output: the set of generalisations $Generalisations$ containing the new generalisations $genName$

```
1  foreach sp ∈ genSet do
2      foreach p ∈ P_sp do
3          pkGeneral ← {(generalCls_attr, val)|(generalCls_attr_genName, val) ∈ p} ;
4          pkSpecific ← {(attr, val)|⟨sp, ( spA_1, ..., spA_{n_sp} ), k_sp⟩ ∈ Classes, attr ∈
           {spA_1, ..., spA_{k_sp}}, (attr, val) ∈ p} ;
5          if ⟨genName, pkGeneral, pkSpecific_1⟩ ∈ Generalisations then
6              pkSpecific_1 ← pkSpecific_1 ∪ {⟨sp, pkSpecific⟩} ;
7          end
8          else Generalisations ←
           Generalisations ∪ {⟨genName, pkGeneral, {⟨sp, pkSpecific⟩}⟩} ;
9      end
10 end
```

Algorithm 4.7: Algorithm `extractGeneralisations`.

4.2.2 Adding Instances and Links

An instance $\langle c, pkAttrs, nonPkAttrs, pkViolAttribs\rangle \in Inst$ is mapped to an object in the UML object diagram according to the sets of attributes defining it. This mapping is realised in terms of Algorithm 4.8.

If all the attributes of the instance are defined and no attribute violates the primary key constraint, the instance is a correct instance. If there exists at least an attribute which is not defined in the instance, the instance is mapped to a broken reference instance. Otherwise, if some of the attributes violate the primary key constraint, the instance is mapped to a primary key violation instance.

A generalisation $\langle genName, pkGeneral, genSet\rangle \in Generalisations$ is mapped to elements in the object diagram according to its generalisation set and the number of general instances for the same specialisation instance. The exact mapping is shown in Algorithm 4.9.

If there are at least two specialisation instances present in the generalisation set, the generalisation is mapped to a disjointness violation generalisation. Otherwise, if there are more general instances for the specialisation instance, the generalisation is a broken generalisation. If none of the aforementioned errors is present in the generalisation, the generalisation is a correct one and is mapped to a normal generalisation.

Moreover, if one of the instances involved in the generalisation is not already among the elements of the object diagram, the instance is mapped to a broken reference instance.

A relationship is mapped to a link between two instances. The number of relationships with the same name starting and ending in the involved instances are taken into account when mapping the relationship. If the multiplicities are within the bounds, the relationship is a correct one. Otherwise, it is mapped to the element in the object diagram corresponding to the violated

input : the set $Inst$ and $Classes$ and the sets of instances in the object diagram
$Instances, EmptyInst, PkViolInst, BrokenRefInst$
output: the updated object diagram

```
1  foreach ⟨c, pkAttr, nonPkAttr, pkViolAttr⟩ ∈ Inst do
2      if ⟨c, {cA_1, ..., cA_n}, k⟩ ∈ Classes then
3          broken ← false ;
4          newPkAttr ← ∅ ;
5          newNonPkAttr ← ∅ ;
6          newPkViolAttr ← ∅ ;
7          if |pkAttr| < k or |pkAttr ∪ nonPkAttr ∪ pkViolAttr| < n then        // some
           attributes are not defined
8              broken ← true ;
9          end
10         foreach attr ∈ {cA_1, ..., cA_k} do
11             if (attr, val) ∈ pkAttr then
12                 newPkAttr ← newPkAttr ∪ {(attr, val)} ;
13                 else newPkAttr ← newPkAttr ∪ {attr} ;
14             end
15         end
16         foreach attr ∈ {cA_{k+1}, ..., cA_n} do
17             if (attr, val) ∈ nonPkAttr then
18                 newNonPkAttr ← newNonPkAttr ∪ {(attr, val)} ;
19                 else if (attr, val) ∈ pkViolAttr then
20                     newPkViolAttr ← newPkViolAttr ∪ {(attr, val)} ;
21                 end
22                 else newNonPkAttr ← newNonPkAttr ∪ {attr} ;
23             end
24         end
25         if broken then
26             BrokenRefInst ←
               BrokenRefInst ∪ {⟨c, newPkAttr, newNonPkAttr ∪ newPkViolAttr⟩} ;
27         end
28         else if newPkViolAttr ≠ ∅ then
29             PkViolInst ← PkViolInst ∪ {⟨c, newPkAttr, newNonPkAttr, newPkViolAttr⟩}
               ;
30         end
31         else Instances ← Instances ∪ {⟨c, newPkAttr, newNonPkAttr⟩} ;
32     end
33 end
```

Algorithm 4.8: Algorithm `addInstances`.

input : the sets $Generalisations$ and $Classes$ and the sets of generalisations in the object diagram $Generalisation$, $BrokenGen$ and $DisjViolGen$
output: the updated object diagram

```
1  foreach ⟨g, pkG, genS⟩ ∈ Generalisations do
2      genI ← getInstance(pkG ) ;
3      if genI = null then          // new broken instance should be added
4          genI ← ⟨gc, pkG ∪ {attr|attr ∈ {gcA_1, ..., gcA_{k_gc}}, ∄(val)((attr, val) ∈
           pkG)}, {gcA_{k_gc+1}, ..., gcA_{n_gc}}⟩ ;
5          BrokenRefInst ← BrokenRefInst ∪ {genI};
6      end
7      if |genS| > 1 then
8          genS_1 ← ∅ ;
9          foreach (sp, pkS) ∈ genS do
10             specI ← getInstance(pkS ) ;
11             if specI = null then          // new broken instance should be
               added
12                 specI ← ⟨sp, pkS ∪ {attr|attr ∈ {spA_1, ..., spA_{k_sp}}, ∄(val)((attr, val) ∈
                   pkS)}, {spA_{k_sp+1}, ..., spA_{n_sp}}⟩ ;
13                 BrokenRefInst ← BrokenRefInst ∪ {specI};
14             end
15             genS_1 ← genS_1 ∪ {specI} ;
16         end
17         DisjViolGen ← DisjViolGen ∪ {⟨g, genI, genS_1⟩} ;
18     end
19     else for (sp, pkS) ∈ genS do
20         specI ← getInstance(pkS ) ;
21         if specI = null then     // new broken instance should be added
22             specI ← (sp, pkS ∪ {attr|attr ∈ {spA_1, ..., spA_{k_sp}}, ∄(val)((attr, val) ∈
               pkS)}, {spA_{k_sp+1}, ..., spA_{n_sp}}) ;
23             BrokenRefInst ← BrokenRefInst ∪ {specI};
24         end
25         if ⟨g, pkG_1, genS_1⟩ ∈ Generalisations, pkG_1 ≠ pkG, specI ∈ genS_1 then
           // multiple generalisation instances
26             BrokenGen ← BrokenGen ∪ {⟨g, genI, specI⟩};
27         end
28         else
29             Generalisation ← Generalisation ∪ {⟨g, genI, specI⟩};
30         end
31     end
32 end
```

Algorithm 4.9: Algorithm `addGeneralisations`.

constraint. The mapping for the associations is described in Algorithm 4.10. The mapping of aggregations and compositions is realised in the same way.

Exactly as in the case of the generalisations, the instances involved in a relationship which are not present among the instances in the object diagram are mapped to broken reference instances. Furthermore, the association instances of the relationships are also added (cf. Algorithm 4.11). If there are more instances associated to one relationship or the association instance belongs to the set $RRViolAssocInstances$, they are mapped to broken association instances.

Up to this point, only the elements present in the answer set were taken into account. However, we also need to represent the lack of elements that should exist but are not present, such as the specialisation instance of a complete generalisation or the mandatory instance or the association instance for a relationship. In order to do that, we need to investigate all the complete generalisations and all the relationships with a mandatory end class in the diagram. Only the generic instances present in $Instances$ are considered when searching for multiplicity or completeness constraint violations. The reason behind that is that all the other instances were not actually present in the answer set and they were only generated as a result of some constraint violation and the goal of the object diagram is to display the errors in the answer set, not to propagate errors.

The complete generalisations are inspected as shown in Algorithm 4.12. For all the complete generalisations in the UML class diagram, if a generalisation instance exists and it does not have any specialisation instance, a completeness violation generalisation is added with the generalisation set being the set of empty instances corresponding to the classes in the generalisation set.

The associations are inspected as shown in Algorithm 4.13. For all the associations in the UML class diagram, if the minimal bound for the target (source) class is greater or equal to 1, it means that for every source (target) instance, there should be at least one association to a target (source) instance. If that is not the case, the target (source) multiplicity constraint is violated and a multiplicity violation association needs to be added.

For aggregations and compositions, the algorithm is exactly the same, only the sets are different.

