RWTH Aachen University
Department of Computer Science III
Aachen
Germany

University of Utrecht
Institute of Information and Computing Sciences
Utrecht
The Netherlands

Computer Science Thesis

Development of a SUpporter for the MApping of GRAph Models for Gras/GXL

Thesis number: INF/SCR-04-66

Els Maes
Student number: 0063983

Aachen, 18 May 2005

DEVELOPMENT OF A SUPPORTER FOR THE MAPPING OF GRAPH MODELS FOR GRAS/GXL

COMPUTER SCIENCE THESIS

PERIOD: 30 AUGUST 2004 - 18 MAY 2005

STUDENT/AUTHOR: ELS MAES

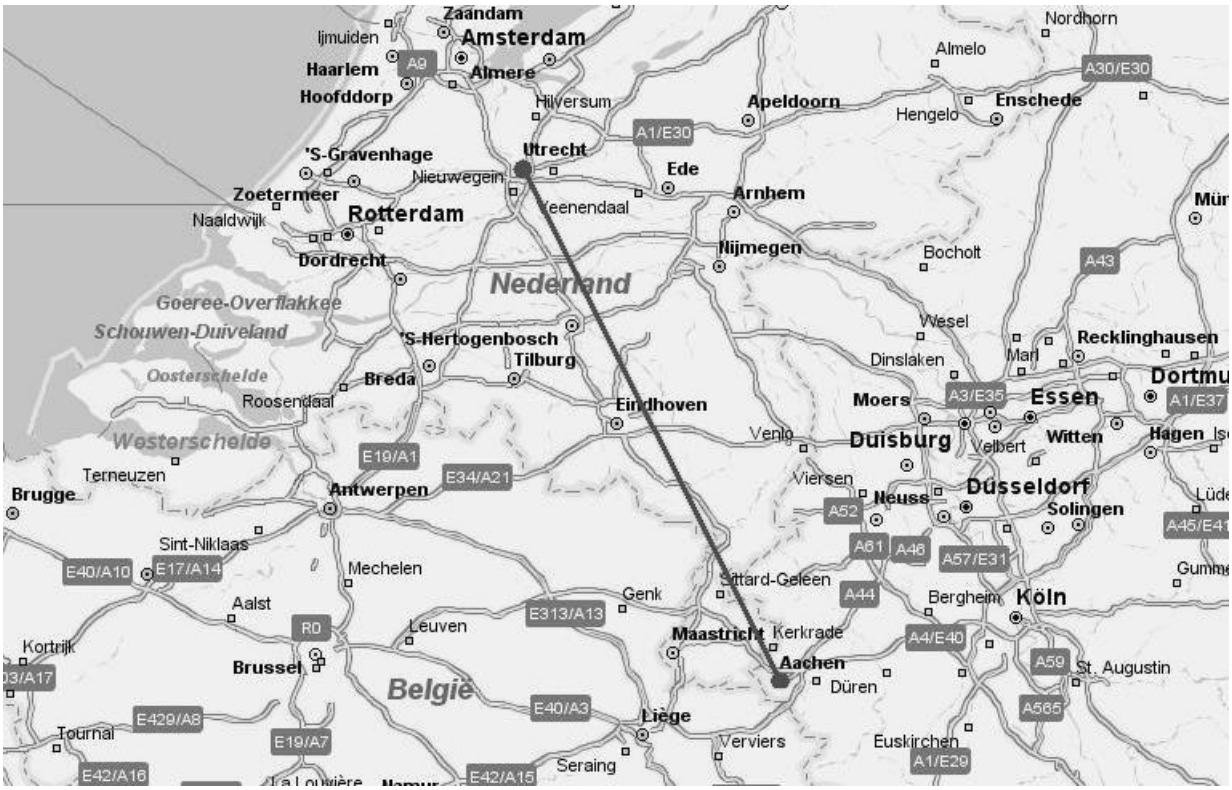
DATE: 18 MAY 2005

EXTERNAL THESIS SUPERVISOR:
DIPL.-INFORM. BORIS BÖHLEN
INTERNAL THESIS SUPERVISOR:
DR. HANS BODLAENDER

RWTH AACHEN, DEPARTMENT OF COMPUTER SCIENCE III
UNIVERSITY OF UTRECHT, INSTITUTE OF INFORMATION AND COMPUTING SCIENCES
EDUCATION OF COMPUTER SCIENCE, ALGORITHMIC SYSTEMS

Preface

In the year 2000, I started my education in Computer Science at the University of Utrecht [Utr] in the Netherlands. As the last task of my education, the fulfilment of a thesis project was planned. In this report, that thesis project will be described. This project was completed at the RWTH Aachen University [RWT] in Germany and accomplished in eight and a half months.



As I am Belgian and studying in the Netherlands, my education was already quite internationally. My residence is not distant of Germany, which developed the idea to accomplish my thesis there. At secondary school, already some German was taught which gave me a reasonable linguistic basis. Nevertheless, in order to communicate properly with my colleagues and supervisor, I seized the opportunity to take a German course with both hands. Erasmus/Socrates, in which frame this thesis was carried out, presented this opportunity. I am very grateful that I could refine my German knowledge and that I was offered the chance to do so.

This diploma thesis is written to finalize my education. It is written for involved teachers, students, and other interested readers. It is assumed that the reader has a mediocre knowledge of Computer Science. I expect that he or she knows what a graph is and also has some knowledge of what a

program looks like. Readers, who plan to use the program, are supposed to have good knowledge of UML and experience on behalf of the methods of Gras/GXL.

At the RWTH Aachen University, diverse projects are fulfilled as dissertation. One of them is the realization of Gras/GXL. Gras/GXL is the frame of this thesis project. It is accomplished at the Software Engineering chair (Department of Computer Science III) of University Professor Dr-Ing. Nagl. Gras/GXL is developed by Boris Böhlen, which is a member of this chair and my external supervisor in this thesis project. The Software Engineering chair does research on the topics Concepts, Languages, Methods, and Tools for Software Engineering. There are, among others, projects in the field of home automation, graph rewriting systems, and process management.

I really enjoyed working on this project. I also have the opinion that I learned a lot during these months in Germany. Therefore, I would like to thank all the people that enabled my education and this thesis. Especially, I would like to express my gratitude to:

- *Prof. Dr.-Ing. Manfred Nagl for the interesting assignment, the possibility to write my thesis at the Department of Computer Science III (Software Engineering) at the RWTH Aachen University and the informative discussion during the thesis presentation,*
- *Prof. Dr. Jan van Leeuwen for giving the possibility to write my thesis in Germany,*
- *Dipl.-Inform. Boris Böhlen for his appreciated, intensive and valuable mentoring, the numerous constructive and informative discussions, and his contribution to my learning of the German language,*
- *Dr. Hans Bodlaender for his highly appreciated mentoring and the informative discussions,*
- *the members of the Department of Computer Science III of the RWTH Aachen University for their support, the creation of a pleasant and comfortable atmosphere, and their patience for me learning the German language,*
- *the members of the Computer Science Institute of the University of Utrecht for giving me the knowledge that was necessary to be able to fulfil this thesis work,*
- *my boyfriend Patrick Luja for supporting me during my complete education, for bearing my tempers in difficult periods and participating in the cheerful moments,*
- *the members of my family for supporting me undoubtedly during my education,*
- *my parents, Gaby and Freddy Maes-Willems, for giving me the possibility to get the education I desired and supporting me unconditionally.*

With this statement, I acknowledge that I fulfilled the work autonomously, I did not use any other than the declared resources and I made the citations recognizable. I will be gladly to give extra information and clarification on behalf of this project to anyone interested.

Els Maes

Aachen, 18 May 2005

Abstract

Different application domains work with a number of complex documents. For simplifying working with these documents, they are stored as graphs and these graphs are stored in databases in a structured way. A database for this purpose is Gras/GXL, a graph oriented database management system.

A graph model defines the structure of the graphs that are stored in Gras/GXL. The corresponding graph schema defines the properties of the graphs like types, attributes, and inheritance relationships. The graph model and graph schema are explicitly general in order to be compatible with a large amount of graph models. The graphs for the specialized application domains need a specialized graph model that defines the specific structure and properties of these graphs. In this specialized graph model, data that is not relevant is left out (for instance entities that cannot be used, like relations), and other data that is not available in the Gras/GXL graph model is added (like a special kind of type). To store the graphs used by the specialized application domain, a mapping between the Gras/GXL graph model and the specialized graph model should be created. Gras/GXL provides a number of methods for manipulating and querying the graphs; for example, methods for creating entities like nodes and edges. When creating an entity for the specialized application domain, a method of the Gras/GXL graph model is used. As this entity is not compatible with the application domain, it has to be converted. To implement an automatic conversion between the Gras/GXL and the specialized graph model, a mapping is created.

The mapping can be created by hand, but as this is a quite tedious and mechanical task, this mapping should be generated by a program, SUMAGRAM. SUMAGRAM is created in this thesis project. SUMAGRAM is a SUpporter for the MApping of GRAph Models. The specialized graph model is modelled using a UML class diagram, which is used as input for SUMAGRAM. The output of SUMAGRAM is the mapping, which is represented by a number of Java files.

SUMAGRAM has to meet several requirements. It first has to be independent of the specialized graph models. It may only contain knowledge of the Gras/GXL graph model. To implement the usability of SUMAGRAM, a methodology for short descriptions has to be invented. These short descriptions, which are called tags, define for every method a possibility to write a method in a few lines, in a structured way. This is not an exhaustive methodology as not every piece of code can be generated with the use of tags. Therefore, SUMAGRAM enables a combination of tags and literal code.

Contents

List of Figures

Chapter 1

Introduction

Many different tools exist for various application domains. Some of these tools work on complex documents. These documents are then internally represented as graphs. This enables the tools to cope with the documents. One of many different documents that can be visualized as a graph is this thesis, which is shown in Figure 1.1. By drawing a graph of a document, it becomes easier to work with it: the structure is clarified and thereby, navigating the document becomes easier.

In Figure 1.1, the imaged graph represents a part of this thesis document. For every chapter, section, subsection, and so on, a representing instance in the graph exists. The structure of the document is displayed through directed edges (in the figure depicted by arrows). It might seem logic that Chapter 2 follows Chapter 1, but there are also a lot of other references: for instance a reference in Section 1.2 addresses to Chapter 2. This relation is illustrated by an edge between these two instances.

In one tool, often a lot of documents are involved, which makes it difficult to grasp the right document of the whole set. To simplify the availability of the documents, it is efficient to store them in a database. In this database, they are apparently stored as graphs. Databases have as an advantage that data can be stored in a stable and structured way, which enables the user to efficiently make use of the data. Another advantage is that a huge amount of data can be stored, in such a way that it is not difficult to recover your data. As there are many documents for the different tools, this is a required and helpful aspect. A database also enables the use of the data by several users at the same time. This data is reliable and consistent because it is updated neatly and structural.

The idea to store documents in databases is not new. In projects such as IPSEN [Nag96] and PROGRES [Sch94], this feature was also requested. The IPSEN (Integrated, Incremental, Interactive Project Support ENvironment) project concerns with integrated and structure-oriented software engineering environments. PROGRES (see Section 3.3.1) is a language and a development environment used for PROgramming with Graph REwriting Systems. These environments use GRAS [KSW95] as a database. GRAS (GRAph Storage) is a graph oriented database system (see Section 2.1), which was developed as part of the IPSEN project. Because GRAS is outdated, a new database management system is developed: Gras/GXL. This system uses the ideas of GRAS, but places everything in a contemporary design.

In the graph database Gras/GXL, graphs are stored. These graphs should meet the structure of the Gras/GXL graph model, which is a very general model. This model applies to different application domains and is hence quite elaborate. Because it is complex, it is preferable that it is simplified

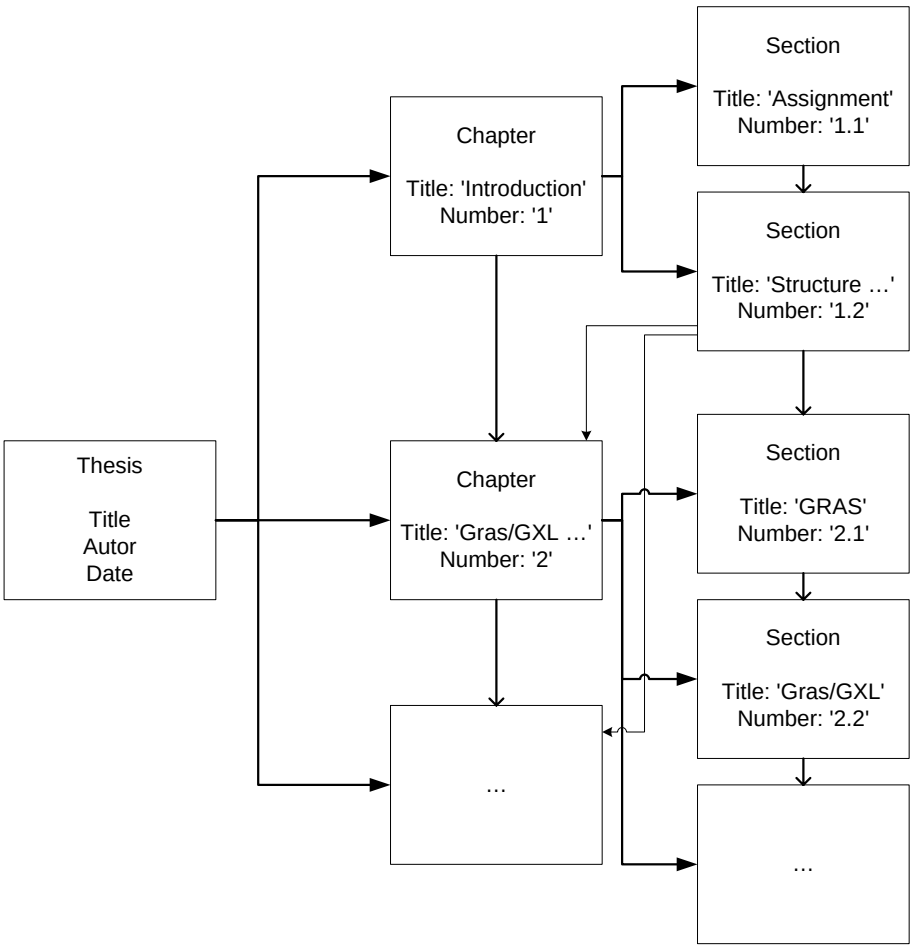


Figure 1.1: Example of a Graph of this Thesis

for each specialized application domain. Therefore, these domains have each a specialized graph model, which is related to the Gras/GXL graph model. This specialized graph model also defines the structure of the graphs related to the respective application domain. To be able to store these specialized graphs, a mapping of their graph model to the Gras/GXL graph model is required.

For this mapping, the code of the specialized graph model is created using the Gras/GXL graph model. The output of methods of the Gras/GXL graph model is converted, such that it is compatible with the specialized graphs. This conversion is called a mapping. This mapping was usually done manually. But, as this is a quite tedious and mechanical task, this mapping should rather be generated. Therefore, this project is called into being. In this project, as a working example, the graph model of PROGRES serves as an input. The mapping as previously described is the output.

The program that is developed is not likely to create a perfect mapping. It is rather a supporter. This led to the program name SUMAGRAM (SUpporter for the Mapping of GRAph Models).

1.1 Assignment

The thesis project concentrates on the creation of a mapping between two graph models. Both a core graph model of the Gras/GXL database management system and a specialized graph model of application domains like PROGRES exist. The graph models are specified using UML diagrams (Unified Modeling Language, [Obj01]). For each class of the specialized graph model, a corresponding class in the core graph model should exist. A mapping should be generated for the specialized graph model. This mapping, which is realized in Java, is the output of the program.

The motivation for this project is the fact that it is really difficult, mechanical, tedious, and error-prone to do the mapping from a universal to a specialized graph model. First, there should be a design of the specialized graph model. Next, code has to be generated, which is strongly dependent on the Gras/GXL graph model. Subsequently, all this code has to be tested.

To create the methods of the mapping manually is not really a problem, but it is remarkable, that when writing the methods for the different specialized graph models, it is every time more or less the same activity. By using the methods of the Gras/GXL graph model, methods of the specialized graph model are implemented. This is not a difficult task, but it is too difficult to perform only a copy-paste action, i.e. copy the code of the Gras/GXL graph model and paste it into the code of the specialized graph model. Automating the task completely or supporting the developer in this task is desirable.

The problem in this project is linked to the Model Driven Architecture (MDA). In the model driven architecture, a platform independent model is transformed to a platform specific model using some extra information. In this project, there is also a platform independent model, namely the Gras/GXL graph model. This model has to be transformed to a platform specific model, the specialized graph model. As, through this reasoning, MDA seemed useful, different tools were considered. But, because every single tool was not sufficient in the desired matters, a new tool is developed.

1.2 Structure of the Thesis

Gras/GXL is the database management system that underlies this project. Gras/GXL follows the concept of GRAS, which is a graph oriented database system. Gras/GXL has a rather complex structure, which is clarified in the second chapter. In the next part, the problem definition and assignment formulation are stated. In addition, the input that is used for creating the final program is mentioned. The input is the PROGRES graph model, which is one of the specialized graph models that are used as an example for working out this project. A view on the related work is given next. Upon the concept of Model Driven Architecture is thrown a light in Chapter 4. The different tools that were considered are evaluated. Chapter 5 declares the solution concept. What was the view on how to solve the problem? Which parts are considered in this view? These questions are answered. The following chapter concerns about the realisation of this project. All the steps are shown in detail. The files that are actually used are exemplified. Also, the encountered problems during the implementation are indicated. In Chapter 7, the example used in the project is exposed, as well as a new example to verify the project results. The different discomforts that were noticed are listed. The last chapter gives a conclusion on all the preceding chapters.

Chapter 2

Gras/GXL, The Graph Database System

This thesis project is embedded in Gras/GXL, which is a graph-oriented database management system. A database management system (DBMS) is needed to simplify the storing of data in the database and the use of data out of the database. A DBMS consists of a number of programs to manage the database. The DBMS has to be able to execute operations on the data that are requested by a user.

The Gras/GXL system provides features for modifications of graphs. Its predecessor GRAS, defines the concepts of Gras/GXL. GRAS has been adapted over the years because of changing requirements and has therefore different versions, going from GRAS over RGRAS to GRAS3.

In the first stage, also some features of GXL (Graph eXchange Language, [HWS00]) were used as a base for Gras/GXL. This clarifies the name of the database system. However, through the multiple transformations during the development of Gras/GXL, the link with GXL has faded. Therefore, GXL will not be digressed upon here. The GXL graph model will, on the other hand, be used as an example in this project and GXL will thus be interpreted in Section 7.2.1.

In the first section the predecessor of Gras/GXL, GRAS, is mentioned. Then Gras/GXL is explained in detail. In Section 2.2 the structure of the database system is clarified. Next, a light is thrown upon the Gras/GXL graph model and the Gras/GXL graph schema. The model and schema define the structure of the graphs that can be stored in the database. For specialized application domains, specialized graph models and schemata are defined. These are described in the last section.

2.1 GRAph Storage

GRAS (GRAph Storage, [KSW95]) is a graph oriented database system, which has evolved over several years and multiple versions, of which RGRAS is the most recent one. The development of GRAS started in 1984. It is developed because data integration was an important issue for integrated, interactive and incrementally working tools and environments. This data is saved in the form of a graph. All the information in the environments should be handled as a consistent whole and modeled and realized in a uniform way. Therefore, a database system was used. This database system is GRAS.

GRAS has been developed within the IPSEN project. The Integrated, Incremental, Interactive Project Support ENvironment (IPSEN, [Nag96]) is developed with the purpose to create an environment that supports as many phases as possible in the software development process. It should also integrate the tools for the different phases. Because GRAS has been used for the implementation of various integrated, structure-oriented environments, its development steadily continued and many versions originated this way.

As the data managed by GRAS is usually very complex, the database system has to be based on a suitable data model, which should provide an easy representation of complex object structures. With the model, objects can be represented as nodes and relations between the objects as edges. Therefore, the decision has been made to use attributed graphs as data model.

One of the aspects of the kernel of GRAS is its ability to efficiently manage medium-sized, complex data of which the size and structure vary dynamically. By this feature, it lends itself to manage, among others, structure-oriented editors, analyzers, and execution tools. On top of the kernel, a number of layers are implemented. These layers serve event handling, change management, etcetera.

PROGRES (see Section 3.3.1) is built using IPSEN. In [KSW95], the relation between GRAS and PROGRES is stated. GRAS supplies the elementary operations on graphs, as there are for instance the creation and deletion of nodes and edges. These operations are checked against the graph schema, which is part of the data definition of PROGRES. Complex graph transformations of PROGRES are mapped onto elementary operations of GRAS. This means that GRAS serves as kernel of a database development environment for PROGRES.

GRAS was first implemented using the programming language Pascal, later Modula-2 was used. In GRAS, only untyped graphs could be stored. The database has to be used by different programs. In the next version, GRAS2, typed graphs could be used. RGRAS (Remotely accessible GRAS) is a successor of GRAS. It has as an extra feature, namely remote accessibility. From here onwards, GRAS refers to the set of all GRAS versions, including RGRAS.

The applications that use GRAS require several functionalities, which originate from the software development environments like IPSEN. More specifically, the manipulation of graphs has to be supported by GRAS. On GRAS, like on other databases, several transactions can be executed. The first requirement is that these transactions are based on the ACID properties [GR92], which are Atomicity, Consistency, Isolation, and Durability. Atomicity is the ability to make sure that either all of the tasks are performed or none of them. Consistency is the claim that a database is consistent when the transaction begins and when it ends. At these moments, no rule may be broken. Isolation means that every operation can be accomplished isolated from all other operations. Other operations cannot see intermediate states from an operation. Durability is the fact that when a transaction has been fulfilled and it was successful, that it will persist and cannot be undone. A second claim to GRAS is that an undo/redo feature has to be available. Because this feature was required for IPSEN, it was decided to integrate it in the database system. In GRAS, the undo/redo functionality is implemented by means of a checkpoint. Between two checkpoints, the adjustments can be undone or redone. As a third feature, also versioning is desired. GRAS provides operations, which can generate a new dependent graph of an existent graph. This new graph is saved in a forward or reverse delta manner. Fourth claim is active behaviour. The application has to be aware in which parts adjustments are made. GRAS contains an event trigger mechanism that informs the application on changes in the database. Another wish concerning the functionalities is the incremental evaluation of attributes. The values of attributes can be dynamically derived of the values of the other attributes. In GRAS, the possibility exists to define a rule for the attributes. As soon as an attribute, which has other

attributes that are derived of it, is changed, the derived attributes should get a status, which makes clear that the value is not up to date. The value will be recalculated as soon as the value is going to be used. The last feature that is claimed is the division of applications. Multiple applications will use the graphs and therefore a distribution of the applications is needed. Applications are divided into groups, which all have a copy of the graph to work on.

As GRAS is not accurate anymore and has some disadvantages, a new database management system is developed. This system is called Gras/GXL. As the requirements upon GRAS are still relevant, they are also applied to Gras/GXL.

2.2 Structure of Gras/GXL

The Gras/GXL system is created at RWTH Aachen University in Germany because the possibilities of existent database management systems, like the different versions of GRAS, were not satisfactory. The predecessors of Gras/GXL can only manage small graphs and do not possess any extensions.

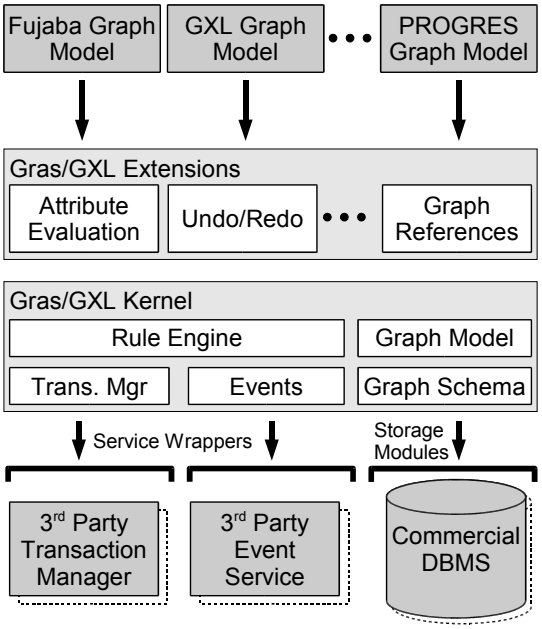


Figure 2.1: Gras/GXL Structure

The Gras/GXL system structure is depicted in Figure 2.1. There are four different parts: the kernel, the extensions, the third-party software, and the specialized graph models. The kernel consists of a graph model, which is elucidated in the next section. The purpose of the graph model is to define the structure of the graphs that are stored in the databases. The graph schema defines the constraints on the graph model. These constraints are on the level of types and classes of the graph elements. The schema is explained in Section 2.4. Beside the graph model and schema, there are events, rules, and transactions. External managers control events and transactions. The graphs are stored in a third-party database after passing through the kernel, during which the structure and properties of the graphs are checked. These databases are arbitrary ones, like PostgreSQL [Mom00] or FastObjects [Ver].

Extensions are built on top of the kernel. These extensions are features that are not necessary or desired for each application. Not all applications use versioning or an undo/redo feature. Therefore, these properties are implemented in Gras/GXL by means of extensions. Possible extensions are attribute evaluation and graph versioning.

Because the graph model is very general, there are specialized graph models, which define the structure of the graphs that ought to be stored in the databases. These specialized graph models are for instance the PROGRES graph model or the GXL graph model.

2.3 Graph Model of Gras/GXL

The Gras/GXL graph model evolved from the GXL graph model. In Figure 2.2, the Gras/GXL graph model is shown as a UML class diagram. In this diagram, the static structure of the graph model is depicted. As the dynamical properties cannot be displayed in a picture, they will be formulated through OCL (Object Constraint Language, [WK99]) constraints.

