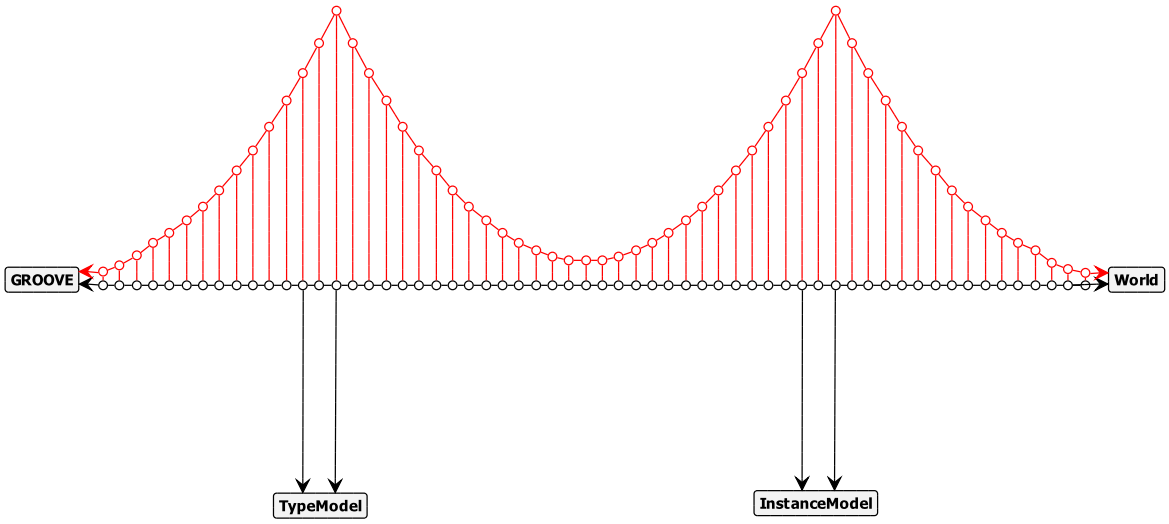
Bridging GROOVE to the world using a conceptual language model



by Harold Bruintjes

UNIVERSITY OF TWENTE.

University of Twente
Faculty of Electrical Engineering, Mathematics and Computer Science
Formal Methods and Tools

Master's Thesis

Bridging GROOVE to the world using a conceptual language model

Author:
Harold Bruintjes

Committee:
Prof. Dr. Arend Rensink
Dr. Ivan Kurtev Ivanov
Dr. Wojciech Mostowski

Monday 12th November, 2012

Preface

My days as a computer scientists started somewhere early in high school, where I discovered that it was possible to take an existing program, and disassemble it into simple machine instructions. It was very interesting to look at how such programs worked, but even more interesting was the fact that these programs could be put back together. At this point I realized that it was possible to make a computer do what you want, rather than being limited to what it allows you to do. Of course I did not understand many of it all, but this is what inspired me to investigate further, and ultimately decide to study in the field of computer science.

Skip ahead a few years and here we are, writing my master's thesis. During my master, there were several occasions where the GROOVE tool was used. I liked the concept of graph transformations and wanted to be able to do something with it for my thesis. Additionally, since I had learnt a lot in the field of formal methods, I wanted something that allowed me to make an implementation of a tool, rather than simply being the user of one. When I talked to Arend Rensink about this, he proposed this project, and it was quickly decided that I would take it.

I would like to thank all my supervisors of this thesis, Arend, Ivan and Wojciech, as well as Maarten, for your help in making this thesis. Your input and feedback have helped me a great deal in understanding the problem, formalizing the solution and writing this thesis. I have learned a great deal from you all. Arend, thank you for helping me put structure in this thesis, and adding improvements mere moments after I found the need for them. Ivan, thank you for helping me around in the world of software modeling and the maze of terminology in it. Wojciech, thank you for your reviews and getting my English up to par, even at there very last moment. And finally Maarten, thank you for helping me writing down the formal specifications during the beginning.

I would also like to thank my fellow students at the FMT-Lab (a.k.a. "Het afstudeerhok"). With you guys no day was ever boring. Freark, Gerjan, Lesley, Paul, Ronald, and everyone else, thank you for your feedback and technical assistance, as well as the pleasant discussions. Also, thumbs up for those nights with pizza, drinks and games.

Abstract

Currently, GROOVE is capable of importing and exporting Ecore models, which can be used by tools such as Eclipse using its underlying modeling framework. This is done using a standalone tool which can transform GROOVE graphs to Ecore models and vice versa. An Ecore type model may be imported as a type graph, and Ecore instance models as instance graphs. GROOVE can then be used to transform these instance graphs, which may then be exported back again to Ecore instance models. However, the generated type and instance graphs can sometimes become too complex and unmanageable, where a simpler graph would be sufficient. It is therefore desirable to improve the current import and export facilities, and give the user the option to generate more simple graphs. Ideally, this process would be configurable so different choices can be made on the resulting outcome. Additionally, it is interesting to add the capability for translating other graph languages to the import and export facilities as well, so as to make GROOVE more interoperable with other tools.

These problems are addressed by defining a conceptual model, which acts as an intermediate layer between GROOVE and possible external languages, such as Ecore. A formal description is given for this conceptual model, providing a well defined foundation for the mappings between itself, the external languages and GROOVE. The mapping between the conceptual model and the external languages is kept simple by combining the concepts found in these external languages into the conceptual model. Then, only the mapping between GROOVE and the conceptual model needs to perform the more heavyweight translation from the conceptual model to the graphs supported by GROOVE.

Contents

Chapter 1 Introduction

1.1 Software development

In the current world of software development, much effort is spent on trying to ensure or proof that software is defect free. The importance of this aspect of software development is clear; many systems depend on software, and software faults can cause damage to these systems, or cause these systems to fail completely. Thus various methods have been created to be able to detect and prevent defects in software. One of these methods is the modeling of software. As software grows more and more complex, it is not always feasible to check for various properties on the software itself. Instead, a model provides a simplified representation of software, which may be used to proof or predict various properties on this software.

Of course, many different representations can be used to model software, and just as many different tools can be used to work with these models. This gives rise to a new question: If a model is created for a tool with a very specific purpose, is it possible to use the same model in another tool with a different purpose? For example, a model for software can be used by one tool to generate an actual program, whilst another tool may be used to verify that certain properties hold on such a model.

1.1.1 Tools and Interoperability

As most tools use a different language to store their models, exchanging models between tools is often not trivial. The language used to store a model has to be translated in some way to another language before it can be used again in another application. This translation can be performed by the application that creates the model, which is referred to as exporting, or the application that reads the model, referred to as importing. Of course applications can support both processes, but most of the time neither is supported. This causes a problem with the interoperability of these tools, as a model that can neither be imported or exported cannot be used by a tool that uses a different language. Simply recreating a model by hand is not always viable, as this is error prone and often very time consuming (especially if multiple, or large, models are involved).

The main advantage of being able to exchange models between various tools is that all of their capabilities can be combined together. Some tools are designed to perform one function particularly well, while others are optimized for other tasks. Therefore, increasing the interoperability between these tools would be beneficial.

1.2 Modeling

Modeling tools often make use of different modeling levels. These different levels are used to describe the actual instances that are being modeled, the models themselves and, in some cases, the model of these models. These different levels are referred to as *instance models*, *type models* and *language models* respectively. Instance models describe actual objects and their associated values, the objects themselves also being considered values. Type models describe the different classes of such objects and their properties, giving a categorization of each such object, and limiting the scope of possible values that may be assigned. Finally, language models classify the elements in a type model.

Of course, language models can be described by other, higher level models, which themselves can be described by models and so on. However, for this thesis these type of models are not required and the three types of models described here are sufficient.

1.2.1 Instance models

Instance models describe actual instances 'in the world'. When considering, for example, the Java programming language, an instance model would represent the state of a program during execution, at which point various instances of objects exist, each with assigned values and residing in a specific place in memory.

1.2.2 Type models

Type models are a description of the possibly types of objects that can be found in an instance model. Leveraging the concepts in Java, a type model would be the equivalent of the code of a program, specifying the various classes of objects and the types of relations between objects.

1.2.3 Language models

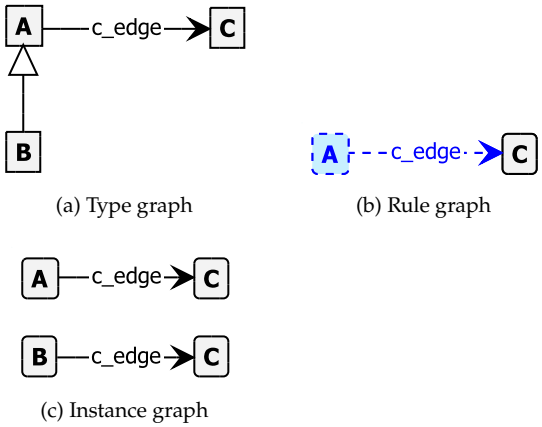
Language models describe the elements that are allowed to occur in a type model. In Java, the language model would describe the allowed syntax and structure of the code of a program, such as class definitions.

1.3 Graphs and GROOVE

Graphs are well suited for modelling object oriented software, as well as visualizing other types of models. It is therefore not surprising that various tools exist that work on such graphs, each supporting their own set of languages to represent these graphs and to transfer them between applications. One such tool is GROOVE [32], which is designed to allow object oriented software to be modeled using graphs, and for these graphs to be verified.

GROOVE is a tool for working with simple graphs and graph transformations [19]. It can be used to model the structure of object oriented systems as graphs, and perform various operations on them. Currently, GROOVE stores these graphs in its own format (based on the GXL format [39]) which is not compatible with other tools. However, in its simplest form these graphs only consist of a series of labeled nodes and edges, which should be conceptually supported by other tools. Therefore, it should be possible to translate the graphs in GROOVE to and from the models of other tools in their respective language.

GROOVE makes use of graph grammars, which for this work can be considered to consist of three kinds of graphs: type graphs, rule graphs, and instance graphs. Type graphs define the structure of the graphs that are allowed as instance graphs or rule graphs. They support concepts such as inheritance, abstractness and multiplicities. An example type graph is shown in fig. 1.1a, showing three node types (`A`, `B` and `C`), an inheritance relation between `B` and `A`, and an edge between `A` and `C`. Valid instance graphs contain only nodes of these types, and only edges between `A` and `C` nodes, and between `B` and `C` nodes (as `B` inherits from `A`). Rule graphs define possible transformations for instance graphs. Rule graphs allow nodes and edges to be removed, created or simply be checked for their presence (or absence). An example rule graph is shown in fig. 1.1b, which will remove any `A` node which has an edge to an existing `C` node. Instance graphs are the graphs that represent an actual



(a) Type graph (b) Rule graph

(c) Instance graph

Figure 1.1: Example GROOVE graphs.

Import	Export
CADP	CADP
DIMACS	-
-	KTH
-	Various image formats
Ecore type	-
Ecore instance	Ecore instance

Table 1.1: Import and export formats supported by GROOVE.

instance model, and are typed by a type graph. Instance graphs can be transformed by rule graphs, resulting in new instance graphs. An example instance graph is shown in fig. 1.1c.

1.3.1 Current importer/exporter

The current version of GROOVE (which at the time of writing is `4.6.0` [20]), is capable of importing and exporting a few external formats as shown in table 1.1. One of these formats is the Ecore model (see section 2.1.1), which is interesting because it is both capable of importing graphs from, and exporting graphs to Ecore models. The Ecore importer and exporter was made as part of the research in [37].

1.4 Research question

Although GROOVE has an importer and exporter for Ecore models, some limitations still apply. The importer is capable of importing both Ecore type models and Ecore instance models, but the exporter can only export Ecore instance models. An additional problem that has arisen is the fact that the graphs that are created for Ecore models tend to become large. The user has no influence on the resulting outcome, which in some cases is too verbose. Furthermore, the importer and exporter for Ecore are not well integrated into GROOVE. They can only be used from the command line, rather than the interface that GROOVE provides itself.

Another point of improvement is the addition of additional languages to import from and export to from GROOVE. As table 1.1 shows, GROOVE already supports a few languages, but other interesting

languages exist as well. It would be beneficial to investigate what other languages exist and which are of interest to integrate into GROOVE.

Therefore, the main research question of this thesis is: *"How can the current import and export capabilities in GROOVE be improved?"*, where the improvement aims at both the current Ecore importer and exporter, as well as the addition of new languages.

1.5 Requirement analysis

Based on experiences with the previous importer/exporter and the desire to add new languages to GROOVE, a few requirements are imposed on the new importer/exporter. Since the aim of this thesis is to improve on the existing importer/exporter, the current capabilities are taken as the baseline. From the experiences with the current importer/exporter, the following issues were identified:

1. The generated graphs are sometimes overly complex;

2. Only instance models can be exported, not type models;

3. The import and export features are not integrated into the GROOVE interface, but are rather implemented as a separate command line interface.

Furthermore, Ecore is the only graph based modeling language supported, which means that support for other popular languages is lacking. Therefore, the aim is to add support for other languages as well, and provide an interface that makes it easier to add other graph based modeling languages which are not covered by this thesis. This can be done by providing some intermediate interface which connects external languages with GROOVE, and promotes code reuse.

1.5.1 Configuration

To tackle the problem of the inability to exert any influence on the generated graph, the ability to configure the import and export process is proposed. This configuration determines what representation is used for the various aspects of models that may be imported or exported. Thus, one model may be represented in different ways, depending on how the importer and exporter are configured.

Some models benefit from a simplified representation, whereas other models may require a complete or more verbose representation. For example, when a model is imported just to be able to get an overview, a simpler representation makes understanding the generated graphs easier. When the model needs to be transformed and exported back again, this process is most useful if most, if not all, information is retained, which requires a more complete and expressive representation.

Since a graph imported with a specific configuration may have to be exported again, the same configuration has to be available. Therefore, some mechanism is required in order to keep track of the used configurations so that they may be reused. This means that the configuration has to be made persistent, and that the user has to be able to select and modify the existing configuration(s).

Finally, if a model is imported or exported and the user wishes to create or modify a configuration, then a GUI should be available in order to do so. This provides for a better workflow when the GROOVE GUI is used to initiate an import or export operation.

1.5.2 Use cases

This section describes the various use cases for the new importer/exporter. These use cases will guide the design decisions and determine what capabilities are needed or desired in order for the importer/exporter to be useful.

Model checking

One of the main use cases is that of model checking using GROOVE. In this case, a model is generated/created outside of GROOVE in one of the supported languages, for example Ecore. The model is then imported into GROOVE, upon which various transformations can be applied to check for various properties.

First, the model is created in one of the external languages. After having done this, it is imported into GROOVE. At this point, the model can be treated as any graph in GROOVE and as such model checking can be performed on the instance graphs (which represent instance models).

Model transformation

Another important use case is that of model transformation. In this case, a model is imported, transformed and then exported back again. Here, GROOVE is used as a model transformation tool. Since GROOVE is capable of performing model transformations, this use case suits GROOVE well.

First, an instance model is created in one of the external languages as with the previous use case. After importing it into GROOVE, GROOVE may be used to transform the generated graph for this instance model. This transformed model may then be exported back again.

Model editing

Editing a model in GROOVE should be supported by the importer/exporter. Although editing the graph does not change the import/export functionality directly, it will affect the required configuration. If a graph is imported with a specific configuration, and it is later on edited in such a way that it does not correspond to that configuration (for instance, a change in naming convention), the configuration will have to be adjusted. The importer/exporter may try to do so automatically, but this will quickly become very complex for all but the most trivial changes. Rather, it will be left up to the user to make the proper adjustments.

There are 3 main case distinctions:

1. Making small changes in the grammar,
2. Changing the structure of graphs in the grammar, or the grammar itself,
3. Breaking the graph grammar.

Small changes in the graph that do not affect its structure with regard to the configuration require no special treatment (no changes in the configuration are required). Exporting the graph can still be handled without any new input required. Larger changes that do affect the graph structure in such a way that the configuration does not match anymore will require the user to modify the configuration accordingly. This requires an interface that allows the user to do this (this interface can be the same as that is offered during the import process). Optionally, the importer/exporter can try to validate the current graphs using this configuration. Finally, it is possible that some changes break the grammar

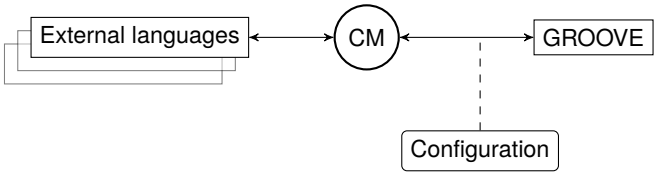


Figure 1.2: Overview of the proposed importer/exporter components.

in such a way that no configuration will allow the exporter to function correctly. In such cases, a message should be generated that indicates what the problem is (or might be), allowing the user to fix the issue.

Model porting

Another use case is that of porting models from one language to another. In this case, GROOVE is only used as an intermediate tool that translates a model from one language to another, without changing it.

First, a model in one external language is imported into GROOVE via the conceptual model. Then, it is exported back again via the conceptual model, but to a different target language. This allows GROOVE to be used as a tool to provide model translations, when an editor for such a model does not provide the option of exporting to one of the languages supported by the exporter of GROOVE.

Model generation

The importer/exporter can also be used to export models created within GROOVE to be used by other tools. This allows operations to be performed on those models that are otherwise out of the scope of GROOVE. For example, a model in GROOVE may be exported to Ecore, which can then be used to create concrete Java classes. The creation of a model requires the creation of a type graph that fits within the conceptual model and optionally accompanying instance graphs.

In this case no configuration exists yet and a new one has to be created. This can be done by copying an existing configuration (adjusting it where necessary) or creating a new one entirely (as is possible during the import process).

1.5.3 Conceptual model

Reusing as much code as possible helps speed up development and improves maintainability. In order to achieve this, a conceptual model (CM) is proposed, which connects the various external languages to GROOVE via some intermediate representation, based on the concepts found in these external languages. By keeping the level of expressiveness of the conceptual model between that of the external languages and the graphs of GROOVE, the import and export process of the various languages can be simplified, as part of the translation process (namely, the part of the conceptual model that GROOVE cannot directly represent) can be reused for each external language.

Based on the requirements and the proposed conceptual model, an overview of how the various elements of the import/export process are connected is shown in fig. 1.2. It is important to recognize that the configuration only affects the translation between the conceptual model and GROOVE, and not between the conceptual model and the external languages.

1.6 Research question refinement

Based on the requirements analysis, various subquestions can be identified that play a role in answering the main research question, where each subquestion answers a part of the main research question.

As part of this thesis, some preliminary research, called the research topics, was performed to identify the boundaries of the problem. Part of this research has been incorporated in this thesis. For example, during the research topics a selection was made of the languages that were going to be added to the new importer and exporter and the conceptual model was defined.