Example 18. Consider the UML class diagram in Figure 3.2 and the following ASP program:

```
person(1,"R. Fellner", "Pazmaniteng 24-9", "01700731601").
person(2,"Maria Muster", "Weihburggasse 26", "01701731651").
course(1,"Logic").
course(2,"Deductive databases").
course(3,"Nonmonotonic Reasoning").
course(4,"Mathematics").
student(1,"Computer Science", "01-10-2010", "15-10-2012", 1).
student(3,"Computer Science", "01-10-2010", "15-10-2012", 3).
teacher(1,"01-10-2000", "20-12-2020", 2).
studies(1,1).
studies(3,2).
```

input : the sets $Relationships$ and $Classes$, $Assoc$ and the sets of associations in the object diagram $Associations$, $TgtMultViolAssoc$, $SrcMultViolAssoc$, $ConjMultViolAssoc$

output: the updated object diagram

```
1  foreach ⟨r, pkS, pkT, pkAI⟩ ∈ Relationships and
   ⟨r, s, t, minS, maxS, minT, maxT, ac⟩ ∈ Assoc do
2      noT ← |{pkT_1 | ⟨r, pkS, pkT_1 , pkAI_1⟩ ∈ Relationships}| ;
3      noS ← |{pkS_1 | ⟨r, pkS_1 , pkT, pkAI_1⟩ ∈ Relationships}| ;
4      tViol :− (minT > noT) or (maxT ≠ -1 and maxT < noT) ;
5      sViol :− (minS > noS) or (maxS ≠ -1 and maxS < noS) ;
6      srcInst ← getInstance(pkS ) ;
7      if srcInst = null then
8          srcInst ← ⟨s, pkS ∪ {attr|attr ∈ {sA_1, ..., sA_ks}, ∄(val)((attr, val) ∈
           pkS)}, {sA_ks+1, ..., sA_ns}⟩ ;
9          BrokenRefInst ← BrokenRefInst ∪ {srcInst};
10     end
11     tgtInst ← getInstance(pkT ) ;
12     if tgtInst = null then
13         tgtInst ← ⟨t, pkT ∪ {attr|attr ∈ {tA_1, ..., tA_kt}, ∄(val)((attr, val) ∈
           pkT)}, {tA_kt+1, ..., tA_nt}⟩ ;
14         BrokenRefInst ← BrokenRefInst ∪ {tgtInst};
15     end
16     if tViol and sViol then
17         rel :− ⟨r, srcInst, tgtInst, noS, minS, maxS, noT, minT, maxT⟩;
18         ConjMultViolAssoc :− ConjMultViolAssoc ∪ {rel } ;
19     end
20     else if tViol then
21         rel :− ⟨r, srcInst, tgtInst, noT, minT, maxT⟩;
22         TgtMultViolAssoc :− TgtMultViolAssoc ∪ {rel } ;
23     end
24     else if sViol then
25         rel :− ⟨r, srcInst, tgtInst, noS, minS, maxS⟩;
26         SrcMultViolAssoc :− SrcMultViolAssoc ∪ {rel } ;
27     end
28     else
29         rel :− ⟨r, srcInst, tgtInst⟩;
30         Associations :− Associations ∪ {rel } ;
31     end
32     addAssociationInstance(rel, pkAI, ac) ;
33 end
```

Algorithm 4.10: Algorithm `addAssociations`.

input : the relationship rel, the set of association instances pkAI, the set $Relationships$ and $RRViolAssocInstances$
output: the updated object diagram

```
1  if |pkAI| > 1 or pkAI ⊆ RRViolAssocInstances then    // broken association
   instances
2      foreach pkAssoc ∈ pkAI do
3          inst :− getInstance(pkAssoc) ;
4          if inst = null then
5              inst ← ⟨ac, pkAssoc ∪ {attr|attr ∈ {acA_1, ..., acA_{k_ac}}, ∄(val)((attr, val) ∈
               pkAssoc)}, {acA_{k_ac+1}, ..., acA_{n_ac}}⟩ ;
6              BrokenRefInst ← BrokenRefInst ∪ {inst};
7          end
8          BrokenAssocInstances :− BrokenAssocInstances ∪ {(rel, inst)};
9      end
10 end
11 else                                    // normal association instances
12     noR ← |{⟨r_1 , pkS_1 , pkT_1 , pkAI_1⟩|⟨r_1 , pkS_1 , pkT_1 , pkAI_1⟩ ∈
       Relationships, pkAssoc ∈ pkAI_1}| ;
13     foreach pkAssoc ∈ pkAI do
14         inst :− getInstance(pkAssoc) ;
15         if inst = null then
16             inst ← ⟨ac, pkAssoc ∪ {attr|attr ∈ {acA_1, ..., acA_{k_ac}}, ∄(val)((attr, val) ∈
               pkAssoc)}, {acA_{k_ac+1}, ..., acA_{n_ac}}⟩ ;
17             BrokenRefInst ← BrokenRefInst ∪ {inst};
18         end
19         if noR > 1 then
20             BrokenAssocInstances :− BrokenAssocInstances ∪ {(rel, inst)};
21         end
22         else AssociationInstances :− AssociationInstances ∪ {(rel, inst)};
23     end
24
```

Algorithm 4.11: Algorithm `addAssociationInstance`.

input : the sets $Generalisations$, $Inst$ and $Classes$ and the set of completeness violation generalisations $ComplViolGen$ in the object diagram
output: the updated object diagram

foreach $\langle g, gc, genS_1, true\rangle \in$ Gen *and* $\langle$ gc, pkG, nonPkAttrs$\rangle \in$ Instances **do**
  if $\nexists\langle$g, pkG, genS$\rangle \in$ Generalisations **then**
    genI $\leftarrow$ `getInstance(`pkG `)` ;
    emptyGenSet $\leftarrow \emptyset$;
    foreach sp $\in$ genS$_1$ **do**
      emptyI $:-$ $\langle$sp, (spA$_1$, ..., spA$_{k_{sp}}$), {spA$_{k_{sp}+1}$, ..., spA$_{n_{sp}}$}$\rangle$;
      emptyGenSet $:-$ emptyGenSet $\cup$ {emptyI};
      EmptyInst $:-$ EmptyInst $\cup$ {emptyI} ;
    end
    ComplViolGen $:-$ ComplViolGen $\cup$ {$\langle$gc, genI, emptyGenSet$\rangle$};
  end
end

Algorithm 4.12: Algorithm `addCompletenessViolationGeneralisations`.

input : the sets $Relationships$, $Classes$, $Assoc$ and the sets of associations in the object diagram $TgtMultViolAssoc$, $SrcMultViolAssoc$
output: the updated object diagram

foreach $\langle$r, s, t, minS, maxS, minT, maxT, ac$\rangle \in$ Assoc **do**
  if minT >= *1 and* $\langle$ s, pkS, nonPkAttrs$\rangle \in$ Instances *and*
  $\nexists\langle$r, pkS, pkT, pkAI$\rangle \in$ Relationships **then**
    srcInst $:-$ `getInstance(`pkS `)` ;
    emptyI $:-$ $\langle$t, {(tA$_1$, ..., tA$_{k_t}$)}, {tA$_{k_t+1}$, ..., tA$_{n_t}$}$\rangle$;
    TgtMultViolAssoc $:-$ TgtMultViolAssoc $\cup$ {$\langle$r, srcInst, emptyI, 0, minT, maxT$\rangle$};
    EmptyInst $:-$ EmptyInst $\cup$ {emptyI} ;
  end
  if minS >= *1 and* $\langle$ t, pkT, nonPkAttrs$\rangle \in$ Instances *and*
  $\nexists\langle$r, pkS, pkT, pkAI$\rangle \in$ Relationships **then**
    tgtInst $:-$ `getInstance(`pkT `)` ;
    emptyI $:-$ $\langle$s, {(sA$_1$, ..., sA$_{k_s}$)}, {sA$_{k_s+1}$, ..., sA$_{n_s}$}$\rangle$;
    SrcMultViolAssoc $:-$ SrcMultViolAssoc $\cup$ {$\langle$r, emptyI, tgtInst, 0, minS, maxS$\rangle$};
    EmptyInst $:-$ EmptyInst $\cup$ {emptyI} ;
  end
end

Algorithm 4.13: Algorithm `addMultiplicityViolationAssociations`.

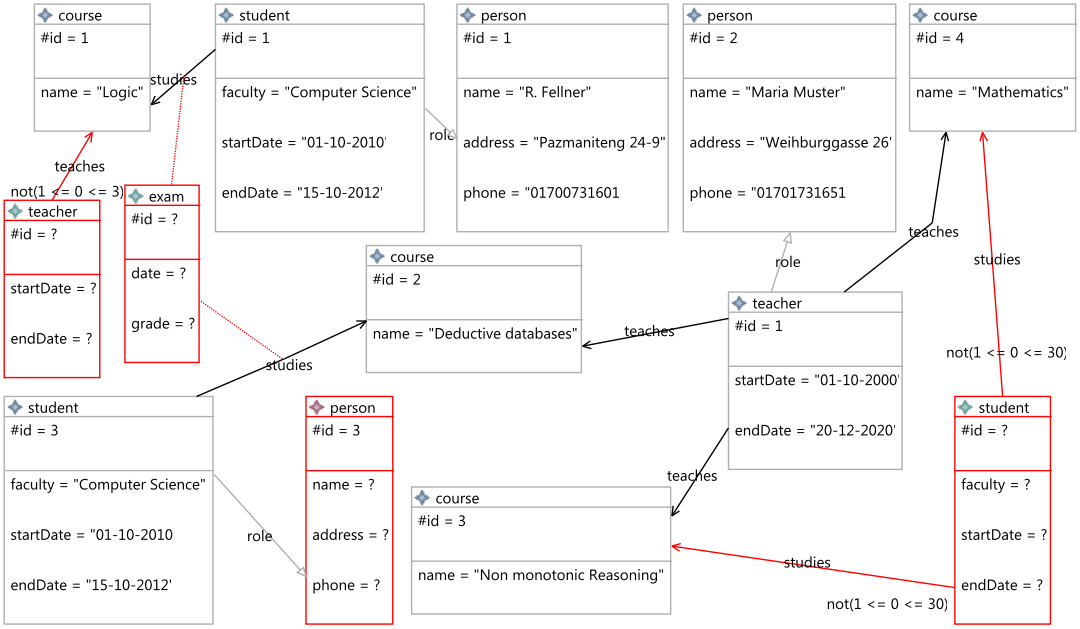

Figure 4.1: Object diagram example

```
teaches(1,2).
teaches(1,3).
teaches(1,4).
```

Then, the UML object diagram is the one in Figure 4.1.

All the literals over the predicates “person”, “course”, “student”, and “teacher” are mapped to the corresponding instances. The associations are represented by the predicates “studies” and “teaches”.

The graphical elements which correspond to certain errors in the diagram are represented in red.

In this example, there are three types of errors: source multiplicity violation associations (for the associations “studies” and “teaches”) for which the empty instances “student” and “teacher” were also introduced, broken reference instance for “person” (as a result of the generalisation “role”), and broken association instance (for the association “studies”).

4.3 Chapter Summary

The UML object diagram is modified in order to display the errors of the problem’s solution. Considering the possible errors that can appear in the answer set, we added primary key violation attributes and instances, empty instances, broken reference instances, broken association instances, source, target and both multiplicity violation relationships, broken generalisations, and disjointness and completeness violation generalisations.