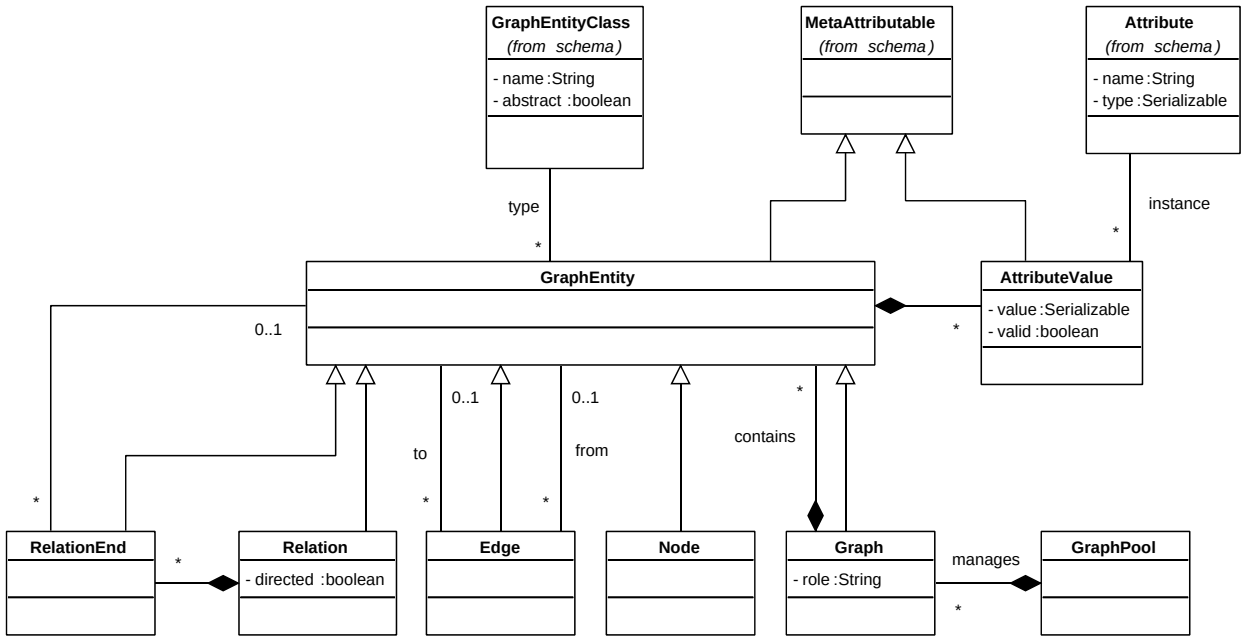


Figure 2.2: Gras/GXL Graph Model

All the graphs that are stored in the database are managed by and stored in a graph pool, which is represented by the class `GraphPool` in the diagram. In the graph pool, an arbitrary amount of graphs can be stored. These graphs are all identified by their role, which should be unambiguous:

context GraphPool
inv: self .manages−>forall(g1,g2 | g1 <> g2 **implies** g1.role <> g2.role)

A graph pool is not indispensable in the graph model; there could also be administered only one graph. The management of the graphs then has to be arranged by the graph model itself, which though has to support several graphs. This has led to the origination of the graph pool.

The graph pool is the manager of the graphs. A graph (`Graph`) is one of the possible graph entities (`GraphEntity`). The graph entities incorporate also edges (`Edge`), nodes (`Node`), relations (`Relation`), and relation ends (`RelationEnd`). A graph consists of an arbitrary number of graph entities. A graph can comprise nodes, edges, relations, and again graphs. Relation ends are also graph entities, but they cannot be directly part of graphs. Instead, a relation end belongs to exactly one relation. This leads to the invariant:

context Graph
inv: self.contains->select(oclIsTypeOf(RelationEnd))->isEmpty()

The reason why relation ends are nevertheless graph entities, is that now, a relation can be defined between relation ends. This can be useful for the definition of ordered edges in a specialized graph model.

Next to n-ary relations, also edges are part of the Gras/GXL graph model. An edge could also be represented by a binary relation. The support of edges in the graph model is however a simplification when it comes to specialized graph models. Otherwise, there had to be defined a binary relation for every edge, and also the relation class. Relations and edges can be defined as ordered or unordered. A relation is only unordered when none of the relation ends is ordered. In all other cases, it is ordered. Consequently, for every relation applies:

context Relation
inv: self.relatesTo->exists(re | re.type.direction <> none)
implies self.directed = true
inv: self.relatesTo->forAll(re | re.type.direction = none)
implies self.directed = false

On the level of the graph entity class (`GraphEntity`) the properties of all elements of a graph are determined. In Gras/GXL, every graph entity may be attributed. This is the responsibility of the attribute value class (`AttributeValue`). Gras/GXL supports every serializable Java class as an attribute value. So not only strings and numbers are supported, but also various collections. The support of serializable classes has the effect that graph entities can be stored in an attribute, also as part of a collection. For every attribute can be identified if the value is valid or not. This enables the implementation of simple extensions, like calculated attributes.

For specialized graph models, the meta-attributable class (`MetaAttributable`) is available. With this class, extensions of Gras/GXL can save extra information. In this class, every serializable Java class is allowed as an attribute value.

Every graph entity is attributed. The Gras/GXL graph model requires that every graph entity has an unambiguous type, which is a subclass of the graph entity class. The graph entity class (`GraphEntityClass`) is specified in the graph schema (see Section 2.4). Although the Gras/GXL graph model does not support untyped graphs, a specialized graph model can be specified that does support untyped graphs.

In Gras/GXL, graphs are only a special kind of graph entity. This implies that there can be relations and edges for connecting or overlapping graphs. In addition, no restrictions exist in Gras/GXL on the level of the source and target of a graph. This means that they do not have to be comprised in the

same graph pool. Even though this is currently not supported by Gras/GXL, it can be implemented through an extension.

Ordered edges and relations are not supported in Gras/GXL. This might be a problem, but most of the time, the ordering of edges or relations can be defined using edges or meta-attributes. By using these attributes, it is nevertheless possible to define ordered edges or relations in a specialized graph model. This feature can also be implemented with an extension.

2.4 Graph Schema of Gras/GXL

Linked to the Gras/GXL graph model, also a graph schema exists. The graph schema is depicted in Figure 2.3. The graph model defines the fundamental structure, in contrary to the graph schema, which defines the different kinds of graph elements and their restrictions.

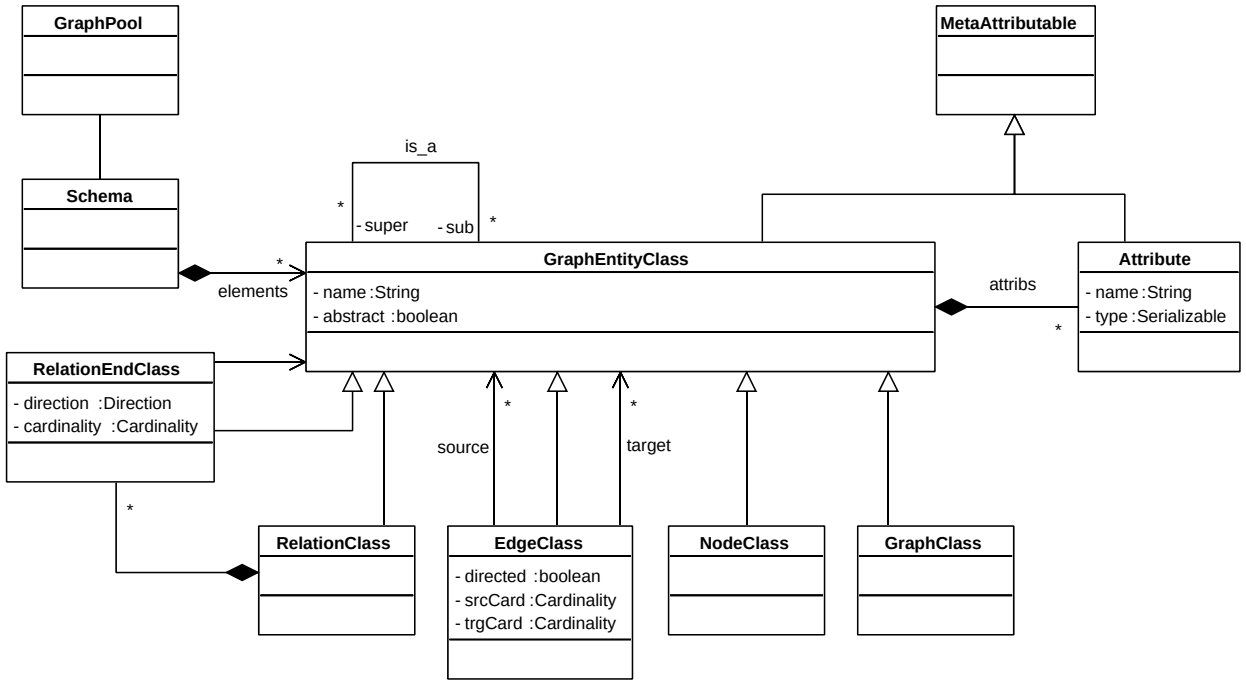


Figure 2.3: Gras/GXL Graph Schema

Identically as for the graph model, the graph schema is also applicable for a wide range of applications and is therefore quite extended. The specialized graph models have also a graph schema that is related to the Gras/GXL graph schema. To the specialized graph schema, no restrictions may be defined. Therefore, the core graph schema provides properties that are common for the different schemata. These properties are for instance cardinality or the definition of attributes.

Every graph pool contains exactly one graph schema that specifies all the kinds of graph entities. If there would be a graph schema for every graph, this would be inconvenient, because relations between graph entities can be stored in different graphs. If this was the case, there should be a feature for the import and export of kinds.

The properties of all kinds of graph entities are defined in the graph entity class (`GraphEntityClass`). For the different kinds of graph entities, separate classes are defined. For instance for nodes, the node class is defined. Every graph entity class has a unique name:

```
context Schema
inv: self.elements->forAll(c1,c2 | c1 <> c2 implies c1.name <> c2.name)
```

In Gras/GXL, also abstract graph entity classes can be used. In contrary to concrete graph classes, there cannot be created instances of these abstract classes in a graph. Therefore, the following rule applies:

```
context Graph
inv: self.contains->forAll(ge | ge.type.abstract = false)
```

The Gras/GXL graph schema supports the inheritance of graph entity classes. Therefore, every graph entity class administers an arbitrary amount of sub- and superclasses. As superclasses are supported, also multiple inheritance is realizable. For the inheritance relationship, two restrictions have to be considered. Cyclic inheritance is not allowed and an abstract graph entity class can only be inherited of another abstract graph entity class:

```
context GraphEntityClass
def: allParents:Set(GraphEntityClass) =
    self.super->union(self.super->collect(p | p.allParents()))
inv: not self.allParents()->includes(self)
inv: self.isAbstract implies (self.super->isEmpty() or
    self.allParents()->forAll(c | c.isAbstract = true))
```

For every graph entity class, an arbitrary number of attributes can be defined. Every attribute is identified by a name that has to be unique for each graph entity class. In addition, the attributes inherited from superclasses have to conform to this rule. Every attribute also has a type that defines which values are valid. Gras/GXL requires that this type implements the `Serializable` interface of Java. Otherwise, these attributes cannot be stored in a database. For the attributes, the following restrictions are defined:

```
context GraphEntityClass
def: inheritedAttributes:Bag(Attribute) =
    self.attribs->union(self.allParents()->collect(g | g.attribs))
inv: self.inheritedAttributes()->forall(a1,a2 | a1 <> a2 implies a1.name <> a2.name)
```

Beside these properties of the graph entity classes, every class also defines some specific properties. The different graph entity classes are `NodeClass`, `EdgeClass`, `GraphClass`, `RelationClass`, and `RelationEndClass`.

The simplest graph entity classes are those for nodes and graphs. These have no other properties than those generally defined for the graph entity classes. It might have been possible, for graph classes, to define instances of which graph entity classes may be created. However, this restriction would make the realization of specialized graph models harder. This restriction can also be defined within a specialized graph model.

On the level of edge classes, it can be defined, which classes are allowed for sources and targets. The inheritance is also respected: subclasses of an allowed class are also allowed. If this behaviour is not appreciated, this can be changed within the program that supports the graph model. Also, the number of incoming and outgoing edges of an edge class can be restricted. This can be done using the `cardinality` attribute. The cardinality of both the source node and the target node can be specified. The Gras/GXL graph schema supports both ordered and unordered edges. This feature is realized using the attribute `directed`. Because Gras/GXL cannot define any restrictions for the specialized graph models, these features can be redefined by the inheritance.

Relations can be connected to any number of graph entities. These graph entities are at that moment linked to relation ends. Therefore, both the relation class and the relation end class have to be defined. For relation end classes, it is defined which graph entity classes are allowed as target, which cardinality the relation ends ought to have, and whether the relation ends are ordered or not. For every relation end class, a name is specified. The relation end classes are linked to the relation classes. This implies the possible relation ends for each relation.

2.5 Specialized Graph Models

Based on the Gras/GXL graph model, specialized graph models [Böh04] are specified. These models are defined in the top layer of Gras/GXL. One of these specialized graph models is the PROGRES graph model, which was used as the starting example for this project. In Section 3.3.2, the details on the PROGRES graph model are specified. Other examples of specialized graph models are the GXL graph model (see Section 7.2) and the Fujaba graph model [FNTZ00].

The Gras/GXL graph model is quite complex and contains very much information that is not useful for all applications. Therefore, these applications provide their own specialized graph model, which discards the redundant information of the general graph model. Another reason for the developers to supply their own graph models is that the Gras/GXL graph model does not consider the extensions that are provided by the Gras/GXL structure. In these extensions, concepts like versioning are included. If an application uses this feature, it would also be useful when this would be embedded in the graph model. As this is impossible for the Gras/GXL graph model, the applications should provide their own specialized graph model in which the extensions are included.

In the specialized graph models, the information that is not needed by the application is ignored and the features and properties that are not included in the Gras/GXL graph model are added. This means that the specialized graph models only define the classes they actually need.

Chapter 3

Problem Definition

In this thesis project, an application should be developed that supports the realization of the mapping between a specialized graph model and the Gras/GXL graph model. The input of the application is the UML class diagram of the specialized graph model. The expected output is a set of Java classes, which realize the mapping. The program can use several techniques and can make use of the data and methods of the Gras/GXL graph model.

In the first section of this chapter, the problem is described in detail. Section 3.2 brightens the concept of the assignment. The last section of this chapter gives a view on PROGRES and its graph model, which is used as an example in this project. A few methods of the PROGRES graph model are defined to give some more insight into the problem.

3.1 The Problem

Many applications use complex documents to work on. These documents can be saved as graphs, to simplify working with and on them. The graphs can be stored with Gras/GXL, the graph oriented database system described in the previous chapter. Gras/GXL possesses a general graph model that is quite rich in comparison with other graph models. It provides a lot of general information, which might appear in a wide range of graph models. The disadvantage of its commonness is that for specialized application domains, the Gras/GXL graph model does not provide the appropriate information. On one side, there is redundant information and on the other side, the information is too limited. In the general graph model, a number of classes are not useful for the specialized graph model. In addition, classes that appear in the specialized graph model do not appear, or appear with fewer properties in the core graph model. Therefore, these applications require a specialized graph model, which contains only the information useful for the specialized application domain.

Graphs of specialized application domains are built out of specific graph entities. For instance, the PROGRES graph model consists of the type `ProgresEdge` instead of the type `Edge` that is part of the Gras/GXL graph model. These types differ from each other, in the way that entities of type `ProgresEdge` cannot be attributed, and entities of type `Edge` can be. Gras/GXL offers elementary operations on graphs: the creation and deletion of nodes, the retrieval of all edges that are related to a certain node, etcetera. These operations return (collections of) Gras/GXL graph elements. As these graph elements are specific for Gras/GXL, and because of the difference between types of the general

graph model and types of the specialized graph models, the specialized graph model cannot work with these general entities. To enable the specialized graph model to work with these entities, it is necessary that these entities are converted to entities of the type provided by the specialized graph model. Such a conversion is called a mapping.

In former times, this mapping was done manually. This task is quite mechanical and straightforward. The methods of Gras/GXL are used, and their output is converted. For example, to retrieve all the nodes of a PROGRES graph, first the method of Gras/GXL, which collects all the nodes of a certain graph, is called. The output of this method is a collection of nodes of Gras/GXL. Because PROGRES is not able to work with these nodes, they have to be converted to nodes of PROGRES. Thus, in the described procedure, first a method of Gras/GXL is called, and then the output is converted to entities of PROGRES. See Section 3.3.3 for a more extensive description of this problem. Unfortunately, it is too difficult to just perform a copy-paste action, because the conversion step also belongs to the procedure and has to be added to every procedure. This conversion step might also be different for different methods. Another disadvantage of creating the mapping manually, is that it is tedious and time-consuming. To create all methods of the specialized graph model by hand, means a lot of work. Every single procedure has to be created by hand, and the conversion step has to be added for every procedure. The last problem that is noticed, is that errors easily appear. When doing such an ample job, it often cannot be helped that errors are made.

Because it is such a mechanical task, it was appropriate to generate the code of the mapping. By automating this task, many disadvantages would be annihilated. Assuming that the graph model is already extant, the user will not have to write completely in depth detailed code, but will be able to describe the program using a short description (see Chapter 6). The basic methods, like converting an entity from Gras/GXL to a specialized graph model, will be definable using such a short description. From the specialized graph models, unit tests can be deduced. With these unit tests, the correctness of the code can be checked. By using unit tests, errors can be found prematurely. Beside the error aspect, it will be also more time efficient to use the program instead of generating the mapping manually.

3.2 The Assignment

The target of the assignment is to create an application that can output the mapping between Gras/GXL and a specialized graph model. It is desirable that the whole process is automated. The program should output every method that is part of a graph model. It is though also considered that it will be impossible to generate every piece of code. Methods that will only be used sporadic and are quite specific will not be generable. For such methods, some other solution has to be found. The delivered input of the program is the UML diagram of the specialized graph model.

Restrictions on the graph model should be specified using OCL (Object Constraint Language). A restriction is, for example, that for the creation of an edge, the source and target nodes should be of a certain specified type. It is desired that these OCL constraints are interpreted and used in the program, and thus have their effect on the output. The advantage is that if the constraints can be included in the graph model and are hence translated by the program, it is not necessary to interpret and check them manually.

The output of the program should be a number of Java files. Because the Gras/GXL methods are defined in Java, it is obvious to create the output also in Java. For each class in the UML diagram of

the specialized graph model, a file should be created. Because of an object-oriented viewpoint, every class is interpreted as an object and gets hence a file including all the methods that are specified for the respective class in the UML diagram.

In the UML diagram, only small amounts of code should be written, in the form of short descriptions. It is also possible to write full code literally in the UML diagram, but this would not solve the problem: the user still has to write the code manually, and that was just the problem to be solved. There has though to be considered that this could not be sufficient. Therefore, it is desired that additional code can be added manually. This code cannot be extracted from the graph models. It is thus necessary that a feature is designed, in which pieces of code can be written that are literally copied into the mapping.

This program has to be tested using the GXL graph model and the PROGRES graph model. As a starting example, the PROGRES graph model is used.

3.3 The PROGRES Graph Model

The PROGRES graph model is the specialized graph model that will be used as an example in order to create the program. As it is only an example, there has to be regarded that the SUMAGRAM will not be in any kind dependent on this specialized graph model.

3.3.1 PROGRES

PROGRES [Sch94] is a language used for PROgramming with Graph REwriting Systems. A graph rewriting system [BR04] is a set of rules that transforms one instance of a given class of graphs into another instance of the same class of graphs. Such a system is often used as a visual and executable specification of abstract data types or as graph manipulating tool. PROGRES is built on the thought that a class of problems exists, which can be modeled with the help of graphs and can be solved with graph rewriting rules. It is a very high-level language, because it is able to specify a problem on a higher abstraction level. With the graph rewriting rules, this problem can be solved. PROGRES is as well a language as a development environment. The graph rewriting systems are the language's underlying formalism. The language PROGRES has a context-free syntax, type checking rules and semantics. It is a multi-paradigm language for visually specifying graph schemata, graph queries, and atomic graph rewriting steps.

3.3.2 The Graph Model

The PROGRES graph model is displayed in Figure 3.1. Usually a corresponding class in the Gras/GXL graph model can be found. This is though not necessarily a one-to-one relationship. For instance for the class `NodeClass` of Gras/GXL, there are two classes, `ProgresNodeType` and `ProgresNodeClass`, in PROGRES.

The graph model defines exactly one graph pool (`ProgresGraphPool`). This graph pool manages an arbitrary number of graphs. A graph is denoted as `ProgresGraph`. Every graph has, as in the core graph model, a unique role. A graph consists of nodes (`ProgresNode`) where each node has a

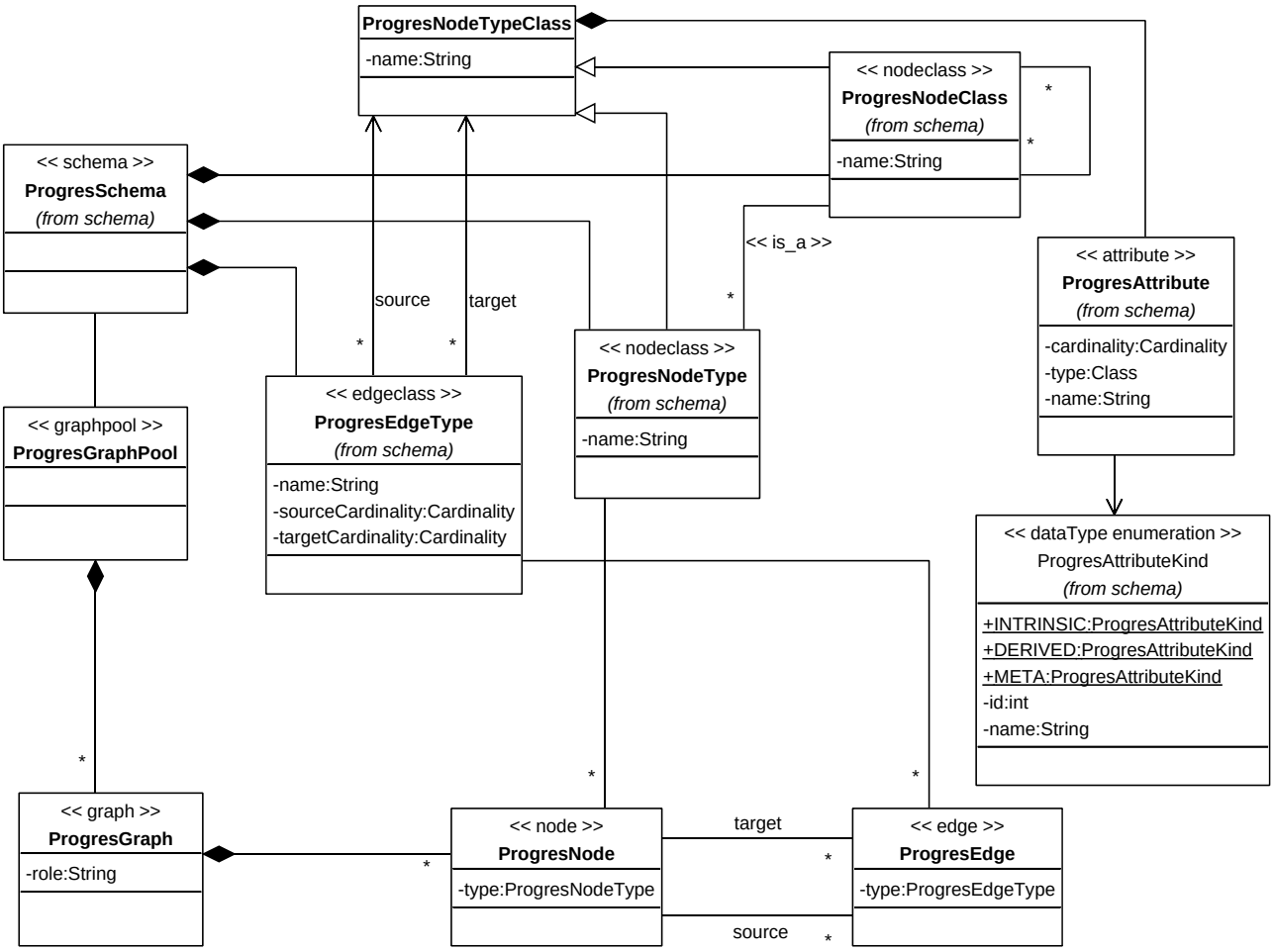


Figure 3.1: UML Class Diagram of PROGRES Graph Model

type (`ProgresNodeType`). Between two nodes, an edge (`ProgresEdge`) can be created. Every node can be related to one or more edges. An edge exists between a source node and a target node. Edges cannot exist independently; this means that they always have to be related to two nodes. Edges do not have an identity, which means they cannot be returned in the scope of a method. For every edge, an edge type (`ProgresEdgeType`) is defined. In PROGRES, no relations are defined, and edges only exist between nodes. There can though be no edges between graphs or no relations between edges.

By the Gras/GXL graph model, it is defined that for every `GraphEntity`, a `GraphEntityClass` has to exist. This condition is satisfied for the `ProgresNode` and `ProgresEdge`, but not for the `ProgresGraph`. As the concept of a `GraphClass` is unknown for PROGRES, a method exists in the `ProgresSchema` that extracts a standard dummy graph class. By this extra method, the condition of existence of a graph class is fulfilled.

There is also one schema (`ProgresSchema`) defined in the graph model. For every specialized graph model, the schema class is obligatory. In the Gras/GXL graph schema, it is stated that a schema should always exist. In addition, a graph pool is an obligatory class in the specialized graph model. Because of the one-to-one relation between the schema and the graph pool, it is required that when a graph pool exists, a schema also has to be available and vice versa. If a graph pool does not exist, there cannot exist graphs; this implies that a schema always has to exist.

The schema of the PROGRES graph model takes care of the different classes and types of the entities. The schema defines node types (`ProgresNodeType`) and node classes (`ProgresNodeClass`). For PROGRES, the nodes have a two-tiered type: both a node class and a node type exist. The node class is abstract and the node type is concrete. When defining a node type, first a node class has to be defined. Every ProgresNodeType is inherited from a ProgresNodeClass. The node type and node class are both subclasses of the type `NodeClass` of Gras/GXL. To support these two entities and their behaviour, a `ProgresNodeTypeClass` is defined. This class is also responsible for other classes that are related to both the node type and the node class.