The following research questions were identified:

1. "What are suitable languages for importing and exporting to and from GROOVE?"
   A choice needs to be made on what languages will be supported by the importer, and possibly what subset of the features of these languages. The various modeling and graph languages in existence all have different features and capabilities which may not be representable by a single conceptual model. It is therefore necessary to find out what languages lend themselves best for inclusion and how much overlap they have. This question has been answered during the research topics phase.
2. "How can different languages be covered while reusing as much code as possible?"
   Ideally, the importer/exporter would be generic enough to be able to import and export different languages while reusing as much code as possible. This helps speed up development of translators for new languages and improves maintainability of the code base (as there is less code to maintain). The answer to this question lies with the conceptual model, which has been defined as part of the research topics.
3. "How to map the languages from and to the conceptual model, and GROOVE from and to the conceptual model?"
   When the various languages are selected (or subparts thereof), a mapping needs to be defined between these languages and the actual conceptual model. The conceptual model will then act as the bridge between languages and GROOVE. Likewise, the conceptual model also needs to be mapped to GROOVE, and in such a way that it is configurable and flexible enough to allow the resulting graph to be either very detailed (with as little loss of information as possible), or simplified in some form to make the graphs more manageable.
4. "How can the import and export facilities be implemented?"
   Part of this thesis is the actual implementation of the exporter and importer, which requires an implementation of the conceptual model as the core. Additionally, the mappings between the conceptual model and both the external languages and GROOVE must be implemented, as well as integrated into the GROOVE GUI. Finally, parsers are required for the external languages as well.
5. "How can the results be validated?"
   The resulting output of the importer/exporter has to be validated in order to determine that the output is correct and conveys the right meaning. Furthermore, the actual implementation must be bench-marked in order to verify that it performs well enough to be used in GROOVE. Therefore, some quantifying (and qualifying) metrics are required to be able to measure the performance, which need to be determined.

1.7 Related work

Many different approaches exist to deal with model transformations. Two main types of transformations can be distinguished: endogenous and exogenous. Endogenous model transformations allow

one model to be translated into another within the same language. Exogenous model transformations allow a model to be transformed from one language to another. Since the model transformations described in this thesis are of an exogenous nature, the focus lies on these type of transformations.

A project that is closely related to this work is EMF Henshin, which provides a model transformation language. The Henshin language supports both endogenous and exogenous model transformations and uses a declarative model transformation language. It allows users to create pattern based transformation rules to transform models. These rules can then be structured into nested transformation units [2]. Henshin also features a state space generator, which can be used to model check the model transformations.

The Object Management Group (OMG) has defined the QVT (Query/View/Transformation) standard, which is comprised of a set of languages that allow model transformations. This set of languages consists of the QVT-Core language, QVT-Operational, QVT-Relations, and the QVT-BlackBox languages. Various implementations of these languages exist, as can be seen in [25].

Similar to QVT is Tefkat [26], which provides a declarative model transformation language. Tefkat makes use of the EMF, allowing it to work with models based on MOF, UML2 and XML Schema. It is closely related to QVT, but its design is based on a number of additional requirements.

The ATLAS Transformation Language (ATL) [23] consists of a hybrid of declarative and imperative constructs. ATL is capable of translating models which are typed by some other meta models. These meta models themselves are then modeled by the MOF. Likewise, ATL specifications make use of the ATL meta model, which itself is modeled by the MOF as well.

MOFLON [1] is a meta modeling framework based on the Fujaba platform. It adds the capability to use the MOF meta model and allows XMI models to be imported and exported. MOFLON can be used with both visual and textual representations and allows modularisation techniques to be applied.

An algebraic approach to model transformations is the MOMENT framework [6]. This framework uses a *ModelGen* operator, which is based on the Maude term rewriting system [12]. This operator supports the QVT-Relations language. By using an algebraic specification, efficient translations are possible, and the algebraic features can be proved (including transformation confluence and termination).

An algebraic semantics for the MOF standard is defined by Boronat and Meseguer [7], which continues the work of the MOMENT framework. This algebraic semantics gives a clear formal definition for the notions of a meta model, a model and the conformance of the model to the meta model. These formal semantics are executable due to the use of the Maude language, and have been implemented as a plugin for the Eclipse IDE by the name of MOMENT2.

AToM^3 is a visual meta modeling tool that is capable of handling a large number of components [15]. AToM^3 can be used to generate a tool to process models specified by graphical formalisms, which can be created using AToM^3. AToM^3 uses graph grammars to represent model transformations.

As a final reference, the work of Czarnecki and Helsen [14] gives an extensive overview of possible model transformation approaches and implementations of these approaches. They provide a classification framework for the various types of model transformations, and distinguish between two major categories: model-to-model transformations and model-to-text transformations.

1.8 Outline

This thesis begins with giving some background information on the GROOVE tool and the external graph languages used in this thesis in chapter 2. Here, some information is given on what these external languages are used for and why they are included in this thesis, as well as some theoretical

background for the GROOVE tool. This chapter aims at answering the first research question. Next, the conceptual model is introduced in chapter 3, which provides the bridge between the external languages and GROOVE, providing a part of the second research question. A full formal specification is given, as well as a few small examples. For a description of its actual implementation, answering the second research question completely, see appendix A. Based on this information, chapters 4 and 5 provide an overview of the mappings between the conceptual model and GROOVE, and the conceptual model and the external languages respectively. These chapters give the answer to the third research question. A more detailed description with more implementation details can be found in appendices B and C. An overview of the implementation, as well as the results of the experiments performed can be found in chapter 6, answering research questions 4 and 5. The final conclusions and some remarks about some further improvements can be found in chapter 7.

Chapter 2 Background

This chapter starts with an overview of the graph languages that were investigated as possible targets for the importer/exporter. From these languages, a selection is made that will actually be included in the final importer/exporter, based on a set of criteria of those languages. The actual details of the selected languages are not discussed until chapter 5 and appendix B. This provides an answer to the first research question which concerns languages suitable for importing and exporting.

This is followed by some theoretical background on GROOVE, or more specifically the graphs that GROOVE uses. This background is used to provide a mapping between these graphs and the conceptual model, which will be discussed in chapter 4 later on.

2.1 Graph languages

This section gives a short overview of a few graph based languages. These languages are a selection within many other graph languages available. This selection is based on a few criteria, which are:

1. Common usage,
2. Existence of a self describing meta model (meta schema),
3. Concrete notation, well defined interpretation,
4. Unique in its kind.

The selection of graph languages and the selection criteria are based on preliminary research from the research topics. For an overview of this research, see appendix D. Appendix D provides a description of each language that has been considered, as well as a description of each selection criterion.

An overview of the languages that have been considered can be found in table 2.1. Based on the criteria given and this overview, a selection was made that was implemented in the final importer/exporter. These were the Ecore, GXL and DOT languages. Of these languages, a short description is given in the following sections.

The most interesting aspect of this choice of languages is its diversity. The Ecore language, being part of the Eclipse Modeling Framework (EMF) is intended to be used to design tools and applications using a structured data model. GXL on the other hand can be used to model graphs instead. These graphs may then be used for any applicable purpose. Finally, DOT is not used for modeling at all but simply allows for graphs to be visualized in various ways. Implementing this combination of languages in the importer/exporter is most likely to give a clear view of the capabilities and limitations of the new importer and exporter.

2.1.1 Ecore

Ecore [16] is a modeling language serialized using XMI based on the (E)MOF specification. Ecore supports concepts such as classes, relations, attributes, types, inheritance and packages (namespaces).

Language	Included?	Reasoning	Commonly used	Meta schema	Concrete notation	Unique in its kind
GXL	Yes	GXL provides an interesting target, as it defines both a language for graphs and a meta schema for this language, allowing GXL to be used as a modeling language for graphs. Additionally, it is intended as a general graph exchange language, which means it adds extra versatility to the GROOVE tool if supported.	✓	✓	✓	✓
XMI	No	XMI defines only an exchange format rather that directly providing a model/graph by itself. It is too general to be supported directly, but rather forms a basis for more concrete languages (such as Ecore).	✓	✓		✓
Ecore	Yes	The Ecore language is taken as the baseline for this research. Therefore, the list of concepts Ecore defines is the minimum subset the importer should support, which can possibly extended by the concepts of other languages.	✓	✓	✓	✓
UML	No	UML is a popular and widely used language for modeling object oriented software. Using the MOF it can be encoded in XMI similar to Ecore. However, the scope of UML is very large and there are multiple combinations of XMI versions and UML versions possible, which are not directly compatible with each other. This makes compatibility with UML using XMI hard (see [37]). Thus, UML is not included.	✓	✓	✓	
GML/XGMML	No	GML and XGMML both provide a syntax for storing graphs, but do not define any model or schema on top. This makes these language less interesting, as virtually any type of graph can be described with no restrictions, making interoperability harder.			✓	✓
GraphML	No	GraphML very closely resembles GXL and a conversion between the two is possible (although not completely without loss of information). Since GXL is already on the list of included languages, and GraphML does not provide a language model of its own, it is not included.	✓		✓	
OWL (RDF(S))	No	The OWL language makes an interesting target as it is used to represent knowledge databases, as opposed to, for example, UML or Ecore, which represent object oriented software artifacts. OWL makes use of RDF(S) as its language model, just like the MOF for Ecore. However, due to time and resource constraints, this language is not included as it is fairly complex.	✓	✓	✓	✓
KM3	No	KM3 is a language that is based on Ecore, and in fact is a textual representation of Ecore. This means that KM3 and Ecore can easily be converted into each other without loss of information, and as such, since Ecore is already defining the baseline, KM3 does not make an interesting target.	✓	✓	✓	
Kermeta	No	Kermeta can be compared to KM3, with the addition of executability properties. In fact, Kermeta can also be converted to Ecore and vice versa just like KM3. So, for the same reasons as KM3, Kermeta is not included.	✓	✓	✓	
DOT	Yes	Like GML, DOT is a very general graph description language, without any direct modeling aspects. It is however a popular language to represent graphs and is often used by tools to visualize graphs. It is fairly simple on the conceptual side which should make the inclusion in an importer fairly easy when other, more complex language are already included.	✓		✓	✓

Table 2.1: Overview of investigated languages.

Since Ecore was used as the main language in the previous importer/exporter [37], and this is the continuation of that work, Ecore will be used as the baseline language. That is, at least every concept in the Ecore model previously supported should be supported by the new importer/exporter.

The main editor for Ecore can be found in the EMF, which can create Ecore type models and instance models. This is a graphical editor with an interface which can generate graphs that closely resemble (UML) class diagrams, although it is also possible to use an editor that works directly on the hierarchical structure of an Ecore document.

2.1.2 GXL

GXL (Graph eXchange Language) [39] is intended to be a standard language for exchanging graphs. GXL is XML based, and originates from merging the Graph Exchange Format (GraX), the Tuple Attribute language (TA) and the PROGRES graph rewriting system format. It is designed to make different software reengineering tools interoperable, with two features that make it particularly suited for software data exchange: its conceptual model is a typed, attributed, directed graph and it can be used to represent instance data as well as schemas describing the data structure [39].

GXL supports, besides standard nodes and edges, relations (which are hyperedges), subgraphs, attributes and types, which can be primitive types or composite types such as sets and tuples. GXL allows both schema graphs (for which a meta schema is defined) and instance graphs to be defined. These graphs are linked to each other via type specifiers using the XML `xlink` specification[1].

GXL currently provides two versions: 1.0 and 1.1. At the moment, 1.0 is the current version, with 1.1 being offered but not yet finalized (although the last change dates from 2002). Since 1.0 is marked as current, version 1.0 will be used in this research.

One of the tools designed to support GXL is GXLGraphPath, which implements a subset of GXL. It stores the graph in a GXL formatted document, and the layout in a separate GXLLayout document (which itself is not written in GXL). However, it does not fully support the entire GXL language (for example, subgraphs appear to be ignored). Being XML-based, GXL can also be edited by any XML editor, such as the Eclipse XML editor, which supports the GXL schema to create a correct GXL document. This does not give any visual feedback however.

2.1.3 DOT

DOT [4] is a simple text based graph language. In addition to defining graphs, it supports specifications for the layout of a graph, although not all the tools that can read DOT files support this. DOT supports the use of subgraphs, attributes and edge ports. A language definition of DOT can be found at [4].

DOT does not define any model concepts. It can easily be imported if layouting is ignored, as it simply defines a set of nodes and connecting edges, optionally with labels.

A collection of tools that can read (and in some cases write) DOT files can be found in the Graphviz suite [17]. It is commonly used for the exchange of graph structures.

[1] `http://www.w3.org/TR/xlink11/`

2.2 GROOVE

An overview of GROOVE was given in section 1.3. However, the graphs in GROOVE have a formal specification which can be used to define a mapping between GROOVE and the conceptual model (to be defined in chapter 4). The formal models given in the following sections are not the complete formalisation of GROOVE, but rather the subset that the mapping makes use of.

GROOVE has a set of labels *Lab* inside its (current) grammar. These labels are used by the type, instance, and rule graphs in GROOVE. The labels can be further subdivided into labels for types, flags and edges. The intersection of these sets has to be empty, and the sets are defined as:

- The set of type labels $Lab_t \subseteq Lab$,
- The set of flag labels $Lab_f \subseteq Lab$,
- The set of edge labels $Lab_e \subseteq Lab$.

Flags are special types of edges which always have an identical source and target node. GROOVE uses flags on nodes to indicate that a certain property holds for such a node. GROOVE also reserves a few labels for its primitive data types, namely

$$Lab_{prim} = \{\text{bool}, \text{int}, \text{real}, \text{string}\}$$

which may not intersect with the aforementioned sets of labels. These labels allow the use of attributes (and values). They represent the types for boolean values, integers (in $\mathbb{N}$), real numbers (in $\mathbb{R}$), and strings.

2.2.1 Type graphs

In GROOVE, a type graph is modeled by the tuple

$$\mathrm{T}_G = \langle N, E, \sqsubseteq, abs, mult, contains \rangle$$

where

- $N \subseteq Lab_t \cup Lab_{prim}$ is the set of nodes in the type graph. This is a subset of the set of labels, used to identify the type of the nodes. Note that both type labels and primitive type labels are used, as primitive types need to be explicitly added to the type graph for them to be used in an instance graph.
- $E \subseteq N \times (Lab_e \cup Lab_f) \times N$ is the set of edges, each of which consists of a source and target node, as well as an edge label or flag label used to identify the edge.
- $\sqsubseteq \subseteq N \times N$ is the inheritance relation, which is a set of tuples of nodes between which an inheritance relation exists. It defines a partially ordered set.
- $abs \subseteq N$ is the set of nodes which are abstract in the type graph. Abstract nodes cannot occur in instance graphs, but their subtypes may.
- $mult : E \to \mathbb{M} \times \mathbb{M}$ where $\mathbb{M} = \mathbb{N} \times (\mathbb{N}^+ \cup \{*\})$ with the lower bound always smaller or equal to the upper bound. It holds that $\forall n \in \mathbb{N} : n < *$. It is the function which maps a multiplicity pair to edges. A multiplicity pair consists of an incoming and outgoing multiplicity. The incoming multiplicity determines the allowed amount of nodes that share the same target node with this edge type. The outgoing multiplicity determines the number of target nodes a single source node may have edges to. A multiplicity consists of two values: the lower bound and the upper

bound. The lower bound determines the minimum of the multiplicity, the upper bound the maximum. Since this range must always contain a positive number of values, the maximum is at least as large as the minimum. For any multiplicity, we define

- $in : \mathbb{M} \times \mathbb{M} \to \mathbb{M}$
- $out : \mathbb{M} \times \mathbb{M} \to \mathbb{M}$

such that $\forall m = (m_{in}, m_{out}) \in mult : in(m) = m_{in} \wedge out(m) = m_{out}$. Furthermore, we say that $\forall m = (lower, upper) \in \mathbb{M}, n \in \mathbb{N} : n \in m \Leftrightarrow lower \leq n \wedge n \leq upper$.

- $contains \subseteq T_e$ is the set of edges which identify a containment relation. A node in an instance graph may not share multiple incoming containment edges, and a cycle of containment edges is not allowed in an instance graph. Furthermore, containment implies the incoming multiplicity of an edge is equal to 0..1, since a node is contained by at most one container.

Note that a GROOVE grammar allows multiple type graphs to be active at the same time. The type graph used by the grammar will then be inferred from all the active type graphs. This allows the various aspects of a type graph to be represented by multiple (sub) type graphs for easier editing. When two type graphs T_{G1} and T_{G2} are combined, the new type graph $T_G{}'$ is defined as

$$T_G{}' = \langle N_1 \cup N_2, E_1 \cup E_2, \sqsubseteq', abs_1 \cup abs_2, mult', contains_1 \cup contains_2 \rangle$$

The set of nodes and edges is simply the union of those of the two type graphs. The inheritance relation is defined as the transitive closure of the union of the subtype relations in the two type graphs. The set $mult'$ is defined such that $\forall e \in E_1 \setminus E_2 : mult'(e) = mult_1(e)$, $\forall e \in E_2 \setminus E_1 : mult'(e) = mult_2(e)$ and $\forall e \in E_1 \cap E_2 : mult'(e) = mult_1(e) \cap mult_2(e)$, where the intersection of two multiplicities is defined as the maximum of both lower bounds and the minimum of both upper bounds.

It is entirely possible that two arbitrary type graphs cannot be combined together into one single type graph, for example because a shared edge has incompatible multiplicities (if one type graph specifies a multiplicity of 0..1 and another 3..4, then no valid multiplicity can be created). Such type graphs cannot be combined by GROOVE and would in fact result in an error. If the merging of two type graphs can be seen as a function, then it could be defined as the partial function (where TG is the complete set of all possible type graphs)

$$merge : TG \times TG \hookrightarrow TG$$

2.2.2 Instance graphs

We assume there exists a global set *Node*, which can be mapped into two disjoint and covering sets $Node^t$ and $Node^v$ which, for every label in Lab_t and Lab_{prim} respectively, provide the set of typed nodes. Thus, there is a mapping

$$nodeType : (Node^t \to Lab_t) \cup (Node^v \to Lab_{prim})$$

Furthermore, the set $Node^v$ is mapped to actual data values in GROOVE by the function $value : Node^v \to \mathbb{B} \cup \mathbb{Z} \cup \mathbb{R} \cup \mathbb{S}$ (see also definition 4). Finally, a set of node identifiers *Id* is defined, which is used to assign a unique identifier to each node in an instance graph.

Like type graphs, multiple instance graphs can be combined into a new instance graph. However, this is beyond the scope of this thesis and only single instance graphs in a grammar will be considered.