The class diagram and the additional preprocessing information (regarding the representation of relationships and classes) are required to create the graphical representation of the solution.

CHAPTER 5

Implementation

5.1 Graphical Editors in Eclipse using the Graphical Modelling Framework

Our approach is implemented as an Eclipse plugin and is integrated with `SeaLion`. One of the advantages of developing on top of Eclipse rather than creating a new system from scratch is the fact that Eclipse is a widespread platform which offers a variety of tools to help the developer. Another advantage is the possibility to develop platform-independent tools.

The *Graphical Modelling Framework* (GMF) [22] is one of the tools provided by Eclipse. It employs a model-driven engineering approach to generate graphical editors and views based on the *Eclipse Modelling Framework* (EMF) [37] and the *Graphical Editing Framework* (GEF) [20]. GMF provides an integration with the *Eclipse Workbench UI*, such as the toolbar, outline, and property view. Furthermore, it offers two distinct views (graphical and tree view) and implements Undo/Redo via Command and Command Stack and selection and creation tools.

The steps involved in the generation of the diagram editor include different definitions, such as domain model, graphical, tooling, and generation definitions.

Ecore files are used to define the DSML (see Section 2.2). Ecore is a metalanguage which allows the user to define platform-independent models. An Ecore file is basically an XML file with a specific syntax which will be used to support the code generation. Further constraints for the DSML can be specified using *audit rules* or the *Object Constraint Language* (OCL) [21]. They are used to validate the graphical models represented in the editor. The validation may be live, when all the changes that lead to a constraint violation are revoked, or it may be done by invoking the validation command offered by Eclipse.

The graphical definition model is used to define the graphical components (such as links, nodes, and attributes) related with the elements in the domain model. The tooling definition model specifies the palette, the creation tools, and the actions for the graphical elements. The mapping definition model binds the domain, the graphical, and the tooling definition models.

After all the graphical components and mappings are defined, the next step involves a model-

to-model transformation from the mapping definition model to the generation model. The last step is a model-to-code transformation which produces the encoding of the diagram editor as a plugin for Eclipse.

In the resulting editor, the domain model and the diagram information are kept in separate synchronised files. The domain model is described in an Ecore file, which can be rendered with the editor by simply right-clicking the file and selecting "Initialise *_diagram file" from the menu, where "*" represents the extension of the Ecore file.

5.2 Architecture

The architecture of the graphical environment is displayed in Figure 5.1. The system guides the ASP development process the entire way from the design phase to the visualisation of the solution.

In the first step, the user describes the graphical representation of a UML class diagram in the UML class diagram editor. An Ecore file containing the domain model of the UML class diagram is created and is kept synchronised by the editor. The Ecore file will be used as input (together with an optional XML file with additional preprocessing information) for the ASP code generator. The output of the ASP code generator is an answer-set program containing the signature of the program and the constraints expressed in Lana, as detailed in Section 3.2.2. The additional preprocessing information (the representation mode of relationships and the way the classes are split in partitioning predicates) is gathered in an XML file. The syntax of this XML file is described by the following *Document Type Definition* (DTD) [12]:

```
<!DOCTYPE information [
<!ELEMENT information (aggregation* association* composition*
    partitioningPredicate*)>
<!ELEMENT aggregation EMPTY>
<!ATTLIST aggregation
    name CDATA #REQUIRED
    representingMode (0|1|2) "0">
<!ELEMENT association EMPTY>
<!ATTLIST association
    name CDATA #REQUIRED
    representingMode (0|1|2) "0">
<!ELEMENT composition EMPTY>
<!ATTLIST composition
    name CDATA #REQUIRED
    representingMode (0|1|2) "0">
<!ELEMENT partitioningPredicate attribute+>
<!ATTLIST partitioningPredicate
    name CDATA #REQUIRED
    description CDATA ""
    class CDATA #REQUIRED>
```

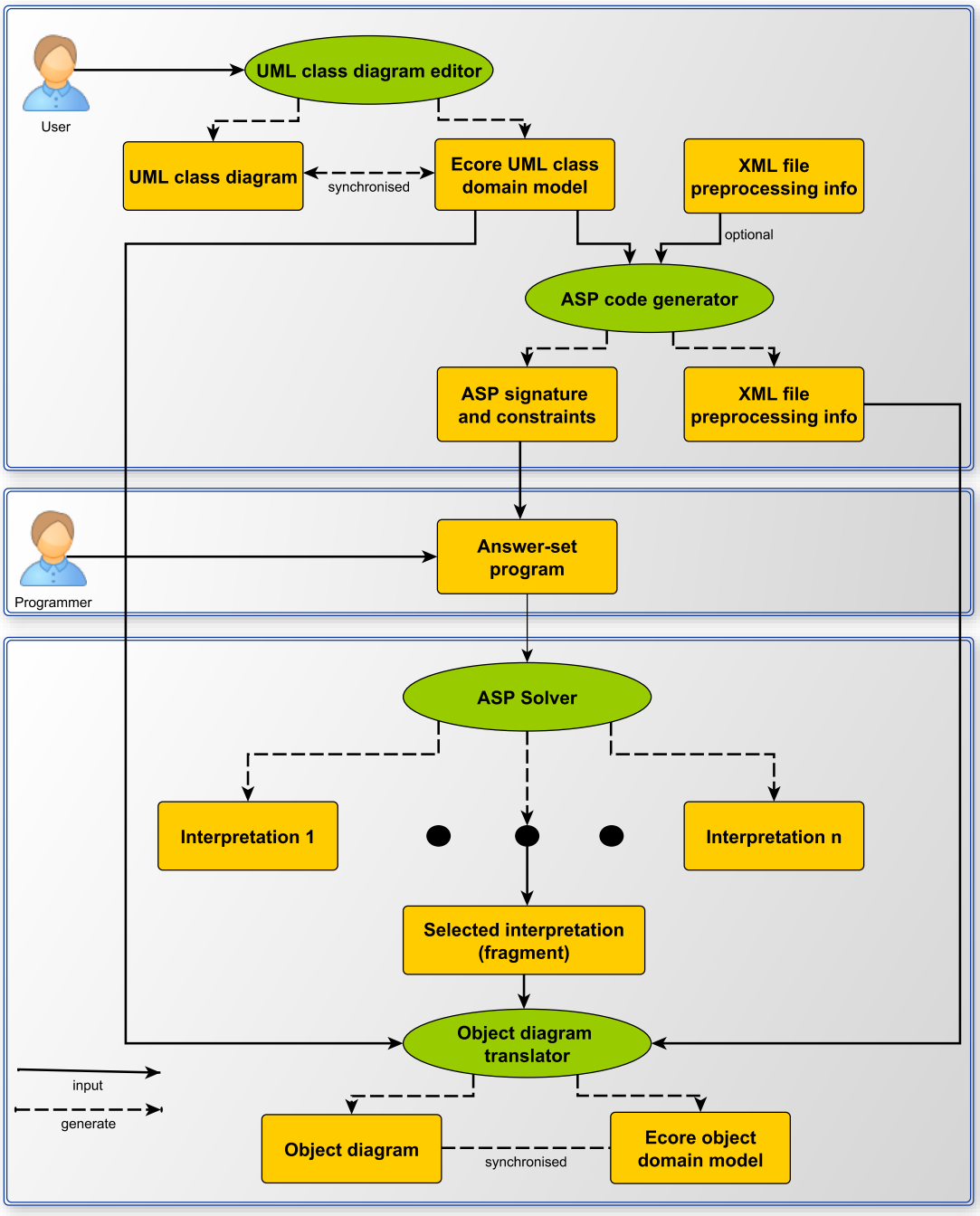

Figure 5.1: The architecture of the graphical modelling environment.

```
<!ELEMENT attribute EMPTY>
<!ATTLIST attribute
    name CDATA #REQUIRED>
]>
```

For every relationship, *representingMode* is 0 if the relationship is represented by adding the foreign key attributes of the mandatory class to the other class, 1 if it is represented by introducing a new predicate containing the foreign key attributes of the classes involved, and 2 if it is represented solely by its association class. The attributes in *partitioningPredicate* represent a subset of the attribute names from a class that are present in this partitioning predicate.

In an XML file representing the output of the ASP code generator, all the attributes of every class from the UML class diagram are present in at least one of its partitioning predicate and the representing mode for every relationship from the UML class diagram is defined. However, this does not have to be true in an XML file used as input.

The programmer (who can be different from the user) proceeds to develop an ASP encoding based on the ASP signature. A solver is used to produce the interpretations of the program from which an interpretation, or a fragment of it, is selected to be visualised. In addition to the selected literals and the Ecore file representing the UML diagram domain model, the XML file with all the preprocessing information is required. Using the UML class diagram as blueprint and the preprocessing information, the translator produces the UML object diagram which will be opened in the UML object diagram editor.

5.3 UML Class Diagram Editor

The meta-model describing the UML class diagram was already depicted in Figure 3.1.

The multiplicities on the references and properties and the containment of the references ensure certain constraints over the graphical editor. The multiplicity 1..1 on all the properties implies that every property of every class has to be set (i.e., the names of the UML class diagram, classes, relationships, the multiplicities of the relationships, and the names and types of the attributes). Some of the properties have default values. The multiplicities are by default 0..1 and the property *complete* for the generalisation is by default *false*. Additional constraints are added using audit rules. Most of the additional constraints are checked in live mode, which implies that any modification leading to their violation is cancelled.

The elements that can be represented in the graphical editor are classes, attributes, primary key attributes, associations, aggregations, compositions, association classes, generalisations, and generalisation sets. The names of all elements contain only alphanumeric characters and they have to start with a letter. Moreover, there cannot be two attributes in the same class or two elements (classes, aggregations, associations, compositions, or generalisations) in the diagram with the same name.