On the level of the edges, a `ProgresEdgeType` is defined. This edge type is dependent on the `EdgeClass` of Gras/GXL. The edge type defines the cardinality of the source and target node of an edge. It also defines the node type and node class for the source and target nodes. This definition is enabled through the `ProgresNodeTypeClass`.

The attribute class (`ProgresAttribute`) defines attributes like name and type. The attributes can have three different kinds: meta, derived or intrinsic. The data type `ProgresAttributeKind` defines these types.

In contrary to the Gras/GXL graph model, the PROGRES graph model contains no relations and no relation ends. Also, the graph entity class is not present in the PROGRES graph model.

The PROGRES graph model will be used, as an example, as input for SUMAGRAM. The graph model will be defined using UML. As output, a mapping from the PROGRES graph model to the Gras/GXL graph model is expected. In the UML diagram, all the classes, attributes and associations are defined. For every class in the diagram, a file has to be generated by the program. In this file, which is a Java class file, the attributes have to be present and the associations in the class diagram have to be interpreted.

3.3.3 Methods of the PROGRES Graph Model

The methods for the PROGRES graph model can be divided into a few different groups. First, there is the creation of entities (nodes, edges, etcetera). Next, there is the declaration of schema entities (node type, node class, edge type). Another group of methods is the retrieval of (collections of) entities under certain conditions, for example the retrieval of all nodes of a graph with a specific node type or the retrieval of the source node of an edge. The retrieval of (a collection of) schema entities is another type of methods. A method that is imaginable in this group, is the retrieval of all node classes from the schema. As an example, the method to retrieve all nodes of a graph is given next:

```
public Collection getAllNodesOfGraph() throws GrasGXLException {
    Collection <? extends Node> collection = graph.getAllNodesOfGraph();
    Collection output = new HashSet();
    for (Node x : collection ){
        output.add(ProgresSchemaFactory.getProgresNode(x));
    }
    return output;
}
```

The line `Collection<? extends Node> collection = graph.getAllNodesOfGraph();` defines the retrieval of all the nodes of a graph, using a method of Gras/GXL. This method outputs

the nodes as type of `Node`. But PROGRES requires a `ProgresNode`. Hence, every entity in the collection of nodes has to be converted. As for every `ProgresNode`, exactly one corresponding `Node` exists; it is possible to store all the `ProgresNodes` in a factory, where it can be looked up afterwards.

```
public ProgresNode createNode(ProgresNodeType type) throws GrasGXLException,
    EntityNotFoundException, SchemaCheckException {
      NodeClass nodeclasstype = type.getBackingNodeClass();
      Node node = graph.createNode( nodeclasstype );
      ProgresNode progresnode = new ProgresNode(node,type,this );
      ProgresSchemaFactory.addProgresNode(progresnode);
      return progresnode;
}
```

For creating a node, a method of Gras/GXL is called. To be able to do this, the method should grab an instance of Gras/GXL on which the method can be called. Therefore, the method `getBackingNodeClass` exists. This method returns the `NodeClass` on which the `ProgresNodeType` is based. After creating the node, it is converted and then stored in the factory. Because when creating a `ProgresGraph`, all the related entities are also created, it can be assumed that, when retrieving a collection of nodes, these nodes are all extant. It will hence not be necessary to convert the nodes by creating a new instance of a `ProgresNode`. Therefore, the factory is called into being. Such a factory stores all the created entities, classes, and types. This will make it easier to retrieve a specific entity related to a certain general entity. A question that might rise on behalf of the factory, is what its relation to and the difference of the database is, in which the graphs are stored. The factory is only created as an intelligent solution in the matter of reducing the creation of entities. When using a factory, the retrieval of existing entities is easier. The factory can also be seen as a cache, where the data is stored temporarily.

Now a short description of PROGRES has been given. This description should be adequate in order to get a good insight into the problem. As this description is not exhaustive, in Section 7.1 the graph model will be revisited to give a complete view on how SUMAGRAM works.

Chapter 4

Related Work

As is the case for most applications, also for SUMAGRAM some related work already exists. Exporting code out of a UML diagram is a concept that is supported by the Model Driven Architecture (MDA). For MDA, already different tools exist. These tools might be useful when solving the problem described in the previous chapter. Therefore, an evaluation of the tools is done.

In the first section, the concept of the MDA is described. What is the reason for developing such a concept? Which building blocks are part of the MDA? In Section 4.2, the different considered tools are given and the motivation for the decision on which tool to use. The next section describes UMT-QVT, which is selected out of the set of tools as a useful tool. Finally, the relationship between the MDA and SUMAGRAM is discussed.

4.1 Model Driven Architecture

The Model Driven Architecture (MDA, [KWB03]) was established by the Object Management Group (OMG, [OMG]) in the year 2000. MDA is created in the line of software development, with the target to solve some problems encountered during the software development process. The first problem is that the realization part of the software development is a lot of work. Lots of code are to be produced in a standard software development process. The second problem is, that with every technological evolvement, the code and models should be remade, which is a time-consuming task. Another difficulty exists in the fact that for systems, there are multiple variants of technology and communication with which the systems need to be able to work. The last complication exists in the requirements: when the requirements of the product change, the whole product should be reviewed. To cope with these problems, the MDA has been developed.

What does "Model Driven Architecture" mean? MDA is applied on models. The definition of a model is formulated by [KWB03] as: "A model is a description of (part of) a system written in a well-defined language. A well-defined language is a language with well-defined form (syntax), and meaning (semantics), which is suitable for automated interpretation by a computer." A model is mostly defined using text and image, where the text can be written in a modeling or natural language. With these models, more understanding is provided and the design and deployment are simplified. By the architecture, the parts and connectors of a system and rules for interactions between systems are specified.

MDA supports the idea to use models for software development. The goals of the MDA are interoperability, portability, and reusability. MDA is language-, vendor-, and middleware-neutral. This means that MDA does not require a specific language, vendor nor middleware. It is entirely independent.

MDA is composed of three necessary parts. First a Platform Independent Model (PIM) is needed. This is a model that describes the essential features, without referring to a specific platform. The PIM is usually built in UML. The second part is the Platform Specific Model (PSM). This is created out of the PIM, and depends on a specific platform. The last part is the code. With the PSM the code is generated. This concept is shown in Figure 4.1.

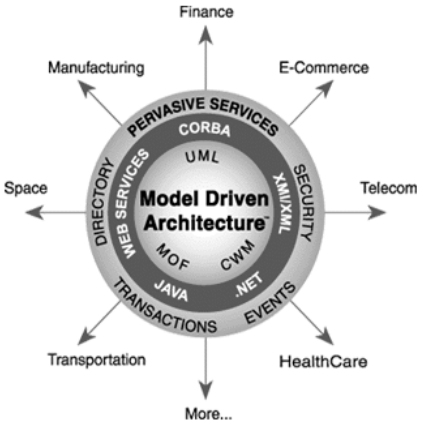


Figure 4.1: The Process of MDA

The first part is the PIM. The target of a PIM is to establish independency of any platform, such that it can be used with several different platforms. It is also stable, because it is technology independent. Out of the PIM, the PSM is created. The PSM combines the PIM and details on a specific platform. To create a PSM out of a PIM a transformation is done by a program. This program might use additional information. Figure 4.2 represents this transformation. The PSM on its turn is converted to a working implementation on a (middleware) platform. Examples of middleware platforms are Web Services, XML/SOAP, EJB, C#/.Net, and Corba.

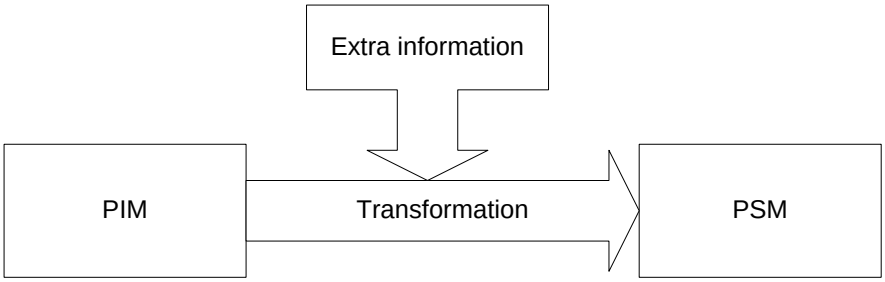


Figure 4.2: Model Transformation from PIM to PSM

The products of the OMG are the building blocks of the MDA: Unified Modeling Language (UML), MetaObjectFacility (MOF), and XML Metadata Interchange (XMI).

4.1.1 Unified Modeling Language

Unified Modeling Language (UML, [Obj01]) is a language specified by the OMG. It is used to specify, visualize, and construct software systems. When developing software, one should think like developing a building. Without a building plan designed by an architect, the building is likely to collapse in time. It is the same for a software project: when there is no plan, it is likely that it will not meet the requirements. A software plan, in technical terms it is called a model, should define a structure such that scalability, robustness, security, and extendibility are guaranteed. The creation of a model has multiple advantages. When modeling an application before the implementation, it can be evaluated if the requirements can be met before starting with the implementation, which is quite an expensive task. Modeling is also an assurance that the functionality is complete and correct. Another advantage is that, after many years when the original designers have already disappeared, maintenance programmers can still use the model to do adjustments and fix bugs. A structured model is also a justified way to deal with the complexity of a large application. When building a rabbit-hutch, a plan is not absolutely needed, but building a skyscraper without plan is pretty hopeless.

UML is not only designed for modeling software systems, also business models or other systems can be modeled using UML. When specifying a software system, different steps can be defined through different diagrams, like class diagrams, use case diagrams, and sequence diagrams. With UML, it is possible to model any type of application for running on any type of hardware and operating system combined with any programming language.

Within the MDA, UML is used to specify the different models. It is nevertheless no strict requirement that UML is used, another modeling language can also be used, although most MDA tools are based on UML. Because a UML model might both be independent of a platform or specific for a platform, it is suitable for use within the MDA.

In the case of SUMAGRAM, the models are specified by means of class diagrams. A class diagram is a structural diagram, where every entity is represented by a class. For every class the attributes and operations can be defined. This can be done quite detailed, by defining the initial value of attributes and specifying the code belonging to the operations. For these classes, relations can be defined by means of associations. These associations can have different types. The composition association defines that a class can be composed of another class. For instance, a graph is composed of nodes; this means that the graph is responsible for the nodes: for the creation and deletion of them. By a composition, one could also speak of a whole and a part. The whole is the graph, and a node is the part. The inheritance relation defines that a class is inherited from another class and hence also inherits the methods and attributes. For instance, the `ProgresNodeType` is inherited from the `ProgresNodeTypeClass`. In addition, relations that are neither an aggregation nor a composition, are defined between classes. There are more types of associations, but the ones mentioned are the most important ones in case of the SUMAGRAM. For all associations, cardinalities and names can be defined. Such a cardinality defines for instance that a graph can exist of zero or more nodes.

Within UML, also constraints can be defined by using OCL (Object Constraint Language, [WK99]). It is a language to formally define conditions. With OCL invariants, preconditions, postconditions, etcetera can be defined. In Section 2.3 a number of OCL constraints were defined.

UML is built upon MOF, which is also defined by the OMG. MOF also plays an import role in MDA.

4.1.2 Meta Object Facility

Meta Object Facility (MOF, [Obj04]) is a standard defined by the OMG. With MOF, the structure of a metamodel can be determined. The UML metamodel is a model following the MOF standard. MOF can be seen as an abstract language, for specifying object-oriented metamodels. In metamodels, language and constructs to build models are defined. For instance, XML schema is a metamodel for XML documents and UML is a metamodel for an application.

As MOF works with metadata, it is related to XMI, the XML Metadata Interchange. XMI is a parallel mapping between MOF metamodels and XML DTDs (Document Type Definition, [BPSM98]), and MOF metadata and XML documents.

4.1.3 XML Metadata Interchange

XML Metadata Interchange (XMI, [Obj02]) is a standard for interchanging information between tools, applications, and repositories. XMI was first established in the late 90s. It is mainly used to exchange information concerning UML diagrams, but it has many more features. An XMI file of a UML diagram contains about all the enclosed information. Every class, association, attribute, etcetera in the UML diagram is represented in the XMI file. Most of the UML tools nowadays have a feature to export the UML diagram to XMI. As the information in the XMI file is represented in XML [BPSM98], it can also be handled as such. All the actions and transformations that can be performed on an XML file can also be performed on an XMI file.

XMI integrates three standards: XML, UML, and MOF. XMI allows the sharing of object models and other metadata. What is actually specified on behalf of a UML class diagram, is described in Section 5.2.

4.2 Choice of Tool

The desired tool should create Java source code out of a UML diagram. At first, the UML diagram is converted to an XMI file. In the second step, this XMI input should be converted to Java source code. On [Mod], a number of MDA tools are defined. These tools use an XMI file as input and generate some output. The tools are evaluated on behalf of their usability in combination with SUMAGRAM. The evaluated tools are listed in the table below.

For the choice of the MDA tool, there were different options. A choice was made from free tools that were available on the Internet. Those were partly evaluated on the base of their age, while the

technique of MDA is quite young. Some of the tools were too young and underdeveloped to use or the website was very limited and inaccurate so that those tools were not further considered.

Name	Similarity to the problem	Website
UMT-QVT Version 0.81 June 2004	Transformation and code generation of UML/XMI models Tool environment language = Java	http://umt-qvt.sourceforge.net/ Open source
MTL-engine Version Beta 2 Dec. 2003		http://modelware.inria.fr/ Freeware
ATL Version 0.2	Language is ATL Only a language, no software?	http://www.sciences.univ-nantes.fr /lina/atl/
Modfact Version 1.0.1 July 2004	Programming language = Java	http://forge.objectweb.org/projects /modfact/ Open source
GMT	Code generation tool Not finished yet?	http://www.eclipse.org/gmt/
OpenMDX Version 1.3.6 August 2004	Supports J2SE, J2EE, and .NET UML modeling tools supported	http://www.openmdx.org/ Open source
AndroMDA Version 2.1.2 Dec. 2003	Takes UML Generates J2EE or other components	http://www.andromda.org/ Open source
Middlegen Version 2.0 January 2004	Generates EJB, Hibernate, JDO and JSP/Struts source code	http://boss.bekk.no/boss/middlegen/ Freeware
Omelet Version 0.0.2 May 2004	General framework for plugging in and integrating models, metamodels and transformations	http://www.eclipse.org/omelet/

After the first selection, three tools remained: OpenMDX, AndroMDA, and UMT-QVT. These are described in the next section.

4.2.1 Which Tool to Use: OpenMDX, AndroMDA, or UMT-QVT?

Three tools were selected out of the set. Among the three tools OpenMDX [Ope04], AndroMDA [And], and UMT-QVT [UMT], further selection had to be done. In Figure 4.3, the tools are listed, with their specific evaluation.

The first selection step for these three tools was to evaluate if these tools were useful for the purpose of the work. Can the input be XMI and is the output Java? For all the tools, XMI or even UML could be used as input. So this was no criterion to lift one tool out of the set. As output, all the tools support J2EE/EJB, but what is requested is the possibility to output simple Java code. AndroMDA could not meet this requirement. UMT-QVT was expected to be able to output Java on the base of the information on the website. OpenMDX is said to support J2SE on which ground could be decided that it was likely to output Java code. During the evaluation of all the tools together, the age (version) was already considered. Here again it is used in the detailed evaluation.

	Input XMI	Output Java (not J2EE/EJB)	Age / Version	Usability / Installability	Support
OpenMDX	+	+	+-	-	-
AndroMDA	+	-	+	+-	?
UMT-QVT	+	+	-	+	+

Figure 4.3: Evaluation Table

Now a more in-depth evaluation is carried out. Can the tool also be easily installed and used? This question might seem superfluous, nevertheless, afterwards it was one of the most important criteria. UMT-QVT was easy to install and quite intuitive to use. For AndroMDA, the installation was fulfilled by means of Ant [Ant] or Maven [Mav]. This is not the easiest way, but with some experience, this installation is no problem. Also the usability of AndroMDA is not bad, although there is no graphical user interface, which makes it a little uncomfortable. For OpenMDX, also an Ant installation was required. The problem with this installation was that the used XML build files were faulty and needed to be adjusted. Also the use of OpenMDX was anything but intuitive. Before the program could be used, some more source code had to be added. How to use the program exactly, is still not clear, which impacted the evaluation of OpenMDX in a negative way. The last criterion is the ability of the developers to deliver sufficient support. As for AndroMDA, no support was needed, thus this cannot be valued. For OpenMDX, the support was valued negatively, as the developers were not able to explain the exact use of the program. For UMT-QVT, the support was satisfiable. Support was given quickly and also the quality was accurate.

On the base of this evaluation, it was decided to use UMT-QVT. This tool is considered in detail in the next section.

4.3 UMT-QVT

UMT-QVT is a UML Model Transformation Tool developed by Sintef [Sin]. QVT stands for Query, View, Transformation. This tool is developed in order to transform models and generate code out of UML or XMI. UMT-QVT is implemented using Java. It is open source. A screenshot of UMT-QVT is displayed in Figure 4.4.

UMT-QVT requires an XMI file as input, which can be transformed to types like J2EE and IDL. In SUMAGRAM, the required output type is Java. UMT-QVT provides transformations to "Java-interfaces" and "Java-based". The output of these transformations was although not satisfiable, which implied that this tool could not be used.

UMT-QVT provided a transformation from the basic XMI file to an XMI-light file. This XMI-light file gives a very good overview of the large XMI file. As this file seems quite useful, it is investigated profoundly if it was appropriate to use it. The answer to this question is given in Section 5.3.

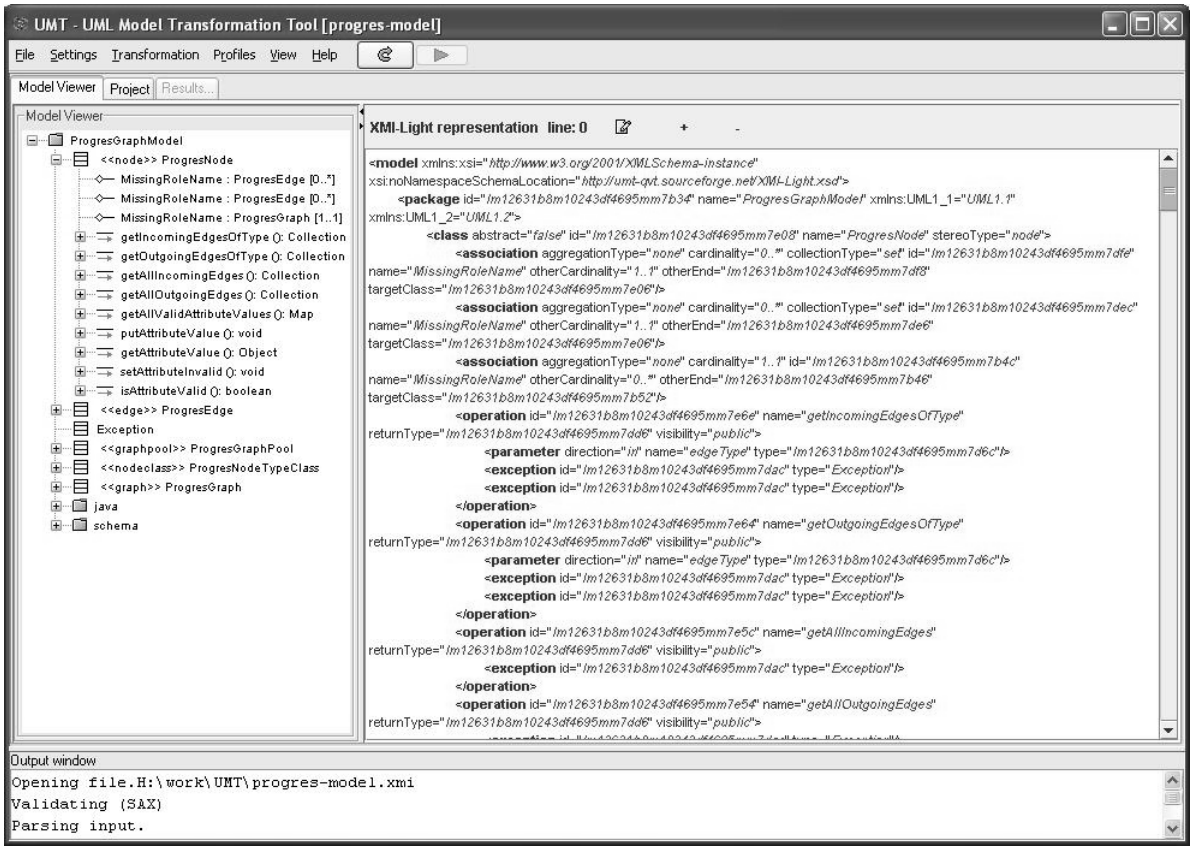


Figure 4.4: UMT-QVT Screenshot

4.4 MDA and SUMAGRAM

The concept of MDA is to take a platform independent model (PIM), transform it into a platform specific model (PSM) and then transform this PSM to code. For SUMAGRAM, there exists the Gras/GXL graph model and a specialized graph model. With the specialized graph model as input, code should be generated by SUMAGRAM.

Obviously, MDA uses two models and outputs code and so does Gras/GXL. Can there any similarities be found? The Gras/GXL graph model is a platform independent model, while the specialized graph model is a platform specific model. For both MDA and SUMAGRAM, the output is code. Even though, this comparison is not quite correct. The specialized graph model is not really created through a transformation of the Gras/GXL graph model: the relationship that exists between the two models, is created by the user in order to store the graphs in the Gras/GXL database. Nevertheless, the conversion from the specialized graph model to the code is similar to the concept where the PSM is transformed to code.

Therefore, the code generation of MDA might be useful in the concept of SUMAGRAM. From the specialized graph model, code should be generated following the concept of SUMAGRAM. This should be possible with MDA, as MDA requires an XMI or UML input, and code output. Unfortunately, as the tools are not applicable, using MDA is no option, and a new-defined solution concept has to be specified for SUMAGRAM.

Chapter 5

Solution Concept

As using an MDA tool was not an option (see Chapter 4), the whole conversion has to be done by a self-implemented application. In this chapter, the concept 5.1 of that program will be explained. As an input, a UML diagram is used. As most UML tools dispose of an export feature to XMI, this is the next file in the chain. This XMI file is transformed into an XML file, using XSLT. Then the implemented Java program is executed, using the XML file as input. The output of this implementation is the mapping in Java code.

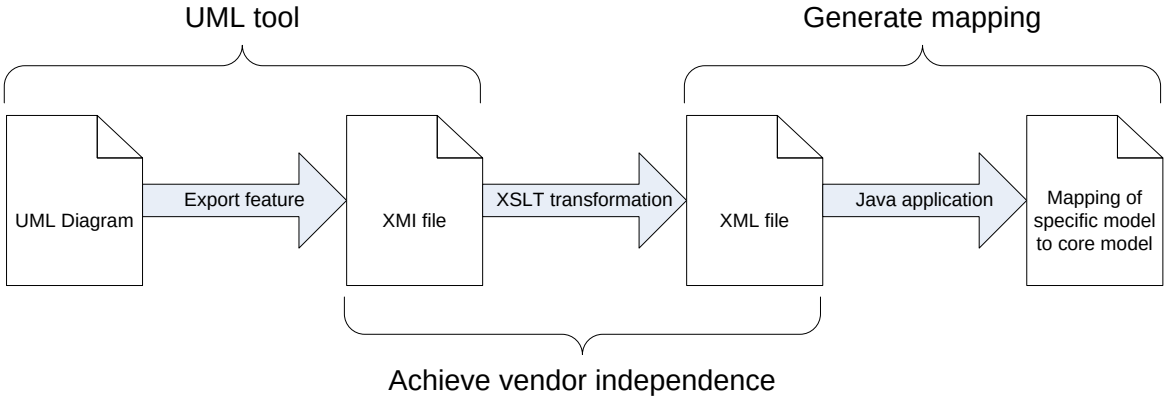


Figure 5.1: Overview of the Solution Concept

The concept of the solution will be explained by means of an example (Figure 5.2). In the first section, the UML diagram of the example will be explained. Subsequently, the XMI file will be demonstrated. In Section 5.3, the transformation from the XMI file to an XML file is described. Afterwards, it is declared how this XML will be converted into the mapping, which consists of Java code. To conclude this chapter, some further plans are revealed.

5.1 UML Class Diagram

In this project, UML class diagrams are used to create the graph models. The different classes of the graph models are specified, as well as the relations between the entities. The PROGRES graph model

is defined with the help of UML class diagrams. In Figure 5.2, a UML class diagram and an instance graph are depicted. This diagram will be used to explain the solution concept and is therefore kept simple. The diagram consists of a graph (`Graph`), a node (`Node`), and an edge (`Edge`). These entities are all classes of the diagram. The displayed graph is composed of nodes and every node can be part of one or more edges.

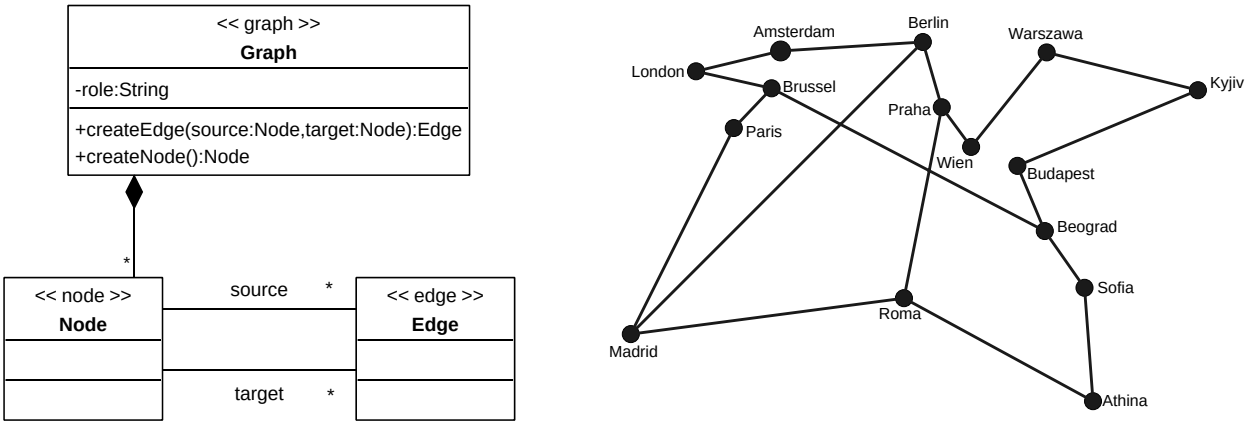


Figure 5.2: Example: a UML Class Diagram and the Corresponding Graph

The `Graph` class has a role as its attribute. In addition, two methods are defined in this class: `createEdge` and `createNode`. These methods can be specified in the UML file. In this example, only for the `createEdge` method, some code is described.