Instance graphs in GROOVE are modeled by the tuple

$$\mathrm{H}_G = \langle H_n, H_e, ident \rangle$$

where

- $H_n \subseteq \mathit{Node}^t \cup \mathit{Node}^v$ is the set of nodes with a type label and the set of value nodes.
- $H_e \subseteq H_n \times E \times H_n$ is the set of edges, consisting of a source and target node and typed by an edge in the type graph.
- $ident : \mathit{Id} \rightarrow H_n$ is the identifier function which maps an identifier onto each node in the instance graph.

Such an instance graph is always typed by some type graph $\mathrm{T}_G = \langle N, E, \sqsubseteq, abs, mult, contains \rangle$.

The types of nodes and edges in an instance graph can be retrieved by the function type, which is defined as

$$\mathrm{type} : (H_n \rightarrow N) \cup (H_e \rightarrow E)$$

which maps nodes and edges in an instance graph to their types in the type graph of the grammar. This function is defined by the instance graph. For the nodes, it holds that $\forall n \in H_n : type(n) = nodeType(n)$. For edges, it holds that $\forall e \in H_e = (src, etype, tgt) \in H_e : type(e) = etype$.

As instance graphs are typed by a type graph, the type graph imposes a few constraints on the allowed instance graphs. These constraints are defined as follows:

- The source of each edge must be properly typed: $\forall e \in H_e : type(src(e)) \sqsubseteq src(type(e))$
- The target of each edge must be properly typed: $\forall e \in H_e : type(tgt(e)) \sqsubseteq tgt(type(e))$
- Abstract nodes may not be instantiated: $\forall n \in H_n : type(n) \notin abs$
- The outgoing multiplicity of each edge type must be adhered to: $\forall te \in E : \forall n \in H_n : |\{e \in H_e | src(e) = n \wedge type(e) = te\}| \in out(mult(te))$
- The incoming multiplicity of each edge type must be adhered to: $\forall te \in E : \forall n \in H_n : |\{e \in H_e | tgt(e) = n \wedge type(e) = te\}| \in in(mult(te))$
- Nodes must be contained by at most one other node: $\forall n \in H_n : |\{e \in H_e | target(e) = n \wedge type(e) \in contains\}| \leq 1$
- There may be no cycle between all containment edges in H_G.

2.2.3 Rules

Just like instance graphs, rule graphs are typed by a type graph. Rule graphs introduce transformation semantics in the graph grammar, by offering the capability to transform one instance graph into an other (possibly identical) instance graph. Both instance graphs are also typed by the same type graph. Rules can match, create and delete nodes, as well as flags and edges. For this reason, rule graphs introduce a few special node and edge types in order to specify these operations, which are matching, creator, deleter, and NAC nodes/edges. Formally, rules use a left hand side and a right hand side as explained in [32], but for this thesis these are collapsed into the aforementioned node types.

Besides the matching, creator, deleter, and NAC nodes and edges, other node types are also supported by rule graphs. For example, quantifier nodes are not directly related to any type graph, but rather allow multiple nodes to be matched by a single rule node under certain circumstances.

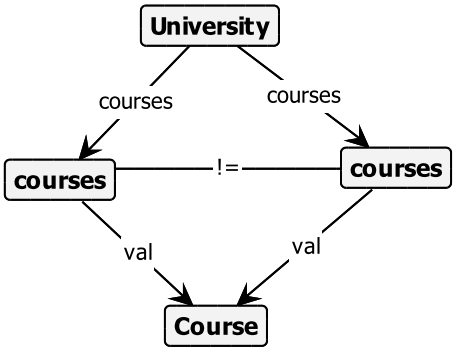


Figure 2.1: Example rule graph with only match nodes. This rule matches if a Student node has two different courses nodes pointing to the same Course node.

Figure 2.2: Example rule graph with creator edge. This rule matches if a Student node has no studyTime edge, and creates one to a PartTime node.

Figure 2.3: Example rule graph with deleter node. This rule matches a University node with a course edge connected to a Course node, and deletes both this edge and the Course node.

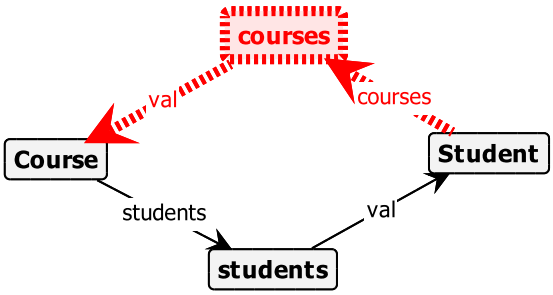


Figure 2.4: Example rule graph with NAC node. This rule matches only if a Course and Student node exist, which are connected by an intermediate students node, but *not* by an intermediate courses node.

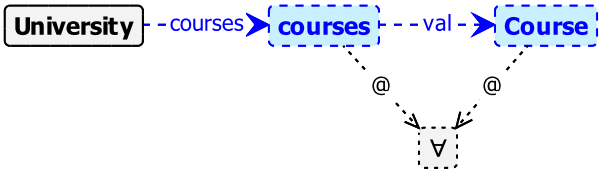


Figure 2.5: Example rule graph with quantifier node. This rule matches a University node and *all* course edges and Course nodes connected to this University node. These edges and Course nodes are then deleted.

A rule can be applied when it can be matched to the current instance graph. Such a match can occur only if all the nodes in the rule graph match. It is possible for a single rule to have multiple matches on any given instance graph. The application of these rule matches may result in any instance graph, possibly identical to another instance graph.

The following types of rule nodes are used by the new importer/exporter:

1. Figure 2.1 shows an example of a rule graph with *matching* nodes and edges, which are displayed with black thin lines. These nodes and edges match when their types occur in the current instance graph. They have no effect on the instance graph when the rule is applied and thus these nodes and edges are kept in the instance graph after the rule is applied. In the example, the rule would match if the instance graph contains a node of type University, two nodes of type courses and a node of type Course with corresponding matching edges between them. (The != edge is a special case which matches when the source and target nodes are not identical). This rule does not transform the instance graph in any way, but simply matches if a certain property holds, in this case if two different courses nodes have a val edge pointing to the same Course node.

2. Creator nodes and edges, displayed with thick (green) lines as seen in fig. 2.2, do not try to match any nodes or edges in the current instance graph (and thus always match), but rather create new nodes and edges of their given type in the transformed instance graph. In the given example, the rule would create a new courses node and a new Course node, with a new edge between them and between the courses and University nodes. This rule effectively adds a single course to a university.

3. Deleter nodes and edges, drawn with thin, dashed (blue) lines such as in fig. 2.3, match when their types occur in the current instance graph. When the rule is applied, these nodes and edges are removed from the instance graph. In the example, the courses edge, courses node, and Course node are first matched, and then removed from the instance graph. This rule effectively removes a single course from a university.

4. NAC (Negative Application Condition) nodes and edges are drawn with thick dashed (red) lines, as shown in fig. 2.4, and match when their types do *not* occur in the current instance graph. In the example, these nodes and edges would match if there does not exist a courses node with an incoming courses edge (from a Student node) and outgoing val edge (toward a Course node). This rule effectively checks if a students relation is not matched by an inverse courses relation.

5. Quantifier nodes, as used in fig. 2.5, allow nodes to be matched optionally and with different counts. Their exact definition is beyond the scope of this thesis, but a definition can be found in [33]. Quantifiers may be nested to arbitrary depth, which allows more complicated logic formulas to be used. In the example, the $\forall$ quantifier node will match any number of courses and Course nodes together. Since the University node will match only a single node in the instance graph, and there is a deleter edge that has to match between the courses nodes and the University node, only those courses (and Course) nodes will be matched that have such an incoming edge from a single University node. This rule effectively removes all courses from a single university.

A grammar can typically contains multiple active rule graphs. Each active rule is evaluated for the current instance graph. Such a rule may match zero or more times on the instance graph. Each match may be evaluated individually, resulting in a new instance graph. Thus, a state space can be generated by following the possible matches.

2.3 Conclusion

Having decided what languages are added to the importer and exporter, the answer to the first research question is readily available. The languages that are included are Ecore, GXL and DOT. Al-

though other interesting languages were found as well, due to limited time and resources only a selection could be included.

The formal description of the GROOVE graphs that are used to represent the various models gives a clear definition of the capabilities of these graphs and provides a foundation to define the mapping between the conceptual model and these graphs.

Chapter 3 Conceptual model

This chapter describes a single conceptual model used to formalize the various concepts found in the previously discussed languages. This model will be used as an intermediate form for a mapping between GROOVE and the various languages. It is based on the distinction between the type model level (which relates to the type graphs in GROOVE, accompanied by rule graphs for various constraints) and the instance model level (which relates to the instance graphs).

One of the main difficulties of translating an external language to GROOVE graphs is the fact that GROOVE has less expressive power than such a language in some cases[1]. For example, a language might impose certain constraints on a graph that GROOVE cannot directly represent. By using a conceptual model that has a larger expressive power than GROOVE, only a single mapping has to be made which has to overcome a large gap in expressive power (this problem will be addressed in chapter 4). The mapping between the various external languages and the conceptual model is considerably simpler, since their expressive power is more closely matched. Thus, all languages can reuse the more difficult mapping between the conceptual model and GROOVE and only have to define a mapping between the language itself and the conceptual model. See also fig. 3.1.

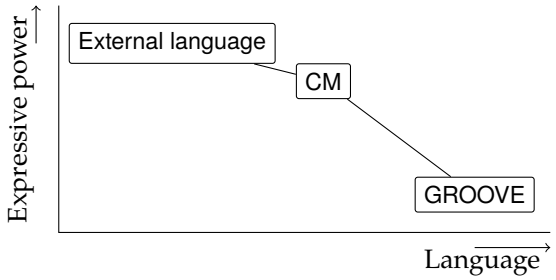


Figure 3.1: Difference in expressive power of the various importer/exporter components.

3.1 Global concepts

Before the type models and instance models can be defined, first some global concepts need to be introduced, which are used by both the instance models and type models.

In both the type model and the instance model some elements have to be identifiable by a name. For this a globally unique set of names is defined, of which a textual representation is written as exampleName.

Definition 1 (Name)

> *Name is a globally fixed set of names (shared between instance models and type models). This set contains at least the names* boolean, integer, real *and* string, *as well as* true, false *and* nil.

Just a set of names is not enough for most languages, as there usually also exists some structure within these names, often referred to as a namespace. Within a namespace, all names have to be unique, but

[1] Here, expressive power should be read as the capability to concisely represent a certain concept.

names can be shared between different namespaces. A combination of a namespace and a name is referred to as an identifier, and uniquely identifies anything within a type model or instance model. The namespace that is not contained within any other namespace is defined as $\perp$ and is also called the *root* namespace.

Definition 2 (Identifier/Namespace)

Identifiers and namespaces are defined as

$$\begin{aligned} \mathit{Id} &= \mathit{Namespace} \times \mathit{Name} \\ \mathit{Namespace} &= \mathit{Id} \cup \{\perp\} \end{aligned}$$

where the set of identifiers is the smallest solution of the given set of equations and $\perp$ is the root namespace.

Note that namespaces are recursively defined up until the root namespace $\perp$.

To write down identifiers a notation may be used which separates namespaces and names using dots, omitting the root namespace. For example, the identifier $\langle \perp, \mathsf{name} \rangle$ can be written as .name, and $\langle \langle \perp, \mathsf{name} \rangle, \mathsf{space} \rangle$ as .name.space.

To distinguish between the different types of data that may be present in an instance model, a set of data types is defined that models the different set of data values. These data types can be used in a type model, of which the values they model may be used in a corresponding instance model.

Definition 3 (Data types)

The set of data types is defined by $\mathit{DataType} = \{\mathsf{boolean}, \mathsf{integer}, \mathsf{real}, \mathsf{string}\}$

Definition 4 (Data type values)

For the various data types a single set defines the possible values. Besides $\mathbb{Z}$ and $\mathbb{R}$ for the number types, the following sets of values are defined:

- $\mathbb{B} = \{\mathsf{true}, \mathsf{false}\}$ *(the set of boolean values)*
- $\mathbb{C} =$ *The set of all printable characters*
- $\mathbb{S} = \mathbb{C}^*$ *(the set of possible strings)*

For certain types in a type model, a 'not assigned' value can be used, indicating there is no actual reference to any value in an instance model. This is defined as the nil value.

Definition 5 (Nil value)

nil defines the unassigned value for some types in the type model.

Definitions 8 and 16 specify in what context this value can be used.

In some cases, the number of assigned values that is allowed by a type model is limited. This is indicated by a multiplicity, which defines a *lower* bound and an *upper* bound. The lower bound defines the minimum number of values that can be assigned, and the upper bound defines the maximum number of values that can be assigned.

Definition 6 (Multiplicity)

A multiplicity is specified by a lower bound (which is any natural number) and an upper bound (which is possibly unbounded). The upper bound is defined by the set $\mathbb{M}$*, defined as* $\mathbb{M} = (\mathbb{N} \setminus \{0\}) \cup \{*\}$*, where* $*$ *means unbounded (that is to say,* $\forall n \in \mathbb{N} \colon n < *$*).*

The set of possible multiplicities is defined as

$$\mathit{Mult} = \mathbb{N} \times \mathbb{M}$$

where $\forall \langle x, y \rangle \in \mathit{Mult} \colon x \leq y$*.*

Members of *Mult* can be expressed as $x..y$, where x defines the lower bound and y defines the upper bound.

3.2 Type models

This section provides the formal definition of type models. This definition, along with other definitions, fully specifies the type model level to which the various language concepts of this level will be mapped.

A type model provides a set of definitions and constraints that describe a set of instance models, which may or may not be valid according to the type model. On the top level, it defines a set of *classes* of which instances (*objects*) may be used within an instance model. These classes contain a set of *fields* which are identified by a name (unique within the class) and are both typed and have a multiplicity. Class instances may assign values to these fields (specific for that instance) which must adhere to both the type and multiplicity of the field. The type model also defines an *inheritance* relation between these classes, which allows classes to inherit from other classes, providing a specialization of that class.

A type model also defines a set of *enumerations* and their values, which defines a unique type with a fixed set of values. Furthermore, a set of *constants* and their types define a symbolic typed value, which relates to a specific value in an instance model. A set of *custom data types* is also provided by the type model, which allows user defined data types to be represented.

Finally, a set of *properties* of the type model specifies the properties an instance model of this type model has to satisfy in order to be valid. These properties specify constraints on the values and structure of such an instance model.

The definition of a type model depends on the definition of various types. These types again depend on the definition of the type model. The solution to this cyclic dependency is the smallest solution to the set of equations given for the types and type model.

The suffix Tm is used when the definition of something depends on any type model Tm, for example $\mathit{Class}_{\mathrm{Tm}}$.

Definition 7 (Type model)

A single type model is defined as

$\mathit{Tm} = \langle \mathit{Class}, \mathit{Enum}, \mathit{UserDataType}, \mathit{Field}, \mathrm{FieldSig}, \mathit{EnumValue}, \mathit{Inh}, \mathit{Prop}, \mathit{Constant}, \mathrm{ConstType} \rangle$

with

- $\mathit{Class} \subseteq \mathit{Id}$ *is the set of classes in Tm.*
- $\mathit{Enum} \subseteq \mathit{Id}$ *is the set of enumerations in Tm.*

- *UserDataType* $\subseteq$ *Id is the set of custom data types in Tm.*
- *Field* $\subseteq$ (*Class* $\times$ *Name*) *is the set that maps a class to a set of field names in Tm.*
- $\mathrm{FieldSig}$: *Field* $\rightarrow$ ($Type_{Tm}$ $\times$ *Mult*) *is the function that maps fields to their type (as defined in definition 8) and multiplicity (as defined in definition 6) in Tm.*
- *EnumValue* $\subseteq$ *Enum* $\times$ *Name is the set of possible values for the enumerations in Tm.*
- *Inh* $\subseteq$ *Class* $\times$ *Class is the direct inheritance relation between classes in Tm.*
- *Prop* $\subseteq$ $Property_{Tm}$ *is the set of properties that apply to Tm (see definition 11).*
- *Constant* $\subseteq$ *Id is the set that contains all possible constants that may be used as a (symbolic) default value.*
- $\mathrm{ConstType}$: *Constant* $\rightarrow$ $Type_{Tm}$ *is the function that maps constants to their respective types.*

where

- *Class, Enum, UserDataType and DataType are pairwise disjoint.*
- *None of the elements in Class* $\cup$ *Enum* $\cup$ *DataType* $\cup$ *UserDataType may be in the namespace of another element in that set.*
- *Inh is an asymmetric relation, of which the transitive closure is irreflexive.*

An example type model is depicted in fig. 3.2. It shows three classes (Person, Student and Course) and a single enumeration (StudyTime). The Person class has two fields, age and name. The Student class has a field follows, which is depicted as a relation between the Student and Course classes. The Student class also has a field studyTime, which makes use of the StudyTime enumeration. The Course class has a field subject, and a field students. Finally, there is an inheritance relation between the classes Person and Student (Student inherits from Person).

The fields in a type model are always associated with a specific type, which defines the set of possible values that may be assigned in an instance model. The possible types are defined by the set of data types, classes and enumerations. The set of types also consists of various aggregations of these types, namely *containers* or *tuples*. Containers provide types for multiple values of the same type, but of which the values may differ in number and ordering. Tuples provide types for the combination of values for arbitrary types, of which the numbering and ordering is always fixed.