Attribute. An attribute is a label *name : type* and the fact that an attribute is a primary key for a class is represented by adding the symbol # on the left side. The containment property of the references *attributes* and *pkAttributes* ensures the fact that no attribute can exist

standalone. Moreover, the multiplicity $1..*$ on the set of attributes in the *pkAttributes* reference ensures the fact that every class has at least a primary key attribute. Once a class is deleted, all the contained attributes are also deleted.

Class. A class is represented by a rectangle with three compartments. The first one contains the name of the class, the second one its primary key attributes, and the third one its non-primary key attributes. The last two compartments are collapsible.

Association. An association is a line connecting two classes, with an open arrow target decoration. The name of the association and the multiplicities are displayed in the centre and respectively at the corresponding ends. An additional audit rule ensures that the minimal bounds cannot exceed the maximal ones.

Aggregation. An aggregation is a special type of association and is a line connecting two classes, with an empty diamond source decoration and an open arrow target decoration.

Composition. A composition is another special type of association and is a line connecting two classes, with a black filled diamond source decoration and an open arrow target decoration. Another difference is the lack of source multiplicity for the composition—the multiplicity is implicitly $1..1$, due to the ownership meaning of the composition.

Association class. An association class is a dotted line connecting a relationship (namely aggregation, association, or composition) to a class. The multiplicity $0..1$ of the reference $associationClass$ in the Ecore model ensures that there is at most one class linked to an association. Furthermore, the containment property of this reference forces the deletion of the association class whenever the relationship it is connected to is deleted.

Generalisation. A generalisation is a line with a triangle target decoration connecting two classes. The name of the generalisation is actually the name of the generalisation set. The generalisation does not have any multiplicity associated to it. However, it does have a boolean attribute named *complete*. This attribute, set by default to *false*, captures the completeness meaning of the generalisation. The additional constraints expressed as audit rules do not allow generalisations from one class to itself or two different generalisations between the same two classes.

Generalisation Set. A generalisation set is a line connecting a generalisation to a class. It is not a standalone element, due to the containment property of the $generalisationSet$ reference. Therefore, the deletion of the generalisation it is associated with leads to its own deletion.

An example of a UML class diagram was presented in Figure 3.2.

Example 19. Another example in which we added the class "module" and the composition "has" from "module" to "course" is depicted in Figure 5.2. In this figure, the creation tools for the elements of the diagram and the tree view are also visible.

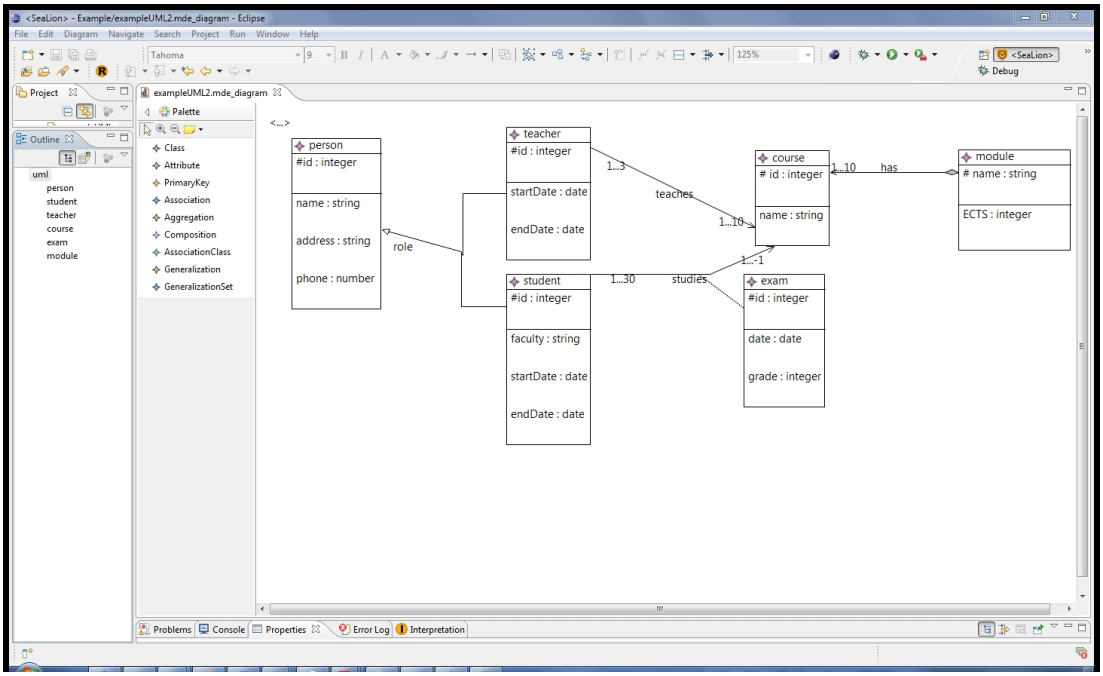

Figure 5.2: The UML class diagram editor.

5.4 The Automatic Translation of UML Class Diagrams to ASP

The input and output of the ASP code generator were already discussed in Section 5.3. In this section, we will discuss only the preprocessing information provided by the user. The dialogue with the user is realised in three different steps.

First, the user specifies the input and the output files and some general preprocessing information which has effect over the entire diagram (the dialogue is represented in Figure 5.3). Optionally, he or she can also provide an initial XML file containing some preprocessing information (which may have been generated before as output or was created manually).

The user can select the desired system (`DLV` or `gringo`) for the encoding and can determine if the program should generate also assertions or only the definitions of terms and atoms. Moreover, the user can decide if the program proceeds to generate all the default predicates. These predicates can be altered afterwards. If the user does not select this option, he or she has to add them manually and finishing at this point is not allowed. However, if this option is selected, the translation can start. That is due to the fact that we need to guarantee that every attribute is present in at least one partitioning predicate.

In the same step, the user can choose to represent all relationships which have an association class solely through the association class. If this option is selected and a relationship does not have an association class, the relationship is handled according to its multiplicities. If the relationship is a many-to-many relationship, it is represented by adding a new predicate. Otherwise, the relationship is a one-to-one (one-to-many) relationship and the foreign key of one class is

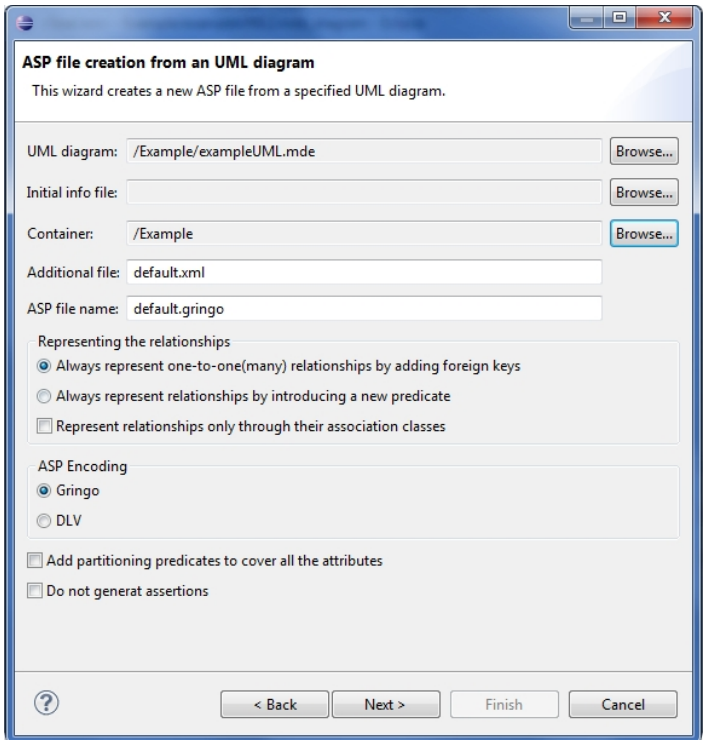

Figure 5.3: Specifying the input and output for the Code Generator.

added to the other, if the option "Always represent one-to-one(many) relationships by adding foreign keys" is selected, or a new predicate is added, if the option "Always represent relationships by introducing a new predicate" is selected. The options regarding the representation of the relationships which can be selected in this step are applied to all relationships in the diagram.

In the next step (see Figure 5.4), the user can change the representation mode for every individual relationship. Only the possible options for the selected relationship are displayed in the dialogue.

The user can also choose to update the predicates of the involved classes (namely the target, source, and association class) so that they will cover all the attributes—for example, that may mean deleting attributes in some predicates and adding a new predicate if the representation mode of the relationship was changed from adding primary keys to adding a new predicate. The predicates are updated for all the relationships that were modified if and only if this option is selected when the button "Next" is pressed. The code generation process cannot start at this moment.

In the last step, the user can add, remove, or modify partitioning predicates (Figure 5.5) for different classes. Every partitioning predicate must include all the primary key attributes of the class. The code generation can start only if all the attributes from the UML class diagram are present in at least a partitioning predicate.