A link between the specialized graph model and the Gras/GXL graph model has to be established. This is done by means of the stereotype. Almost every class in the specialized graph model gets a stereotype. This stereotype refers to the corresponding class of the Gras/GXL graph model. For instance, the class `ProgresGraph` of the PROGRES graph model is related to the class `Graph` of the Gras/GXL graph model. This relationship has to be defined in order to be able to grasp the methods of Gras/GXL. For instance, when a new edge should be created, the method of Gras/GXL for creating this new edge is needed. When the stereotype is defined, SUMAGRAM knows that it can use the method in the `Graph` class of Gras/GXL. If no stereotype is defined, no method can be used. This example might seem easy, because the names ProgresGraph and Graph are alike and the relationship between the two might seem obvious. Nevertheless, this is no self-evidence. One could also imagine that the names are not so alike. For instance, there might be a German who defines a 'Knoten' class which is related to the `Node` class. If this happens, the relationship is less obvious.

To represent the UML class diagram adequately, a number of tools exist. Out of this range, two tools were selected: Rational Rose [Ros] and Poseidon for UML [Pos]. These tools were evaluated and compared. The most important feature that was required is the export feature to XMI, which is indispensable in this project. Next, they are evaluated on the matter of usability and quality of the delivered XMI file.

5.1.1 Poseidon for UML or Rational Rose

Poseidon for UML is developed by Gentleware since the year 2000. The version that is currently available is version 3.0. Poseidon delivers the possibility to create all kinds of UML diagrams. It also

possesses a feature for exporting the UML diagram in XMI format. Poseidon is easy to use and can define many details of the UML diagram in an easy and user-friendly way. The delivered XMI file contains all details of the UML class diagram. The XMI version that is exported is version 1.2.

Rational Rose is developed by Rational Software since the early nineties. With Rose, all kinds of UML diagrams can be designed. By a UML/XMI add-in, the export of a UML diagram to an XMI file is enabled. Rose is quite difficult to use. To realize a composition association, an aggregation relation should be created with the "call by value" option selected. Poseidon on the contrary, has a simple checkbox "composition" which can be enabled. In the XMI file, the details of the UML diagram are represented. The exported file has the format of XMI version 1.0 or 1.1.

In both the XMI files, the same information is contained. This means that on that matter, no diversification can be found. The user-friendliness on the other hand, is evaluated better for Poseidon. Also, the fact that the exported XMI has a newer version is a reason to rate Poseidon higher. Hence, Poseidon is used as the tool responsible for maintaining the UML class diagrams and exporting them to XMI.

5.2 XML Metadata Interchange

Poseidon exports a UML diagram to an XMI file. In the delivered XMI file, every detail of the UML diagram is contained. The XMI file is dependent on the used XMI exporter and the XMI version. The XMI exporter is defined by the developer of the UML tool. This means that an XMI file exported by Poseidon looks different compared to the one exported by Rational Rose, even though it concerns exactly the same UML diagram. The XMI file consists of lots of information. Because an XMI file of a small example like in Figure 5.2, already counts five pages, the most important parts are lifted out.

The XMI file starts with a definition of the used version. As XMI is a kind of XML, the used XML version is defined. Also the XMI version, the namespace and the creation date of the file are defined in the header of the XMI file.

```
<?xml version='1.0' encoding='UTF-8' ?>
<XMI xmi.version='1.2' xmlns:UML='org.omg.xmi.namespace.UML' timestamp='Tue␣May␣03␣11:04:34␣CEST␣2005'>
</XMI>
```

For every class in the UML class diagram, a corresponding entity is created in the XMI file. The class that will be used as an example is the `Graph` class. It looks as follows in the XMI file:

```
<UML:Class xmi.id='Im23c9c469m102f337f5f0mm7e72' name='Graph' visibility='public' isSpecification='false' isRoot='false'
      isLeaf='false' isAbstract='false'  isActive=' false '>
   <UML:ModelElement.stereotype>
      <UML:Stereotype xmi.idref='Im791a2191m1033723d2d7mm7d70'/>
   </UML:ModelElement.stereotype>
   <UML:Classifier.feature>
      <UML:Attribute xmi.id='I7d277254m102f90c4e68mm7db2' name='role' visibility='private' isSpecification='false'
            ownerScope='instance' changeability='changeable'>
         <UML:StructuralFeature.type>
            <UML:DataType xmi.idref='I7d277254m102f90c4e68mm7da1'/>
         </UML:StructuralFeature.type>
      </UML:Attribute>
```

```
      <UML:Operation xmi.id='I7d277254m102f90c4e68mm7da0' name='createEdge' visibility='public' isSpecification='false'
            ownerScope='instance' isQuery='false' concurrency='sequential' isRoot='false' isLeaf='false' isAbstract='false'>
         <UML:BehavioralFeature.parameter>
            <UML:Parameter xmi.id='I7d277254m102f90c4e68mm7d9d' name='return' isSpecification='false' kind='return'>
               <UML:Parameter.type>
                  <UML:Class xmi.idref='Im23c9c469m102f337f5f0mm7e4c'/>
               </UML:Parameter.type>
            </UML:Parameter>
            <UML:Parameter xmi.id='Im58e4ebb0m103a1c66056mm7d4c' name='source' isSpecification='false' kind='in'>
               <UML:Parameter.type>
                  <UML:Class xmi.idref='Im23c9c469m102f337f5f0mm7e5f'/>
               </UML:Parameter.type>
            </UML:Parameter>
            <UML:Parameter xmi.id='Im58e4ebb0m103a1c66056mm7d1f' name='target' isSpecification='false' kind='in'>
               <UML:Parameter.type>
                  <UML:Class xmi.idref='Im23c9c469m102f337f5f0mm7e5f'/>
               </UML:Parameter.type>
            </UML:Parameter>
         </UML:BehavioralFeature.parameter>
      </UML:Operation>
      <UML:Method xmi.id='I7d277254m102f90c4e68mm7d9e' isSpecification='false' isQuery='false'>
         <UML:Method.body>
            <UML:ProcedureExpression xmi.id='I7d277254m102f90c4e68mm7d9f' language='java' body='␣␣␣␣␣␣␣␣return␣new␣
                  Edge(source,target);&#10;'/>
         </UML:Method.body>
         <UML:Method.specification>
            <UML:Operation xmi.idref='I7d277254m102f90c4e68mm7da0'/>
         </UML:Method.specification>
      </UML:Method>
<!-- ... -->
   </UML:Classifier.feature>
</UML:Class>
```

Every entity in the XMI file has a unique identifier and a name. They are respectively denoted by `xmi.id` and `name`. In the UML class diagram, on behalf of the classes, a number of properties can be defined, like abstract and root. These properties are also available in the XMI file, in the shape of attributes like `isAbstract` and `isRoot`. The lines including the identifier, name and properties form the header of the entity.

The stereotype that is defined in the UML class is represented by the tag `UML:Stereotype`. For this entity, only the reference is given. The actual definition of the stereotype is defined elsewhere in the file.

In the UML class diagram, per class, attributes and operations are defined. These are also existent in the XMI file. Attributes and operations are interpreted within a `UML:Classifier.feature` tag. An attribute is represented by the tag `UML:Attribute`. The attributes are defined by an identifier and a name. For the example of the `Graph` class, the name is `role`. There are a number of other properties defined that are not of major importance. The attribute `role` has the Java type `String`. This type is elsewhere defined as the data type with the identifier `I7d277254m102f90c4e68mm7da1`. In the attribute entity, this identifier is already mentioned. These identifiers are used in the XMI file for referring. As can be seen, at the level of the stereotype, only an identifier is mentioned, and the specification can be found elsewhere. This makes the XMI file hard to read, and difficult to process the data because the specifications often have to be looked up to obtain the required information.

The next entity defined for a class, is the operation, which is defined by the tag `UML:Operation`. The operation that is defined here, is the public operation `createEdge`. This operation has two input parameters: the source and target node between which the edge should be created. These parameters are both of type `Node`, which is defined by the tag `UML:Parameter.type`. The type is defined by a reference to a class, which is the `Node` class in this case. The output parameter is in XMI denoted by the parameter with kind `return`. The name is also by default `return`, but can be changed if desired. The type of this parameter can be either an actual type or `void`. For this operation, the return type is `Edge`. The contents of this operation is defined by the tag `UML:Method`. For the method is defined to which operation it belongs. In addition, the actions that should be accomplished for this method are defined, accompanied by the language, which is Java in this case.

Beside classes, also associations are defined in detail. The association upon which a light will be thrown is the source relation between the class `Node` and `Edge`.

```
<UML:Association xmi.id='Im23c9c469m102f337f5f0mm7dd8' name='source' isSpecification='false' isRoot='false' isLeaf='false'
    isAbstract='false'>
  <UML:Association.connection>
    <UML:AssociationEnd xmi.id='Im23c9c469m102f337f5f0mm7dde' visibility='public' isSpecification='false' isNavigable='false'
        ordering='unordered' aggregation='none' targetScope='instance' changeability='changeable'>
      <UML:AssociationEnd.multiplicity>
        <UML:Multiplicity xmi.id='Im23c9c469m102f337f5f0mm7ddc'>
          <UML:Multiplicity.range>
            <UML:MultiplicityRange xmi.id='Im23c9c469m102f337f5f0mm7ddd' lower='1' upper='1'/>
          </UML:Multiplicity.range>
        </UML:Multiplicity>
      </UML:AssociationEnd.multiplicity>
      <UML:AssociationEnd.participant>
        <UML:Class xmi.idref='Im23c9c469m102f337f5f0mm7e5f'/>
      </UML:AssociationEnd.participant>
    </UML:AssociationEnd>
    <UML:AssociationEnd xmi.id='Im23c9c469m102f337f5f0mm7ddb' visibility='public' isSpecification='false' isNavigable='false'
        ordering='unordered' aggregation='none' targetScope='instance' changeability='changeable'>
      <UML:AssociationEnd.multiplicity>
        <UML:Multiplicity xmi.id='Im23c9c469m102f337f5f0mm7dbb'>
          <UML:Multiplicity.range>
            <UML:MultiplicityRange xmi.id='Im23c9c469m102f337f5f0mm7dba' lower='0' upper='-1'/>
          </UML:Multiplicity.range>
        </UML:Multiplicity>
      </UML:AssociationEnd.multiplicity>
      <UML:AssociationEnd.participant>
        <UML:Class xmi.idref='Im23c9c469m102f337f5f0mm7e4c'/>
      </UML:AssociationEnd.participant>
    </UML:AssociationEnd>
  </UML:Association.connection>
</UML:Association>
```

The header of an association consists of an identifier and a name. An association is characterized by two association ends, in this case one association end is `Node` and the other is `Edge`. For an association end, among others, the visibility, the navigability, the ordering, and the aggregation are defined. For the association end, also the multiplicity is given, specified by a lower and upper range. Last but not least, also the referring class is defined, in this case respectively the `Node` and `Edge` class.

Also for the stereotype, the properties are listed. For the `Graph` class, the stereotype is `graph`. It is specified by an identifier, a name and some other properties.

```
<UML:Stereotype xmi.id='Im791a2191m1033723d2d7mm7d70' name='graph' visibility='public' isSpecification='false' isRoot='false'
      isLeaf='false' isAbstract='false'>
   <UML:Stereotype.baseClass>Class</UML:Stereotype.baseClass>
</UML:Stereotype>
```

So for every UML diagram, an XMI file can be obtained. This XMI file is unfortunately rather complex. Therefore, it was considered to convert the XMI file to an XML file. This conversion also has some other advantages, which will be clarified in the next section.

5.3 The Transformation from XMI to XML

UMT-QVT already provided the concept of an XMI-light file. The transformation that was responsible for the creation of the XMI-light was observed. This transformation is done by means of an XSLT file. EXtensible Stylesheet Language Transformations (XSLT, [Cla99]) is a language for transforming XML documents to other documents. In the project, the XMI file is a derivative of an XML file and hence XSLT can be used for the transformation.

The result of the transformation using the XSLT provided by UMT-QVT, delivered an XML file (previously called XMI-light file). This XML file did not contain all the elements that were necessary for SUMAGRAM. For instance, if an attribute is `public` or `private` was not written in the XML file. The XSLT file was adjusted such that these details were copied too. The XML file resulting after this transformation, could be used for SUMAGRAM.

The transformation from XMI to XML is not indispensable. It is possible to convert the XMI file right away to Java code. Nevertheless, the use of this transformation has some essential advantages. The rationale behind the transformation step is to achieve a tool and vendor independent file. Because the XMI file is dependent on the tool, this would mean that SUMAGRAM could only be used in combination with Poseidon. The difference between the output of Rational Rose and Poseidon is shown in Figure 5.3.

```
<UML:Operation xmi.id='S.125.0958.53.3' name='createEdge' visibility='public'
 isSpecification ='false' ownerScope='instance' isQuery='false' concurrency='sequential'
isRoot='false' isLeaf='false' isAbstract='false'
 specification ='return␣new␣Edge(source,target);'/>
```

(a) Rational Rose

```
<UML:Method.body>
    <UML:ProcedureExpression xmi.id='I7d277254m102f90c4e68mm7d9f' language='java'
    body='␣␣␣␣␣␣␣␣return␣new␣Edge(source,target);&#10;'/>
</UML:Method.body>
```

(b) Poseidon

Figure 5.3: Comparison of the XMI Output of Rational Rose and Poseidon

Figure 5.3(a) shows the XMI part (outputted by Rational Rose) that represents the body of the method. A `specification` attribute is added to the tag `UML:Operation`. For Poseidon (Figure 5.3(b)), the body is shown by means of a tag `UML:Method.body`. Because of this difference, the transformation to an independent file is necessary. This is done by transforming the XMI file. When

another tool than Poseidon is used, the XSLT conversion should be adapted. Then an XML file can be obtained that is suitable to use by the program. Another advantage is that the XML file is more readable than the XMI file, which also has a small influence on the run time of the program.

If another tool is used, the XSLT file can be adapted. This implies that the user knows the structure and elements of the XML file. This structure is defined in a DTD file. A Document Type Definition file specifies which elements are allowed and required in the XML file. The DTD file belonging to the XML file of SUMAGRAM looks like shown below. A number of elements are represented: for an association, it is defined which possible attributes should be present.

```
<?xml version="1.0" encoding="UTF−8"?>
<!ELEMENT association EMPTY>
<!ATTLIST association
    name CDATA #REQUIRED
    id CDATA #REQUIRED
    targetClass CDATA #REQUIRED
    cardinality CDATA #REQUIRED
    otherCardinality CDATA #REQUIRED
    collectionType CDATA #IMPLIED
    aggregationType CDATA #REQUIRED
    otherAggregationType CDATA #REQUIRED
    navigation CDATA #REQUIRED
    otherNavigation CDATA #REQUIRED
    otherEnd CDATA #IMPLIED
>
<!ELEMENT attribute EMPTY>
<!ATTLIST attribute
    name CDATA #REQUIRED
    id CDATA #REQUIRED
    type CDATA #REQUIRED
    changeability CDATA #REQUIRED
    cardinality CDATA #REQUIRED
    class CDATA #IMPLIED
    datatype CDATA #IMPLIED
    initialValue CDATA #IMPLIED
>
<!ELEMENT class (attribute*, association*, operation*)>
<!ATTLIST class
    name CDATA #REQUIRED
    id CDATA #REQUIRED
    stereotype CDATA #IMPLIED
    superClass CDATA #IMPLIED
    abstract CDATA #REQUIRED
>
<!ELEMENT method EMPTY>
<!ATTLIST method
    name CDATA #REQUIRED
>
<!ELEMENT operation (returntype, parameter*, method)>
<!ATTLIST operation
    name CDATA #REQUIRED
    visibility CDATA #REQUIRED
>
<!ELEMENT package (package+ | class+)>
<!ATTLIST package
    name CDATA #REQUIRED
    id CDATA #REQUIRED
>
```

```
<!ELEMENT parameter EMPTY>
<!ATTLIST parameter
   name CDATA #REQUIRED
   type CDATA #REQUIRED
   direction  CDATA #REQUIRED
>
<!ELEMENT returntype EMPTY>
<!ATTLIST returntype
   name CDATA #REQUIRED
>
<!ELEMENT sourceroot (class | package | attribute)+>
<!ATTLIST sourceroot
   changeability CDATA #REQUIRED
>
```

In the DTD all the elements that can occur in the XML file are listed. They are listed in alphabetical order. First, the `association` is listed with all its possible attributes: `name`, `id`, `targetClass`, etcetera. The attributes `otherEnd` and `targetClass` might seem identical, but they are not. The `targetClass` defines the class on the other association end. The `otherEnd` specifies the identifier of the `association` that defines the other association end. The element `attribute` defines the attributes of a class. In addition, the properties are listed in the DTD file. A `class` can consist of multiple `attributes`, `associations` and `operations`. In the attribute list of a `class` are the name, id, etcetera defined. The next element is the `method`. The element `operation` is defined by means of `returntype`, multiple `parameters` and `method`. Consequently, the elements `package`, `parameter`, `returntype` and `sourceroot` are defined including the attributes.

The file that is obtained after the transformation from XMI to XML is displayed below. This file is a lot smaller than the original XMI file. Only the necessary data is copied into the XML file and the number of references is reduced to a minimum. The created XML file should be transformed to the desired output: the mapping. This transformation is the last phase of the program.

```
<?xml version="1.0" encoding="UTF−8"?>
<sourceroot changeability="">
   <class name="Graph" id="Im23c9c469m102f337f5f0mm7e72" stereotype="graph" abstract="false">
      <attribute name="name" id="I7d277254m102f90c4e68mm7db2"
            type="I7d277254m102f90c4e68mm7da1" changeability="changeable" cardinality=""
            datatype="String"/>
      <association name="" id="Im23c9c469m102f337f5f0mm7d52"
            targetClass="Im23c9c469m102f337f5f0mm7e5f" cardinality="0..*" otherCardinality="1..1"
            collectionType="set" aggregationType="composite" otherAggregationType="none"
            navigation="true" otherNavigation="true" otherEnd="Im23c9c469m102f337f5f0mm7d55"/>
      <operation name="createEdge" visibility="public">
         <returntype name="Edge"/>
         <method name="␣␣␣␣␣␣␣␣return␣new␣Edge();&#xA;"/>
      </operation>
      <operation name="createNode" visibility="public">
         <returntype name="Node"/>
         <method name=""/>
      </operation>
   </class>
```

```
    <class name="Node" id="Im23c9c469m102f337f5f0mm7e5f" stereotype="node" abstract="false">
      <association name="source" id="Im23c9c469m102f337f5f0mm7ddb"
          targetClass="Im23c9c469m102f337f5f0mm7e4c" cardinality="0..*" otherCardinality="1..1"
          collectionType="set" aggregationType="none" otherAggregationType="none"
          navigation="false" otherNavigation="false"/>
      <association name="target" id="Im23c9c469m102f337f5f0mm7d9a"
          targetClass="Im23c9c469m102f337f5f0mm7e4c" cardinality="0..*" otherCardinality="1..1"
          collectionType="set" aggregationType="none" otherAggregationType="none"
          navigation="false" otherNavigation="false"/>
      <association name="" id="Im23c9c469m102f337f5f0mm7d55"
          targetClass="Im23c9c469m102f337f5f0mm7e72" cardinality="1..1" otherCardinality="0..*"
          aggregationType="none" otherAggregationType="composite" navigation="true"
          otherNavigation="true" otherEnd="Im23c9c469m102f337f5f0mm7d52"/>
    </class>
    <class name="Edge" id="Im23c9c469m102f337f5f0mm7e4c" stereotype="edge" abstract="false">
      <association name="source" id="Im23c9c469m102f337f5f0mm7dde"
          targetClass="Im23c9c469m102f337f5f0mm7e5f" cardinality="1..1" otherCardinality="0..*"
          aggregationType="none" otherAggregationType="none" navigation="false"
          otherNavigation="false"/>
      <association name="target" id="Im23c9c469m102f337f5f0mm7d9d"
          targetClass="Im23c9c469m102f337f5f0mm7e5f" cardinality="1..1" otherCardinality="0..*"
          aggregationType="none" otherAggregationType="none" navigation="false"
          otherNavigation="false"/>
    </class>
    <package name="java" id="I7d277254m102f90c4e68mm7db4">
      <package name="lang" id="I7d277254m102f90c4e68mm7db5"/>
    </package>
  </sourceroot>
```

5.4 From XML to the Mapping

After interpreting the XML file and reading in the information in Java, the information has to be interpreted further. As already mentioned in earlier chapters, it is desired to write the code into the UML diagram in a short description form or tag. Through this short form, the user is supported by creating a mapping. Without this short form, he has to write the entire code by hand, as is stated in the problem definition.

The idea for this short description form expanded, because the code pieces in the mapping show similarities. Some actions occur quite often. For instance, methods or procedures like the creation of an entity, and the translation of an entity from Gras/GXL to the specialized application domain. These actions appear in different methods, so it is useful to create a kind of macro, which can create a standard piece of code. A macro is used to handle often occurring actions. This macro has to be defined in a way, that the program knows which methods of Gras/GXL can be used. But also such that the program is in no way dependent on the specialized application domain. For instance for PROGRES, it would be easy to define in the program that a `ProgresNodeType` and a `ProgresNodeClass` both exist. By writing everything into the program, it becomes quite easy for the user and he will only have to write very little information himself. However, if these entities

were literally mentioned in the program, it could not anymore be used for the GXL graph model. Therefore, independency is very important to keep an eye on. This problem will be explained more in-depth in Section 6.7.

5.5 Further Plans

It was also considered that there should be a possibility to use constraints. Therefore, it was considered to include OCL constraints in the project. It would be efficient to include these constraints in the UML diagram. In Poseidon, these constraints can be included in the UML diagram. They can be added on various levels like classes, attributes, operations, and associations.

This solution concept now should be transformed to an actual application. The realisation of this application is extensively described in the next chapter.

Chapter 6

Realisation

The methodology of this thesis project can be divided into different parts. In Figure 6.1, the development of SUMAGRAM is shown again. The first part is the creation of a UML diagram. In this case, the UML diagram is created using the tool Poseidon. As an example, the PROGRES graph model is used. In the next step, the UML diagram is exported as an XMI file. This can be done using a feature of Poseidon. With this file, an XML file is created which contains the relevant data of the original XMI file. The transformation between the original XMI file and the XML file is done using an XSLT file. From the resulting XML file, the target is to create files containing the mapping. This step is fulfilled using a Java application.

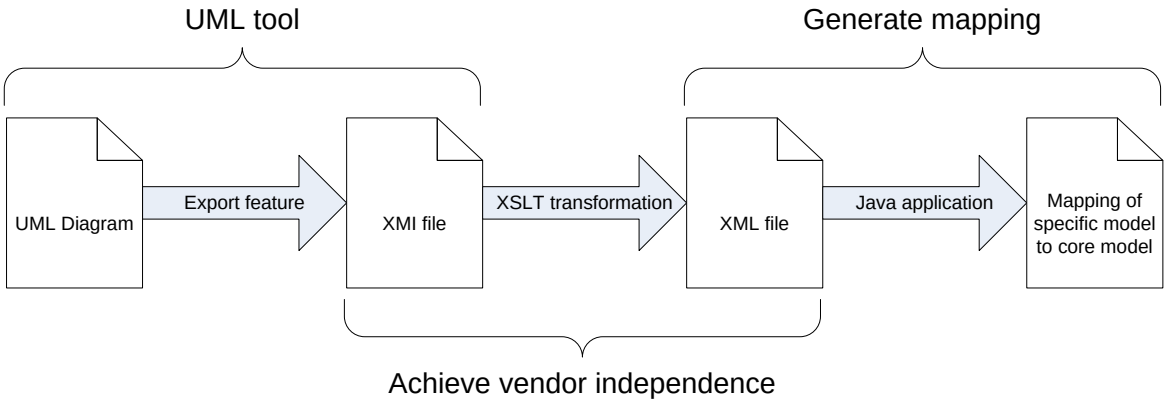


Figure 6.1: Overview of the Solution Concept

The first section of this chapter will explain the expected contents of the UML class diagram. Section 6.2 concentrates on the XSLT file that is included in SUMAGRAM and how it will be handled by the program. The following subject is the parsing of the XML file. Also the creation of the SchemaFactory class is explained. Section 6.5 handles the creation of the Java files systematically. How the tags are interpreted, is elucidated in the next section. The last section reveals any problems that are encountered during the implementation.