Definition 8 (Types)

Given any type model Tm, the set of types is defined as

$$Type_{Tm} = DataType \cup ClassType_{Tm} \cup Enum_{Tm} \cup UserDataType_{Tm} \cup Container_{Tm} \cup TupleType_{Tm}$$

The $ClassType_{Tm}$ *set defines both a set of* nullable *and* proper *classes. Nullable classes are classes for which the* nil *(see definition 5) value is valid, and proper classes are those classes for which the* nil *value is not valid (hence both sets of classes are disjoint). The* $ClassType_{Tm}$ *set is defined as*

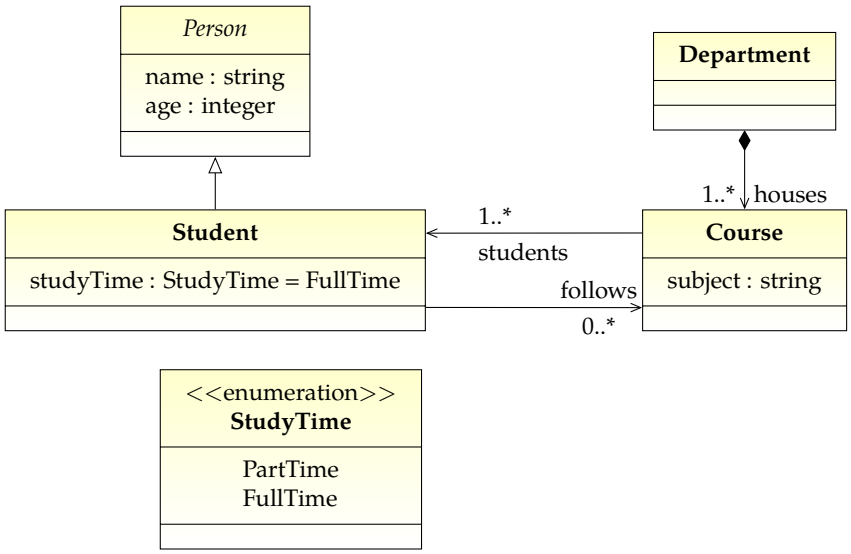
$$ClassType_{Tm} = \{\mathit{nullable}, \mathit{proper}\} \times Class_{Tm}$$

Given a $C \in Class_{Tm}$, $?C = \langle \mathit{nullable}, C \rangle$ *is the nullable variant of C, and* $!C = \langle \mathit{proper}, C \rangle$ *the proper variant.*

A container is a type that may contain multiple values in an instance. Containers define the type of values they contain, and the multiplicity of the container. They are defined by

$$Container_{Tm} = \{\mathit{bagof}, \mathit{setof}, \mathit{seqof}, \mathit{ordof}\} \times Type_{Tm}$$

For the interpretation of the values in $\{\mathit{bagof}, \mathit{setof}, \mathit{seqof}, \mathit{ordof}\}$*, see definition 14.*

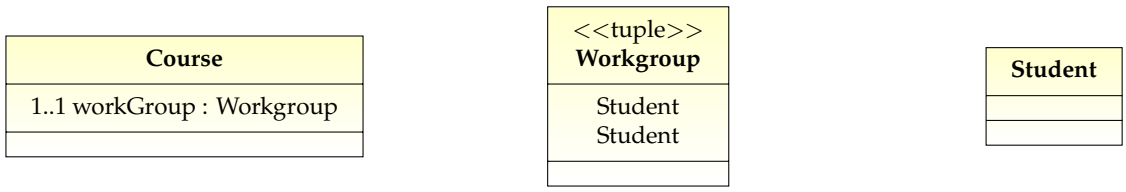


(a) Graphical representation of a type model

$$
\begin{aligned}
\mathit{Class}_{\mathrm{Tm}} &= \{\text{.Person}, \text{.Student}, \text{.Course}, \text{.Department}\} \\
\mathit{Enum}_{\mathrm{Tm}} &= \{\text{.StudyTime}\} \\
\mathit{Field}_{\mathrm{Tm}} &= \{\langle \text{.Person}, \text{age}\rangle, \langle \text{.Person}, \text{name}\rangle, \langle \text{.Student}, \text{studyTime}\rangle, \langle \text{.Student}, \text{follows}\rangle, \\
&\quad \langle \text{.Course}, \text{subject}\rangle, \langle \text{.Course}, \text{students}\rangle, \langle \text{.Department}, \text{houses}\rangle\} \\
\mathrm{FieldSig}_{\mathrm{Tm}} &= \{\langle\langle \text{.Person}, \text{age}\rangle, \langle \text{integer}, 1..1\rangle\rangle, \langle\langle \text{.Person}, \text{name}\rangle, \langle \text{string}, 1..1\rangle\rangle, \\
&\quad \langle\langle \text{.Student}, \text{studyTime}\rangle, \langle \text{.StudyTime}, 1..1\rangle\rangle, \\
&\quad \langle\langle \text{.Student}, \text{follows}\rangle, \langle \text{bagof}, \langle \text{!.Course}\rangle, 0..*\rangle\rangle, \langle\langle \text{.Course}, \text{subject}\rangle, \langle \text{string}, 1..1\rangle\rangle, \\
&\quad \langle\langle \text{.Course}, \text{students}\rangle, \langle\langle \text{bagof}, \text{!.Student}\rangle, 1..*\rangle\rangle, \\
&\quad \langle\langle \text{.Department}, \text{houses}\rangle, \langle\langle \text{bagof}, \text{!.Course}\rangle, 0..*\rangle\rangle\} \\
\mathit{EnumValue}_{\mathrm{Tm}} &= \{\langle \text{.StudyTime}, \text{FullTime}\rangle, \langle \text{.StudyTime}, \text{PartTime}\rangle\} \\
\mathit{Inh}_{\mathrm{Tm}} &= \{\langle \text{.Student}, \text{.Person}\rangle\} \\
\mathit{Prop}_{\mathrm{Tm}} &= \{\langle \text{abstract}, \text{.Person}\rangle, \langle \text{keyset}, \text{follows}, \{\langle \text{.Course}, \text{subject}\rangle\}\rangle, \\
&\quad \langle \text{identity}, \{\langle \text{.Person}, \text{age}\rangle, \langle \text{.Person}, \text{name}\rangle\}\rangle, \\
&\quad \langle \text{containment}, \langle \text{.Department}, \text{houses}\rangle\rangle, \\
&\quad \langle \text{opposite}, \langle \text{.Student}, \text{follows}\rangle, \langle \text{.Course}, \text{students}\rangle\rangle, \\
&\quad \langle \text{opposite}, \langle \text{.Course}, \text{students}\rangle, \langle \text{.Student}, \text{follows}\rangle\rangle, \\
&\quad \langle \text{defaultValue}, \langle \text{.Student}, \text{studyTime}\rangle, \text{.Constant.StudyTime.FullTime}\rangle\} \\
\mathit{Constant}_{\mathrm{Tm}} &= \{\text{.Constant.StudyTime.FullTime}, \text{.Constant.StudyTime.PartTime}\} \\
\mathrm{ConstType}_{\mathrm{Tm}} &= \{\langle \text{.Constant.StudyTime.FullTime}, \text{.StudyTime}\rangle\}
\end{aligned}
$$

(b) Formal description of a type model

Figure 3.2: Example type model.



(a) Graphical representation of a type model with a tuple.

$$\begin{aligned} \mathit{Class}_{\mathrm{Tm}} &= \{\mathsf{.Student}, \mathsf{.Course}\} \\ \mathit{Field}_{\mathrm{Tm}} &= \{\langle \mathsf{.Course}, \mathsf{workGroups} \rangle\} \\ \mathrm{FieldSig}_{\mathrm{Tm}} &= \{\langle \langle \mathsf{.Course}, \mathsf{workGroup} \rangle, \langle \langle \mathsf{tuple}, \langle \mathsf{.Student}, \mathsf{.Student} \rangle \rangle, 1..1 \rangle \rangle\} \end{aligned}$$

(b) (Partial) formal description of a type model with a tuple.

Figure 3.3: Example of a very simple type model with a tuple.

$\mathit{TupleType}_{Tm}$ is the set of types that is defined by an ordered list of types (which may be equal). It is defined by

$$\mathit{TupleType}_{Tm} = \{\mathsf{tuple}\} \times \mathit{Type}_{Tm}{}^{*}$$

The set of types is recursively defined as the smallest solution of the given set of equations for Type_{Tm}, $\mathit{Container}_{Tm}$ and $\mathit{TupleType}_{Tm}$.

Note that the $\mathit{TupleType}_{\mathrm{Tm}}$ set defines tuples with a single cardinality from $\mathbb{N}$, which means the number of values is always fixed (as opposed to containers which do not impose any limit on the amount of values they contain). The example in fig. 3.3 shows a type model that defines a class Course with a single field of which the type is a tuple. This tuple consists of two Students. Note that the elements of a tuple are not fields (even though they appear similar in the example) as can be seen by the formal definition of the type model.

In the example in fig. 3.2, the various fields make use of different types. The fields depicted as a relation between two classes are actually of a type based on the $\mathit{Class}_{\mathrm{Tm}}$ set. For example, the students field of class Course is typed by the Student class. As no further constraints are given for this container relation, it can be assumed to be a bagof container, where ordering does not matter and multiple identical values are allowed. Additionally, this type is a container type, as it represents multiple Students. The studyTime field of the Student class is typed by the StudyTime enumeration. Finally, the Person class shows two fields that are typed by some of the data types available, integer and string for the age and name fields respectively.

Definition 9 (Field)

Given any type model Tm, the Field_{Tm} relation defines a binary relation between classes and fields. To retrieve the set of fields for a given class (and the fields inherited from superclasses), the following function is defined:

$$\mathrm{fields}\colon \mathit{Class}_{Tm} \to \mathcal{P}(\mathit{Field}_{Tm})$$

such that

$$\mathrm{fields}_{Tm}(c) = \{f \in \mathit{Field}_{Tm} \mid f = \langle c', n \rangle \wedge c \sqsubseteq_{Tm} c'\}$$

Given any type model Tm, the $\mathrm{FieldSig}_{Tm}$ function defines a mapping between fields and their attributes. To retrieve the various components of this function, the following mappings are defined

- $\mathrm{class}\colon \mathit{Field}_{Tm} \to \mathit{Class}_{Tm}$
- $\mathrm{type}\colon \mathit{Field}_{Tm} \to \mathit{Type}_{Tm}$
- $\mathrm{lower}\colon \mathit{Field}_{Tm} \to \mathbb{N}$

$Type_{\mathrm{Tm}}$	Multiplicity
$\{\mathsf{proper}\} \times Class_{\mathrm{Tm}}$	1..1
$\{\mathsf{nullable}\} \times Class_{\mathrm{Tm}}$	0..1
$Container_{\mathrm{Tm}}$	x..y ($0 \leq x \leq y \wedge 1 \leq y$)
$DataType$	1..1
$Enum_{\mathrm{Tm}}$	1..1
$UserDataType_{\mathrm{Tm}}$	1..1
$TupleType_{\mathrm{Tm}}$	1..1

Table 3.1: Possible multiplicities for types.

- $\mathrm{upper}\colon Field_{Tm} \to \mathbb{M}$

such that

- $\mathrm{class}_{Tm}(f) = class$ *if and only if* $f = \langle class, name \rangle$
- $\mathrm{lower}_{Tm}(f) = lower$ *if and only if* $\mathrm{FieldSig}_{Tm}(f) = \langle type, \langle lower, upper \rangle \rangle$
- $\mathrm{upper}_{Tm}(f) = upper$ *if and only if* $\mathrm{FieldSig}_{Tm}(f) = \langle type, \langle lower, upper \rangle \rangle$

Fields can be separated into relation and attribute sets, where attributes reference (containers or tuples of) data types, user data types and enumerations, and relations reference other types. They are defined by

- $Attr_{Tm} = \{f \in Field_{Tm} | \, \mathrm{type}(f) \in (DataType_{Tm} \cup Enum_{Tm} \cup UserDataType_{Tm}) \vee \mathrm{type}(f) \in \{\mathsf{setof}, \mathsf{bagof}, \mathsf{ordof}, \mathsf{seqof}\} \times (DataType_{Tm} \cup Enum_{Tm} \cup UserDataType_{Tm})\}$
- $Rel_{Tm} = Field_{Tm} \setminus Attr_{Tm}$

Note that fields with a tuple type are not considered relations, but attributes. This is because even though tuples can contain classes, they are not classes themselves.

Taking for example the field follows from the Student class in the type model example, the following properties can be identified: The type refers to Course, which is an element of the $Class_{\mathrm{Tm}}$ set. The lower and upper values are 0 and * respectively (which means it can be an arbitrary number of courses). Furthermore, the follows field is an element of the Rel_{Tm} set (it is a class container type) and part of the $\mathrm{fields}_{\mathrm{Tm}}(\mathsf{Student})$ set, which is $\{\mathsf{studyTime}, \mathsf{follows}\}$.

The various types have an underling subtype relation, which generalizes inheritance. A subtype defines a specialization of a supertype, and as such all values valid for the subtype are also valid for the supertype.

Definition 10 (Subtypes)

Given any type model Tm, $\sqsubseteq_{Tm} \subseteq Type_{Tm} \times Type_{Tm}$ *defines the subtype relation, which is a generalization of inheritance (*Inh_{Tm}*). It is a reflexive partial order relation, for which the following rules can be defined (with* $t_1, t_2, t_3 \in Type_{Tm}$*,* c_1 *and* $c_2 \in Class_{Tm}$*):*

Transitivity:

$$\frac{t_1 \sqsubseteq_{Tm} t_2 \qquad t_2 \sqsubseteq_{Tm} t_3}{t_1 \sqsubseteq_{Tm} t_3}$$

Reflexivity

$$\frac{}{t_1 \sqsubseteq_{Tm} t_1}$$

Generalization of inheritance

$$\frac{c_1 \,\mathrm{Inh}_{Tm}\, c_2}{?\, c_1 \sqsubseteq_{Tm} ?\, c_2} \qquad \frac{c_1 \,\mathrm{Inh}_{Tm}\, c_2}{!\, c_1 \sqsubseteq_{Tm} !\, c_2}$$

Nullable/Proper classes

$$\frac{}{!\, c_1 \sqsubseteq_{Tm} ?\, c_1}$$

Thus, in the example, $\langle \mathsf{nullable}, \mathsf{Student}\rangle \sqsubseteq_{\mathrm{Tm}} \langle \mathsf{nullable}, \mathsf{Person}\rangle$ (since Student $\mathrm{Inh}_{\mathrm{Tm}}$ Person). Furthermore, it also holds that $\langle \mathsf{proper}, \mathsf{Student}\rangle \sqsubseteq_{\mathrm{Tm}} \langle \mathsf{nullable}, \mathsf{Student}\rangle$, as a proper class is a subtype of a nullable class.

A type model may specify a set of properties which specify a set of constraints, which an instance model has to *satisfy* in order to be valid. The following properties are defined:

- The abstract property. This property, specified for a specific class in the type model, forbids the instantiation of that class in any instance model. As such, it is satisfied when no object exists in an instance model which is an instance of that class.

- The readonly property. This property, specified for a field in the type model, forbids a new value being assigned to the field in any instance model. This property only affects the possible transformations of an instance model, and is always satisfied for any specific instance model.

- The keyset property. This property is specified for a set of attributes of a class and a relation towards that class. It ensures that each instance of that class within the given relation, can uniquely be identified by the values of the set of attributes. It is satisfied when two objects have pairwise identical values for these attributes, they must be the same object if they are the target of the given relation.

- The identity property. This property is specified for a set of attributes. Similar to the keyset property, the values of this set of attributes uniquely identify any object of the class of the attributes, though globally in an instance model rather than within a single relation as is the case of the keyset property. As such, it is satisfied when no two objects exist in an instance model that are an instance of the class and that have pairwise the same value for all the attributes.

- The containment property. This property, specified for a relation, states that all objects that are the target of this relation are contained within a single source object. Objects can be contained by at most one other object, and contained cycles are not allowed. When an instance model transformation removes a container object, all contained objects are removed as well. The constraint is satisfied when an object is the target of no more than one containment relation, and there exists no cycle between containment relations.

- The opposite property. This property specifies that two relations are opposite of each other, which means that for each instance of one of those relations, another instance exists that has an opposite target and source. It is satisfied when for each pair of objects, for each relation that exists between these objects, a reverse relations exists if both these relations are opposite.

- The defaultValue property. This property specifies a default value for a field which has not been explicitly assigned a value in an isntance model. It specifies a constant for a field, which represents a value in an instance model. It is always satisfied, and influences the behavior of possible model transformations.

Definition 11 (Type model Properties)

For a type model Tm a set of properties $Property_{Tm}$ is defined which contains all the possible properties. This set is defined as

$$\begin{aligned} Property_{Tm} = &\{\langle \textit{abstract}, c\rangle | c \in Class_{Tm}\} \cup \{\langle \textit{readonly}, f\rangle | f \in Field_{Tm}\} \\ &\cup \{\langle \textit{keyset}, r, A\rangle | r \in Rel_{Tm} \wedge A \subseteq Attr_{Tm}\} \cup \{\langle \textit{identity}, A\rangle | A \subseteq Attr_{Tm}\} \\ &\cup \{\langle \textit{containment}, r\rangle | r \in Rel_{Tm}\} \cup \{\langle \textit{opposite}, r, r'\rangle | r, r' \in Rel_{Tm} \wedge r \neq r'\} \\ &\cup \{\langle \textit{defaultValue}, f, v\rangle | f \in Field_{Tm} \wedge v \in Constant_{Tm}\} \end{aligned}$$

where

- $\langle \textit{identity}, A\rangle$ *is defined such that* $\exists c \in Class_{Tm} : \forall \langle ac, an\rangle \in A : c = an$.
- $\langle \textit{keyset}, r, A\rangle$ *is defined such that* $\forall \langle ac, an\rangle \in A : \mathrm{type}_{Tm}(r) = an$, *and* $\mathrm{type}_{Tm}(r) \in (\{\textit{setof}, \textit{ordof}\} \times ClassType_{Tm})$.
- $\langle \textit{opposite}, r, r'\rangle$ *is defined such that when* $r = \langle c1, n1\rangle$, $r' = \langle c2, n2\rangle$, *then* $c1 = \mathrm{type}_{Tm}(r') \wedge c2 = \mathrm{type}_{Tm}(r)$, $\mathrm{type}_{Tm}(r) \notin \{\textit{bagof}, \textit{seqof}\} \times Type_{Tm}$ *and finally* $\mathrm{type}_{Tm}(r') \notin \{\textit{bagof}, \textit{seqof}\} \times Type_{Tm}$ *(containers must have unique values).*
- $\langle \textit{defaultValue}, f, v\rangle$ *is defined such that* $type(f) = \mathrm{ConstType}_{Tm}(v)$.

The example also shows a few properties:

- The Person class is declared abstract (indicated by an italicized font for its name).
- The houses relation of the Department class is a containment relation (shown by the filled diamond at one end of the containment relation).
- The follows and students relations are opposites (as indicated by the formal definition).
- The Course subject is the keyset of the follows relationship (as indicated by the formal definition). This means that a student can follow only one Course on a specific subject at any given time.
- The name and age fields of a Person are its identity (as indicated by the formal definition). Although these details are most likely not unique in the world, they may be in the given instance model.
- Finally, the field studyTime has a default value of FullTime.

With the given definitions, it is possible to define an inconsistent type model. Such a type model is correct according to the given definitions, but does not specify any valid instance model. For example, because of conflicting properties or multiplicities. Therefore, a definition is given for a *consistent* type model, based on the consistency of the multiplicities and properties within the type model.

Definition 12 (Type model consistency)

The multiplicities of the fields in $Field_{Tm}$ are consistent *if it holds that:*

$$\mathrm{lower}_{Tm}(f) = \begin{cases} 1, & \textit{if } \mathrm{type}(f) \in DataType \cup Enum_{Tm} \cup UserDataType \cup TupleType_{Tm} \\ & \qquad \cup (\textit{proper} \times Class_{Tm}) \\ 0, & \textit{if } \mathrm{type}(f) \in (\textit{nullable} \times Class_{Tm}) \\ 0 \textit{ or more}, & \textit{otherwise} \end{cases}$$

$$\mathrm{upper}_{Tm}(f) = \begin{cases} 1 \textit{ or more}, & \textit{if } \mathrm{type}(f) \in Container_{Tm} \\ 1, & \textit{otherwise} \end{cases}$$

as indicated by table 3.1.