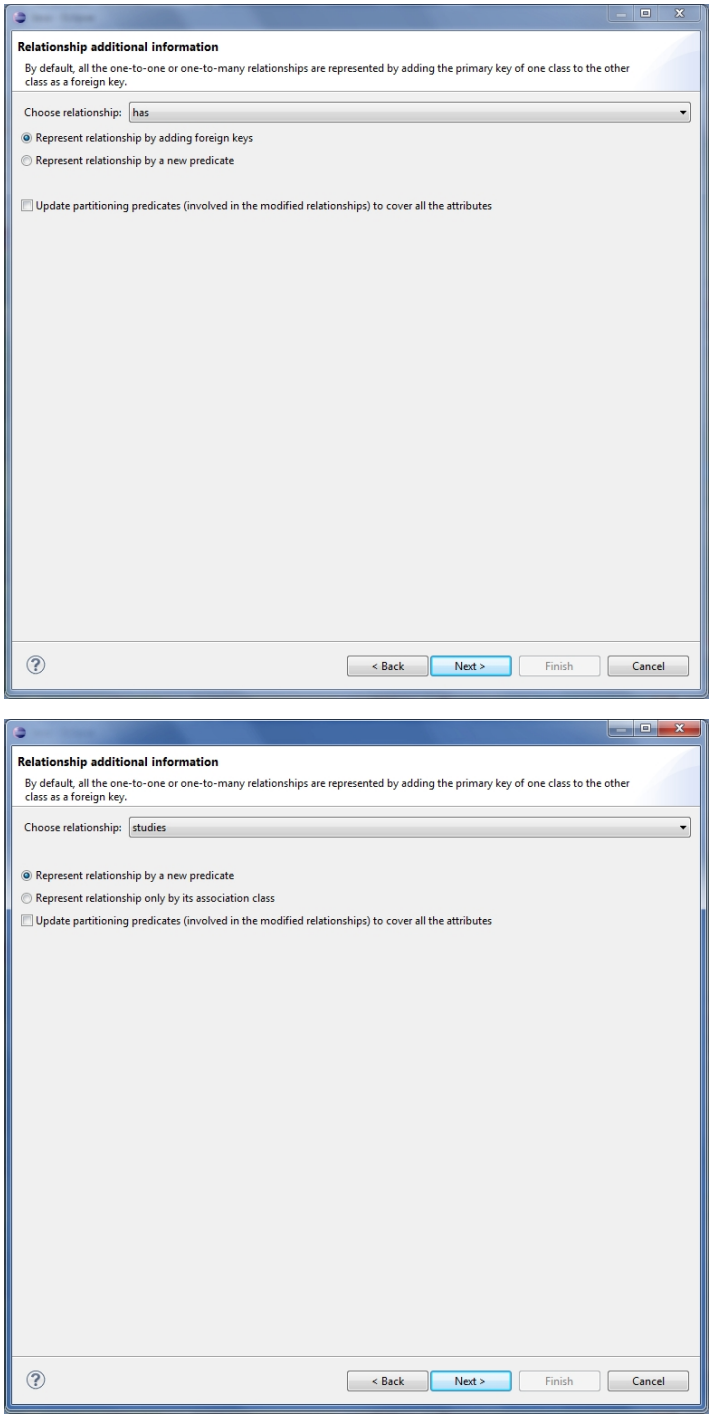

Figure 5.4: Representing individual relationships.

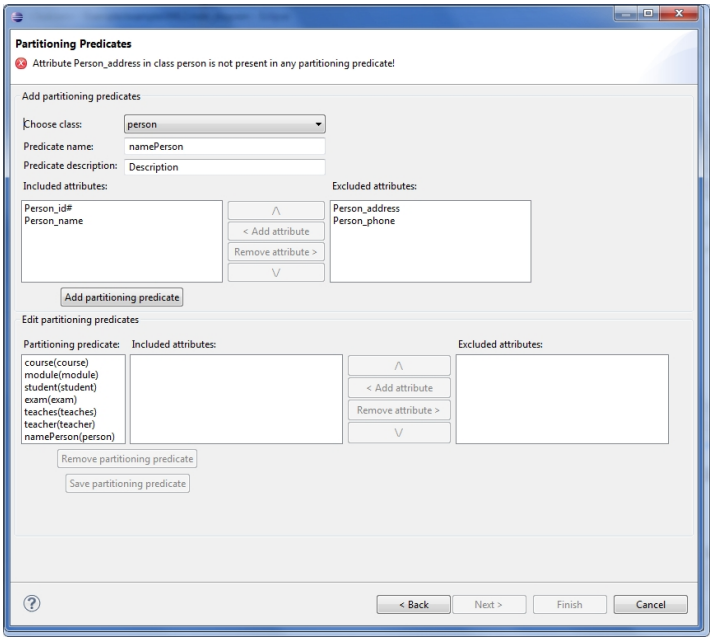

Figure 5.5: Defining partitioning predicates.

5.5 UML Object Diagram Editor

The elements of the UML object diagram editor are divided in two categories: legal elements and elements that depict different types of errors. The legal elements correspond one-to-one to the elements of the UML class diagram, except the generalisation set whose existence in a UML object diagram implies either a disjointness or completeness constraint violation of a generalisation.

We did add some constraints on the elements in the graphical editor, even though the object diagrams are automatically generated and these constraints are taken into account in the translation process.

Attribute. An attribute is a label *name* = *value* and the fact that an attribute is a primary key for an instance is represented by adding the symbol # on the left side. No attribute can exist standalone. Once an instance is deleted, all the contained attributes are also deleted.

Instance. An instance is represented by a rectangle with three compartments. The first one contains the name of the class, the second one its primary key attributes, and the third one its non-primary key attributes. The last two compartments are collapsible.

Association. An association is a line connecting two instances, with an open arrow target decoration. The name of the association is displayed in the centre of the association.

Aggregation. An aggregation is a line connecting two instances, with an empty diamond source decoration and an open arrow target decoration.

Composition. A composition is a line connecting two instances, with a filled diamond source decoration and an open arrow target decoration.

Association instance. An association instance is a dotted line connecting a relationship (namely aggregation, association, or composition) to an instance. No association instance can exist without the relationship they are linked to.

Generalisation. A generalisation is a line with a triangle target decoration connecting two instances. The generalisation does not have a generalisation set because a general instance can have at most one specialisation instance for a legal generalisation.

The additional elements are used to depict different types of errors in the answer set (relative to the UML class diagram used as blueprint and the preprocessing information provided as input). These elements are represented using a different colour (normally red) from the one used by the legal elements.

Primary Key Violation Attribute. A primary key violation attribute is a red rectangle with two compartments. The first one contains the name of the attribute and the second its values (more than one).

Primary Key Violation Instance. A primary key violation instance is a blue rectangle with four compartments. The first one contains the name of the class, the second one its primary key attributes, the third one its non-primary key legal attributes, and the fourth one its primary key violation attributes.

Empty Instance. An empty instance is a red rectangle with three compartments. The first one contains the name of the class, the second one the names of its primary key attributes, and the third one the names of its non-primary key attributes. No attribute has any value defined.

Broken Reference Instance. A broken reference instance is a red rectangle with three compartments. The first one contains the name of the class, the second one its primary key attributes, and the third one its non-primary key attributes. A broken reference instance must have at least an attribute (be it primary key or non-primary key) which has no value. However, among the non-primary key attributes, there may exist also primary key violation attributes.

Broken Association Instance. A broken association instance is a dotted red line connecting a relationship to an instance.

Source/Target/Both Multiplicity Violation Relationship. The source/target/both multiplicity relationship (aggregation, association, composition) is a red line decorated in the same way like its corresponding legal relationship, but it also shows the multiplicity (resp., multiplicities) which is (resp., are) violated together with the actual number of relationships with this name in which the instance is (resp., the instances are) involved. The expression which shows how the multiplicity constraint has been violated has the form

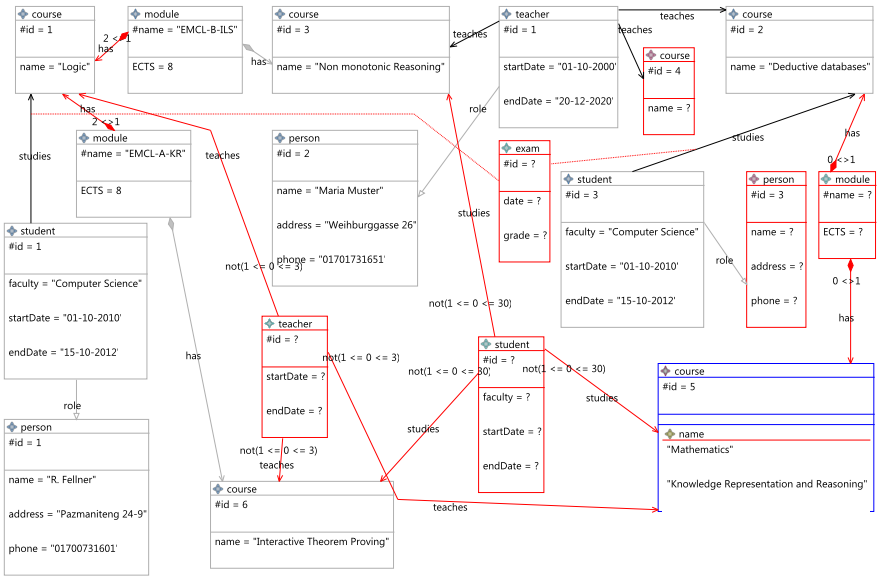

Figure 5.6: Object diagram for Example 20.

not$(min <= no <= max)$, where min and max are the bounds defined in the UML class diagram and no represents the actual number of relationships involving the instance.

Broken Generalisation. A broken generalisation is a red line with a triangle target decoration connecting two instances.

Disjointness Violation Generalisation. A disjointness violation generalisation is a red line with a triangle target decoration connecting two instances. A disjointness violation generalisation must have a generalisation set containing at least two different instances.

Completeness Violation Generalisation. A completeness violation generalisation is a red line with a triangle target decoration connecting two instances for which the specialisation instance is an empty instance. A completeness violation generalisation may have a generalisation set containing only empty instances.

Generalisation Set. A generalisation set is a red line connecting a disjointness or completeness violation generalisation with an instance.

The UML graphical editor allows the user to edit object diagrams, but it does not provide synchronisation with the interpretation.

An example of UML object diagram is represented in Figure 4.1.