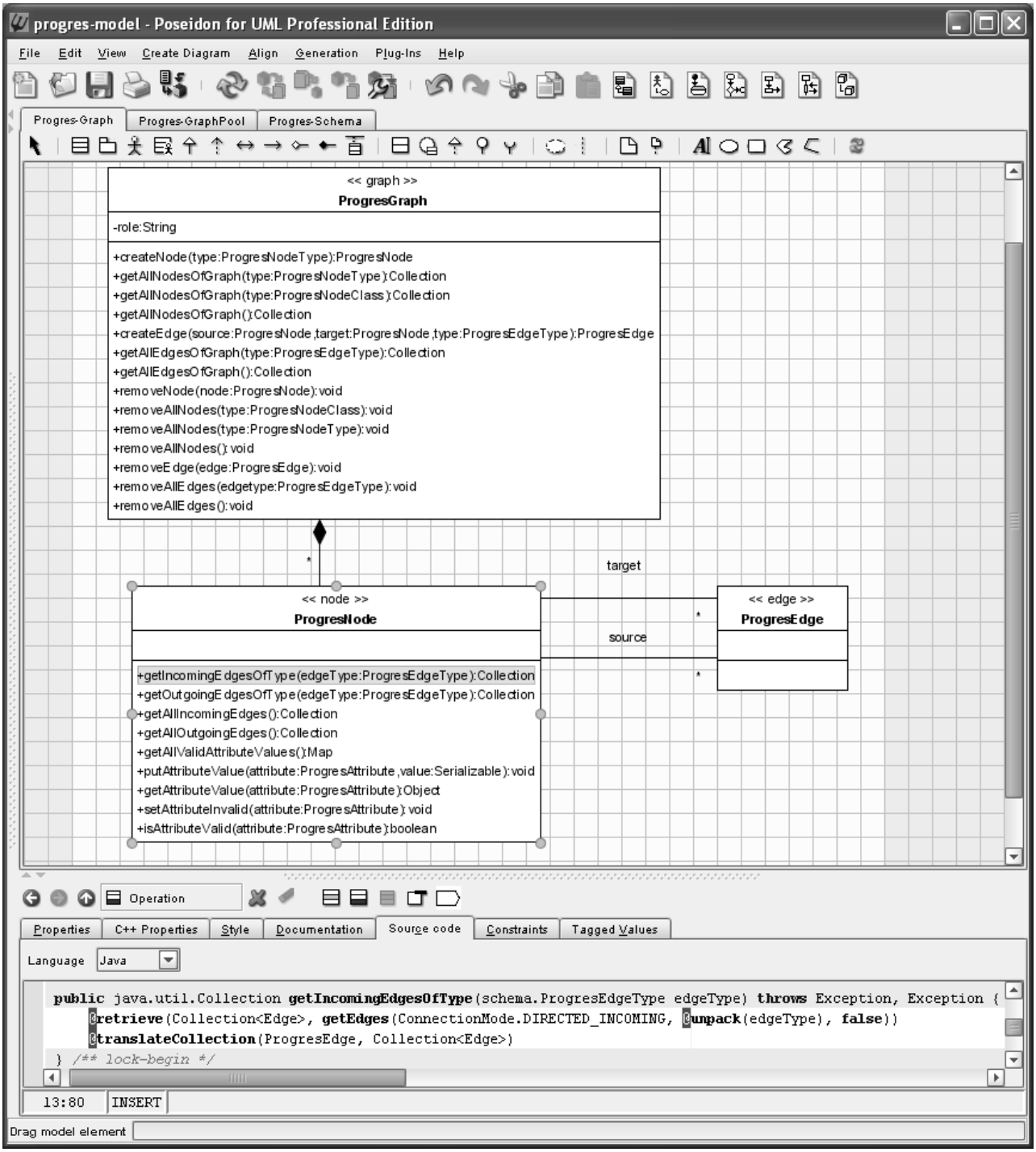


Figure 6.2: Screenshot of Poseidon

6.1 Contents of the UML Class Diagram

The input of SUMAGRAM is the UML class diagram of a specialized graph model. An example of a specialized graph model is the PROGRES graph model, which is described extensively in Section 3.3.2. The depiction of the graph model is done using Poseidon. However, any other UML case tool

will do. In the UML diagram, all classes need to be created. The attributes for every class are added to the classes. These attributes are defined with name and type. The associations define the relations between the classes and must also be drawn. Last but not least, the operations should be listed per class.

In the UML diagram, it is possible to literally write the code of the operations. As one of the requirements for SUMAGRAM was that the user did not have to write full code, another method had to be found. The idea to use tags originated (see Section 5.4 and Section 6.1.1). The purpose was to create a short description of the code that had to be inserted on the indicated position. This description had to be satisfactory for the program to know what code exactly had to be inserted. In addition, the tags have to be easy to use and to the point. One major thing to keep in mind was that the program has to be independent of the specialized graph model. It may only contain data of the Gras/GXL graph model.

In Figure 6.2, a screenshot of Poseidon is displayed. It shows that for every class, the operations are created. For every operation, the source code can be written down. In the picture, the operation `getIncomingEdgesOfType` is selected. For this method the source code is defined. This is done using tags. A tag always starts with the character '@'.

6.1.1 Tags

With tags, a short description of the method should be given, such that the program is able to interpret the given information. The first definition of the tags was a description of the method (usually the name) and information concerning the involved entities of both the core model and the specialized graph model. As an example the method `createNode` of the PROGRES graph model will be used, which creates a new node in the graph. The tag would look like this:

```
createnode-node-progresnode
```

The tag consists of the description of the method that should be outputted, the second argument is the entity of Gras/GXL that corresponds with this method. Finally, the output type is given. When using the tags in this format, the program would not be independent of the specialized graph model, and almost all the code had to be included in the program. This seemed not right. When all code would be included in the program, this would have the consequence that if Gras/GXL methods change, the output will not be consistent. It would be an improvement if the information of the core model could be imported through the UML diagram. Consequently, this kind of tags is considered to be unuseful and the tags had to be reformed.

The second idea for the tags was to make groups based on types of methods. Each group is a type of method that occurs often. For example, the retrieval of an instance or the creation of an entity represent both a group. If necessary, the method of the corresponding Gras/GXL class is given as a parameter. Also, the expected output type and sometimes the input type are given. From this description, later on, the extended code is created.

Tag name	Parameters
retrieve	output type, method
translateEntity	output type, input type
translateCollection	output type, input type
unpack	variable
create	output type, method
declare	output type, method
check	method, reaction when false
write	method

There are several types of tags. These types are described in the table above. A tag is prefixed with the sign '@'. This sign informs the program that there is a tag at hand. When there is no sign at the beginning of the phrase, then the program will interpret the following text as literal code, and it will copy this code literally into the mapping code. The program will evaluate the code of the operation line by line, lines starting with an @-sign are tags and lines not starting with @ are interpreted as literal code. This means that combinations of both tags and literal code are accepted by the program.

The tag *retrieve* has as parameters an output type and a method. The method is a method of the Gras/GXL graph model that returns a certain output type. This output type can be a collection of instances of entities or an instance of a specific entity. For example: `retrieve(NodeClass,getNodeClass)` retrieves a `NodeClass`. This node class is needed for a method that wants a `ProgresNodeClass`. After a *retrieve* tag, usually a *translate* tag is encountered.

The *translate* tag makes sure that the retrieved class is converted to a class of the specific model. There are two kinds of translation tags, a *translateEntity* tag, which translates one instance, and a *translateCollection* tag, which translates a collection of instances. The translation tags have as first argument an output type and as second argument an input type. An example of the *translateEntity* tag is `translateEntity(ProgresEdgeType,EdgeClass)`. For the *translateCollection* tag, the example looks like: `translateCollection(ProgresNodeClass, Collection<NodeClass>)`. Of course, the output of the *translateCollection* method should also be a collection of `ProgresNodeClass` but the argument simply explains to the program that every `NodeClass` in the set should be translated to a `ProgresNodeClass`. The returned instances are looked up in the factory where the classes of the schema of the specialized graph model are stored.

The combination of a *retrieve* tag and a *translateCollection* tag looks as shown below. These tags are defined in the method `getAllSuperClasses` of the class `ProgresNodeClass`. The first tag defines that the superclasses of the current node class should be retrieved. The current node class is in this case the `NodeClass` corresponding to the current `ProgresNodeClass`. The output of the method defined by the first tag is a collection of `NodeClasses`. With the second tag, the content of the collection will be translated. The elements in the collection are of type `NodeClass` and have to be converted to `ProgresNodeClasses`.

```
@retrieve(Collection<NodeClass>,getAllSuperClasses())
@translateCollection(ProgresNodeClass,Collection<NodeClass>)
```

The *unpack* tag is often included in another tag. For almost every class of the specialized graph model, a corresponding class of the Gras/GXL graph model exists. The *unpack* tag returns for a specialized

class its corresponding Gras/GXL class. The parameter for the *unpack* tag is a variable, that is most of the time a parameter of the method.

With the *create* tag, a new instance of an entity of the core graph model can be created. Examples of entities are `Edge`, `Node` or `Graph`. The tag `create(Node,createNode(@unpack(type)))` is an example for the *create* tag. The parameters are the output type and the method to retrieve this output. Afterwards, this created entity also has to be converted to an instance of the specialized graph model. This is done by the previously described translation tags. On the same level as the *create* tag, the *declare* tag exists, which is responsible for the declaration of new entity classes like `NodeClass`, `EdgeClass` or `GraphClass`. An example for a *declare* tag is `declare(NodeClass,declareNodeClass(name,false))`. This tag has two parameters: an output type and a method. These entity classes should also be translated. After creating an entity or an entity class, it should be stored in the factory.

The *create* and the *declare* tag resemble a lot. The difference exists in the responsibilities. The *create* tag is responsible for the creation of new entities belonging to the graph model (`Edge`, `Node` or `Graph`). The *declare* tag on the other hand, is responsible for the declaration of new entity classes, which appear in the graph schema (`NodeClass`, `EdgeClass` or `GraphClass`).

Another tag is the *check* tag. This tag asserts if new instances are in line with the schema rules. For instance, when a new edge is created, it is checked that the source and target are valid. The *check* tag receives as a parameter a method (which must return a boolean) and a corresponding reaction. An example is `check(isSourceValid(source, type),throw new SchemaCheckException("The type is not valid for the source."))`. The example shows the parameters: the first one is a method that returns a boolean value, the second parameter gives the reaction if the boolean value is false.

The last and most simple tag is the *write* tag. This tag writes a method that is defined in the parameter. This method can return a value or instance, but this depends on the return type of the method. If this return type is `void` nothing is returned, else the created value or instance of the written method is returned. The tag `write(getAttributeValue(@unpack(attribute)))` is an example of a *write* tag. The method in the *write* tag cannot be written as literal code into the UML diagram because it has to be applied onto the instance belonging to the stereotype of the class.

The parameters of the tags can contain another indicator. The used prefix is the number sign (#). An example is given next; this method is included in a *write* tag in order to remove a graph with a certain name: `@write(removeTopLevelGraph(#GraphPool.getTopLevelGraphByName(name)))`. This indicates that an instance of a certain type is wanted. This instance is not straightaway available because of two reasons. The first one is that the place where the instance is needed is not the usual place. The second one is that the instance is not the instance corresponding with the stereotype. In the given example, the instance is wanted in the parameter of the method. On this location, the program does not expect that it has to write some self-defined output. The number sign explains to the program that there is an instance of a type expected, on a given location. The type is defined immediately after the sign, the location is the location where it is defined. SUMAGRAM replaces the sign including the type by a variable corresponding to the defined type.

6.1.2 Input

It is important that all the desired information of the specialized graph model is available in the UML diagram. This is the most appropriate way to give input to the program. Input can also be given through external files. The information in these files then has to be embedded into the XML file. In the UML diagram, there has to be defined if it is possible to change the attributes. This can be done by describing the changeability in the constraints window of the model element of the diagram. This changeability affects the existence of several methods. In Poseidon, it is possible to add constraints on the level of the model, which is the highest possible level. This constraint is written as if it is an OCL constraint. The constraint is denoted as an invariant, with the name `changeability`. It can have two values: `frozen` or `changeable`.

6.2 XSLT file

As is already described in Section 5.2, the XMI file contains all the information of the UML class diagram. This XMI file is transformed into an XML file, using an XSLT file, which was originally defined within the tool UMT-QVT (see Section 4.3). On detail level, this file was not satisfactory. Some extra code had to be added. For instance, if an attribute was `public` or `private` was not translated into the XML file. By the XSLT transformation, the important information of the XMI file is selected and rewritten in a more readable and independent format. The resulting file is an XML file. To obtain this XML file, the XSLT transformation is included in SUMAGRAM in the shape of the piece of code defined below.

```
StreamSource sourceXMI = new StreamSource(PATH + "progres-model.xmi");
StreamSource sourceXSLT = new StreamSource(PATH + "xmitransform.xsl");
StreamResult result = new StreamResult(PATH + "result.xml");

try {
    System.setProperty("javax.xml.transform.TransformerFactory",
        "net.sf.saxon.TransformerFactoryImpl");
    TransformerFactory transformerFactory = TransformerFactory.newInstance();
    Transformer transformer = transformerFactory.newTransformer(sourceXSLT);
    transformer.transform(sourceXMI, result);
    new ParseFile(PATH + "result.xml");
    }
catch (Exception e) {
    e.printStackTrace();
}
```

The XMI file and XSLT file are both source files, and the XML file is a result file. By using a `TransformerFactory`, the transformation is accomplished. After this transformation, the XML file can be parsed. This action is introduced by the rule `new ParseFile(PATH + "result.xml")`.

6.3 Parsing the XML File

After obtaining the XML file, different paths could be treaded. The first idea was to use another XSLT file to convert the XML to Java. Therefore, a function should exist with which multiple output files can be created from one XML file. As the output of SUMAGRAM consists of a number of different Java files (one for each class in the UML diagram), such a function was necessary. With XSLT version 2.0 [Kay05b], this was possible. XSLT provided the function `xsl:result-document`. Despite this function was working well, it was not possible to create the output completely satisfying.

Because using only XSLT was quite difficult and complex, the decision to combine XSLT with another tool was taken. As the mapping code is defined in Java, it seemed logical to use Java for creating the output. Therefore, XSLT was combined with a Java application. This had also the advantage that it is not hard to combine them. By defining a namespace referring to a Java file (`xmlns:cm="java:i3.grasgxl.ext.gmgen.xslt.CreateMethod"`) a method occurring in this Java file can be called. All the information of the XML file had to be transported using parameters for the method. This working method was in the beginning satisfying. Nevertheless, it turned out to be difficult to obtain a certain attribute of the XML file. This encountered problem was the base for the decision to leave out the XSLT entirely and using only Java for the last transformation. With Java, this problem was nonexistent. Using Java, the XML file could easily be searched for a certain attribute. Because it was a lot easier, this working method was used.

To parse the XMI file, JDOM [Har02] was used. JDOM is an XML API, which is able to read and write XML files. It is a Java-based solution for accessing, manipulating and outputting XML data from Java code. JDOM is one of many solutions for dealing with XML. It is perfectly capable of interoperating with other tools like SAX (Simple API for XML, [Kay05a]) and DOM (Document Object Model, [PLHW05]). JDOM resembles DOM, but is still different. JDOM is developed for Java, in contrary to DOM, which is developed for multiple languages. Therefore, it is easier to integrate in Java, especially for someone who is already familiar with Java. JDOM can handle a lot of actions easier than DOM, which results in a simpler and more readable code.

JDOM is able to parse the XML file. With JDOM, elements and attributes can be selected. On the highest level, the `XMIProject` is created. It manages the elements in the project. The project stores all the classes. For every class in the UML class diagram, an `XMIClass` is produced. In these classes, the information (like attributes) is represented. In addition, the associations are interpreted and included in the classes as attributes. In the classes, the operations can be found. They are represented by an `XMIOperation`. When the whole XML file is parsed and the data is present in the different Java classes, the *real* work can start: the creation of the output files and the editing of the operations.

6.4 Creation of the SchemaFactory Class

Section 3.3.3 already described the factory briefly. The factory creates instances for every entity. The `ProgresSchemaFactory` stores for every class the entities in a `HashMap`. The entities are stored in the form of a key-value pair. For every instance of a class, a unique instance of its corresponding Gras/GXL class exists.

```
public class ProgresSchemaFactory{
    static Map<GraphEntityClass,ProgresEdgeType> edgeclassFactory = new
        HashMap<GraphEntityClass,ProgresEdgeType>();
    static Map<GraphEntityClass,ProgresNodeType> progresNodeTypeFactory = new
        HashMap<GraphEntityClass,ProgresNodeType>();
/* ... */
    static Map<GraphEntity,ProgresEdge> edgeFactory = new
        HashMap<GraphEntity,ProgresEdge>();
    static Map<Attribute, ProgresAttribute > attributeFactory = new
        HashMap<Attribute,ProgresAttribute>() ;
/* ... */

    public static void addProgresEdgeType(ProgresEdgeType progresedgetype){
        edgeclassFactory .put(progresedgetype.getEdgeClass() ,progresedgetype) ;
    }
    public static ProgresEdgeType getProgresEdgeType(GraphEntityClass edgeclass ){
        return edgeclassFactory . get ( edgeclass ) ;
    }
    public static void addProgresNodeType(ProgresNodeType progresnodetype){
        ProgresNodeTypeFactory.put(progresnodetype.getNodeClass() ,progresnodetype) ;
    }
    public static ProgresNodeType getProgresNodeType(GraphEntityClass nodeclass ){
        return ProgresNodeTypeFactory.get ( nodeclass ) ;
    }
/* ... */
    public static void addProgresEdge(ProgresEdge progresedge){
        edgeFactory.put(progresedge.getEdge() ,progresedge) ;
    }
    public static ProgresEdge getProgresEdge(GraphEntity edge){
        return edgeFactory.get(edge);
    }
    public static void addProgresAttribute ( ProgresAttribute progresattribute ){
        attributeFactory .put( progresattribute . getAttribute () , progresattribute ) ;
    }
    public static ProgresAttribute getProgresAttribute ( Attribute attribute ){
        return attributeFactory . get ( attribute ) ;
    }
/* ... */
}
```

For every factory, two methods are defined: a `get` and an `add` method. The `add` method is used after every creation or declaration of a new instance. Then the newly created instance is stored in the factory using the `add` method. The `get` method is used to retrieve a certain instance, corresponding to a Gras/GXL class. This class is used as parameter for the `get` method.

6.5 Creating the Java Files

For every class in the UML class diagram, a Java file should be created. These Java files all have the same name as the corresponding UML class. In the Java files, attributes and operations should be contained. The attributes of the classes are created in three steps. First, the stereotype is interpreted, then the associations are processed and as third, the attributes of the UML classes are added. With the attributes, the constructor can be composed. Afterwards, the operations should be written in the Java files. This is done with the help of the tags.

6.5.1 Stereotype

The stereotype of a class is defined in the UML class diagram. For instance, the stereotype of the `ProgresGraph` is `graph`. The stereotype corresponds to a class of the Gras/GXL graph model. Often this class has the same name as the stereotype with the first character uppercased. As this cannot be seen as a valid and exact rule, a lookup table has been established. The lookup table can also be used to define variable names and find the superclasses of the Gras/GXL classes.

```
<?xml version="1.0" encoding="UTF-8"?>
<lookuptable>
<stereotype name="relationend" javatype="RelationEnd" variable="relationend"
    superclass="graphentity"/>
<stereotype name="relation" javatype="Relation" variable="relation"
    superclass="graphentity"/>
<stereotype name="node" javatype="Node" variable="node" superclass="graphentity"/>
<stereotype name="edge" javatype="Edge" variable="edge" superclass="graphentity"/>
<stereotype name="graph" javatype="Graph" variable="graph" superclass="graphentity"/>
<!-- ... -->
<stereotype name="edgeclass" javatype="EdgeClass" variable="edgeclass"
    superclass="graphentityclass"/>
<stereotype name="nodeclass" javatype="NodeClass" variable="nodeclass"
    superclass="graphentityclass"/>
<stereotype name="graphclass" javatype="GraphClass" variable="graphclass"
    superclass="graphentityclass"/>
<stereotype name="schema" javatype="Schema" variable="schema"/>
<stereotype name="graphpool" javatype="GraphPool" variable="graphpool"/>
<stereotype name="graphpoolfactory" javatype="GraphPoolFactory"
    variable="graphpoolfactory"/>
<!-- ... -->
<stereotype name="attributevalue" javatype="AttributeValue" variable="attributevalue"
    superclass="metaattributable"/>
</lookuptable>
```

By using this lookup table, the corresponding Gras/GXL class can be found for a class of the specialized graph model. This class is needed to define an attribute for the class of a specialized graph model. This attribute is needed to call methods of Gras/GXL. When using these methods, which are passed by the tags, a reference is needed to a Gras/GXL class. The attribute defined by the stereotype is such a reference.

6.5.2 Associations

Attributes of a class can also be defined by interpreting the associations. The `association` variable is an element in the XML file that is a child of the `class` element. In the example code that is shown below, is the class `ProgresNode` defined, with the associations.

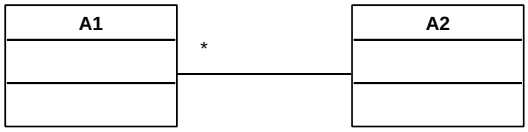
```
<class name="ProgresNode" id="Im12631b8m10243df4695mm7e08" stereotype="node"
    abstract="false">
     <association name="source" id="Im12631b8m10243df4695mm7dfe"
          targetClass="Im12631b8m10243df4695mm7e06" cardinality="0..*"
           otherCardinality ="1..1" collectionType ="set" aggregationType="none"
          otherAggregationType="none" navigation="true" otherNavigation="true"
          otherEnd="Im12631b8m10243df4695mm7df8"/><!--ProgresEdge-->
     <association name="target" id="Im12631b8m10243df4695mm7dec"
          targetClass="Im12631b8m10243df4695mm7e06" cardinality="0..*"
           otherCardinality ="1..1" collectionType ="set" aggregationType="none"
          otherAggregationType="none" navigation="true" otherNavigation="true"
          otherEnd="Im12631b8m10243df4695mm7de6"/><!--ProgresEdge-->
     <association name="" id="Im12631b8m10243df4695mm7d92"
          targetClass="Im12631b8m10243df4695mm7d26" cardinality="1..1"
           otherCardinality ="0..*" aggregationType="none" otherAggregationType="none"
          navigation="false" otherNavigation="false"/><!--ProgresNodeType-->
     <association name="" id="Im12631b8m10243df4695mm7b4c"
          targetClass="Im12631b8m10243df4695mm7b52" cardinality="1..1"
           otherCardinality ="0..*" aggregationType="none" otherAggregationType="composite"
          navigation="true" otherNavigation="true"
          otherEnd="Im12631b8m10243df4695mm7b46"/><!--ProgresGraph-->
<!-- ... -->
</class>
```

There are four associations defined. In comments, the target classes of the associations are named. The first association is the source association between `ProgresNode` and `ProgresEdge`. The association has a unique identifier. The target class is also defined by its identifier. The target class in this case is the `ProgresEdge` class. In addition, a number of properties of the association are defined. By the prefix "other", the values of the other association end are defined. If this information was not available, the data had to be looked up in the definition of the target class. For the `source` association, the cardinality is `0..*`, this means that the `ProgresNode` can be the source of zero or more `ProgresEdges`, the other way around, the source of the `ProgresEdge` can be exactly one `ProgresNode`.

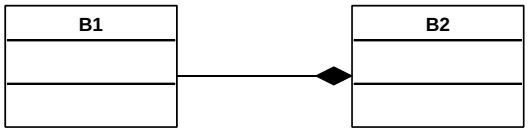
Which association will be part of an attribute? The if-statement defines for which associations attributes can be outlined. The associations that will be responsible for the creation of an attribute in a class are depicted in Figure 6.3. The if-statement exists of three parts, for every part there is a picture in the figure. The first part (Figure 6.3(a)) defines that if the `cardinality` is `0..*` (this is if there is a * depicted on the side of the current class), then the current class should get an attribute that is an instance of the class on the other association end. In the XML code, this would mean that the first association defines, that the `ProgresNode` class receives an attribute of the `ProgresEdge` class. The next part (Figure 6.3(b)) of the if-statement says that the association end should be the receiving end of a composition (`otherAggregationType="composite"`). The last association in the XML part, defines that there is also an attribute of the `ProgresGraph`. Figure 6.3(c) requires that no navigation is possible and the cardinality of the association is one-on-one. If the association

properties equal `cardinality="1..1" otherCardinality="1..1" navigation="false" otherNavigation="false"`, then the current class gets an attribute of the target class of this association.

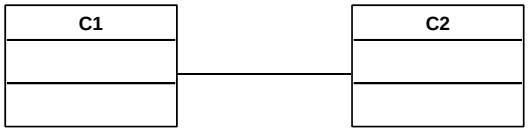
```
if (
( association . getAttributeValue (" otherCardinality ") . equals (" 0..* ")) // Figure 6.3( a )
|| ( association . getAttributeValue ("otherAggregationType") . equals ("composite"))
// Figure 6.3( b )
|| ( association . getAttributeValue ("navigation") . equals (" false ")
      && association . getAttributeValue ("otherNavigation") . equals (" false ")
      && association . getAttributeValue (" cardinality ") . equals (" 1..1 ")
      && association . getAttributeValue (" otherCardinality ") . equals (" 1..1 ")))
// Figure 6.3( c )
){
      // the current class gets an attribute
      // of the target class
}
```



(a) Cardinality



(b) Composition



(c) One-on-one relation

Figure 6.3: Associations

6.5.3 Attributes of the UML Classes

More attributes of the Java class are defined by the attributes of the corresponding class in the UML class diagram. These attributes are copied with name and type into the Java class. On the attributes in the Java class, methods are defined. In particular, the `get` and `set` methods are defined. The `get` methods are defined standard, the `set` methods are only defined if the changeability property has the value `changeable`.

6.5.4 Inheritance

If there is an inheritance relationship in the UML class diagram, this has consequences on the output files. In Figure 6.4, an inheritance relationship is depicted. This relationship defines, that the subclass (D2) can use the methods of its superclass (D1).

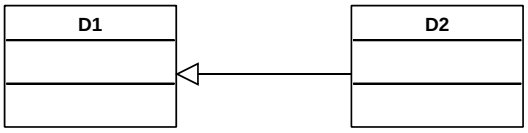


Figure 6.4: Inheritance Relationship

When a class is a subclass of another class, this means often that it is a class of the same kind but more specific. For instance, the class "mountain bike" is a subclass of the class "bicycle". The subclass is more specific then its superclass. Applied to this project, it means that the subclass also needs the specification of the superclass. This is done by calling the constructor of the superclass in the subclass. By means of a 'super-statement', this constructor is called: `super(p1,p2)` where `p1` and `p2` are the parameters of the constructor of the superclass. If the superclass has more than two parameters, the super method will have an equal number of parameters. The parameters of the superclass have to be available on the level of the subclass. Therefore, it has to be checked if these parameters are contained in the subclass, else they have to be added. For example, the `ProgresNodeClass` is a subclass of the `ProgresNodeTypeClass`. In the next code piece, it is defined how the super-statement is applied.

```
ProgresNodeClass(NodeClass nodeclass , ProgresSchema progresschema, String name) {
super( nodeclass ,name);
this .progresschema = progresschema;
}
```

The constructor of the `ProgresNodeTypeClass` looks as follows:

```
ProgresNodeTypeClass(NodeClass nodeclass, String name) {
this . nodeclass = nodeclass ;
this .name = name;
}
```

It can be perceived that the `ProgresNodeClass` calls the constructor of the `ProgresNodeTypeClass` by means of the super-statement, and with the parameters that are necessary in the constructor.