The properties in $Prop_{Tm}$ *are* consistent *if the following holds:*

- $\langle \textit{keyset}, r, A\rangle \in Prop_{Tm} \wedge \langle \textit{keyset}, r, A'\rangle \in Prop_{Tm} \Rightarrow A = A'$ *(unique for r).*
- $\langle \textit{opposite}, r, r'\rangle \in Prop_{Tm} \wedge \langle \textit{opposite}, r, r''\rangle \in Prop_{Tm} \Rightarrow r' = r''$ *(unique for r).*
- $\langle \textit{opposite}, r, r'\rangle \in Prop_{Tm} \Leftrightarrow \langle \textit{opposite}, r', r\rangle \in Prop_{Tm}$ *(symmetry).*
- $\langle \textit{defaultValue}, f, v\rangle \in Prop_{Tm} \wedge \langle \textit{defaultValue}, f, v'\rangle \in Prop_{Tm} \Rightarrow v = v'$ *(unique for f).*
- $\langle \textit{containment}, r\rangle \wedge \langle \textit{opposite}, r, r'\rangle \Rightarrow \mathrm{upper}_{Tm}(r') = 1$ *(opposite of containment relation must have upper bound of 1).*

Then a type model Tm is consistent *if and only if*

- *The multiplicities of all fields in* $Field_{Tm}$ *are consistent.*
- *All properties in* $Prop_{Tm}$ *are consistent.*

3.3 Instance models

An instance model represents an instance of a type model. An instance model consists of a set of objects, which are mapped to both a class they instantiate and an optional identifier. All objects are an instance of a specific class, and as such are typed by that class (and implicitly its superclasses). An instance model also specifies the values for each field of an object. The set of fields for each object is determined by its class and all its superclasses, which are defined by the subtype relation of the type model. Finally, the instance model specifies a set of default values, which maps the constants from the type model to actual values in the instance model, allowing default values to be assigned to fields.

As with the type model and type definitions, there is a cyclic dependency between instance models and values. In the same manner, the solution is set to be the smallest solution to the set of equations for the instance model and values.

The suffix Im is used when the definition of something depends on any instance model Im, which itself depends on the definition of any type model Tm. For example $Object_{\mathrm{Im}}$.

Definition 13 (Instance Model)

For a type model $Tm = \langle Class, Enum, UserDataType, Field, \mathrm{FieldSig}, EnumValue, Inh, Prop, Constant, \mathrm{ConstType}\rangle$*, a single instance model Im is a tuple*

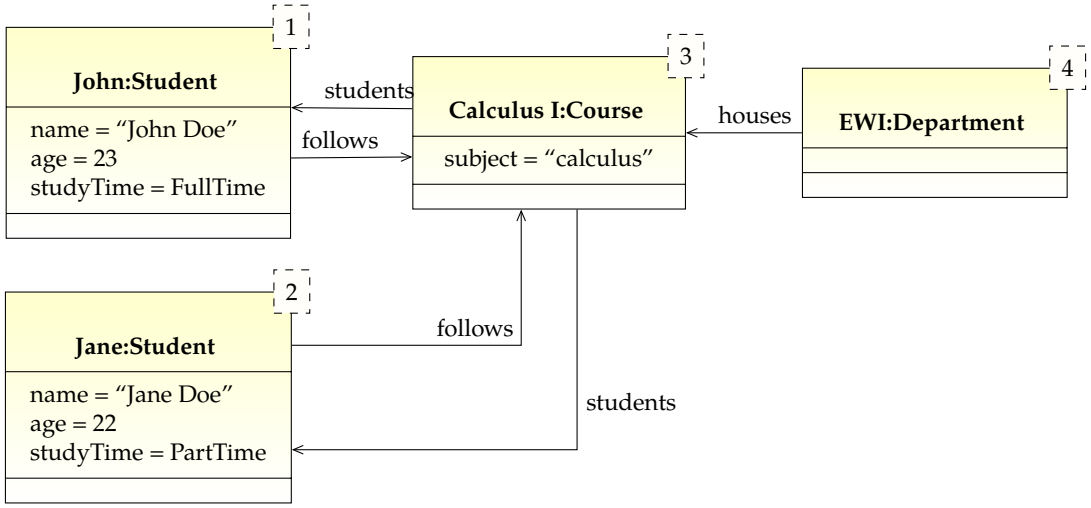
$$Im = \langle Object, \mathrm{ObjectClass}, \mathrm{ObjectId}, \mathrm{FieldValue}, \mathrm{DefaultValue}\rangle$$

where

- *Object is the set of objects (class instances) in Im.*
- $\mathrm{ObjectClass}$: $Object \to Class_{Tm}$ *is the function that maps each object in Im to a class.*
- $\mathrm{ObjectId}$: $Object \hookrightarrow Name$ *is the partial function that maps each object in Im to a unique identifier.*
- $\mathrm{FieldValue}$: $(Object \times Field_{Tm}) \hookrightarrow Value_{Im}$ *is the partial function between each* $Field_{Tm}$ *of an* $Object_{Im}$ *and a* $Value_{Im}$ *(see definition 14).*
- $\mathrm{DefaultValue}$: $Constant_{Tm} \to Value_{Im}$ *is the function that maps the constants from the type model to actual values in the instance model.*

where

- $\forall \langle o, n\rangle, \langle o', n'\rangle \in \mathrm{ObjectId} : n = n' \Rightarrow o = o'$



(a) Graphical representation of an instance model.

$$
\begin{aligned}
\mathit{Object}_{\mathrm{Im}} &= \{1, 2, 3, 4\} \\
\mathit{ObjectId}_{\mathrm{Im}} &= \{\langle 1, .\mathsf{John}\rangle, \langle 2, .\mathsf{Jane}\rangle, \langle 3, .\mathsf{Calculus\ I}\rangle, \langle 4, .\mathsf{EWI}\rangle\} \\
\mathit{FieldValue}_{\mathrm{Im}} &= \{\langle\langle 1, \langle .\mathsf{Person}, \mathsf{age}\rangle\rangle, \langle \mathsf{int}, 23\rangle\rangle, \\
&\quad \langle\langle 1, \langle .\mathsf{Person}, \mathsf{name}\rangle\rangle, \langle \mathsf{string}, \text{“John Doe”}\rangle\rangle, \\
&\quad \langle\langle 1, \langle .\mathsf{Student}, \mathsf{studyTime}\rangle\rangle, \langle \mathsf{enum}, \langle .\mathsf{StudyTime}, \mathsf{FullTime}\rangle\rangle\rangle, \\
&\quad \langle\langle 1, \langle .\mathsf{Student}, \mathsf{follows}\rangle\rangle, \langle \mathsf{bagof}, \langle\langle \mathsf{obj}, 3\rangle\rangle\rangle\rangle, \\
&\quad \langle\langle 2, \langle .\mathsf{Person}, \mathsf{age}\rangle\rangle, \langle \mathsf{int}, 22\rangle\rangle, \\
&\quad \langle\langle 2, \langle .\mathsf{Person}, \mathsf{name}\rangle\rangle, \langle \mathsf{string}, \text{“Jane Doe”}\rangle\rangle, \\
&\quad \langle\langle 2, \langle .\mathsf{Student}, \mathsf{studyTime}\rangle\rangle, \langle \mathsf{enum}, \langle .\mathsf{StudyTime}, \mathsf{PartTime}\rangle\rangle\rangle, \\
&\quad \langle\langle 2, \langle .\mathsf{Student}, \mathsf{follows}\rangle\rangle, \langle \mathsf{bagof}, \langle\langle \mathsf{obj}, 3\rangle\rangle\rangle\rangle, \\
&\quad \langle\langle 3, \langle .\mathsf{Course}, \mathsf{subject}\rangle\rangle, \langle \mathsf{string}, \text{“calculus”}\rangle\rangle, \\
&\quad \langle\langle 3, \langle .\mathsf{Course}, \mathsf{students}\rangle\rangle, \langle \mathsf{bagof}, \langle\langle \mathsf{obj}, 1\rangle, \langle \mathsf{obj}, 2\rangle\rangle\rangle\rangle, \\
&\quad \langle\langle 4, \langle .\mathsf{Department}, \mathsf{houses}\rangle\rangle, \langle \mathsf{bagof}, \langle\langle \mathsf{obj}, 3\rangle\rangle\rangle\rangle\} \\
\mathit{DefaultValue}_{\mathrm{Im}} &= \{\langle .\mathsf{Constant.StudyTime.FullTime}, \langle \mathsf{enum}, \langle .\mathsf{StudyTime}, \mathsf{FullTime}\rangle\rangle\rangle\}
\end{aligned}
$$

(b) Formal description of an instance model.

Figure 3.4: Example instance model.

- $\forall o \in \mathit{Object}, f \in \mathit{Field}_{Tm}\colon \langle o, f\rangle \in \operatorname{dom} \mathit{FieldValue} \Leftrightarrow \mathit{ObjectClass}(o) \sqsubseteq_{Tm} \operatorname{class}(f)$.

An example model is represented by fig. 3.4. It is based on the type model from the example in fig. 3.2. It shows two instantiations of the Student class: the John and Jane objects. Furthermore, there is one instantiation of the Course class (Calculus I) and one instantiation of the department class (EWI). The $\mathrm{ObjectClass}_{\mathrm{Im}}$ of each object in the instance model is represented by the text after the colon in the header of each object. The $\mathrm{ObjectId}_{\mathrm{Im}}$ is represented by the text preceding the colon. The Student objects have values assigned for all fields, including the fields of the classes they inherit from. This also holds for the Course object. For attributes, this is represented by an assignment, such as age = 23 (meaning the value 23 is assigned to the age field). For relations, this is represented by an arrow from one object to another, with the name of the field shown on the arrow.

Note that the objects themselves are represented by elements from $\mathbb{N}$. The conceptual model does not give a concrete specification for elements in the $\mathit{Object}_{\mathrm{Im}}$ set, but by convention objects (or in graph terms, nodes) are represented by numbers.

For each instance model a set of possible values is defined by the values for all data types, the possible enumerations of the type model and the objects in the instance model. Each value has a symbol that defines its type, allowing the values in an instance model to be typed by the types in the type model. This also allows values with identical content but different type to be separated. For example, any value in $\mathbb{Z} \cap \mathbb{R}$ (which can be of type integer or real). Multiple values can be aggregated by container values and tuple values, which are typed by container and tuple types respectively.

Definition 14 (Values)

Given any instance model Im, the set of values is Value_{Im}.
The set of values is then defined as

$$\mathit{Value}_{Im} = \mathit{AtomValue}_{Im} \cup \mathit{ContainerValue}_{Im} \cup \mathit{TupleValue}_{Im}$$

with

- $\mathit{AtomValue}_{Im} = \mathit{ClassValue}_{Im} \cup \mathit{LiteralValue} \cup (\{\mathit{enum}\} \times \mathit{EnumValue}_{Tm}) \cup (\{\mathit{data} \times \mathbb{S}\})$
- $\mathit{LiteralValue} = (\{\mathit{bool}\} \times \mathbb{B}) \cup (\{\mathit{int}\} \times \mathbb{Z}) \cup (\{\mathit{real}\} \times \mathbb{R}) \cup (\{\mathit{string}\} \times \mathbb{S})$
- $\mathit{ClassValue}_{Im} = \{\mathit{obj}\} \times (\mathit{Object}_{Im} \cup \{\mathit{nil}\})$
- $\mathit{ContainerValue}_{Im} = \{\mathit{setof}, \mathit{bagof}, \mathit{seqof}, \mathit{ordof}\} \times \mathit{Value}_{Im}{}^{*}$
  (where $\mathit{Value}_{Im}{}^{*}$ *allows containers to recursively contain other containers.)*
- $\mathit{TupleValue}_{Im} = \{\mathit{tuple}\} \times \mathit{Value}_{Im}{}^{*}$

The set of values is recursively defined as the smallest solution of the given set of equations for Value_{Im}, $\mathit{ContainerValue}_{Im}$ *and* $\mathit{TupleValue}_{Im}$.

For custom data types, the value is an element from the set $\mathbb{S}$. This way, these data types can be considered opaque, with the value being a string representation of the data type. Thus, the data type can be stored in the conceptual model, but it cannot be further interpreted.

Containers attributed as setof or ordof are considered to have unique values, whereas containers attributed as bagof or seqof are not. This means for example that a tuple with two or more identical values is not a valid value for a container attributed as setof or ordof, see also definition 16.

Additionally, the values of a container attributed as bagof or setof are considered unordered, and seqof or ordof ordered. This affects the equivalency of containers, as defined in definition 15.

In the example, the set of atomic values that are assigned consists of $\{\langle \text{string}, \text{“John Doe”}\rangle, \langle \text{string}, \text{“Jane Doe”}\rangle, \langle \text{string}, \text{“calculus”}\rangle, \langle \text{int}, 22\rangle, \langle \text{int}, 23\rangle, \langle \text{enum}, \text{PartTime}\rangle, \langle \text{enum}, \text{FullTime}\rangle\}$. Note that none of

the objects are in an atomic assigned value, all the relations in the type model are container types, and as such all the objects are contained in a container value as well. For example, the container value for the students field of the Course objects is $\langle \mathsf{bagof}, \langle\langle \mathsf{obj}, 1\rangle, \langle \mathsf{obj}, 2\rangle\rangle\rangle$ (in no particular order, as the relation is of a bag container type).

For each instance model a value equivalence relation is also defined. This relation allows aggregate values to be compared, and more specifically defines equivalency for unordered container values.

Definition 15 (Value equivalency)

Two values are equivalent ($\equiv \subseteq Value_{Im} \times Value_{Im}$) if both the type is identical and the actual value content is equivalent. It consists of the smallest reflexive relation between values and the relations defined by the rules given next.

For atomic values equivalence is defined as

$$\frac{v_1 \in Value_{Im} \qquad v_2 \in Value_{Im} \qquad v_1 = v_2}{v_1 \equiv v2}$$

Sequences and ordered sets are equivalent if the values in their tuples are pairwise equivalent.

Sequence container equivalency

$$\frac{c_1 = \langle \mathsf{seqof}, \langle v_1, \cdots, v_n\rangle\rangle \qquad c_2 = \langle \mathsf{seqof}, \langle u_1, \cdots, u_n\rangle\rangle \qquad v_1 \equiv u_1, \cdots, v_n \equiv u_n}{c_1 \equiv c_2}$$

Ordered set container equivalency

$$\frac{c_1 = \langle \mathsf{ordof}, \langle v_1, \cdots, v_n\rangle\rangle \qquad c_2 = \langle \mathsf{ordof}, \langle u_1, \cdots, u_n\rangle\rangle \qquad v_1 \equiv u_1, \cdots, v_n \equiv u_n}{c_1 \equiv c_2}$$

Sets and bags are equivalent if there exists a bijective function which maps elements from one set/bag to the other, such that the mapped values are equivalent. Set container equivalency

$$\frac{c_1 = \langle \mathsf{setof}, \langle v_1, \cdots, v_n\rangle\rangle \qquad c_2 = \langle \mathsf{setof}, \langle u_1, \cdots, u_n\rangle\rangle \qquad \exists\, \mathrm{f} \colon \{1, \cdots, n\} \rightarrowtail\!\!\!\!\!\rightarrow \{1, \cdots, n\} \colon v_i \equiv u_{\mathrm{f}(i)}}{c_1 \equiv c_2}$$

Bag container equivalency

$$\frac{c_1 = \langle \mathsf{bagof}, \langle v_1, \cdots, v_n\rangle\rangle \qquad c_2 = \langle \mathsf{bagof}, \langle u_1, \cdots, u_n\rangle\rangle \qquad \exists\, \mathrm{f} \colon \{1, \cdots, n\} \rightarrowtail\!\!\!\!\!\rightarrow \{1, \cdots, n\} \colon v_i \equiv u_{\mathrm{f}(i)}}{c_1 \equiv c_2}$$

Tuple equivalency

$$\frac{c_1 = \langle \mathsf{tuple}, \langle v_1, \cdots, v_n\rangle\rangle \qquad c_2 = \langle \mathsf{tuple}, \langle u_1, \cdots, u_n\rangle\rangle \qquad v_1 \equiv u_1, \cdots, v_n \equiv u_n}{c_1 \equiv c_2}$$

In the example, the value $\langle \mathsf{bagof}, \langle\langle \mathsf{obj}, \text{John Doe}\rangle, \langle \mathsf{obj}, \text{Jane Doe}\rangle\rangle\rangle$ would thus be equivalent to $\langle \mathsf{bagof}, \langle\langle \mathsf{obj}, \text{Jane Doe}\rangle, \langle \mathsf{obj}, \text{John Doe}\rangle\rangle\rangle$, as the ordering does not matter for bag container types.

For each type in $Type_{\mathrm{Tm}}$, there exists a set of values from $Value_{\mathrm{Im}}$ which is considered *valid*. This is defined by a relation $Valid_{\mathrm{Im}} \subseteq (Type_{\mathrm{Tm}} \times Value_{\mathrm{Im}})$ which defines a tuple for each valid value given a type.