Example 20. Consider the following program and the UML class diagram from Figure 5.2 with the default ASP translation of its elements, except for the composition “has”, which we chose to translate by adding the predicate “has(Module_name_src_has,Course_id_tgt_has)”- Then, the UML object diagram corresponding to the solution is the one shown in Figure 5.6.

```
person(1,"R. Fellner", "Pazmaniteng 24-9", "01700731601").
person(2,"Maria Muster", "Weihburggasse 26", "01701731651").
course(1,"Logic").
course(2,"Deductive databases").
course(3,"Nonmonotonic Reasoning").
course(5,"Mathematics").
course(5,"Knowledge Representation and Reasoning").
course(6,"Interactive Theorem Proving").
module("EMCL-B-ILS", 8).
module("EMCL-A-KR", 8).
has("EMCL-B-ILS",1).
has("EMCL-B-ILS",3).
has("EMCL-A-KR",6).
has("EMCL-A-KR",1).
student(1,"Computer Science", "01-10-2010", "15-10-2012", 1).
student(3,"Computer Science", "01-10-2010", "15-10-2012", 3).
teacher(1,"01-10-2000", "20-12-2020", 2).
studies(1,1).
studies(3,2).
teaches(1,2).
teaches(1,3).
teaches(1,4).
```

5.6 The Automatic Translation of Interpretations to UML Object Diagrams

Note that the input and the output provided by the user have already been discussed in Section 5.3, and the translation of interpretations has already been detailed in Chapter 4.

The only aspect that was not yet discussed is how the literals from an answer set are mapped to UML object elements when only a fragment of the interpretation is selected. If the user selects to visualise only a fragment of an interpretation, the object diagram will still represent the entire interpretation, with the observation that only the instances that are present (either through their primary keys or their foreign keys) in the selected literals are visible. Furthermore, only the relationships between two visible instances are displayed. All the other elements are hidden.

The user can choose from the menu to display all the elements of the diagram.

Example 21. We consider the same UML class diagram, the same ASP translation of its elements, and the same program used in Example 20. If the user chooses to visualise only the literals

```
course(5, "Knowledge Representation and Reasoning"),
module("EMCL-A-KR", 8),
student(3, "Computer Science", "01-10-2010", "15-10-2012", 3),
studies(1, 1).
```

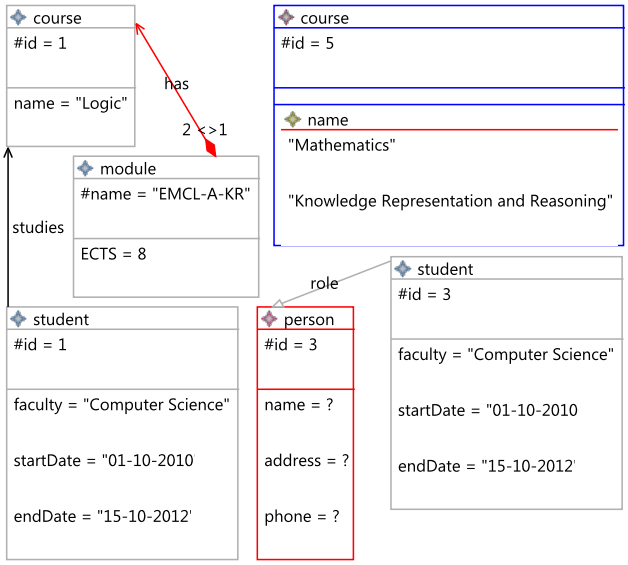

Figure 5.7: Object diagram for Example 21.

the corresponding UML object diagram is represented in Figure 5.7.

As one can easily see, even if the literal "course(5,"Mathematics")" is not present among the selected literals, its presence in the interpretation still introduces the primary key violation instance. Furthermore, even if in the UML object diagram, only one composition "has" for the course "Knowledge Representation and Reasoning" is visible, there exists another one in the interpretation and thus this composition violates the source multiplicity and it is represented accordingly.

5.7 Chapter Summary

The graphical modelling environment is a round-trip graphical tool which guides the user in the development of answer-set programs.

It implements different tools—two UML diagram editors, an ASP code generator, and a UML object diagram translator.

The code generator allows the user a certain amount of flexibility.

Concerning the visualisation of the solution, the user may select to visualise a fragment of an interpretation, instead of the entire interpretation. However, even though the resulting UML object diagram contains only the graphical elements which are present in the fragment, the elements are represented in the context of the UML object diagram of the entire interpretation.

CHAPTER 6

Example

In this chapter, we illustrate the intended usage of the tool described previously. In order to do this, we consider a basic example, which requires us to solve the scheduling of courses.

We assume that each weekday is divided into one-hour timeslots which are spread over one week. A course is held for a fixed number of hours per week and a lesson is a part of a course corresponding to a timeslot. Given a set of teachers and courses, our encoding must create a time schedule which allocates lessons in such a way that a teacher does not hold more than one lesson at a time.

In a first step, we represent the requirements of the problem as a UML class diagram depicted in Figure 6.1. Due to the association "teaches" and its association class "lesson", a teacher cannot teach two different lessons in the same timeslot and a lesson cannot be taught by two different teachers. Furthermore, due to the composition "lessons", a lesson is associated to one and only one course.

Afterwards, we generate the predicate signatures and constraints of the program, a fragment of which is represented below:

```
%**
@block schedule

@term Course_id
id of course

@term Course_name
name of course

@term Course_hoursPerWeek
hoursPerWeek of course

@term TimeSlot_day
```

<...>

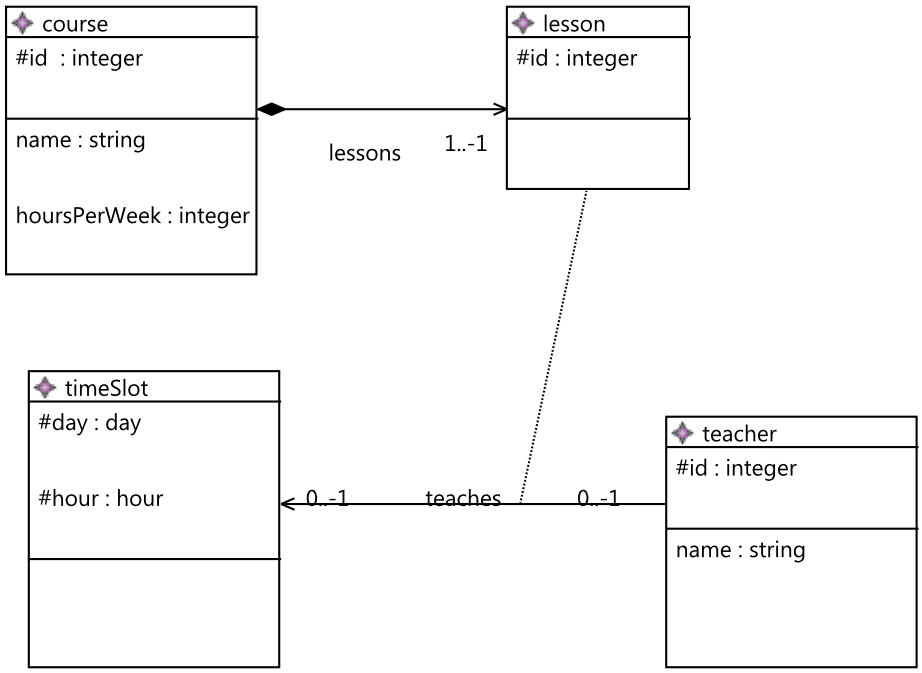

Figure 6.1: The UML class diagram of the scheduling problem.

```
day of timeslot

@term TimeSlot_hour
hour of timeslot

@term Lesson_id
id of lesson

@term Course_id_src_lessons
id foreign key attribute referencing course
@samerangeas Course_id

@term Teacher_id_src_teaches
id foreign key attribute referencing teacher
@samerangeas Teacher_id

@term TimeSlot_day_tgt_teaches
day foreign key attribute referencing timeslot
```

```
@samerangeas TimeSlot_day

@term TimeSlot_hour_tgt_teaches
hour foreign key attribute referencing timeslot
@samerangeas TimeSlot_hour

@term Teacher_id
id of teacher

@term Teacher_name
name of teacher

@atom course(Course_id,Course_name,Course_hoursPerWeek)
completely describes course
@atom timeSlot(TimeSlot_day,TimeSlot_hour)
completely describes timeSlot
@atom lesson(Lesson_id,Course_id_src_lessons,Teacher_id_src_teaches,
   TimeSlot_day_tgt_teaches,TimeSlot_hour_tgt_teaches)
completely describes lesson
@atom teaches(Teacher_id_src_teaches,TimeSlot_day_tgt_teaches,
   TimeSlot_hour_tgt_teaches)
completely describes teaches
@atom teacher(Teacher_id,Teacher_name)
completely describes teacher

@assert associationInstanceViolation_teaches_lesson
every relationship teaches must have an association instance lesson
@never associationInstanceViolation_teaches_lesson
associationInstanceViolation_teaches_lesson :-
   teaches_complete(Teacher_id_src_teaches, TimeSlot_day_tgt_teaches,
      TimeSlot_hour_tgt_teaches), not 1 lesson_complete(Lesson_id,
      Course_id_src_lessons, Teacher_id_src_teaches,
      TimeSlot_day_tgt_teaches, TimeSlot_hour_tgt_teaches) 1.
teaches_complete(Teacher_id_src_teaches, TimeSlot_day_tgt_teaches,
      TimeSlot_hour_tgt_teaches) :- teaches(Teacher_id_src_teaches,
      TimeSlot_day_tgt_teaches,TimeSlot_hour_tgt_teaches).
lesson_complete(Lesson_id, Course_id_src_lessons,
      Teacher_id_src_teaches, TimeSlot_day_tgt_teaches,
      TimeSlot_hour_tgt_teaches) :- lesson(Lesson_id,
      Course_id_src_lessons, Teacher_id_src_teaches,
      TimeSlot_day_tgt_teaches, TimeSlot_hour_tgt_teaches).
```

```
...


*%

%**
*%
```

We proceed with the encoding of the problem:

```
hour(1..4).
day("Monday").
day("Tuesday").
day("Wednesday").
day("Thursday").
day("Friday").

timeSlot(D,H) :- day(D), hour(H).
lesson(Id*100+(1..Nr),Id,TId):-course(Id,_,Nr), teaches(TId,Id).
1{lessonSchedule(Id,D,H) : timeSlot(D,H)}1 :- lesson(Id,_,_).
lesson(Id, CId, TId, D, H) :-  lessonSchedule(Id, D, H),
   lesson(Id, CId, TId).
teaches(TId, D, H) :- lesson(_, _, TId, D, H).
```