For the subclass, it also has to be defined that it is a subclass of another class. This can be done by specifying in the Java class that the class 'extends' another class. This is done by adding to the class name the keyword `extends` followed by the name of the superclass.

6.5.5 Adding the Constructor to the Java Files

Of course, after defining the attributes, first the constructor of the Java file has to be created. The name of the constructor must be the same name as the corresponding class (and the file). For every attribute, a private field is created and the public `get` and `set` methods are written in the Java file.

The definition of the attributes is quite straightforward. Nevertheless, now the real methods, which are defined in the UML diagram through tags, have to be interpreted and written in the Java files.

6.5.6 Operations

In the UML class diagram, every operation is defined. The name of the method is given, including the return type, the parameters and the exceptions that can rise for this operation. These data are stored in the `XMIOperation` class. By means of this class, for every operation its specific properties can be loaded. Before it is significant to interpret the tags, the operation has to be announced by a header. This header contains if the method is `public` or `private`, what the return type is, the method name, and the parameters between parentheses. Also, the exceptions that can be thrown are given. These exceptions are already included in the UML class diagram. For example, for the operation `createNode` the method looks like: `public ProgresNode createNode(ProgresNodeType type) throws GrasGXLException, EntityNotFoundException, SchemaCheckException`. In the next section, it is declared how the tags are interpreted.

6.6 Interpreting the Tags

The tags that are defined in Section 6.1.1 need to be interpreted. Every tag has some specific properties and the parameters not always denote the same information, which is shown in the table in Section 6.1.1. Sometimes the first parameter is a method, in other cases it is an output type, etcetera. The parameters of the tags are, per tag, denoted in a map as a key-value pair. This makes it more efficient to recover a parameter in the course of the program. The tags themselves are defined using an enumeration. With an `enum`, Java provides the facility to create enumerated types with fields and methods. For every tag, some methods should be added, that are defined in the evaluator. Hence, the `enum` type would be a comfortable solution, because it enables to assign every evaluator to a tag. By adding the tags to an enumeration, the names of the tags are fixed. The tag names thus have to be written exactly as is specified in the enumeration. This means that the tag names are all lowercased.

6.6.1 The AbstractEvaluateTag Class

An `AbstractEvaluateTag` class is defined, to support the different evaluators of the tags. For every tag, a separate evaluator that extends the abstract class is realized. The `AbstractEvaluateTag` class has two methods: the `evaluate` method and the `isResponsible` method. The specific evaluators redefine these methods. The `isResponsible` method returns true if the evaluator is responsible for the tag. The `evaluate` method has three parameters: the operation for which it is defined, the parameter map, and the previous tag. The operation defines all the information concerning the operation: return type, parameters, etcetera. It is useful to know which parameters the method possesses, in order to create the method. By the parameter map, the parameters defined in the tag are available for the evaluator. Moreover, the previous tag is needed when a translation has to be done, in order to know if the instance that should be transformed is new or already extant. If it is new, then the previous tag will be a *declare* or *create* tag. Also, a new instance of an entity of the specialized graph model has to be created, using the former instance. Then the

conversed instance has to be stored in the factory. If the instance is not new, then it can be retrieved of the factory.

6.6.2 The *unpack* Tag

`EvaluateUnpackTag` is the class in SUMAGRAM for interpreting the *unpack* tag. The *unpack* tag is defined in order to return the corresponding Gras/GXL class for a parameter. The *unpack* tags are often included in another tag, where an instance of the superclass is needed. The tag is extracted from the other tag. The unpacked class is assigned to a variable. On the place of the tag, this variable is included. In the next table, two examples are given. For each example, the top line defines the method name, the return type and its parameters. The second line gives the input (the tag with parameters) and the output.

Input	**Output**
public Collection getAllEdgesOfGraph(ProgresEdgeType type)	
@retrieve(Collection<Edge>, getAllEdgesOfGraph(@unpack(type),false))	EdgeClass edgeclasstype = type.getBackingEdgeClass(); Collection<? extends Edge> collection = graph.getAllEdgesOfGraph(edgeclasstype,false);
public ProgresNode createNode(ProgresNodeType type)	
@create(Node, createNode(@unpack(type)))	NodeClass nodeclasstype = type.getBackingNodeClass(); Node node = graph.createNode(nodeclasstype);

The first example is a method where all the edges of a graph have to be found, with a certain type. This type is an instance of `ProgresEdgeType`. The method `getAllEdgesOfGraph` requires as input an `EdgeClass` and a boolean (`false`) that indicates if also the subclasses should be included in the output. The second example defines the creation of a node. The method `createNode` of Gras/GXL expects a `NodeClass`, as there is no `NodeClass` available on the level of this method, it has to be retrieved. The method provides a `ProgresNodeType`, which has as a superclass the `NodeClass`.

Every class provides a `getBacking`-method, which returns the instance of its corresponding Gras/GXL class. First, the parameter belonging to the variable in the *unpack* tag has to be found, such that the type is clear. Next, the class, belonging to the type, has to be looked up. With the definition of this class, also the corresponding Gras/GXL class can be selected. The instance of that class is stored in a variable. The type is the Gras/GXL class and the variable is the Gras/GXL class lowercased and concatenated to the parameter name. This concatenation is done in order to avoid duplicate variable names. When only using the class name lowercased, the risk exists that his name already appears in the class. By creating such a concatenation, this risk is declined. Sometimes this concatenation might deliver a weird name (for instance `graphgraph`), but this is not as big a problem as having duplicate names. To the created variable, the instance is assigned. This instance is retrieved by applying the `getBacking`-method to the stereotype class.

6.6.3 The *retrieve* Tag

The class `EvaluateRetrieveTag` is responsible for the actions surrounding the *retrieve* tag. The idea of a *retrieve* tag is to gather an entity or a collection, by using a method of Gras/GXL. A few examples are listed in the next table. The input is a tag as explained previously, the output is the code as expected in the Java file.

Input	**Output**
public Collection getAllEdgesOfGraph(ProgresEdgeType type)	
@retrieve(Collection<Edge>, getAllEdgesOfGraph(@unpack(type),false))	Collection<? extends Edge> collection = graph.getAllEdgesOfGraph(edgeclasstype,false);
public ProgresGraph getGraph(String name)	
@retrieve(Graph, getTopLevelGraphByName(name))	Graph graph = graphpool.getTopLevelGraphByName(name);

The first example is a method to retrieve all the edges of a graph that have a certain edge type. The first parameter of the tag defines the type of the output. The output in this case is a `Collection`. The variable name is the lowercased output type. As in Gras/GXL, the methods use the type `Collection<? extends Edge>`, the type as denoted in the parameter is rewritten. `Collection<? extends Edge>` means that every element in the collection should be an element of which the type extends an `Edge`. Version 5.0 of Java recommends that for a `Collection` it is specified what the content is. The definition of the content is defined within the angle brackets. The content definition `Collection<? extends Edge>` is more unrestrained than using only `Edge` between the brackets, in the sense that every element that is a kind of `Edge` can be stored in the collection. In order not to bother the user with this extra work, the program will rewrite the `Collection<Edge>` to a `Collection<? extends Edge>`. The easiest would probably be not defining the type, but then there can be added elements to the collection of a wrong type, which can deliver problems when using the contents of the collection.

The specified method is a method of the `Graph` class of Gras/GXL. Therefore, an instance of this class is needed to call this method. This instance is the attribute `graph`, which can be grasped by means of the stereotype. The parameters of the method are given within the second parameter of the tag. As first parameter of the method, an instance of type `EdgeClass` is expected. No instance of type `EdgeClass` is available, only an instance of type `ProgresEdgeType` is vacant. This `ProgresEdgeType` has the `EdgeClass` as corresponding Gras/GXL class. The instance of the corresponding class that is included in the `ProgresEdgeType` class, can be used as parameter for the method. To retrieve this instance, the *unpack* tag is used.

In the second example, a method to retrieve a graph with a specified name is described. The method that is provided for this purpose is the method `getTopLevelGraphByName`. As output, an instance of type `Graph` is expected. This method is provided by the class `GraphPool` and uses a `String name` as parameter, which is the name of the graph that should be collected, in the format of a `String`.

By interpreting this tag, first there has to be defined if the output type concerns a `Collection`, a `Map` or an instance of an entity of Gras/GXL. This check is done using a pattern matcher. There has to be kept in mind, that a `Collection` is the superclass of types like `List` and `Vector`. A

`Collection` is interpreted as some text, followed by an opening angle bracket (<) then again some text and closed by a closing angle bracket (>). A `Map` has two strings separated by a comma between the angle brackets. The patterns also take into account that there may be spaces included. If the output type does not correspond to one of these two patterns, then it must be an entity of Gras/GXL.

For each selection, a different road is treaded. In the case of a `Collection`, the different parts of the output are concatenated. What is needed in the output is an output type, followed by a variable, then an equal sign, followed by the instance on which the Gras/GXL method can be run followed by a point and the method like defined in the parameter of the tag, with the suitable method parameters, and of course once in a while some spaces are added. The output type is defined as the part before the opening angle bracket in the pattern, which will usually be `Collection`, followed by the angle brackets and the contents, which is rewritten using `"? extends"`. Subsequent a space and the variable are added. The variable is the lowercased type, left of the angle brackets. After the equal sign, the instance of the corresponding Gras/GXL class is inserted. This is looked up in the lookup table. Thereafter, the method is written.

In case the output type is a `Map`, the output is almost the same, except for the code between the angle brackets, which will be a key-value-pair. For the last possibility, when the output is an instance of a Gras/GXL entity, the output type is literally the first parameter of the tag. The rest of the code is the same as for the `Collection` and the `Map`.

6.6.4 The *translateEntity* Tag

The target of the *translateEntity* tag is to translate an instance of an entity of Gras/GXL to an instance of an entity of a specialized graph model. The responsible class in the code of SUMAGRAM, is the `EvaluateTranslateEntityTag` class. For the *translateEntity* tag, two cases can be distinguished. The first one is the translation of an instance in the case that the previous tag was a *declare* or *create* tag. The second one includes all other cases. The difference lies in the use of the factory. When creating or declaring an instance, it has to be added to the factory, in the other cases, the instance can be loaded of the factory. The two cases are both exemplified in the next table.

Input	**Output**
public ProgresNode createNode(ProgresNodeType type)	
@translateEntity(ProgresNode,Node)	ProgresNode progresnode = new ProgresNode(node,type,this); ProgresSchemaFactory.addProgresNode(progresnode); return progresnode;
public ProgresGraph getGraph(String name)	
@translateEntity(ProgresGraph,Graph)	return ProgresSchemaFactory.getProgresGraph(graph);

The first example is included in the method `createNode`. The node is created by means of a *create* tag. This means that after the translation of the created Gras/GXL node, it has to be stored in the factory. The second example represents the translation of a Gras/GXL graph, after the retrieval for which a Gras/GXL method is used. The first parameter of a *translateEntity* tag is the output type, the second parameter states the input type. The input type should always be equal to the output type of the tag that precedes the *translateEntity* tag.

By the program, this tag is interpreted by first checking the situation. When there was a creation or declaration, the first thing to do is to create a new instance of the output type. This is done by writing down the type and the name of the corresponding variable, which is the type lowercased. This is followed by an equal sign. Then the keyword `new` and again the output type are inserted. Between parentheses, the parameters needed by the constructor of the output type are defined. This is done in the following way. First, the class of the output type is searched in the set of all classes. Here, the parameters are defined in the right order. Then there has to be evaluated which variable corresponds with the defined parameter. If the type of the parameter corresponds to the input type on the tag level, then the variable, corresponding to the input type, is used. In the second case, it is checked if the parameter type equals the current class. Then the keyword `this` will be included. In the last case, the parameters that belong to the operation are evaluated. It is checked if there is a parameter that corresponds with name and type. If this cannot be found, there is checked if there is a parameter with the right type. If no corresponding parameter is found, the initial value of the class parameter is used. It is the responsibility of the user, that there can be found a parameter. How the user can accomplish this, will be explained in the Section 6.7. After creating this instance, it should be added to the factory. Next, it will be returned.

When the previous tag was no creation or declaration, the instance is simply looked up in the factory and returned. The parameter of the `get` method of the factory, is the instance that is created previously, and that should be translated.

6.6.5 The *translateCollection* Tag

The `EvaluateTranslateCollectionTag` class is responsible for interpreting the *translateCollection* tag. This tag converts every instance in a set to instances of another class. The first parameter of the tag denotes the output type of a single instance, the second parameter denotes the type of the set.

Input	**Output**
public Collection getAllNodesOfGraph()	

@translateCollection(ProgresNode, Collection<Node>)	Collection output = new HashSet();
	for (Node x : collection){
	output.add(ProgresSchemaFactory.getProgresNode(x));
	}
	return output;

public Map getAllValidAttributeValues()	

@translateCollection(ProgresAttribute, Map<Attribute,Serializable>)	Map output = new HashMap();
	for (Attribute x : map.keySet()){
	output.put(
	ProgresSchemaFactory.getProgresAttribute(x),
	map.get(x));
	}
	return output;

In the table above, two examples of the *translateCollection* tag and the outputted code are given. The first tag is a tag included in the method `getAllNodesOfGraph`. After retrieving a collection

of nodes using a method of Gras/GXL, every `Node` in this collection has to be converted to a `ProgresNode`. This is done by looping through the set of nodes, and, for every `Node`, retrieve the corresponding `ProgresNode` out of the factory.

Again, a difference has to be made between the actions when a `Map` is encountered and the actions when there is a `Collection`. This is again done by a pattern. When the output type is a `Map`, a new instance of a `HashMap` will be created. If the output type is a `Collection`, an instance of a `HashSet` will be created. Both Java classes have a different behaviour and other methods. To add new elements to a `HashMap`, the method `put` exists, which consists of two arguments, a key and a value. For a `HashSet`, the method for adding elements is called `add`. This difference has to be regarded.

When implementing the method in the case of the `Map`, first the instance of the `HashMap` has to be created. Then a loop through all the key-value pairs of the map of which the elements should be converted, is implemented. This loop is done by means of the `keySet` method of the `HashMap`. This method returns a set of all keys that appear in the `HashMap`. These keys are all of a type of Gras/GXL. These keys all have to be converted, the corresponding value for every key remains unchanged. Within the loop, the newly created `HashMap` is filled. First, the keys are converted to entities of the specialized graph model. This is done by finding the converted entities in the factory. Then the value that was part of the original key is attached to the new key. This key-value pair is put into the `HashMap`. The created map is returned to the program.

The implementation of the `Collection` is less complicated. The first step is to create the instance of the `HashSet`. In this case, also a loop is created to the existing collection, of which the elements have to be converted. For every element in the collection, the corresponding element is searched in the factory and added to the new `HashSet`. The filled set is then returned to the program.

6.6.6 The *create* Tag

By means of the `EvaluateCreateTag` class, the desired output of the *create* tag is created. With the *create* tag, the establishing of new entities can be done.

Input	**Output**
public ProgresNode createNode(ProgresNodeType type)	
@create(Node, createNode(@unpack(type)))	Node node = graph.createNode(nodeclasstype);
public ProgresGraph createGraph(String name)	
@create(Graph, createTopLevelGraph(name, #ProgresSchema.getStandardGraphClass()))	Graph graph = graphpool.createTopLevelGraph(name,progresschema.getStandardGraphClass());

In the table above, two examples are given for the *create* tag. The first example is the creation of a `Node`; the second example is the creation of a `Graph`. The idea is that afterwards, these entities are transformed to respectively a `ProgresNode` and a `ProgresGraph`.

The output is composed by first defining the output type, which is one of the parameters of the tag. The variable is again the lowercased output type. To this variable, the output value of the method,

defined in the second parameter of the creation tag, is assigned. This method had to be applied to the instance corresponding with the stereotype. The variable name of the instance is retrieved from the lookup table.

6.6.7 The *declare* Tag

For the *declare* tag, the responsibility rests on the `EvaluateDeclareTag` class. This tag provides the possibility to declare new schema entities, like `NodeClass`, `EdgeClass`, and `GraphClass`.

Input	**Output**
public ProgresNodeType declareNodeType(String name, ProgresNodeClass typeOf)	
@declare(NodeClass, declareNodeClass(name,false))	NodeClass nodeclass = schema.declareNodeClass(name,false);
public ProgresEdgeType declareEdgeType(String name, ProgresNodeTypeClass source, Cardinality sourceCardinality, ProgresNodeTypeClass target, Cardinality targetCardinality)	
@declare(EdgeClass, declareDirectedEdgeClass(name, false, (GraphEntityClass) source, sourceCardinality, (GraphEntityClass) target, targetCardinality))	EdgeClass edgeclass = schema.declareDirectedEdgeClass(name, false, (GraphEntityClass) source, sourceCardinality, (GraphEntityClass) target, targetCardinality);

In the first example, a new `NodeClass` is defined. The method for this declaration needs two parameters, the first parameter is a name and the second one is a boolean value. This boolean value defines whether the class is an abstract class or not. The boolean value is in this case `false`, because this tag is part of the declaration of a new `ProgresNodeType`, which is a concrete class. The second example creates a new `EdgeClass`, which is created for the purpose of a new `ProgresEdgeType`. The first parameter of the method is the name of the new `ProgresEdgeType`, then a boolean is defined that indicates whether it concerns an abstract class or not. The next parameters define for the source and target node the classes and cardinalities. As in Gras/GXL, an `Edge` is defined between two `GraphEntityClasses` and for the PROGRES graph model, these classes do not exist, a conversion has to be done by using a typecast.

The output type is the first part of the output, then the variable is defined as the lowercased output type. After the variable, an equal sign follows to which the method succeeds. This method is applied to the instance belonging to the stereotype. The method itself with its parameters is literally copied.

6.6.8 The *check* Tag

The *check* tag is handled by the `EvaluateCheckTag` class. As this tag only appears seldom, only one example is given. This tag appears within the `createEdge` method.

Input	Output
public ProgresEdge createEdge(ProgresNode source,ProgresNode target,ProgresEdgeType type)	

@check(isSourceValid(source, type),
throw new SchemaCheckException("The type
is not valid for the source."))
	if(!(progresgraphpool.getProgresSchema()
	.isSourceValid(source, type))){
	throw new SchemaCheckException(
	"The type is not valid for the source.");

With the *check* tag, there can be asserted that no inconsistencies on behalf of the schema exist when creating a new entity. For instance, when creating a new `Edge`, it can be checked that the nodes are valid. This means that the node has to be of the `ProgresNodeType` that is specified for the `ProgresEdgeType`. With the *check* tag, a method is called that returns a boolean value. This boolean value reports if the `Node` is valid with regard to the `Edge`.

The check method is composed by creating an if-statement, which is defined by the first parameter of the tag. This method is defined in the `ProgresSchema` class. Therefore, an instance of this class has to be found. This method is defined in the `ProgresGraph` class. This class can operate on an instance of the `ProgresGraphPool` class, which is related to the `ProgresSchema` class. These classes can be found by looping through the set of classes and finding the classes with respectively the stereotypes `graphpool` and `schema`. The defined reaction should be given when the boolean value returned by the method is `false`. This implies the negation in the if-statement. Within the if-statement, the reaction as defined in the second parameter of the tag is returned.

6.6.9 Literal Code

The method that is called using the *check* tag, is a method that is defined on the level of the `Schema` class of the specialized graph model. The method should actually be defined using an OCL constraint, but in the scope of the project it is defined as literal code in the `ProgresSchema` class. The `isSourceValid` method is defined as follows:

```
public boolean isSourceValid (ProgresNode source, ProgresEdgeType type) throws
    GrasGXLException, EntityNotFoundException {
      if (! source.getProgresNodeType().getBackingNodeClass().isSubClassOf(
      type.getSource().getBackingNodeClass())) {
            return false;
      }
      if (!( type. getSourceCardinality ().getMinimum() < source.getAllOutgoingEdges().size()+1
           && type. getSourceCardinality ().getMaximum() > source.getAllOutgoingEdges().size()+1)
           || (type. getSourceCardinality ().getMinimum() == -1 &&
           type.getSourceCardinality ().getMaximum() == -1)) {
            return false;
      }
      return true;
}
```

If this method returns the value `true`, then the source is valid. This method is written literally in the UML class diagram. However, it is actually interpreted as if it was also created by means of tags. On the level of the `createEdge` method, it is not known that this method is not generated. This is an example of how literal and generated code can be used simultaneously without any problems.

6.6.10 The *write* Tag

The *write* tag is handled by the `EvaluateWriteTag` class. The first example given in the next table defines a method of the `ProgresGraphPool` class, which states if there exists a graph with the defined name. The second example is taken out of the `ProgresNode` class. It is used to define an attribute value for a specific node. The *write* tag writes a method into a Java class, taking into account what the return type is and that it has to be applied on a class of the Gras/GXL graph model.

Input	**Output**
public boolean existsGraph(String name)	
@write(existsTopLevelGraph(name))	return graphpool.existsTopLevelGraph(name)
public void putAttributeValue(ProgresAttribute attribute, Serializable value)	
@write(putAttributeValue(@unpack(attribute), value))	node.putAttributeValue(attributeattribute, value);)

The method belonging to the first example has a boolean value as return type. When the program perceives this, it returns the value defined by the method. The second method has `void` as return type, so nothing will be returned by the method.

The output is composed by first checking the return type, everything different from `void` causes the program to write "return" before the real method is written. Then the variable belonging to the stereotype is written followed by a point and the method as defined in the first parameter of the tag. The method may contain an *unpack* tag, which will then be interpreted previously.

6.7 Problems with the Implementation

As with all application developments, also with the development of SUMAGRAM, some problems have arisen. One of the greater issues encountered during the implementation of SUMAGRAM was handling strings. As it is already a known problem in compiler construction, the programmer can define the text structure arbitrarily. In SUMAGRAM, the created source code is copied literally. The parameters of the tags have to be interpreted. They have a quite simple structure, but it still has to be analyzed appropriately. The parameters are actually strings. A string can consist of different characters, like letters, digits, spaces, and also parentheses and commas. The style of every programmer might differ in the spacing before and after parameters. Because these spaces might influence the output of SUMAGRAM, the strings are all trimmed after loading them. With the method `trim` provided by Java, the leading and trailing spaces of a string are removed. Another aspect on behalf of the strings is reading the tags and its parameters correctly. The tag `@retrieve(Collection<Edge>, getEdges(ConnectionMode.DIRECTED_OUTGOING, @unpack(edgeType), false))` has to be read correctly. The first question is: which parameters belong to the tag, and which to the method? It is possible to use the commas as separators, but this might be faulty because the parameters of the methods are also separated by commas. Therefore, it has to be checked how many brackets occur, and by that, the parameters can be selected. The next method shows how the parameters can be selected out of a tag. The `params` parameter of the method contains the parameters of the tag. The `operation` parameter defines the operation to which the tag belongs.

```
private String getVariable(String params, XMIOperation operation) throws GMGenException {
    int brackets = 0;
    int typer = 0;

    for (int i=0; i<params.length(); ++i) {
        if (params.charAt(i) == ',' && brackets == 0 && typer == 0) {
            return params.substring(0, i);
        }

        if (params.charAt(i) == '<')
            ++typer;
        if (params.charAt(i) == '>') {
            if (typer > 0)
                --typer;
            else
                throw new GMGenException("The number of brackets is incorrect for
                    operation " + operation.getName());
        }

        if (params.charAt(i) == '(')
            ++brackets;
        if (params.charAt(i) == ')') {
            if (brackets > 0)
                --brackets;
            else
                throw new GMGenException("The number of brackets is incorrect for
                    operation " + operation.getName());
        }
    }
    if (brackets != 0 || typer != 0) {
        throw new GMGenException("The number of brackets is incorrect for operation " +
            operation.getName());
    }
    return params;
}
```

Another big issue, was to keep SUMAGRAM independent of the PROGRES example. In the program, it may not be defined that `ProgresNodeTypes` exist, simply because they do not exist outside of PROGRES. This was something that needed some attention. This problem was also a reason to redevelop the tags. In the lookup table, also a stereotype `NodeType` was defined. Just by cause of inattention, every class was entered, leaving out the Progres prefix. When this mistake was rectified, it was not clear how to write the variables exactly, because they used to be defined in the lookup table.

Retrieving the parameters based on the associations delivered also some difficulties. The self-loop association of the `ProgresNodeClass` brought about a parameter of the type `ProgresNodeClass` in the class `ProgresNodeClass`. By means of the `NodeClass` of Gras/GXL, the related class can be defined which can be of type `ProgresNodeClass`. The extra parameter is thus not necessary and should not be added.

When there are two attributes in a class with the same type, it has to be watched that the names of the `get` and `set` methods differ. For instance for the class `ProgresEdge`, two `ProgresNodes` exist.

An idea is to define the methods as `getProgresNode`, but this would mean that there are two such methods in one class (with the same parameter list), which is illegal. Therefore, it is checked if the same type occurs more than once in the attribute list. If this is the case, the attributes are called by the names instead of the types. This is not the usual procedure, because it is easier for the user to denote that he wants a `ProgresEdge` than that he wants a `progresedge`, because the first notation is the ordinary one. Only if there is defined a name of an attribute by the UML class diagram, this name is used in the method name. For example, to retrieve the source node, the method `getSource` will be used.

For the usability of the program, it is necessary that the names of the attributes are unambiguous. It is therefore advised to always use the same name for entities that have the same purpose. For instance, when a source node is desired, this node should at any time receive the same name. For example, for the `ProgresEdge`, in the attributes two `ProgresNodes` are defined: the `source` and `target`. Then it is important, for the `createEdge` method, to use the same names, otherwise it might be the case that they are swopped and the node that was intended to be the source is actually the target and vice versa. Because the program checks if there exists a parameter with the same name and type, this naming conventions should be followed.

Another point is to be careful with the naming. It is not important what the method names are for the specialized graph model, but it can easily occur that for instance the method `getAllEdgesofGraph` is created in the specialized graph model, but the corresponding method of Gras/GXL is the method `getAllEdgesOfGraph` (notice that it is written differently: "of" versus "Of"). When now the method name is written in the UML class diagram, it can happen that the wrong name is used as the Gras/GXL method, which will then throw an exception.