Definition 16 (Valid type values)

The $Valid_{Im}$ set contains tuples which indicate what values are valid for a given type, which is defined by

$$Valid \subseteq (Type \times Value)$$

An element $\langle T, v\rangle \in Valid_{Im}$ may be written as

$$\frac{}{v : T}$$

The contents of the $Valid_{Im}$ set may then be derived as follows:

Data type values:

$$\frac{v \in \mathbb{B}}{\langle \mathit{bool}, v\rangle : \mathit{boolean}} \qquad \frac{v \in \mathbb{Z}}{\langle \mathit{int}, v\rangle : \mathit{integer}}$$

$$\frac{v \in \mathbb{R}}{\langle \mathit{real}, v\rangle : \mathit{real}} \qquad \frac{v \in \mathbb{S}}{\langle \mathit{string}, v\rangle : \mathit{string}}$$

Class values

$$\frac{\mathrm{ObjectClass}_{Im}(o) = c \qquad !\, c \sqsubseteq_{Tm} t \qquad t \in ClassType_{Tm}}{\langle \mathit{obj}, o\rangle : t}$$

$$\frac{t \in \{\mathit{nullable}\} \times Class_{Tm}}{\langle \mathit{obj}, \mathit{nil}\rangle : t}$$

Enumeration values

$$\frac{\langle ename, eval\rangle \in EnumValue_{Tm} \qquad ename \in Enum_{Tm}}{\langle \mathit{enum}, \langle ename, eval\rangle\rangle : ename}$$

Container values

$$\frac{v_1 : T, \cdots, v_n : T \qquad \langle v_1, \cdots, v_n\rangle \,\mathrm{distinct} \qquad \langle \mathit{setof}, T\rangle \in Container_{Tm}}{\langle \mathit{setof}, \langle v_1, \cdots, v_n\rangle\rangle : \langle \mathit{setof}, T\rangle}$$

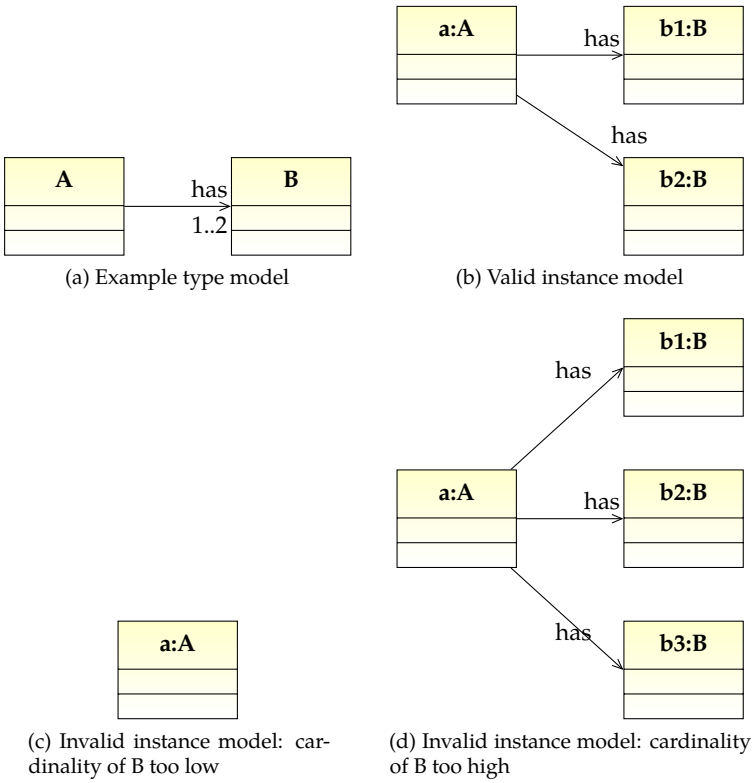
$$\frac{v_1 : T, \cdots, v_n : T \qquad \langle \mathit{bagof}, T\rangle \in Container_{Tm}}{\langle \mathit{bagof}, \langle v_1, \cdots, v_n\rangle\rangle : \langle \mathit{bagof}, T\rangle}$$

$$\frac{v_1 : T, \cdots, v_n : T \qquad \langle v_1, \cdots, v_n\rangle \,\mathrm{distinct} \qquad \langle \mathit{ordof}, T\rangle \in Container_{Tm}}{\langle \mathit{ordof}, \langle v_1, \cdots, v_n\rangle\rangle : \langle \mathit{ordof}, T\rangle}$$

$$\frac{v_1 : T, \cdots, v_n : T \qquad \langle \mathit{seqof}, T\rangle \in Container_{Tm}}{\langle \mathit{seqof}, \langle v_1, \cdots, v_n\rangle\rangle : \langle \mathit{seqof}, T\rangle}$$

TupleType values

$$\frac{v_1 : T_1, \cdots, v_n : T_n \qquad \langle \mathit{tuple}, \langle T_1, \cdots, T_n\rangle\rangle \in TupleType_{Tm}}{\langle \mathit{tuple}, \langle v_1, \cdots, v_n\rangle\rangle : \langle \mathit{tuple}, \langle T_1, \cdots, T_n\rangle\rangle}$$



(a) Example type model

(b) Valid instance model

(c) Invalid instance model: cardinality of B too low

(d) Invalid instance model: cardinality of B too high

Figure 3.5: Examples of valid and invalid multiplicities.

The validity of an instance model depends on the multiplicity of field values. The multiplicities specified in the type model also depend on the type, so both the type of a value has to be correct, and its multiplicity. The multiplicity is of most influence for container values, as they can contain an arbitrary amount of values.

Definition 17 (Multiplicity validity)

A field value $\langle\langle \mathit{object}, \mathit{field}\rangle, \mathit{value}\rangle \in \mathrm{FieldValue}_{Im}$ *has a valid multiplicity if the following property holds:*

$$\mathit{value} : \mathrm{type}(\mathit{field}) \wedge$$
$$\mathit{value} = \langle t, \langle v_1, \cdots, v_n\rangle\rangle \in \mathit{ContainerValue}_{Im} \Rightarrow \mathrm{lower}(\mathit{field}) \leq n \leq \mathrm{upper}(\mathit{field})$$

This may be written as $\mathrm{validMul}(\langle\langle \mathit{object}, \mathit{field}\rangle, \mathit{value}\rangle)$.

Figure 3.5 shows some examples of different multiplicities in instance models. More specifically, fig. 3.5a shows a type model that specifies a multiplicity of 1..2 for the has relation. Figures 3.5c and 3.5d show two instance models that have an invalid multiplicity (too low and too high respectively), whereas fig. 3.5b shows an instance model with correct multiplicity (an alternative correct instance model could have only a single instance of class B).

In order to simplify reasoning over value assignments, the $\mathrm{edgeCount}$ and edge operators are defined, which specify the number of relations (and the existence thereof) between any two objects.

Definition 18 (Value edges)

Let $a, b \in \mathit{Object}_{Im}$ *and* $r \in \mathit{Field}_{Tm}$ *where* $r \in \mathrm{fields}_{Tm}(\mathrm{ObjectClass}_{Im}(a))$. *Furthermore, we*

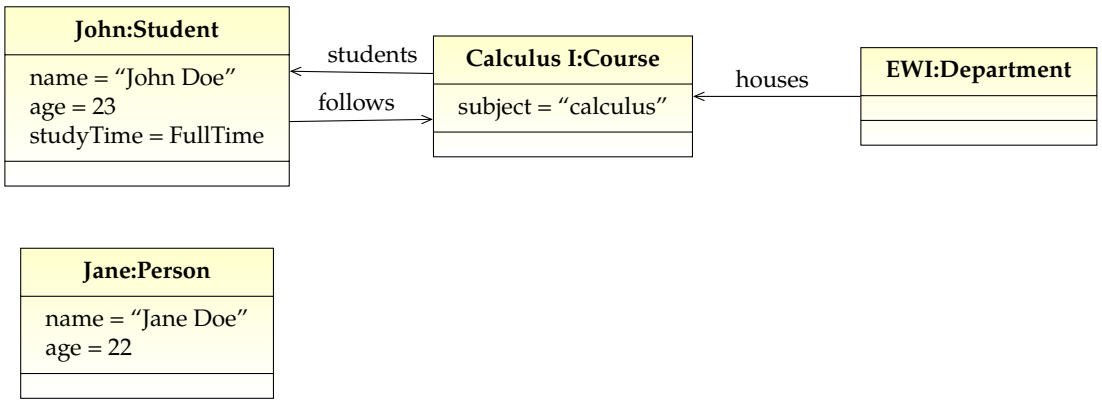


Figure 3.6: Model not satisfying the abstract property.

define $\text{containerCount}_{Im}(a, r, b)$ *as*

$$\text{containerCount}_{Im}(a, r, b) = |\{i \in \mathbb{N} | \langle\langle\langle \textit{obj}, a\rangle, r\rangle, \langle v_1, \cdots, v_n\rangle\rangle \in \text{FieldValue}_{Im} \wedge v_i = \langle \textit{obj}, b\rangle\}|$$

Then $\text{edgeCount}_{Im}(a, r, b)$ *is defined as*

$$\text{edgeCount}_{Im}(a, r, b) = \begin{cases} 0, & \textit{if } \text{type}(r) \notin \textit{Container}_{Tm} \\ & \quad \wedge \langle\langle\langle \textit{obj}, a\rangle, r\rangle, \langle \textit{obj}, b\rangle\rangle \notin \text{FieldValue}_{Im} \\ 1, & \textit{if } \text{type}(r) \notin \textit{Container}_{Tm} \\ & \quad \wedge \langle\langle\langle \textit{obj}, a\rangle, r\rangle, \langle \textit{obj}, b\rangle\rangle \in \text{FieldValue}_{Im} \\ \text{containerCount}_{Im}(a, r, b), & \textit{otherwise} \end{cases}$$

The $\text{edge}_{Im}(a, r, b)$ *predicate is defined as*

$$\text{edge}_{Im}(a, r, b) = \text{edgeCount}_{Im}(a, r, b) \geq 1$$

As previously mentioned, the properties specified in a type model must be satisfied by the instance model in order for it to be valid. For each property there is a satisfaction formula defined, which must hold for a given instance model for that instance model to be valid. The following definition specifies such a formula for each possible property in a type model.

Definition 19 (Property satisfaction)

Given an instance model Im and a type model Tm, a property $p \in \textit{Prop}_{Tm}$ *can be satisfied, written as* $Im \models p$*, if the satisfaction formula holds for p.*

- *The satisfaction formula for an abstract property* $\langle \textit{abstract}, c\rangle$ *given an instance model Im is defined as*

  $$\nexists o \in \textit{Object}_{Im} : \text{ObjectClass}_{Im}(o) = c$$

  The example in fig. 3.4 shows an instance model that satisfies this formula for the Person class, as no direct instantiations of the Person class exist. Figure 3.6 shows an instance model that does not satisfy the property, as the Person class has been instantiated (by the object Jane).
- *Let, for a constraint* $\langle \textit{identity}, A\rangle$*,* $ac \in \textit{Class}_{Tm} \wedge \forall a \in A : a = \langle ac, n\rangle$*. The satisfaction formula for the identity property given an instance model Im is then defined as*

  $$\begin{aligned} \forall o, o' \in \textit{Object}_{Im} : \text{ObjectClass}_{Im}(o) = ac \wedge \text{ObjectClass}_{Im}(o') = ac \\ \wedge \forall a \in A : \text{FieldValue}_{Im}(\langle o, a\rangle) \equiv \text{FieldValue}_{Im}(\langle o', a\rangle) \\ \Rightarrow o = o' \end{aligned}$$

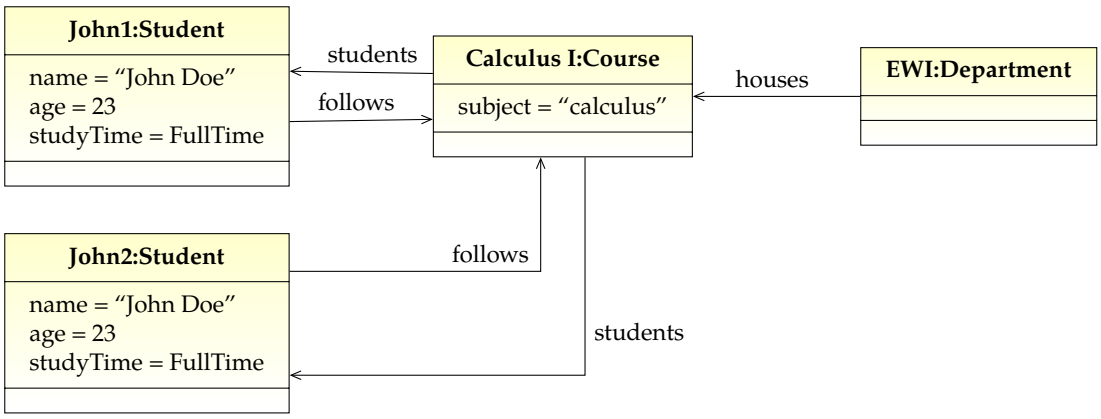


Figure 3.7: Model not satisfying the identity property.

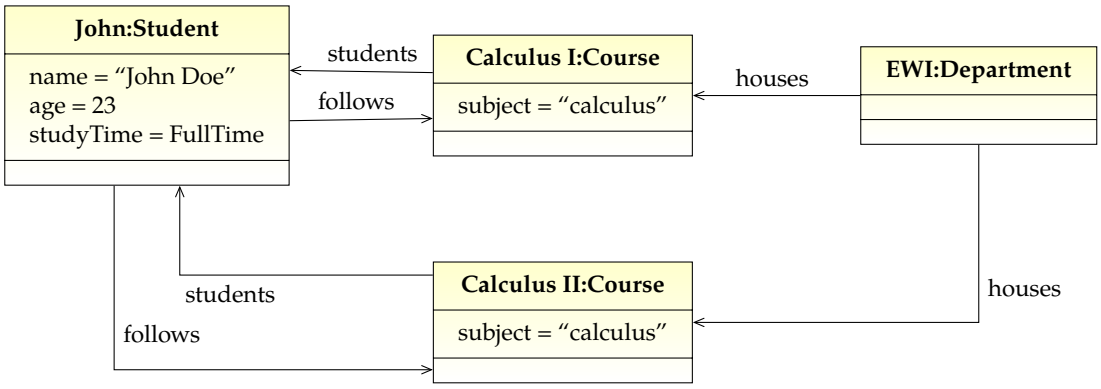


Figure 3.8: Model not satisfying the keyset property.

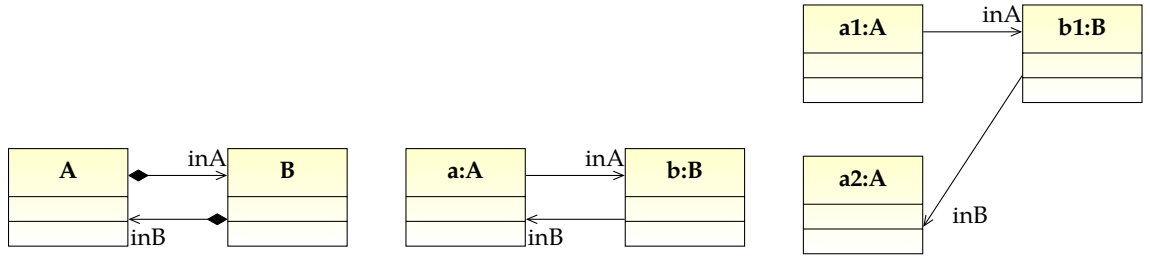
Assume the type model in fig. 3.2 specifies an identity property for the name *and* age *attributes of the* Person *object. Figure 3.7 shows an instance model that does not satisfy the property, as both* Student *objects (which inherit from the* Person *class) do not have a unique set of values for the* name *and* age *attributes. In fig. 3.4 this is not the case and the property would be satisfied.*

- *The satisfaction formula for a* keyset *property* $\langle \textsf{keyset}, r, A \rangle$ *given an instance model Im is defined as*

$$\begin{aligned} \forall o, o', p \in \mathit{Object}_{\mathit{Im}} \colon & \langle \mathrm{ObjectClass}_{\mathit{Im}}(p), r \rangle \in \mathit{Field}_{\mathit{Tm}} \\ & \wedge \mathrm{edge}_{\mathit{Im}}(p, r, o) \wedge \mathrm{edge}_{\mathit{Im}}(p, r, o') \\ & \wedge \forall a \in A \colon \mathrm{FieldValue}_{\mathit{Im}}(\langle o, a \rangle) \equiv \mathrm{FieldValue}_{\mathit{Im}}(\langle o', a \rangle) \\ \Rightarrow & \; o = o' \end{aligned}$$

- *The* containment *property* $\langle \textsf{containment}, R \rangle$ *is satisfied for an instance model Im, when any object in Im, which is the target for a containment relation, is contained by no more than 1 object, and there are no cycles in the instance model given the containment values.*
  Let $CR = \{R | R \in \mathit{Rel}_{\mathit{Tm}} \wedge \langle \textsf{containment}, R \rangle \in \mathit{Cons}_{\mathit{Tm}}\}$ *be the set of all containment relations in a type model Tm. The satisfaction formula for the* containment *property given an instance model Im is then defined as*

$$\begin{aligned} \forall o \in \mathit{Object}_{\mathit{Im}} \colon & |\{\langle \langle \mathit{fo}, \mathit{ff} \rangle, \mathit{fv} \rangle | \langle \langle \mathit{fo}, \mathit{ff} \rangle, \mathit{fv} \rangle \in \mathrm{FieldValue}_{\mathit{Im}} \wedge \langle \textsf{obj}, o \rangle = \mathit{fv} \wedge \mathit{ff} \in CR\}| \leq 1 \\ & \wedge \{(\mathit{fo}, \mathit{fv}) | \langle \langle \mathit{fo}, \mathit{ff} \rangle, \mathit{fv} \rangle \in \mathrm{FieldValue}_{\mathit{Im}} \wedge \mathit{ff} \in CR\} \textit{ is acyclic} \end{aligned}$$



(a) Example type model with two containment relations

(b) Model not satisfying the containment property

(c) Model satisfying the containment property

Figure 3.9: Examples of the containment property.

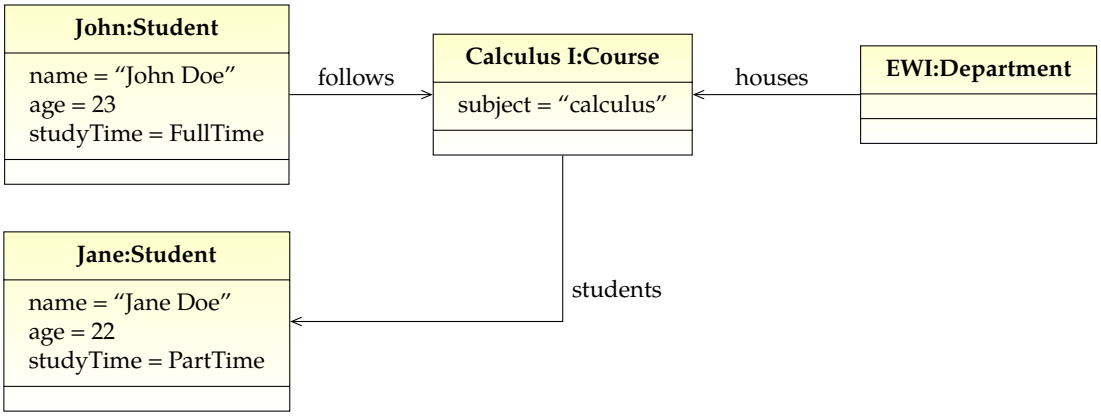


Figure 3.10: Model not satisfying the opposite property.

Figure 3.9a shows a type model with two containment relations, in opposite direction. The model given in fig. 3.9b does not satisfy this containment property, as there exists a cycle of containment relations. This is corrected in the instance model in fig. 3.9c, where such a cycle does not exist (and each object is containment by at most one other object).

- *The satisfaction formula for an* opposite *property* $\langle \textsf{opposite}, r, r' \rangle$ *given an instance model Im is defined as*

$$\forall o, o' \in \mathit{Object}_{\mathit{Im}} : \mathrm{edgeCount}_{\mathit{Im}}(o, r, o') = \mathrm{edgeCount}_{\mathit{Im}}(o', r', o)$$

In fig. 3.10, an example model is shown which does not satisfy the opposite *property for the* students *and* follows *relations. Although the number of relations is equal, they do not have the same source and target objects (in opposite direction). The example model in fig. 3.4 does satisfy the property.*

With the previous definitions, it is now possible to define when an instance model itself is valid, given its type model.