Considering the input

```
course(1,"Maths I",4).
course(2,"Maths II",5).
course(3,"Maths III",3).
course(4,"Maths IV",2).
course(5,"Maths V",6).
course(6,"Anorganic Chemistry",3).
course(7,"Organic Chemistry",3).
course(8,"Thermodynamics",2).

teacher(1,"Max Musterman").
teacher(2,"John Richard").
teaches(1, 1..5).
teaches(2, 6..8).
```

we notice that in the solution all the lessons are mapped to the same timeslot, which makes it invalid. Therefore, we have to assume that we made a mistake in the encoding. If we activate the assertions, we additionally notice the presence of the constraint violation atom "association-InstanceViolation_teaches_lesson". Furthermore, the broken associations in the corresponding

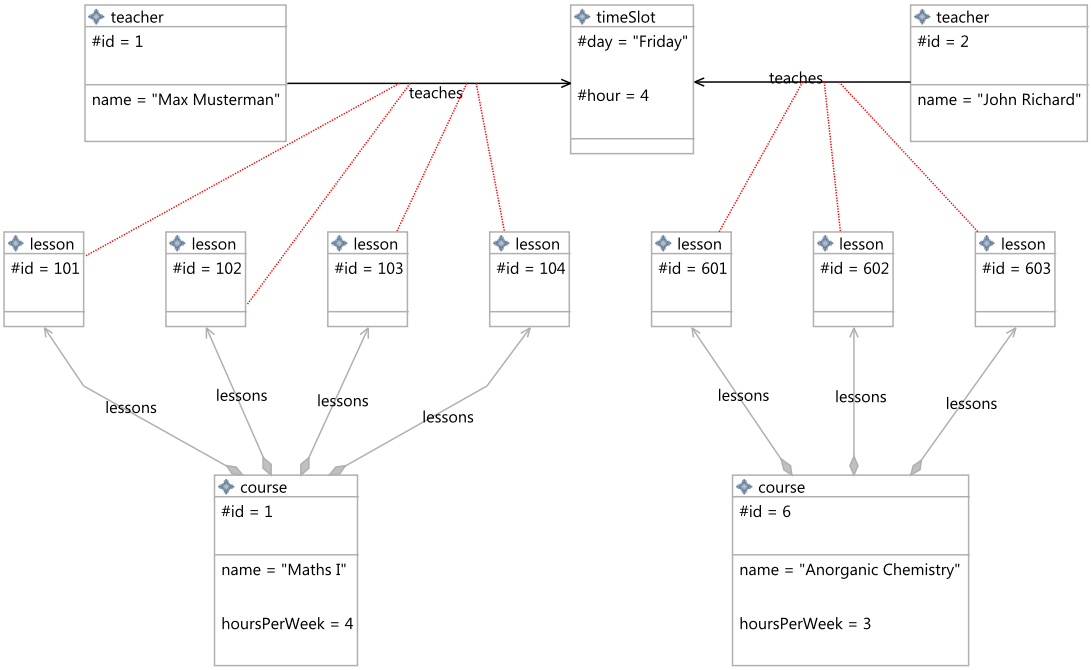

Figure 6.2: The UML class diagram of the scheduling problem.

UML class diagram (a fragment of which is displayed in Figure 6.2) lead us to the same conclusion.

After further analysis of the ASP program, we determine that in the initial program we omitted a constraint regarding the fact that a teacher cannot teach two different lessons at the same time. This constraint is

```
:- teacher(TId,_), timeSlot(D,H), 2{lesson(Id,CId,TId,D,H)}.
```

In the presence of the new rule, the scheduling of lessons is

```
lesson(101, 1, 1, "Wednesday", 2).
lesson(102, 1, 1, "Tuesday", 2).
lesson(103, 1, 1, "Monday", 3).
lesson(104, 1, 1, "Friday", 1).
lesson(201, 2, 1, "Friday", 2).
lesson(202, 2, 1, "Thursday", 4).
lesson(203, 2, 1, "Tuesday", 1).
lesson(204, 2, 1, "Thursday", 2).
lesson(205, 2, 1, "Monday", 2).
lesson(301, 3, 1, "Monday", 1).
lesson(302, 3, 1, "Wednesday", 3).
lesson(303, 3, 1, "Wednesday", 4).
```

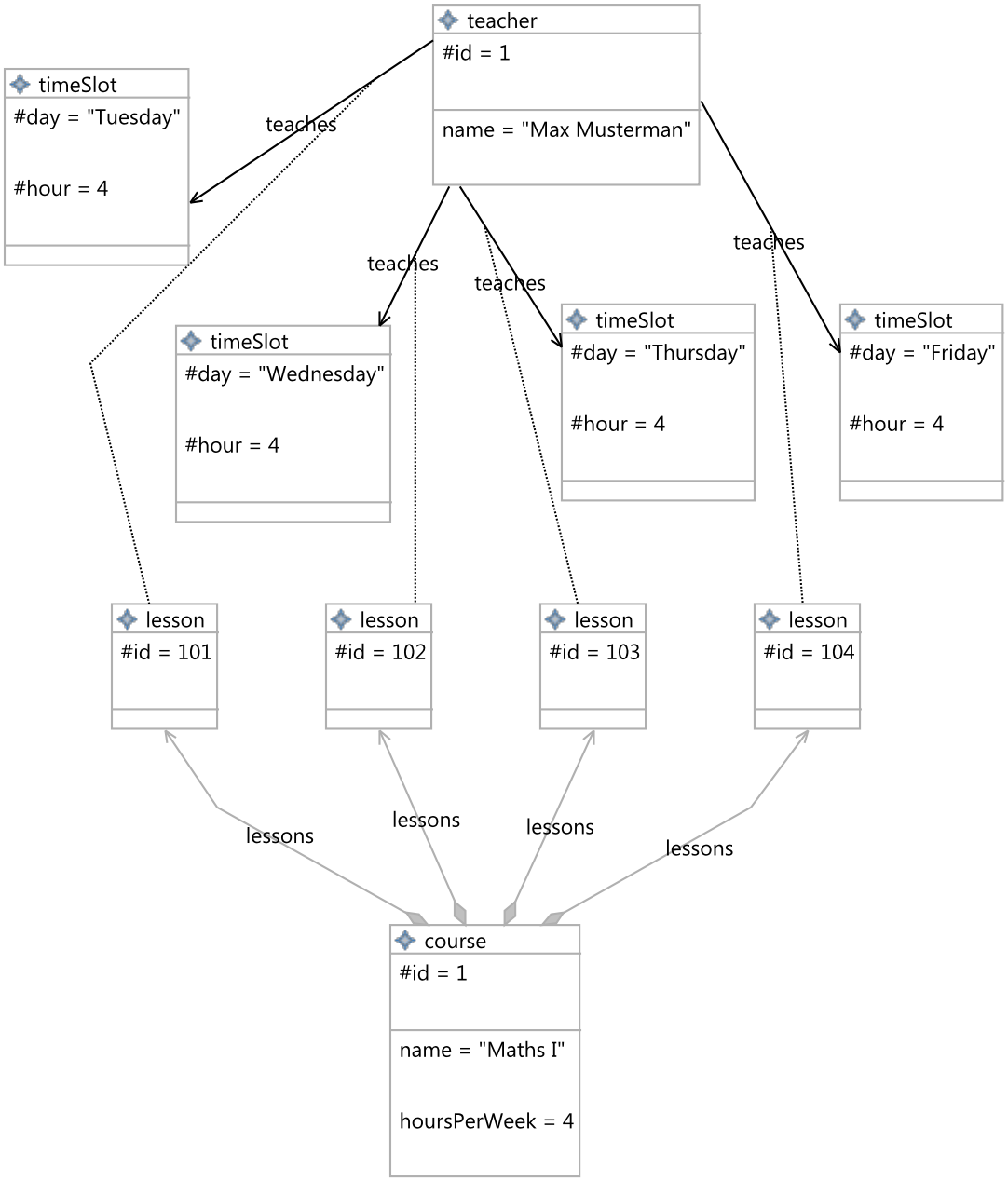

Figure 6.3: Part of the UML object diagram of the solution.

```
lesson(401, 4, 1, "Tuesday", 3).
lesson(402, 4, 1, "Wednesday", 1).
lesson(501, 5, 1, "Thursday", 3).
lesson(502, 5, 1, "Friday", 4).
lesson(503, 5, 1, "Thursday", 1).
lesson(504, 5, 1, "Tuesday", 4).
lesson(505, 5, 1, "Monday", 4).
lesson(506, 5, 1, "Friday", 3).
lesson(601, 6, 2, "Wednesday", 1).
lesson(602, 6, 2, "Tuesday", 1).
lesson(603, 6, 2, "Monday", 1).
lesson(701, 7, 2, "Monday", 2).
lesson(702, 7, 2, "Friday", 1).
lesson(703, 7, 2, "Thursday", 1).
lesson(801, 8, 2, "Wednesday", 2).
lesson(802, 8, 2, "Tuesday", 2).
```

This is indeed a valid solution which can be attested by the lack of any constraint violation atoms in the answer set as well as by the lack of any errors in the corresponding UML object diagram (partially represented in Figure 6.3).

CHAPTER 7

Conclusions and Related Work

In this thesis, we presented a solution to apply model-driven techniques used in software engineering in order to ease the development process of answer-set programs.

On the one hand, MDE has extensively been applied to model relational databases. There are several open-source tools handling the visualisation of existent tables as UML class diagrams and keeping the database synchronised by generating scripts for creating new tables and relationships or for modifying existing ones. `ArgoUML` [2], `UModel` [39], `Agile Data` [1], and `MagicDraw` [29] are some examples of such tools. The tools are used to model relational databases—schemas, tables, relationships, views, indexes, default values, and different constraints such as primary or foreign keys. They all use UML class diagrams and stereotypes to represent the database diagrams.