For the specialized graph model, it is not allowed to use names for the classes, which are equal to the names of Gras/GXL. This is for instance not sensible for the *translateEntity* tag. This would namely cause the next case: `@translateEntity(Graph,Graph)` which does not make a lot of sense and this would confuse the program. Therefore, it is obligatory to give names that are not equal to the Gras/GXL names.

In the PROGRES graph model, a method `getAllNodeTypesOfNodeClass` exist for the `ProgresNodeClass`. This method first retreives a collection with all the subclasses of a node class, and then it retrieves the node types for each node class in the set. This is a kind of recursive method. As there is no suitable combination of tags to implement this method, there is decided to write this method as literal code into the UML class diagram. Another possibility is to make a combination of tags and literal code, which can also be done.

```
public Collection getAllNodeTypesOfNodeClass() throws SchemaCheckException, GrasGXLException {
    Collection output = new HashSet();
    Collection <? extends GraphEntityClass> collection = getAllSubClassesOfNodeClass();
    output.addAll(getAllDirectNodeTypesOfNodeClass());
    for (GraphEntityClass x : collection ){
        output.addAll (((ProgresNodeClass)x).getAllDirectNodeTypesOfNodeClass());
    }
    return output;
}
```

Chapter 7

Examples of Specialized Graph Models

SUMAGRAM receives a specialized graph model as input. To develop SUMAGRAM, two examples of specialized graph models have been used. The first example is the PROGRES graph model, which has been used during the development. The GXL graph model is the second example. After the completion of the development, it has been used to check if there were any inconsistencies.

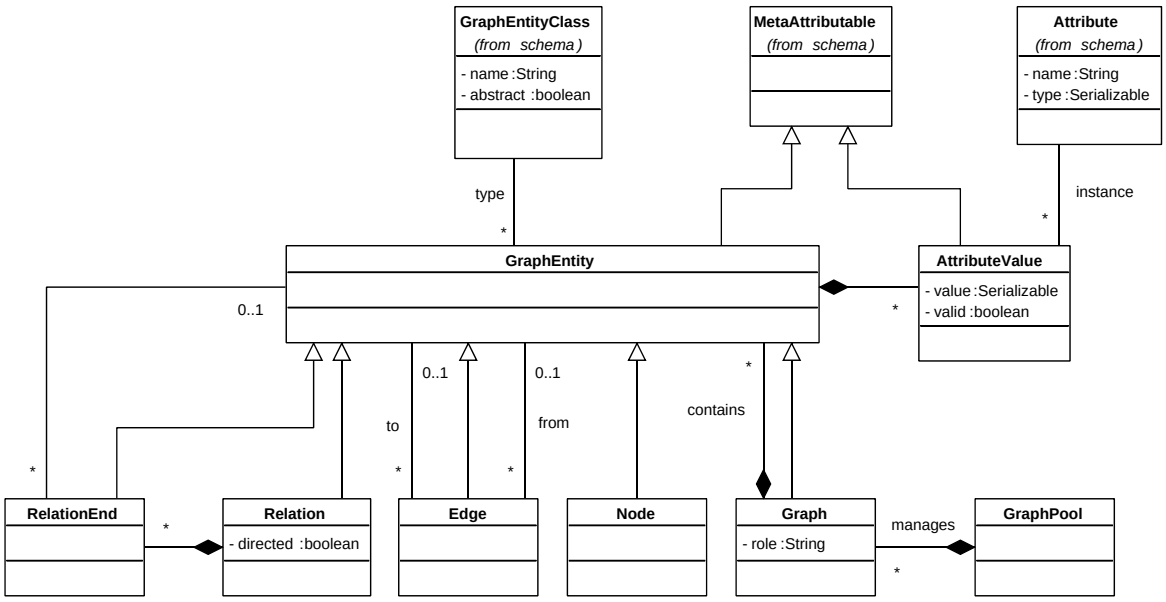


Figure 7.1: Gras/GXL Graph Model

A link between the Gras/GXL graph model and the specialized graph model always has to be established. It is not obligatory to denote a stereotype for every class of the specialized graph model, but if there are no stereotypes at all, the methods of Gras/GXL cannot be mapped to the specialized graph model. The Gras/GXL graph model and schema are again depicted in Figures 7.1 and 7.2, to denote the link with the specialized graph models.

In the first section, the PROGRES graph model is described and how it has been used as an example. Section 7.2 explains the example of the GXL graph model.

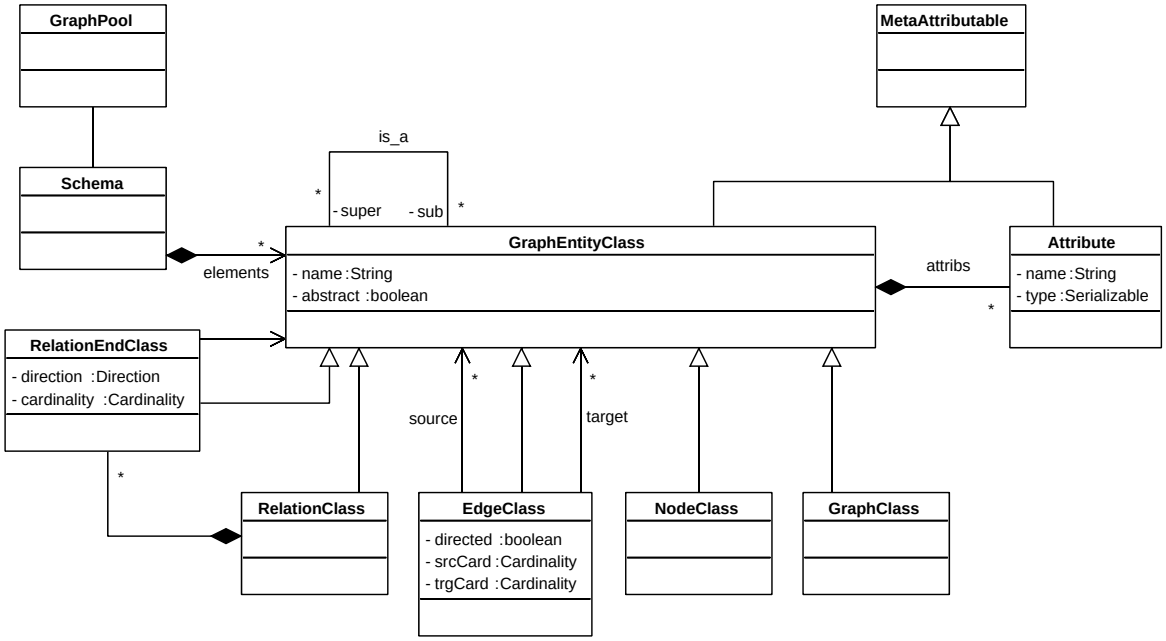


Figure 7.2: Gras/GXL Graph Schema

7.1 PROGRES Graph Model

The PROGRES graph model is the example, which is used to develop SUMAGRAM. The graph model has already been explained in Section 3.3. The PROGRES graph model is visualized in Figure 7.4.

First, for every class in the PROGRES graph model, a corresponding class in the Gras/GXL graph model has to be found. This class will then be used to denote the stereotype. This correspondence is perceived by means of properties. The class `ProgresEdge` is a class that connects two nodes with each other. The class `Edge` of Gras/GXL has the same characteristic. Therefore, it is logical to give the `ProgresEdge` the stereotype `edge`. As can be seen in Figure 7.4, every class has a stereotype. Denoting the stereotypes for the PROGRES graph model is straightforward, because every class can be linked to a class of Gras/GXL by examining the names and characteristics.

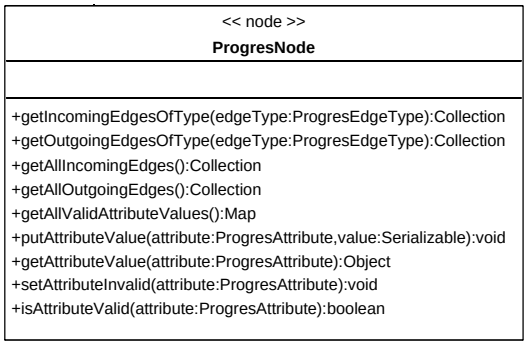


Figure 7.3: The ProgresNode Class

To exemplify the whole process, one class of the PROGRES graph model will be lifted out: the `ProgresNode` class. This class is interesting on behalf of the relationship to the rest of the graph model. In addition, it contains many methods, such that a good overview can be gained of how the tags are interpreted and written in the output file.

The `ProgresNode` class contains a number of different operations, as can be perceived in Figure 7.3. The XMI file belonging to the UML class diagram is converted to an XML file. The part of that XML file that is about the `ProgresNode` class, is displayed in Figures 7.5 and 7.6.

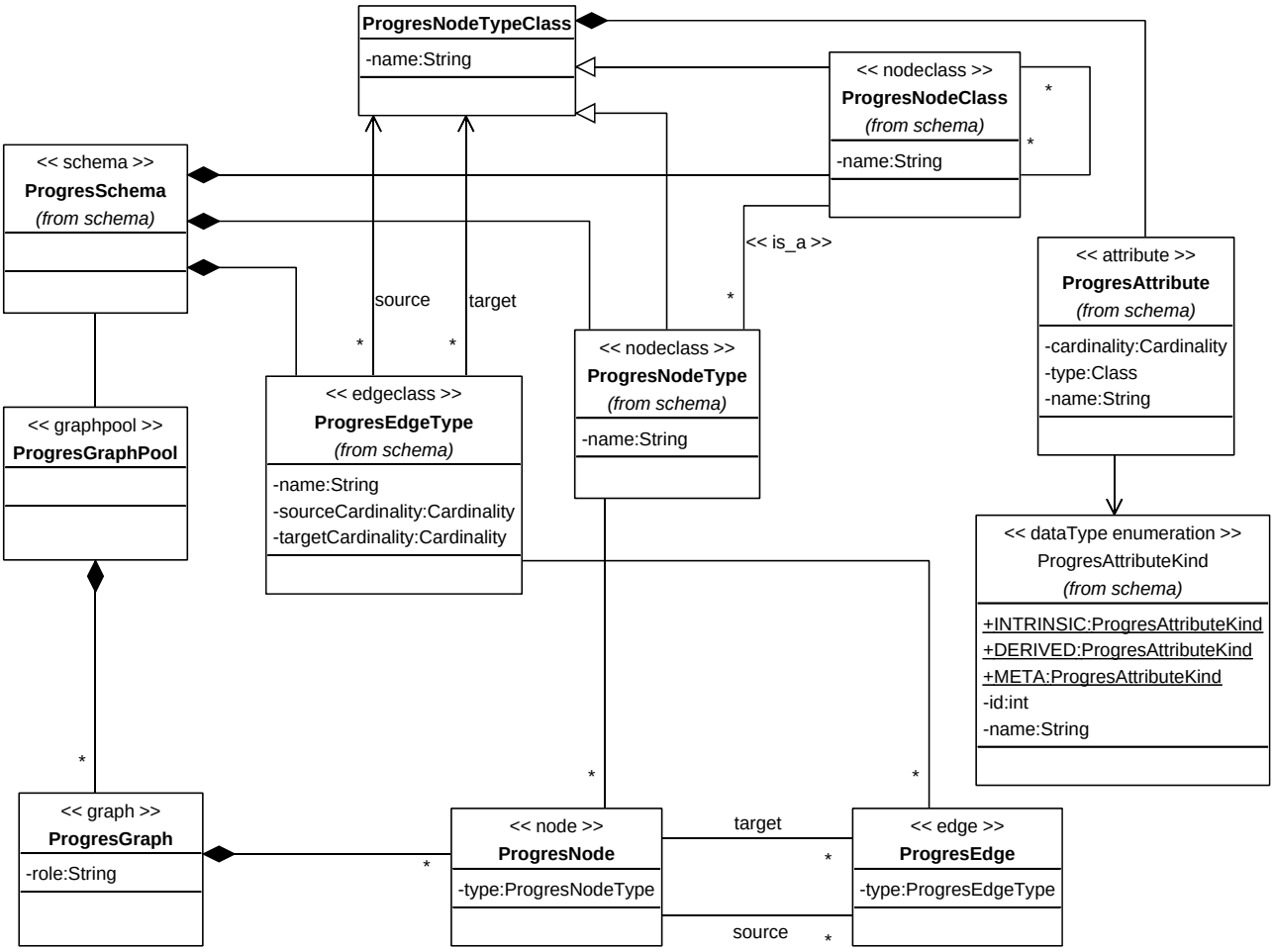


Figure 7.4: PROGRES Graph Model

Figure 7.5 shows the associations. The `ProgresNode` class has a number of associations. Two associations connect the `ProgresNode` class with the `ProgresEdge` class. These associations both have a multiplicity of zero or more. This association can be read as: a `ProgresNode` can be part of multiple `ProgresEdges`. This implies that the `ProgresEdge` class receives two attributes: `ProgresNode source` and `ProgresNode target`. The `ProgresNode` receives no attribute caused by these two associations. The next association links the `ProgresNode` class to the `ProgresNodeType` class. Here, also a multiplicity of zero or more exists. This association results in an attribute of type `ProgresNodeType`. The last association is the one with the `ProgresGraph` class. This relationship is a composition. This results in an attribute of type `ProgresGraph`.

Figure 7.6 shows the operations of the `ProgresNode` class. Every operation is defined by specifying a return type, a method body and the parameters. For every method, a tag is defined.

The Java file of the `ProgresNode` class is shown in Figure 7.7. The methods are not mentioned in the file, because they are handled later on step by step. In the file, first the `imports` are written. Then, after the class header, the fields are mentioned. The `ProgresNodeType` and `ProgresGraph` fields are created due to the associations. The `Node` field is created as a consequence of the stereotype of the `ProgresNode` class. This stereotype is `node`, which refers to the class `Node` of the Gras/GXL graph model. Then the constructor is created using the fields. For every field, a `get` method is created. As

```
<class name="ProgresNode" id="Im12631b8m10243df4695mm7e08" stereotype="node" abstract=" false ">
  <association name="source" id="Im12631b8m10243df4695mm7dfe"
      targetClass="Im12631b8m10243df4695mm7e06" cardinality="0..*" otherCardinality =" 1..1 "
       collectionType ="set" aggregationType="none" otherAggregationType="none"
      navigation="true" otherNavigation="true" otherEnd="Im12631b8m10243df4695mm7df8"/>
  <association name="target" id="Im12631b8m10243df4695mm7dec"
      targetClass="Im12631b8m10243df4695mm7e06" cardinality="0..*" otherCardinality =" 1..1 "
       collectionType ="set" aggregationType="none" otherAggregationType="none"
      navigation="true" otherNavigation="true" otherEnd="Im12631b8m10243df4695mm7de6"/>
  <association name="" id="Im12631b8m10243df4695mm7d92"
      targetClass="Im12631b8m10243df4695mm7d26" cardinality="1..1" otherCardinality =" 0..* "
      aggregationType="none" otherAggregationType="none" navigation=" false "
      otherNavigation=" false "/>
  <association name="" id="Im12631b8m10243df4695mm7b4c"
      targetClass="Im12631b8m10243df4695mm7b52" cardinality="1..1" otherCardinality =" 0..* "
      aggregationType="none" otherAggregationType="composite" navigation="true"
      otherNavigation="true" otherEnd="Im12631b8m10243df4695mm7b46"/>
<!-- ... -->
</class>
```

Figure 7.5: The Associations of the ProgresNode Class in XML Format

the `changeability` of the PROGRES graph model is set to `frozen`, no `set` methods are created. Also, the `getBackingNode` method that is used for the *unpack* tag is defined.

```
<class name="ProgresNode" id="Im12631b8m10243df4695mm7e08" stereotype="node" abstract=" false ">
<!-- ... -->
  <operation name="getIncomingEdgesOfType" visibility ="public ">
    <returntype name="Collection"/>
    <parameter name="edgeType" type="ProgresEdgeType" direction="in"/>
    <exception id="Im12631b8m10243df4695mm7db6" type="Exception" name="EntityNotFoundException"/>
    <exception id="Im12631b8m10243df4695mm7dae" type="Exception" name="GrasGXLException"/>
    <method name="          @retrieve ( Collection &lt;Edge&gt;, getEdges(ConnectionMode.DIRECTED_INCOMING,
        @unpack(edgeType), false),Node)&#xA;      @translateCollection (ProgresEdge, Collection &lt;Edge&gt;)&#xA;"/>
  </operation>
  <operation name="getOutgoingEdgesOfType" visibility ="public ">
    <returntype name="Collection"/>
    <parameter name="edgeType" type="ProgresEdgeType" direction="in"/>
    <exception id="Im12631b8m10243df4695mm7db6" type="Exception" name="EntityNotFoundException"/>
    <exception id="Im12631b8m10243df4695mm7dae" type="Exception" name="GrasGXLException"/>
    <method name="          @retrieve ( Collection &lt;Edge&gt;,getEdges(ConnectionMode.DIRECTED_OUTGOING,
        @unpack(edgeType), false),Node)&#xA;      @translateCollection (ProgresEdge, Collection &lt;Edge&gt;)&#xA;"/>
  </operation>
  <operation name="getAllIncomingEdges" visibility ="public ">
    <returntype name="Collection"/>
    <exception id="Im12631b8m10243df4695mm7dae" type="Exception" name="GrasGXLException"/>
    <method name="          @retrieve ( Collection &lt;Edge&gt;,getEdges(ConnectionMode.DIRECTED_INCOMING))&#xA;
        @translateCollection (ProgresEdge, Collection &lt;Edge&gt;)&#xA;"/>
  </operation>
  <operation name="getAllOutgoingEdges" visibility ="public ">
    <returntype name="Collection"/>
    <exception id="Im12631b8m10243df4695mm7dae" type="Exception" name="GrasGXLException"/>
    <method name="          @retrieve ( Collection &lt;Edge&gt;,getEdges(ConnectionMode.DIRECTED_OUTGOING))&#xA;
        @translateCollection (ProgresEdge, Collection &lt;Edge&gt;)&#xA;"/>
  </operation>
```

```
  <operation name=" getAllValidAttributeValues "  visibility ="public">
    <returntype name="Map"/>
    <exception id="Im12631b8m10243df4695mm7dae" type="Exception" name="GrasGXLException"/>
    <method name="        @retrieve (Map&lt;Attribute , Serializable &gt;, getAllValidAttributeValues )&#xA;
        @translateCollection ( ProgresAttribute ,Map&lt;Attribute , Serializable &gt;)&#xA;"/>
  </operation>
  <operation name="putAttributeValue"  visibility ="public">
    <returntype name="void"/>
    <parameter name="attribute "  type=" ProgresAttribute "  direction ="in"/>
    <parameter name="value" type=" Serializable "  direction ="in"/>
    <exception id="Im12631b8m10243df4695mm7db6" type="Exception" name="EntityNotFoundException"/>
    <exception id="Im12631b8m10243df4695mm7dae" type="Exception" name="GrasGXLException"/>
    <method name="        @write( putAttributeValue (@unpack( attribute ), value))&#xA;"/>
  </operation>
  <operation name="getAttributeValue"  visibility ="public">
    <returntype name="Object"/>
    <parameter name="attribute "  type=" ProgresAttribute "  direction ="in"/>
    <exception id="Im12631b8m10243df4695mm7db6" type="Exception" name="EntityNotFoundException"/>
    <exception id="Im12631b8m10243df4695mm7dae" type="Exception" name="GrasGXLException"/>
    <exception id="Im12631b8m10243df4695mm7db2" type="Exception" name="InvalidValueException"/>
    <method name="        @write( getAttributeValue (@unpack( attribute )))&#xA;"/>
  </operation>
  <operation name=" setAttributeInvalid "  visibility ="public">
    <returntype name="void"/>
    <parameter name="attribute "  type=" ProgresAttribute "  direction ="in"/>
    <exception id="Im12631b8m10243df4695mm7db6" type="Exception" name="EntityNotFoundException"/>
    <exception id="Im12631b8m10243df4695mm7dae" type="Exception" name="GrasGXLException"/>
    <method name="        @write( setAttributeInvalid ( attribute . getBackingAttribute ()))&#xA;"/>
  </operation>
  <operation name=" isAttributeValid "  visibility ="public">
    <returntype name="boolean"/>
    <parameter name="attribute "  type=" ProgresAttribute "  direction ="in"/>
    <exception id="Im12631b8m10243df4695mm7db6" type="Exception" name="EntityNotFoundException"/>
    <exception id="Im12631b8m10243df4695mm7dae" type="Exception" name="GrasGXLException"/>
    <method name="        @write( isAttributeValid ( attribute . getBackingAttribute ()))&#xA;"/>
  </operation>
</class>
```

Figure 7.6: The Operations of the ProgresNode Class in XML Format

The first method is `getIncomingEdgesOfType`. This method returns all the edges that come into the current node, of the specified `ProgresEdgeType`. The code of this method is defined in the UML class diagram using the following tags:

```
@retrieve(Collection<Edge>, getEdges(ConnectionMode.DIRECTED_INCOMING,
@unpack(edgeType), false))
@translateCollection(ProgresEdge, Collection<Edge>)
```

```
public Collection getIncomingEdgesOfType(ProgresEdgeType edgeType) throws
    EntityNotFoundException, GrasGXLException {
  EdgeClass edgeclassedgeType = edgeType.getBackingEdgeClass();
  Collection <? extends Edge> collection =
      node.getEdges(ConnectionMode.DIRECTED_INCOMING, edgeclassedgeType, false);
  Collection output = new HashSet();
  for (Edge x : collection ){
    output.add(ProgresSchemaFactory.getProgresEdge(x));
  }
  return output;
}
```

```
import java.util.*;
import java.io.*;
import i3.grasgxl.exception.*;
import i3.grasgxl.gm.core.*;
import i3.grasgxl.gm.core.schema.*;

public class ProgresNode{
    private Node node;
    private ProgresNodeType progresnodetype;
    private ProgresGraph progresgraph;

    ProgresNode(Node node, ProgresNodeType progresnodetype, ProgresGraph progresgraph) {
        this.node = node;
        this.progresnodetype = progresnodetype;
        this.progresgraph = progresgraph;
    }

    public Node getNode() {
        return node;
    }
    public ProgresNodeType getProgresNodeType() {
        return progresnodetype;
    }
    public ProgresGraph getProgresGraph() {
        return progresgraph;
    }
    public Node getBackingNode() {
        return node;
    }

// methods of the ProgresNode class
}
```

Figure 7.7: The Java File of the ProgresNode Class

Three tags are used: the *retrieve* tag, the *unpack* tag and the *translateCollection* tag. The *unpack* tag is lifted out by SUMAGRAM, because it will be handled before the *retrieve* tag. The definition of the variable `edgeclassedgeType` is the result of the interpretation of the *unpack* tag. The *retrieve* tag is interpreted next. The `collection` variable is created with the use of the *retrieve* tag. The next piece of code, starting with the assignment of the variable `output` until the `return` statement, is a result of the *translateCollection* tag.

The method `getOutgoingEdgesOfType` resembles the previous method strongly. The only difference is the first parameter of the Gras/GXL method, which is changed from `ConnectionMode.DIRECTED_INCOMING` into `ConnectionMode.DIRECTED_OUTGOING`.

The method `getAllIncomingEdges` is the following method that is defined. The target is to collect all the incoming edges of the current node, independent of their type. This method looks like the previous methods, with the difference that no *unpack* tag appears. The tags and the corresponding method are shown next. As the `getAllOutgoingEdges` method is identically except for the "outgoing" characteristic, it will not be described further.

```
@retrieve(Collection<Edge>,getEdges(ConnectionMode.DIRECTED_INCOMING))
@translateCollection(ProgresEdge,Collection<Edge>)
```

```
public Collection getAllIncomingEdges() throws GrasGXLException {
    Collection <? extends Edge> collection =
        node.getEdges(ConnectionMode.DIRECTED_INCOMING);
    Collection output = new HashSet();
    for (Edge x : collection ){
        output.add(ProgresSchemaFactory.getProgresEdge(x));
    }
    return output;
}
```

The method `getAllValidAttributeValues` should retrieve all the valid attribute values for the current node. The structure of this method looks like the previous ones, because there is also first a *retrieve* tag followed by a *translateCollection* tag. The difference for this method is the fact that the output type is not a `Collection` but a `Map`.

```
@retrieve(Map<Attribute,Serializable>,getAllValidAttributeValues())
@translateCollection(ProgresAttribute,Map<Attribute,Serializable>)
```

```
public Map getAllValidAttributeValues () throws GrasGXLException {
    Map<Attribute, Serializable > map = node. getAllValidAttributeValues () ;
    Map output = new HashMap();
    for ( Attribute x : map.keySet() ){
        output.put(ProgresSchemaFactory. getProgresAttribute (x),map.get(x));
    }
    return output;
}
```

The next methods are all methods that contain a *write* tag, with an *unpack* tag included. Therefore, only the first one is reviewed in detail. The first method is the `putAttributeValue` method. The other ones are the `getAttributeValue` method, the `setAttributeInvalid` method, and the `isAttributeValid` method.

The method `putAttributeValue` defines for a certain `ProgresAttribute` its value. This value is dependent on an `Attribute`, which implies that this `Attribute` needs to be unpacked, before denoting the value to it. The `putAttributeValue` method (that occurs in the code) is a method of Gras/GXL. This method is applied to a `Node` of Gras/GXL. This node is retrieved as the corresponding Gras/GXL class. The `ProgresAttribute` and the `value` that should be denoted are given as parameters of the method.

```
@write(putAttributeValue(@unpack(attribute), value))
```

```
public void putAttributeValue ( ProgresAttribute attribute , Serializable value) throws
    EntityNotFoundException, GrasGXLException {
    Attribute attributeattribute = attribute . getBackingAttribute () ;
    node. putAttributeValue ( attributeattribute , value) ;
}
```

The PROGRES graph model resembles the Gras/GXL graph model, and therefore the mapping was not very complicated. The PROGRES graph model was a good first example, and allowed the implementation from SUMAGRAM from scratch. The advantage of the simplicity of the graph model was that it delivered no big complications during the implementation, and SUMAGRAM could be implemented neatly and structured. The only issue was the two-tiered node class. As there exist both a `ProgresNodeType` and a `ProgresNodeClass`, this delivered some difficulties, but nevertheless the PROGRES graph model was not complicated to map to the Gras/GXL graph model. However, to ensure that SUMAGRAM was not dependent on the PROGRES graph model, another example has to be used. This example had to be quite different from the PROGRES graph model. Thus we chose the GXL graph model.