Definition 20 (Model validity)

An instance model Im is said to be valid if and only if

- *All values are correctly typed:* $\forall \langle \langle \mathit{obj}, \mathit{field} \rangle, \mathit{val} \rangle \in \mathrm{FieldValue}_{\mathit{Im}} : \mathit{val} : \mathrm{type}_{\mathit{Tm}}(\mathit{field})$
- *All container multiplicities are valid:* $\forall \mathit{fv} \in \mathrm{FieldValue}_{\mathit{Im}} : \ \mathrm{validMul}(\mathit{fv})$.
- *All properties are satisfied:* $\forall p \in \mathit{Prop}_{\mathit{Tm}} : \mathit{Im} \models p$

- *All default values have the correct type:*
  $\forall c \in \mathit{Constant}_{Tm} : \mathrm{DefaultValue}_{Im}(c) : \mathrm{ConstType}_{Tm}(c)$
- *Tm is consistent, as defined in definition 12.*

This is written as $Tm \vdash Im$.

3.4 Conclusions

This chapter has introduced the conceptual model. The conceptual model provides a bridge between the various external languages and GROOVE graphs. This chapter only provides a formal definition, the actual implementation is discussed in appendix A. By defining this conceptual model, an answer has been found for the second research question. By using this conceptual model as an intermediate layer, it provides a framework that can be used to implement the various mappings between the external languages and GROOVE. These mappings are discussed in chapters 4 and 5.

Chapter 4 Mapping CM-GROOVE

This chapter describes how the conceptual model is mapped to GROOVE graphs, based on the definitions from section 2.2 and chapter 3. First, a general description is given of this mapping for the default graph structure. Later on, a description is given of the various configuration possibilities which affect the way graphs are generated and interpreted. The definition of the mapping is divided between the type and instance models. The type models are mapped to type graphs and rule graphs, as will be described, whereas instance models are mapped to instance graphs.

The mapping between GROOVE and the conceptual model is configurable. This means that for several parts of the conceptual model multiple (partial) mappings exists. Which mapping is chosen then depends on the configuration. In some cases these mappings influence each other, which means that only the entire configuration determines the entire mapping, which is not always simply the conjunction of each partial mapping determined by a single element of the configuration. This chapter first describes the mapping for the default configuration (which is the configuration used if the user does not actively change it). For the parts of the mapping that may be influenced by the configuration, section 4.4 will describe the alternatives. This section also gives an overview of all the available configuration options.

4.1 Global level

The *Id* set in the conceptual model can be mapped to the *Lab* set in GROOVE. However, since an identifier consists both of a name and a namespace, and a label is just a string, some conversion is required. The configuration determines how an identifier is mapped. The result is always a string which is mapped to the *Lab* set in the GROOVE type graph. This string consists of any of the namespaces of the identifier, separated by the configured separator symbol. Optionally, some part of the identifier may have been rewritten by a manual override. The other way around, any override is applied in reverse. Then, the label is split into namespaces and names using the separator symbol, resulting in the final identifier in *Id*.

The globally defined data types are mapped to data type nodes in GROOVE. For example, the string data type is mapped to the string type in GROOVE. Values are mapped similarly, in the instance graphs. Thus, *DataType* is mapped to Lab_{prim}. The value mappings are kept the same, as both the conceptual model and GROOVE use the exact same sets of values.

The global level also specifies the Nil value. This value is mapped to a special Nil node in the type graph (with the configurable label `Nil`), which is a direct subtype of every nullable class in the type graph. Thus, this node can be instantiated in the instance graph and be used as a value for every reference to a nullable class. For an example of this, see fig. 4.2, which shows two classes with nullable types, and the Nil node.

4.2 Type model

Mapping the type model to GROOVE would ideally consist of a mapping between a type model and a type graph only. However, this is not always possible because of the various constraints in the type model which cannot be represented by a type graph alone. Thus, other graphs may be generated as

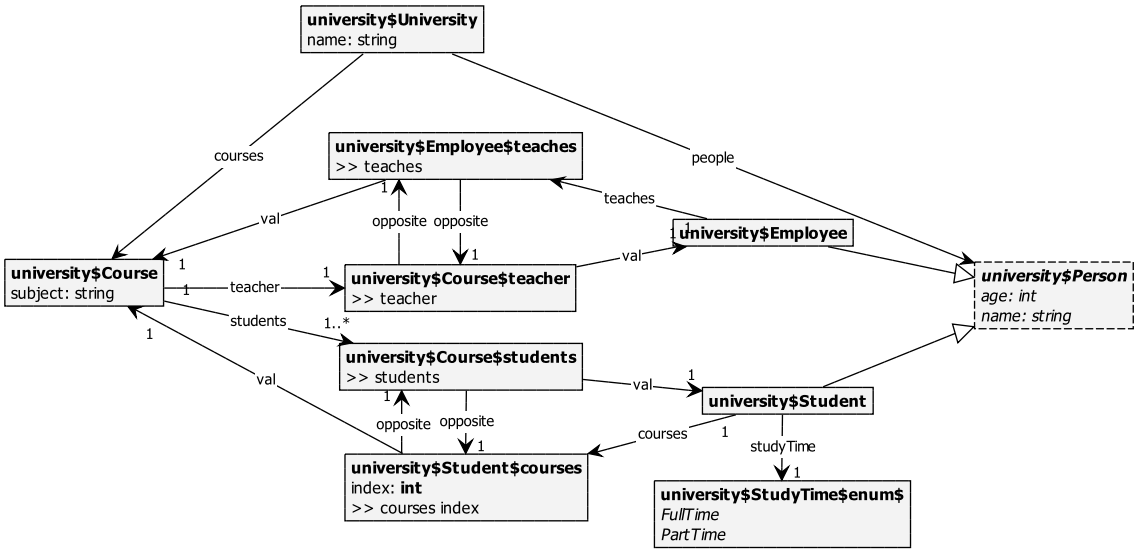


Figure 4.1: Example graph representation in GROOVE for the University type model example.

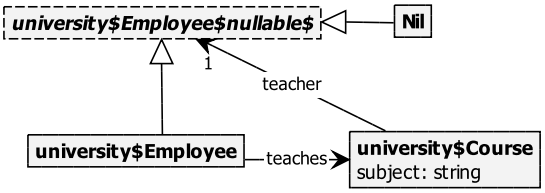


Figure 4.2: Example typegraph with nullable classes.

well, which are usually rule graphs (though a meta graph may be generated as well, see section 4.2.2).

A type model is defined as

$$\mathrm{Tm} = \langle \mathit{Class}, \mathit{Enum}, \mathit{UserDataType}, \mathit{Field}, \mathrm{FieldSig}, \mathit{EnumValue}, \mathit{Inh}, \mathit{Prop}, \mathit{Constant}, \mathrm{ConstType} \rangle$$

whereas a type graph in GROOVE is defined as

$$\mathrm{T}_G = \langle N, E, \sqsubseteq, abs, mult, contains \rangle$$

The various parts of the type model are mapped as follows:

1. The set of classes *Class* is mapped to the set N in the type graph by mapping the identifier of the class to the label of the type node. However, the conceptual model distinguishes two types of classes: proper classes and nullable classes. In order to accommodate for this, a class in the conceptual model is mapped to two type nodes in the type graph. One node represents the nullable class (which is a supertype of the Nil node), a second node represents the proper class, where the second node is a subtype of the first (as a proper class is a subtype of a nullable class). To differentiate between the two, a configurable string (by default `$nullable$`) is appended to the label of the nullable class node. An example of this is shown in fig. 4.2.

2. The set of enumerations *Enum* is mapped to the set N as well. However, the identifier is not directly mapped, but rather mapped to a label with a postfix appended at the end, to be able to distinguish class and enumeration nodes in the type graph (by default `$enum$`). The mapping

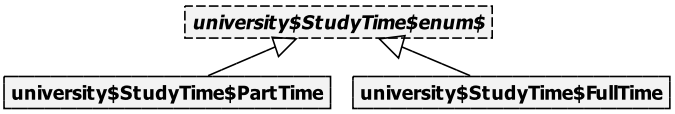


Figure 4.3: Type graph for an enumeration mapped to nodes.

for the *EnumValue* set is configurable. By default, it is mapped to nodes in the set N. The name part of each tuple in *EnumValue* is mapped to an identifier, consisting of that name and the identifier of the corresponding enumeration as its namespace. This identifier is then mapped to a label, which is used for a node in the type graph representing the enumeration value. The value node will be a subclass of the node representing the enumeration itself. Figure 4.3 shows an example of an enumeration mapped to nodes. More details about the configuration of the enumeration mapping can be found in section 4.4.

3. The *Field* set is mapped to edges in the type graph from the corresponding class node to the field node (which is determined based on the $\mathrm{FieldSig}$ function). The label of the edge is directly mapped from the name of the field. $\mathrm{FieldSig}$ is mapped to field nodes and combined with the edge of the corresponding element in the *Field* set. The type of the field is mapped to the node corresponding to that type in the type graph, with the edge representing the field connecting the node representing the class to the node representing the field type. The multiplicity does require some changes. In the conceptual model, the multiplicity of a field with a nullable class is always 0..1 (optional). However, if the configuration is set to use nullable classes, in GROOVE such a field is required to always have a value, which may be Nil. So, the multiplicity is mapped to an outgoing multiplicity in GROOVE of 1..1, the type of the field being that of a nullable class. Furthermore, GROOVE allows to specify an incoming multiplicity as well. The incoming multiplicity is not defined by the conceptual model, and is left at its default value (which is 0..*). For other types, the multiplicity is always 1..1, with the exception of container types. For these types, the multiplicity is exactly that of the one in the conceptual model, allowing a field to have no value assigned at all.

4. The *Inh* set is directly related to the $\sqsubseteq$ set in the type graph. The nullable version of the classes in the *Inh* set are mapped to the type nodes in the $\sqsubseteq$ set, and the pairs in *Inh* are subsequently mapped to $\sqsubseteq$.

5. The *Prop* set is mapped as follows for each possible property:

   - Abstract: The abstract property is mapped by adding the node from N to the set *abs* in the type graph. Figure 4.4 shows an example of this.
   - Containment: The containment property is mapped to the *contains* set in the type graph, by mapping the relation to the generated edge for that relation. An example is shown in fig. 4.5 and also in fig. 4.28.
   - Identity and keyset: These properties are mostly similar, except that for the Keyset property an additional check is added for the relation under which the keyset must hold. The rule graph matches if the number of values for a given field matches and at least one of those values is equal to the other. The test of value equality differs between different container type values, and between values with or without intermediate nodes. If a value is not a container value, it is considered an unique, unsorted container with one element. Examples of keyset and identity property constraint graphs are shown in figs. 4.6 and 4.7. A complete overview of the possible constraint checks is shown in fig. 4.11.
   - DefaultValue: This property is mapped to a rule which adds the value specified by the property to the instance graph if it is missing, of which an example is shown in fig. 4.10. Additionally, the value may be applied in the initial instance graph if the configuration is set to do so. See also section 4.4.

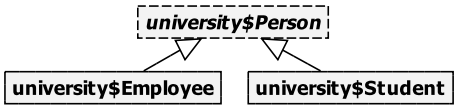


Figure 4.4: Mapping of the abstract property to abstract nodes. In this case, the class Person is declared abstract (but its subclasses are not).

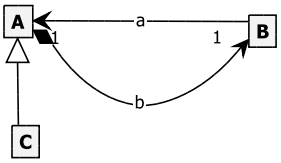


Figure 4.5: The containment property in GROOVE. In this example, the field b of class A is declared to be a containment relation.

- Opposite: The property is mapped to a constraint rule that checks if either relation from the opposite property is missing from the instance graph, of the other relation does exist. It may optionally be mapped to two edges between two intermediate nodes of the relation in the type graph, each for one direction of the opposite property. See figs. 4.8 and 4.9 for example graphs of the opposite constraint checks and opposite edges.

6. The *Constant* set is mapped to transformation rules which apply a default value when no value is assigned. For the exact mapping, see the above information on the DefaultValue property.
7. The $\mathrm{ConstType}$ function is, just like the *Constant* set, mapped with the DefaultValue property.

The type model also specifies types for containers and tuples. Containers are used for fields which may contain multiple instances of a given type. These containers have two properties: uniqueness and ordering. A container type may be mapped to a special node in the type graph, but may also directly refer to the node of the type that is contained within the container. This choice between using a special node for a container (an intermediate node) or directly referring to the underlying type can be based on various settings as well. However, whatever the configuration is, the following circumstances always require the use of an intermediate node:

- The underlying type of the container is again a container.
- The container allows the same item to occur twice; GROOVE does not support the same edge twice between any two nodes.

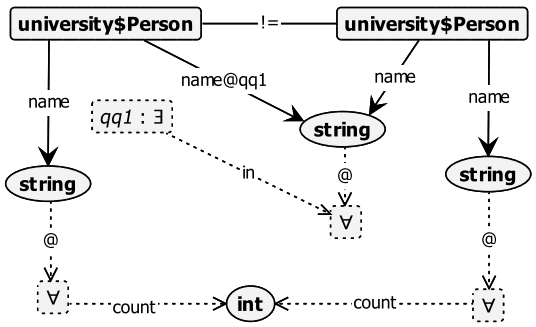


Figure 4.6: Example of an identity constraint rule. This rule checks that no two instances of the Person class have the same value of the name field.

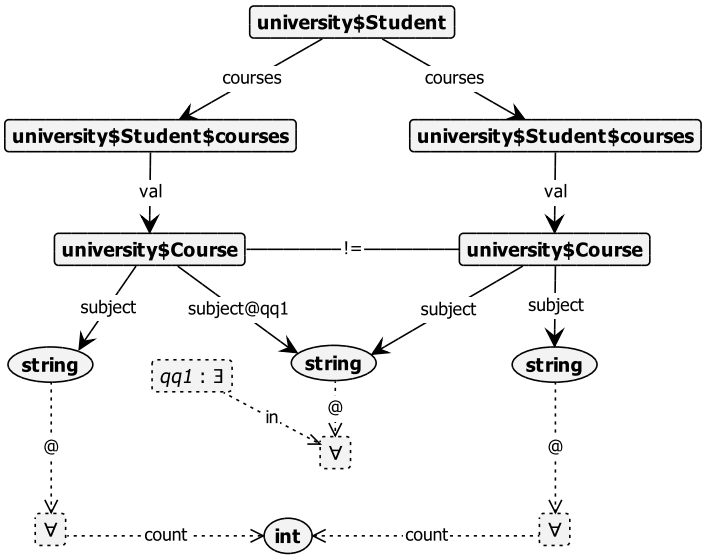


Figure 4.7: Example of a keyset constraint rule. This rule checks that no instance of the Student class has two Course instances with the same subject in the courses relation.

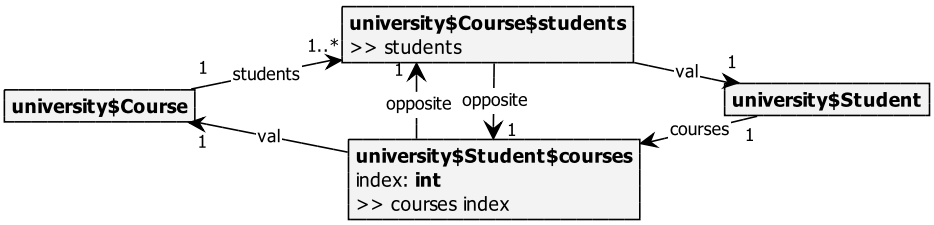


Figure 4.8: Example type graph for opposite edges.

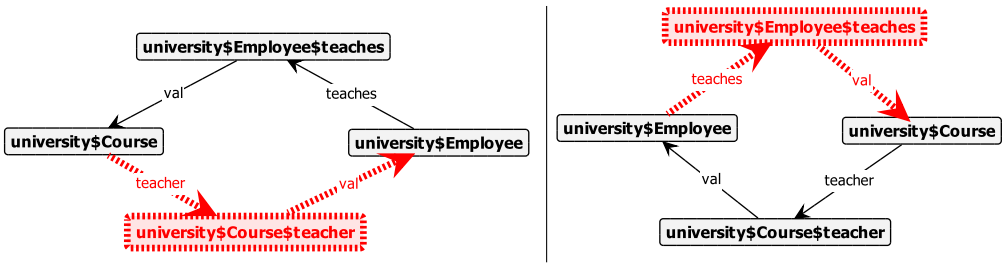


Figure 4.9: Example constraint rules for opposite fields.

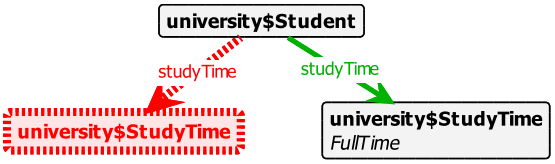
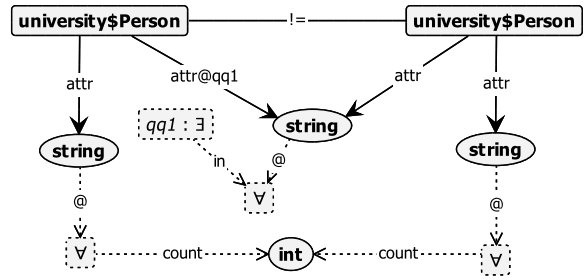
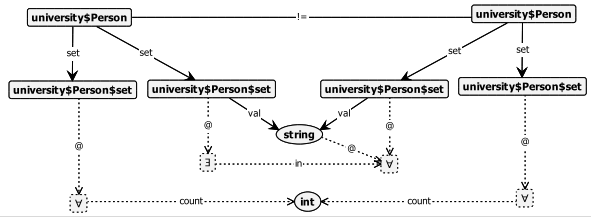


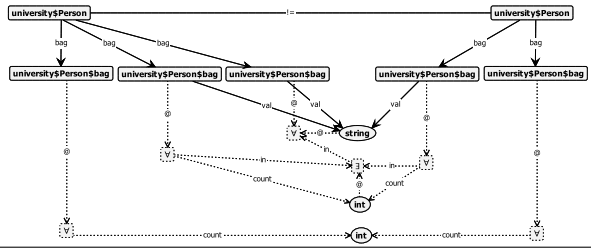
Figure 4.10: A rule graph for the defaultValue property.



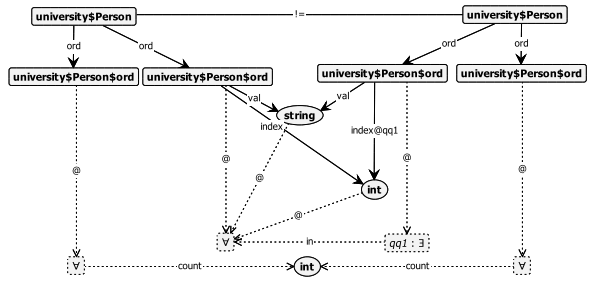
Identity check for values without an intermediate node. This rule checks that for all values of attr of one Person node, an equal value exists for the attr set of another Person, as well as that the size of both containers is equal.