On the other hand, adapting MDE techniques in ASP is still not a fully explored area. A first step towards using graphical models to support the design of answer-set programs has been realised by `VIDEAS`, which introduces model-to-code generation mappings and code generation to ASP. `VIDEAS` uses ER diagrams to model answer-set programs and as a basis to automatically generate constraints before the programmer proceeds with an encoding. In addition, `FactBuilder` provides the user a modality to define a consistent fact base. `FactBuilder` is a command-line tool which ensures the satisfaction of constraints represented in the ER diagram when facts are entered.

Our approach involves further developments in adapting MDE for ASP which results in an Eclipse plugin consisting of two graphical modelling editors (namely the UML class diagram and the UML object diagram editors), a code generator from UML class diagrams to ASP, and the translation of answer sets to UML object diagrams. The generated ASP code does not impose any constraints on the subsequent development process.

Future work may involve the consideration of additional elements of the UML class diagram, such as enumerations to specify the type of the attributes, default values, relationship inheritance, relationships involving an arbitrary number of classes, and multiplicities for attributes and qualified associations. Furthermore, keeping the ASP encoding and the UML class diagram synchronised in both directions, with the possibility to illustrate the changes involved

and the elements that are affected (directly or indirectly), may be beneficial.

The functionality of the UML object diagram editor can be extended from using it solely to visualise the answer sets and the inconsistencies with the problem model to allowing the user to visually edit the interpretations. In the current state, the editor can be used to edit graphical models, but the changes are not reflected in the interpretation.

Bibliography

[1] Agile Data - A UML Profile for Data Modeling. http://www.agiledata.org/essays/umlDataModelingProfile.html.

[2] ArgoUML Database Modeling. http://argouml-db.tigris.org/documentation/UML_Model.htm.

[3] Marcello Balduccini. Representing Constraint Satisfaction Problems in Answer Set Programming. In *Proceedings of the 2nd Workshop on Answer Set Programming and Other Computing Paradigms* (*ASPOCP 2009*), 2009.

[4] Marcello Balduccini and Gelfond Michael. Diagnostic Reasoning with A-Prolog. *Theory and Practice of Logic Programming*, 3(4-5):425–461, 2003.

[5] Chitta Baral. *Knowledge Representation, Reasoning and Declarative Problem Solving*. Cambridge University Press, 2003.

[6] Robert Bihlmeyer, Wolfgang Faber, Giuseppe Ielpa, Vincenzino Lio, and Gerald Pfeifer. DLV User Manual. http://www.dlvsystem.com/dlvsystem/html/DLV_User_Manual.html.

[7] Francesco Calimeri and Francesco Ricca. On the Application of the Answer Set Programming System DLV in Industry: A Report from the Field. *ALP Newsletter*, March 2012. http://www.cs.nmsu.edu/ALP/wp-content/uploads/2012/03/cali-ricc-alp-apps.pdf.

[8] Samarjit Chakraborty. Formal Languages and Automata Theory-Regular Expressions and Finite Automata. *Computer Engineering and Networks Laboratory*, 2003.

[9] Andrea De Lucia, Carmine Gravino, Rocco Oliveto, and Genoveffa Tortora. An Experimental Comparison of ER and UML Class Diagrams for Data Modelling. *Empirical Software Engineering*, 15(5):455–492, 2010.

[10] Marina De Vos, Doga Gizem Kisa, Johannes Oetsch, Jörg Pührer, and Hans Tompits. Annotating Answer-Set Programs in LANA. *Theory and Practice of Logic Programming*, 12(4-5):619–637, 2012.

[11] Yannis Dimopoulos, Bernhard Nebel, and Jana Koehler. Encoding planning problems in nonmonotonic logic programs. In Sam Steel and Rachid Alami, editors, *Recent Advances in AI Planning*, volume 1348 of *Lecture Notes in Computer Science*, pages 169–181. Springer Berlin / Heidelberg, 1997.

[12] DTD Tutorial. `http://www.w3schools.com/dtd/default.asp`.

[13] Thomas Eiter, Giovambattista Ianni, Thomas Lukasiewicz, Roman Schindlauer, and Hans Tompits. Combining answer set programming with description logics for the semantic web. *Artificial Intelligence*, 172(12–13):1495 – 1539, 2008.

[14] Wolfgang Faber, Gerald Pfeifer, Nicola Leone, Tina Dell'Armi, and Giuseppe Ielpa. Design and Implementation of Aggregate Functions in the DLV System. *Theory and Practice of Logic Programming*, 8(5–6):545–580, 2008.

[15] Onofrio Febbraro, Kristian Reale, and Francesco Ricca. A Visual Interface for Drawing ASP Programs. In Wolfgang Faber and Nicola Leone, editors, *Proceedings of the 25th Italian Conference on Computational Logic* (*CILC 2010*), volume 598 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2010.

[16] Onofrio Febbraro, Kristian Reale, and Francesco Ricca. ASPIDE: Integrated Development Environment for Answer Set Programming. In *Proceedings of the 11th International Conference on Logic Programming and Nonmonotonic Reasoning* (*LPNMR 2011*), volume 6645 of *Lecture Notes in Computer Science*, pages 317–330. Springer, 2011.

[17] Martin Fowler. *UML Distilled: A Brief Guide to the Standard Object Modeling Language*. Addison-Wesley Professional, 2004.

[18] Martin Gebser, Roland Kaminski, Benjamin Kaufmann, Max Ostrowski, Torsten Schaub, and Sven Thiele. A User's Guide to gringo, clasp, clingo, and iclingo. Unpublished draft, 2008. `http://downloads.sourceforge.net/potassco/guide.pdf?use_mirror=`.

[19] M. Gelfond and V. Lifschitz. The Stable Model Semantics for Logic Programming. In *Proceedings of the 5th International Conference and Symposium on Logic Programming* (*ICLP '88*), pages 1070–1080. MIT Press, 1998.

[20] Graphical Editing Framework. `http://www.eclipse.org/gef/`.

[21] Graphical Modeling Framework Constraints. `http://wiki.eclipse.org/GMF_Constraints`.

[22] Graphical Modeling Framework Tutorial. `http://wiki.eclipse.org/Graphical_Modeling_Framework/Tutorial`.

[23] David C. Hay. *UML and Data Modelling: A Reconciliation*. Technics Publications, 2011.

[24] IDPDraw. `http://dtai.cs.kuleuven.be/krr/software`.

[25] iGROM. http://igrom.sourceforge.net/.

[26] Tom Jewett. Database design with UML and SQL. http://www.tomjewett.com/dbdesign/. Department of Computer Engineering and Computer Science, California State University, Long Beach.

[27] Christian Kloimüllner, Johannes Oetsch, Jörg Pührer, and Hans Tompits. Kara: A System for Visualising and Visual Editing of Interpretations for Answer-Set Programs. In *Proceedings of the 19th International Conference on Applications of Declarative Programming and Knowledge Management* (*INAP 2011*) *and 25th Workshop on Logic Programming* (*WLP 2011*), pages 152–164, 1843-11-06, 2011. INFSYS Research Report.

[28] Vladimir Lifschitz. Answer Set Planning. In *Proceedings of the 16th International Conference on Logic Programming* (*ICLP '99*), pages 23–37. The MIT Press, 1999.

[29] MagicDraw - Applying UML for Relational Data Modeling. http://www.magicdraw.com/files/articles/Sep04%20Applying%20UML%20for%20Relational%20Data%20Modeling.html.

[30] Joaquin Miller and Jishnu Mukerj. Model Driven Architecture (MDA) Document Number ormsc/2001-07-01. http://www.omg.org/cgi-bin/apps/doc?ormsc/01-07-01.pdf, 2001.

[31] Johannes Oetsch, Jörg Pührer, Martina Seidl, Hans Tompits, and Patrick Zwickl. VIDEAS: A Development Tool for Answer-Set Programs based on Model-Driven Engineering Technology. In *Proceedings of the 11th International Conference on Logic Programming and Nonmonotonic Reasoning* (*LPNMR 2011*), volume 6645 of *Lecture Notes in Computer Science*, pages 382–387. Springer, 2011.

[32] Johannes Oetsch, Jörg Pührer, and Hans Tompits. Methods and Methodologies for Developing Answer-Set Programs—Project Description. In *Technical Communications of the 26th International Conference on Logic Programming* (*ICLP 2010*), volume 7 of *Leibniz International Proceedings in Informatics, Schloss Dagstuhl—Leibniz-Zentrum für Informatik*, pages 154–161, 2010.

[33] Johannes Oetsch, Jörg Pührer, and Hans Tompits. The SeaLion has Landed: An IDE for Answer-Set Programming—Preliminary Report. In *19th International Conference on Applications of Declarative Programming and Knowledge Management* (*INAP 2011*) *and 25th Workshop on Logic Programming* (*WLP 2011*), pages 141–151. INFSYS Research Report 1843-11-06, 2011.

[34] Cliffe Owen, Marina De Vos, Martin Brain, and Julian Padget. ASPVIZ: Declarative Visualisation and Animation using Answer Set Programming. In *Proceedings of the 24th International Conference on Logic Programming* (*ICLP 2008*), volume 5366 of *Lecture Notes in Computer Science*, pages 724–728, 2008.

[35] Peter Phin-Shan Chen. The Entity-Relationship Model—Toward a Unified View of Data. *ACM Transactions on Database Systems*, 1(1):9–36, 1976.

[36] Douglas C. Schmidt. Model-Driven Engineering. *IEEE Computer*, 39(2):41–47, 2006.

[37] Dave Steinberg, Frank Budinsky, Marcelo Paternostro, and Ed Merks. *EMF: Eclipse Modeling Framework*. Addison-Wesley Professional, 2008.

[38] Adrian Sureshkumar, Marina De Vos, Martin Brain, and John Fitch. APE: An AnsProlog* Environment. In *Proceedings of the 1st International Workshop on Software Engineering for Answer Set Programming* (*SEA 2007*), volume 281 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2007.

[39] UModel Database Modeling Tool. `http://www.altova.com/umodel/uml-database-diagrams.html`.