7.2 GXL Graph Model

GXL is the Graph Exchange Language. It is an XML based exchange format, which is used for sharing data between tools. Typed, attributed and ordered directed graphs form the base for GXL. The major target for GXL is to provide interoperability between tools which use graphs for the representation of internal data.

7.2.1 Graph eXchange Language

The Graph eXchange Language (GXL, [HWS00]) originated in the year 1998. The target of GXL was to enable the exchange of information between reengineering applications. Already some useful tools existed, but the problem was they performed only parts of the desired features. They had for instance a functional graph query language, but no visualisation of the internally used graphs.

One of the existent tools was Gupro [KWDE98]. It uses as a query language GReQL [Kam96]. The major disadvantage was the fact that Gupro did not support graphical visualisation of the internally used graphs. Another tool was Rigi [MK88]: it did support the visualisation aspect, but had no functional language.

Graphs are often used to represent complex internal documents. This characteristic is also useful for most of the reengineering tools, which requests the implementation of a general exchange language for graphs. This feature is represented by GraX [EKW99]. While graphs are used in a lot of application domains, the exchange format GraphXML [HM00] was created.

With GraX, only TGraphs [JE95] could be exchanged. TGraphs are ordered graphs of which the nodes and edges could be attributed and typed. More complex graph elements like n-ary relations were not directly created. Among others, this problem was the cause for GXL to be developed. To the

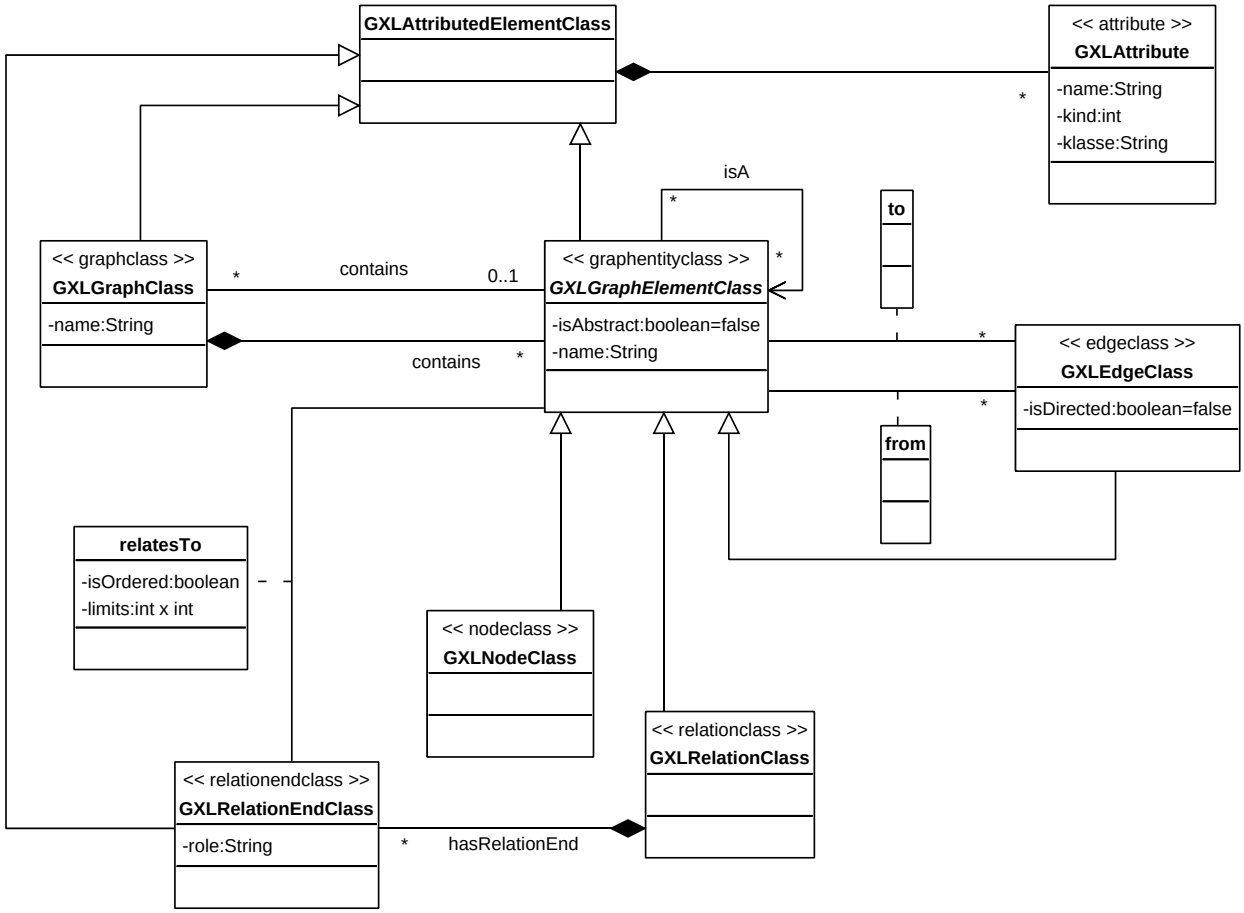


Figure 7.8: GXL Graph Schema

developed graph model, the concepts for hierarchical graphs and hypergraphs were added, because they are used in various graph models. GXL can as such be seen as the superset of these graph models.

GXL supports the exchange of graphs as well as their graph schemata. GXL is a sublanguage of XML [BPSM98]. The graph models of GXL have as a property that every element can be attributed and also attributes themselves can be attributed. GXL has also some disadvantages, which will be rectified in future versions. It does not support relations between graphs. These must be simulated with surrogate graph elements. Another problem is the support of references of graphs and graph elements.

7.2.2 The Graph Model

The basis of the GXL graph model is the `GXLGraphPool`, which corresponds to the `GraphPool` class of Gras/GXL. This means that this graph pool is responsible for the creation and deletion of the graphs and for storing the different graphs. To the graph pool, a one-to-one relation with the `GXLSchema` exists. The schema class is responsible for the creation and use of the different schema entities: `GXLNodeClass`, `GXLEdgeClass`, `GXLGraphClass`, and `GXLRelationClass`. These classes are respectively related to the `NodeClass`, the `EdgeClass`, the `GraphClass`, and

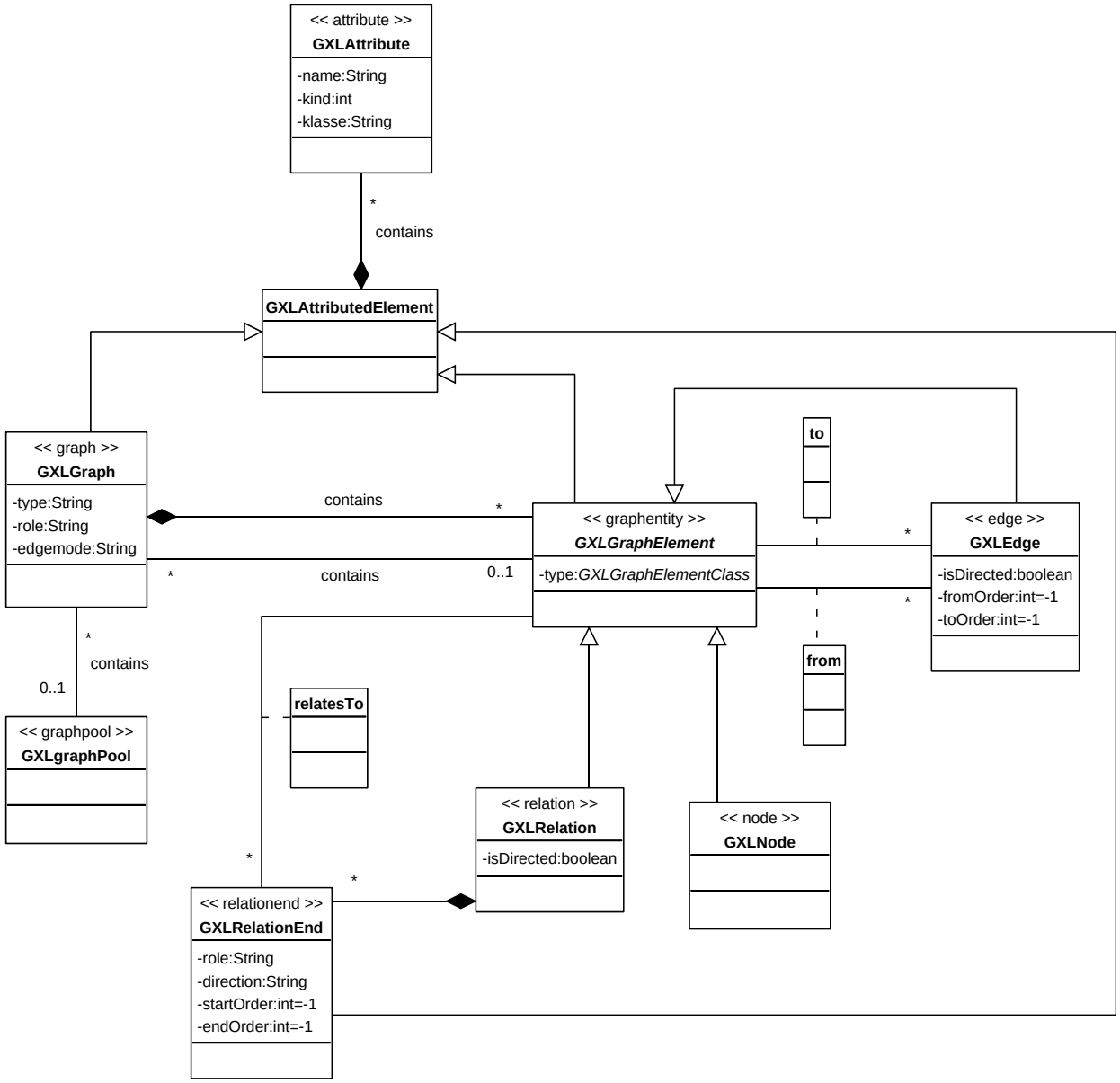


Figure 7.9: GXL Graph Model

the `RelationClass`. These classes have an abstract superclass: the `GXLGraphElementClass`. The `GXLRelationClass` is composed of zero or more entities of `GXLRelationEndClass`. It is therefore also responsible for the declaration of the `GXLRelationEndClass`.

The `GXLAttributedElementClass` is the superclass of the `GXLGraphClass`, the `GXLGraphElementClass`, and the `GXLRelationEndClass`. The `GXLAttributedElementClass` has no corresponding class in Gras/GXL and therefore, it has no stereotype.

The `GXLEdgeClass` defines the definition of the relations between the different `GXLGraphElementClasses`. There are a `from` and a `to` association, which define that an edge can come from a graph element and go to a graph element. The `GXLRelationEndClass` is

related to the GXLGraphElementClass in the way that every relation end is related to a graph element.

The GXLGraphClass is composed of entities of the GXLGraphElementClass. But, the other way around, the GXLGraphClass also is a type of a GXLGraphElementClass.

The GXLAttributedElementClass is composed of zero or more GXLAttributes, which is also part of the GXLAttributedElement class. With the GXLAttributedElement class, the view is turned to the GXL graph model. The GXLAttributedElement class is the superclass of GXLGraph, GXLGraphElement, and GXLRelationEnd. The class GXLGraphElement is an abstract class that is the superclass of GXLNode, GXLRelation, and GXLEdge. A GXLRelation can be composed of multiple GXLRelationEnds.

The GXLRelationEnd is related to the GXLGraphElement class, because every relation end is related to a graph element. A GXLEdge is an edge from a GXLGraphElement to a GXLGraphElement, which is most of the time another one but might also be the same one.

The GXLGraphPool contains a number of graphs. These graphs of type GXLGraph are composed of a number of GXLGraphElements. A GXLGraphElement can also be a GXLGraph, which implies that a graph can consist of graphs.

The edges and relations of the GXL graph model have the properties that they can be directed or undirected. This property is denoted by means of a boolean value. Edges and relations also can have an ordering. This ordering is implemented independent of the Gras/GXL graph model, because this feature is not supported by Gras/GXL.

The graphs of GXL can be untyped. As Gras/GXL only supports typed graphs, this property is ignored, and it is pretended that all graphs do have a type.

7.2.3 The Implementation of the GXL Graph Model

The GXL graph model contains ordered edges. Although this feature is not available in Gras/GXL, SUMAGRAM tolerates to write this code. This is done by the use of literal code. The methods that are specified on behalf of the ordering of edges, should be inserted in the UML class diagram through literal code. For the implementation of the GXL graph model, there is claimed that the methods for the ordering are existent. For instance, for the class GXLEdge, a method setOrderOfEdge exists. This method has two parameters: the fromOrder and the toOrder. The method is included as follows in the UML diagram:

```
this.fromOrder = fromOrder;
this.toOrder = toOrder;
@write(arrangeOrdering(this,fromOrder,toOrder))
```

The result in the Java output files is shown below. The literal code is integrated into the generated code.

```
public void setOrderOfEdge(int fromOrder, int toOrder) throws GrasGXLException {
        this.fromOrder = fromOrder;
        this.toOrder = toOrder;
        edge.arrangeOrdering(this,fromOrder,toOrder);
}
```

On all locations, this combination of literal and generated code has to be written. For instance, also for the method `getAllIncomingEdges`, the edges in the collection will be ordered by means of an ordering method.

Another method that is considered is the `declareGraphClass` method of the `GXLSchema` class. This class is a concrete class of the GXL graph schema. The method retrieves one parameter: the `name` in String format. The tags are defined as follows:

```
@declare(GraphClass,declareGraphClass(name, true))
@translateEntity(GXLGraphClass,GraphClass)
```

The corresponding source code is depicted next.

```
public GXLGraphClass declareGraphClass(String name) throws GrasGXLException {
    GraphClass graphclass = schema.declareGraphClass(name, true);
    GXLGraphClass gxlgraphclass = new GXLGraphClass(graphclass,null, this ,name);
    GXLSchemaFactory.addGXLGraphClass(gxlgraphclass);
    return gxlgraphclass ;
}
```

The source code belonging to the *declare* tag is the assignment of the `graphclass` variable. The rest of the source code is created due to the *translateEntity* tag.

As the mapping of the methods of the GXL graph model resembles the mapping of the PROGRES graph model, no more examples will be given. The idea of the mapping is the same for most of the methods.

During the specification of the GXL graph model, a few issues have arisen. First, there had to be decided that for the `GXLAttributedElementClass` no stereotype could be defined. In the beginning, there was thought that a stereotype always had to exist for every class of the specialized graph model. This example showed though that having a stereotype is no requirement. Without any stereotype, no mapping can be created, but a stereotype is not indispensable for every class.

In Poseidon, it seems that there is a small error on behalf of including exceptions. In the case of SUMAGRAM, a number of different exceptions can be used: `GrasGXLException`, `SchemaCheckException`, etcetera. It is though only possible to include new exceptions in Poseidon. Because the exceptions occur more often, it is desired that they can be reused. It is nevertheless not possible to add an already existing exception to an operation. This is a known error in Poseidon, which will be rectified in one of the coming releases. For the PROGRES graph model, every exception is added by hand in the XMI file. Nevertheless, for the GXL graph model, the exceptions were all handled by the `GrasGXLException`, which is a superclass of all exceptions. This property is implemented in SUMAGRAM: if no exceptions are included, default the `GrasGXLException` will be used for every method.

The attributes like `fromOrder` and `toOrder` of `GXLEdge` are optional values. This means that they are not by default assigned. Therefore, these attributes receive an optional value, which can be interpreted by SUMAGRAM, in case that a value for these attributes is required. Concrete, this means that on every location where an instance of this attribute is needed and no instance is available, the optional value will be inserted. As this optional value is chosen in a way that it is harmless (it should be the default value), the program will work correctly. An example is the `fromOrder`. The initial value is `-1`. This means that if the value is not assigned during the construction of the `GXLEdge`, the

value `-1` will be used. SUMAGRAM knows that this value is the value to denote that no order is defined, and thus this will not cause any problems.

Another issue that was encountered, is the naming of attributes. In the GXL graph model, there are three relations related to the `GXLGraph` class, they all have the name `contains`. As the composition relation is ignored, this would deliver for the `GXLGraph` class, two different attributes both with the name `contains`. To avoid this, the naming of attributes has changed. To an attribute of an association, the name of the target class is added. The `GXLGraph` class thus has two attributes with respectively the variable names `containsGXLGraphElement` and `containsGXLGraphPool`.

After the testing of SUMAGRAM with the GXL graph model, a number of larger and smaller errors have been rectified. Therefore, it can be said that SUMAGRAM now functions without errors. The code that is generated compiles faultlessly. Of course, it still can be extended as is denoted in the next chapter.

Chapter 8

Conclusions

The requirement on behalf of SUMAGRAM was to develop an application that supports the user by creating a mapping from a specialized graph model to the Gras/GXL graph model. This mapping was formerly created by hand, but as this is a mechanical, tedious, and error-prone task, it is desired that the mapping is generated by an application, SUMAGRAM. With SUMAGRAM, the user only has to add a small amount of code from which the mapping is generated. Of course, writing literal code also has to be possible.

The specialized graph model is designed by means of a UML class diagram, which will be transformed into an XMI file. This XMI file is converted to an intermediate format, which is an XML file. This XML file will be converted to the mapping, which is a Java application.

SUMAGRAM is able to let the user develop source code by means of tags. With these tags, the user can give a short description of the content of a method. Each type of tag denotes a group of methods that occurs frequently, like the retrieval of a graph entity or the declaration of a schema entity. These tags should be inserted in the UML class diagram. By creating the source code using the tags, the user can create a mapping in a straightforward and structured manner. The use of the tags is not complicated, which means that the usability of SUMAGRAM can be valued positive.

The features of SUMAGRAM ensure that the information that is given by the user, is complete. Also when the inputted data is not entirely consistent, the program is able to rectify some of the data.

8.1 Future

Some extensions can still be made on behalf of SUMAGRAM. The implementation of OCL constraints was not accomplished. It is wishful that methods like the `isSourceValid` method can be implemented by means of OCL constraints. In Poseidon, it is possible to include OCL constraints. This feature should be explored in order to implement OCL constraints. Afterwards, also the interpretation of the OCL constraints has to be implemented in SUMAGRAM.

Up to now, it is not possible to test whether the generated code conforms to the specialized graph model. Therefore, it is desirable that unit tests [Lin02] can be generated out of the specialized graph model in order to test the generated mapping code. But first, the OCL interpretation should be

included into SUMAGRAM. With the UML class diagram and the OCL constraints, SUMAGRAM should generate a framework for test cases, in order to check what is specified with SUMAGRAM. If such a framework is available, the user can enter tests in it. With these test cases, for a specialized graph model, the user can test if the output that is generated by SUMAGRAM is the output that was desired.

Bibliography

[And] AndroMDA website. http://www.andromda.org/.

[Ant] Ant website. http://ant.apache.org/.

[Böh04] Boris Böhlen. Specific graph models and their mappings to a common model. In John L. Pfaltz, Manfred Nagl, and Boris Böhlen, editors, *Applications of Graph Transformations with Industrial Relevance, Second International Workshop, AGTIVE 2003*, volume 3062 of *Lecture Notes in Computer Science*, pages 45–60. Springer-Verlag, Heidelberg, 2004.

[BPSM98] Tim Bray, Jean Paoli, and C. M. Sperberg-McQueen, editors. *Extensible Markup Language (XML) 1.0*. W3C, February 1998. http://www.w3.org/TR/1998/REC-xml-19980210.

[BR04] Boris Böhlen and Ulrike Ranger. Concepts for specifying complex graph transformation systems. In Hartmut Ehrig, Gregor Engels, and Francesco Parisi-Presicce, editors, *Graph Transformations: Second International Conference, ICGT 2004*, volume 3256 of *Lecture Notes in Computer Science*, pages xx–xx. Springer-Verlag, Heidelberg, 2004.

[Cla99] James Clark, editor. *XSL Transformations (XSLT)*. W3C, November 1999. http://www.w3.org/TR/1999/REC-xslt-19991116, Version 1.0.

[EEKR00] Hartmut Ehrig, Gregor Engels, Hans-Jörg Kreowski, and Grzegorz Rozenberg, editors. *Theory and Application of Graph Transformations, 6th International Workshop, TAGT'98*, volume 1764 of *Lecture Notes in Computer Science*. Springer-Verlag, Heidelberg, 2000.

[EKW99] Jürgen Ebert, Bernt Kullbach, and Andreas Winter. GraX - an interchange format for reengineering tools. In *Proceedings of the 6th Working Conference on Reverse Engineering (WCRE '99)*, pages 89–98. IEEE Computer Society Press, Los Alamitos, CA, USA, 1999.

[FNTZ00] T. Fischer, J. Niere, L. Torunski, and A. Zündorf. Story diagrams: A new graph rewrite language based on the Unified Modeling Language and Java. In Ehrig et al. [EEKR00].

[GR92] Jim Gray and Andreas Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann Publishers Inc., 1992.

[Har02] Elliotte Rusty Harold. *Processing XML with Java - A guide to SAX, DOM, JDOM, JAXP, and TrAX*. Pearson Education, 2002.

[HM00] Ivan Herman and M. Scott Marshall. GraphXML - An XML-Based Graph Description Format. In *Graph Drawing*, pages 52–62, 2000.

[HWS00] Richard Holt, Andreas Winter, and Andy Schürr. GXL: Towards a standard exchange format. In *Proceedings of the 7th Working Conference on Reverse Engineering (WCRE '00)*, pages 162–171. IEEE Computer Society Press, Los Alamitos, CA, USA, 2000.

[JE95] A. Franzke J. Ebert. A declarative approach to graph based modeling. In G. Tinhofer E. Mayr, G. Schmidt, editor, *Graphtheoretic Concepts in Computer Science*, number 903 in Lecture Notes in Computer Science, pages 38–50. Springer-Verlag, Heidelberg, 1995.

[Kam96] M. Kamp. GReQL - eine Anfragesprache für das GUPRO-Repository 1.1. Projektbericht 8/96, Universität Koblenz-Landau, Institut für Softwaretechnik, Koblenz, Januar 1996. `http://www.uni-koblenz.de/~ist/Projektberichte/1996/ist08-96.ps.gz`.

[Kay05a] Michael Kay. *Saxonica: XSLT and XQuery processing*. 2005. `http://www.saxonica.com/`.

[Kay05b] Michael Kay, editor. *XSL Transformations (XSLT)*. W3C, February 2005. `http://www.w3.org/TR/2005/WD-xslt20-20050211`, Version 2.0.

[KSW95] Norbert Kiesel, Andy Schürr, and Bernhard Westfechtel. GRAS, a graph-oriented (software) engineering database system. *Information Systems*, 20(1):21–51, 1995.

[KWB03] Anneke Kleppe, Jos Warmer, and Wim Bast. *MDA Explained: The Model Driven Architecture: Practice and Promise*. Object Technology Series. Addison Wesley, Reading, MA, USA, 2003.

[KWDE98] Bernt Kullbach, Andreas Winter, Peter Dahm, and Jürgen Ebert. *Program Comprehension in Multi-Language Systems*. IEEE Computer Society Press, Los Alamitos, CA, USA, 1998.

[Lin02] Johannes Link. *Unit Tests mit Java: Der Test-First-Ansatz*. dpunkt.verlag, Heidelberg, 2002.

[Mav] Maven website. `http://maven.apache.org/`.

[MK88] H. A. Müller and K. Klashinsky. Rigi-a system for programming-in-the-large. In *ICSE '88: Proceedings of the 10th international conference on Software engineering*, pages 80–86, Los Alamitos, CA, USA, 1988. IEEE Computer Society Press.

[Mod] modelbased website. `http://www.modelbased.net/`.

[Mom00] Bruce Momjian. *PostgreSQL: Introduction and Concepts*. Addison Wesley, Reading, MA, USA, 2000.

[Nag96] Manfred Nagl, editor. *Building Tightly Integrated Software Development Environments: The IPSEN Approach*, volume 1170 of *Lecture Notes in Computer Science*. Springer-Verlag, Heidelberg, 1996.

[Obj01] Object Management Group, Needham, MA, USA. *Unified Modeling Language (UML) Specification, version 1.4*, September 2001. `http://www.omg.org/cgi-bin/doc?formal/01-09-67.pdf`.

[Obj02] Object Management Group, Needham, MA, USA. *XML Metadata Interchange (XMI), version 1.2*, January 2002. `http://www.omg.org/cgi-bin/doc?formal/02-01-01.pdf`.

[Obj04] Object Management Group, Needham, MA, USA. *Meta-Object Facility (MOF) Specification, Version 2.0*, 2004. http://www.omg.org/uml.

[OMG] OMG Website. http://www.omg.org/.

[Ope04] *OpenMDX Introduction - Model Driven Architecture and Development*. 2004. http://www.openmdx.org,Version 1.0.

[PLHW05] Ray Whitmer Philippe Le Hégaret and Lauren Wood, editors. *Document Object Model*. W3C, 19 edition, January 2005. http://www.w3.org/DOM/.

[Pos] Poseidon for UML website. http://www.gentleware.com/.

[Ros] Rational Rose website. http://www-306.ibm.com/software/rational/.

[RWT] RWTH Aachen University website. http://www-i3.informatik.rwth-aachen.de/.

[Sch94] A. Schürr. PROGRES, a visual language and environment for programming with graph rewrite systems. Technical Report AIB 94-11, Aachen University of Technology, 1994.

[Sin] Sintef website. http://www.sintef.no/.

[UMT] UMT-QVT website. http://umt-qvt.sourceforge.net/.

[Utr] University of Utrecht website. http://www.cs.uu.nl/.

[Ver] Versant. Versant fastobjects t7. http://www.versant.com/products/fastobjects.

[WK99] Jos Warmer and Anneke Kleppe. *The Object Constraint Language — Precise Modeling with UML*. Object Technology Series. Addison Wesley, Reading, MA, USA, 1999.