Identity check for values with an intermediate node, in a unique but unordered container. The difference with the above rule graph is that now a check is made for the existence of an intermediate node, rather than a direct edge.



Identity check for values with an intermediate node, in a non-unique and unordered container. This is the most complex rule, and checks that for each value of one container, the count of that value is the same for both containers. This, combined with a check for the total container size, matches containers with multiple identical values.



Identity check for values with an intermediate node, in an ordered container. Since each element has an index, it suffices to check that for each element in one container, an equal element with the same index exists in another container.

Figure 4.11: Example constraint rules for opposite fields.

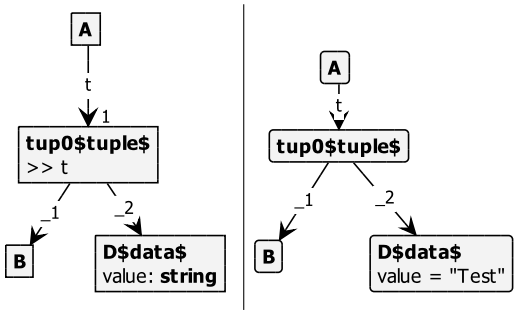


Figure 4.12: Example of a tuple in a type graph (left) and an instance graph (right).

- The items in the container are ordered, and cannot be modified in such a way that they can represent this ordering by themselves (for example, they are shared between different ordered containers). An intermediate node is required to store this ordering.
- An opposite edge is required between the relations.

If an intermediate node is created, first a mapping is made between the container and an identifier. This identifier is based on the field of which the type is the container, or on the identifier of the container that contains this container. The identifier consists of the identifier of the class of the field and the name of the field. For sub-containers, the identifiers are based on the identifier of the above container and a postfix indicating the intermediate node is an element of another container (by default `$elem$`). The identifier of the container is then mapped to a node in N, with an edge from the field or above container to the node, and from the node to the node in N representing the type contained within the container.

The intermediate nodes for a container will be marked as an 'edge node', which will be drawn as a small dot in instance graphs. This causes the node to more closely resemble an edge, making the graph appear less convoluted. This is purely a graphical feature and does not modify the underlying model in any way. See also fig. 4.16.

Tuples are represented by nodes like classes are. However, the elements in a tuple have no name, but are rather accessed by their position in the tuple. Each tuple will first be mapped to an identifier. This identifier gives no guarantee about its name other than that it is unique within the type graph. However, to be able to distinguish it as a tuple, a postfix is added (by default `$tuple$`). The tuple node will have an edge going to each node representing one of the types in the tuple. The edge will be labeled with an index, ranging from 1 to the number of elements in the tuple. An example of a tuple used in a type graph and in an instance graph is given in fig. 4.12. This tuple contains both a single instance of a class of type `B` and a custom data type `D`.

4.2.1 Additional constraint checks

In addition to the mappings described before, additional constraints are added to the type graph, and extra rule graphs may be generated as well. These constraints and rules help prevent invalid instance models from being created/generated.

- For intermediate nodes, it is required that each intermediate node has exactly one parent node, such that the intermediate node is an actual value of another node. This is enforced by setting the incoming multiplicity of the edge pointing to the intermediate node to 1, ensuring the intermediate node is always owned by exactly one parent.
- Intermediate nodes always have to point to exactly one value as well. This, combined with the

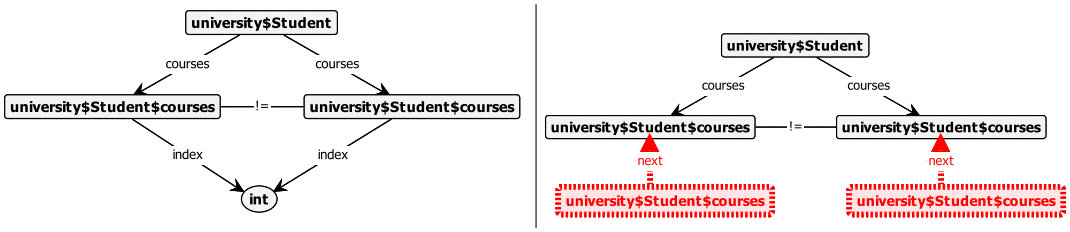


Figure 4.13: Example constraint rule for ordered containers using an index and a next edge respectively.

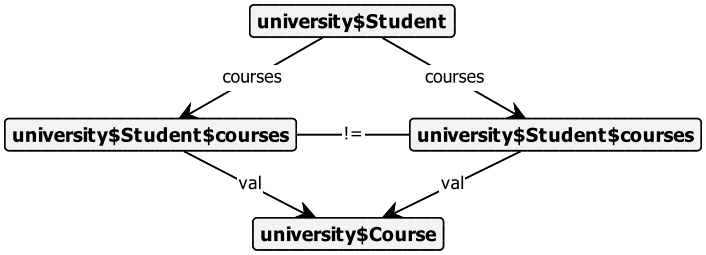


Figure 4.14: Example constraint rule for unique values.

previous constraint an intermediate node always forms an edge between two nodes. To enforce this, the outgoing value edge of the intermediate node has an outgoing multiplicity of 1.

- For ordered containers, constraints need to be added to make sure all elements in the container are well ordered. For an ordering based on edges pointing to the next and previous elements, it is required that between each pair of consecutive nodes exactly one edge exists pointing to the next node (and optionally the previous). This is enforced by specifying that each node in an ordered container has at most one incoming and at most one outgoing next (and optionally previous) edge. Furthermore, a constraint rule is added to make sure the container has no two different nodes without an incoming edge (only one such node is allowed to exist). For index based ordering, all the values in the container need to have a unique value within the container. An outgoing multiplicity for the index value of 1 ensures each intermediate node has an index. Uniqueness is enforced by a constraint rule that ensures no two nodes in the container have the same index value. For examples of these rule graphs, see fig. 4.13.

- For containers with unique values, constraint rules are added to ensure each value is unique. These constraint rules check that for each pair of values in the container, the nodes are not identical. An example of such a rule graph is shown in fig. 4.14.

- To ensure opposite edges are always present between two intermediate nodes of two opposite relations, the outgoing multiplicity of opposite edges is set to exactly 1, so that each intermediate node will point to an opposite intermediate. The rule graphs generated for opposite properties ensure that these intermediate nodes always come in pairs, which guarantees the opposite edges will always occur in opposite pairs as well.

4.2.2 Meta graph

The meta graph provides a bridge between the type model and the language model this type model is based on. It has nodes for classes (and nullable classes), enumerations, containers (one for each type), tuples, data types and intermediate nodes. Each of these nodes is marked as the supertype of the nodes generated for a type graph that represent these various elements.

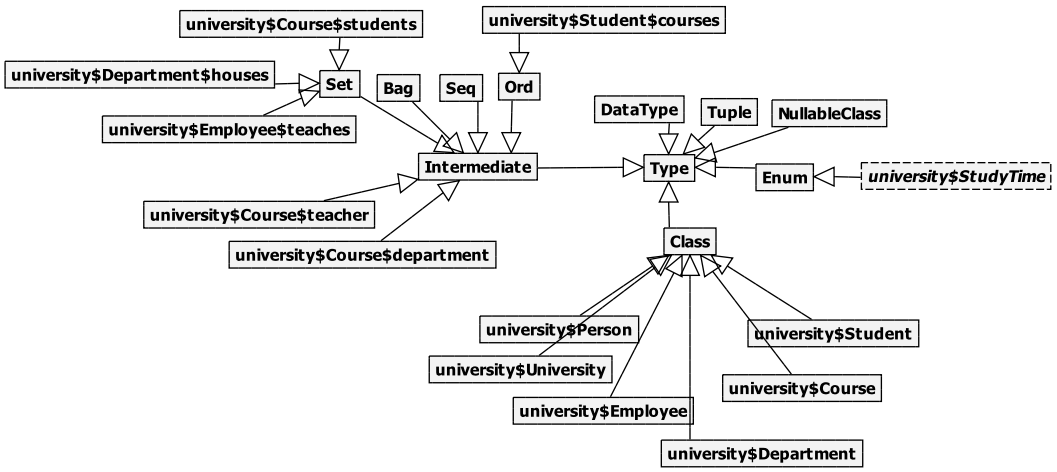


Figure 4.15: Meta graph of University example.

The meta graph provides an optional graph structure which is used in aiding the exporter in determining what each node in the graph represents in the conceptual model. The meta graph adds some extra complexity to the grammar, as the user has to ensure that it stays synchronized with other type graphs when changes are made to the grammar. However, as the meta graph can store some of the information required to create a type model for the grammar, other graphs may be simplified, which makes them easier to work with.

The example in fig. 4.15 shows a meta graph which may accompany the type graph shown in fig. 4.1. It shows how the various nodes are related to the meta nodes for Class and Enum, as well as the various intermediate nodes. Identifying the intermediate nodes in the meta graph, and the exact type of container they may represent, removes the need for extra constraints graphs for example, which would otherwise be needed to identify the various constraints for a container.

4.3 Instance model

The mapping between an instance model in the conceptual model and GROOVE is based on a single instance graph, for which a type graph and rules for the type model of the instance model are present. An instance model Im is defined as

$$\text{Im} = \langle \mathit{Object}, \text{ObjectClass}, \text{ObjectId}, \text{FieldValue}, \text{DefaultValue} \rangle$$

and instance graphs are defined as

$$\text{H}_G = \langle H_n, H_e, \mathit{ident} \rangle$$

The instance model is mapped as follows:

- The *Object* set can directly be mapped to H_n based on the ObjectClass function, which determines the type of the object and thus the node. Each element in the *Object* set is mapped directly to a node in the H_n set.
- The ObjectId function is mapped to the *ident* set directly.

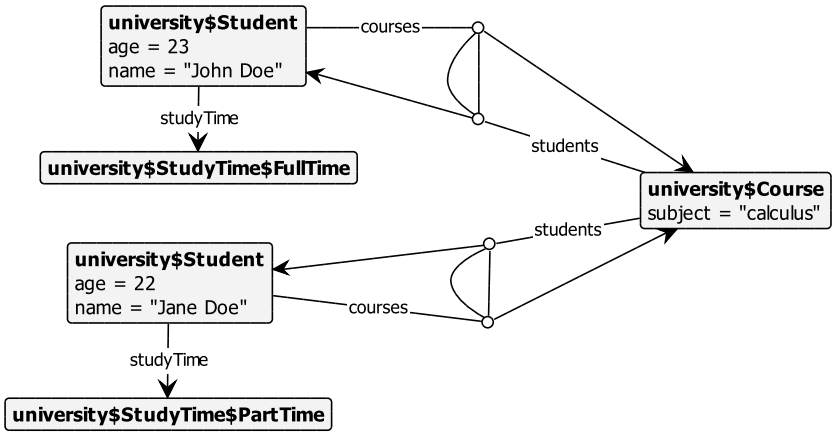


Figure 4.16: An instance graph representation of the example University instance model.

- The $\mathrm{FieldValue}$ function is mapped by creating an edge (in H_e) between the node corresponding to the object and the node corresponding to the value of the field. The label is the edge is that of the label used in the type graph mapping of the field. The exact same structure is used as with the type graphs, but now multiple nodes maybe created as container values may contain multiple other values. For each such a contained value, an edge is created between the container node and the value nodes, with optionally an extra intermediate node.
- The$\mathrm{DefaultValue}$ function is not directly mapped. Rather, when a defaultValue property applies, a value of the $\mathrm{DefaultValue}$ function may be used to directly assign a value to a field for a given object. This only applies if the configuration has been set to do so. The value that is applied is determined by the constant that is mapped from the defaultValue property via the type model.

An example instance graph for the University model (in part) is shown in fig. 4.16, using the default configuration. Attribute values are shown inside the nodes representing object instances. Intermediate nodes are drawn as small circles, which gives a better indication of their meaning. Opposite edges have been added to the relations between the Course and Student nodes.

Data type values are mapped directly to GROOVE data types. There is no conversion required and each type can be mapped directly. For custom data types, their corresponding nodes are instantiated from the type graph, with an additional string node for its literal value.

4.4 Configuration

The configuration will allow the models to be represented by different graph structures that can be targeted towards more complete information and stricter type checking, or simpler graphs that present a better overview of the model. The other way around, transforming graphs into models, the configuration can be used to determine what graph structure corresponds to what model element. This allows different graphs structures to be used for modeling the same information (although in some cases more detailed than others).

The configuration is a list of settings that apply to the entire model, or only parts selected by the user. These settings may influence or interfere with each other. Some options are incompatible with others. For example, not using intermediate nodes but not allowing ordering information to be placed on values either.

The configuration options can be structured using a feature tree. In figs. 4.17 and 4.18 this tree is shown. It has been split into two parts: one part contains all the options for the various possible graph

structures and features. The second tree contains the options concerning the use of the properties in the conceptual model and the generated constraint graphs (rule graphs), as well as the generation of an optional meta graph.

In the tree, abstract features represent subparts of the entire configuration, making up the tree. The concrete nodes are the actual possible options. If such a feature (option) is marked optional, that means it may be turned on or off independently of any other option under the same node. If the options together form alternatives, then exactly one of these must be chosen.

- The `Names` option allows the *namespace* separator and the namespace usage to be configured. The `Separator` option has two alternatives:
  - `$` : The dollar symbol is used as a separator
  - `_` : The underscore symbol is used as a separator

  Figure 4.19 shows the effects of this option.
- The `Namespaces` option has three alternatives:
  - `Full` : The entire namespace is used in the graphs. This results in the most verbose and complete graphs.
  - `NonAmbiguous` : Only the parts of the namespace required to disambiguate all the identifiers are added to the graphs. All names are guaranteed to be unique but part of the namespace information is lost.
  - `Flat` : No namespace information is added to the graph. Identifiers (which are simple names) may conflict with each other as a result, but the resulting names in the graph are the simplest.

  Figure 4.20 shows the effects of this option.
- The `Overrides` option allows the *Id* mapping to be overridden. This allows a specific *Id* or *Namespace* to be represented in a completely custom manner. This applies to both the importing and exporting of identifiers. This only applies to an *Id* as a whole, which may be the namespace of another *Id*.
- When importing or exporting a type model, optionally a *meta graph* can be generated or used, controlled by the `MetaSchema` option. This allows certain information to be stored in a separate type graph, which allows the other graphs to be simplified. For example, the meta graph keeps track of the intended meaning of a node in a type graph (for instance, a node representing either a *Class* or an *Enum*), which allows more simple names to be used for the labels of these nodes.

  When using a meta graph, various other elements in the grammar can be left out without losing information required to be able to export the required models. For example, the constraint rules created for the various container types, which are also used to determine their types, can be left out. Although this disables the constraint checking in GROOVE, fewer rule graphs are required without losing the ability to correctly export the type of the container. This also holds for the ability to distinguish various node types in the type graph, such as class, enumeration and data type nodes. These are normally distinguished by adding a postfix to their labels, but with the use of a meta graph this is no longer required, simplifying the labels of these nodes.

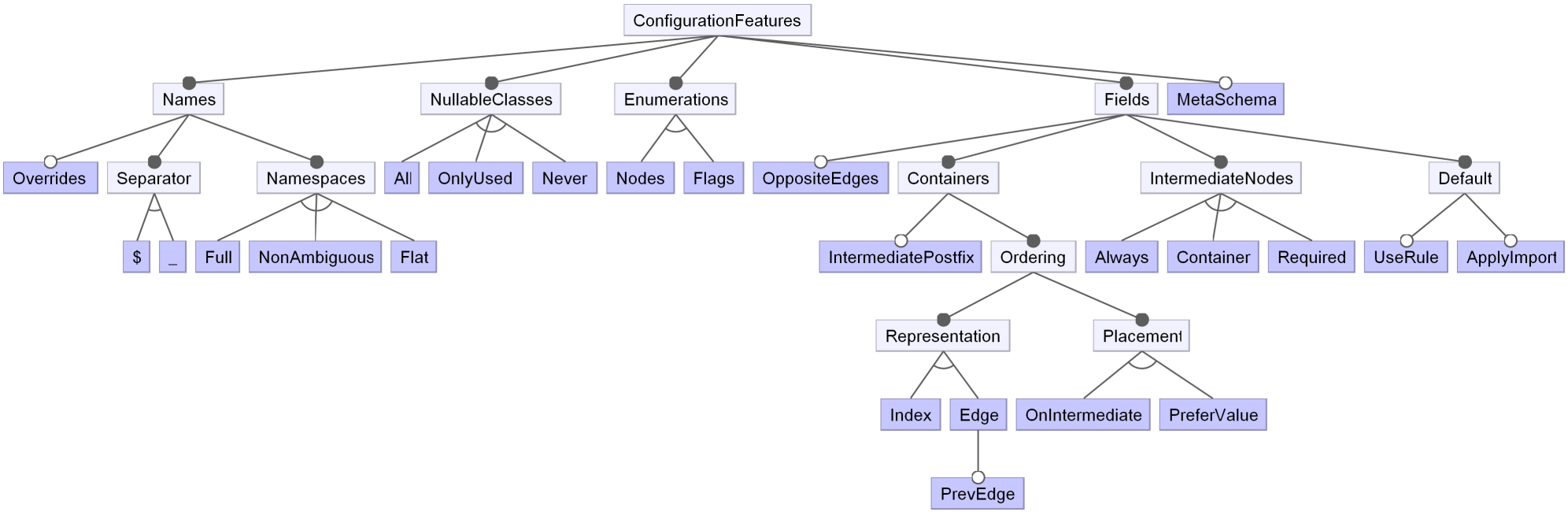


Figure 4.17: Configuration tree for graph features.

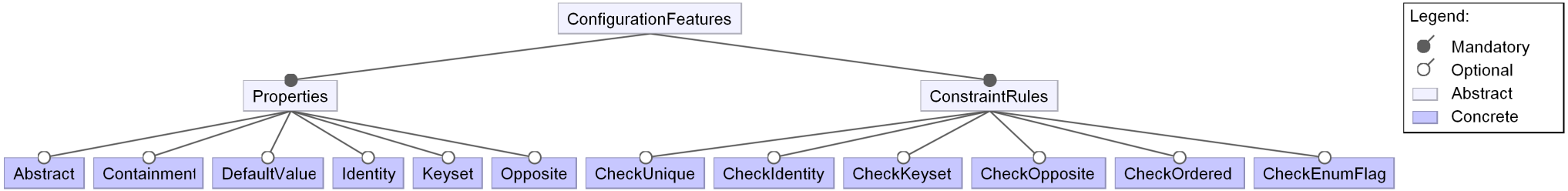


Figure 4.18: Configuration tree for properties and rule graphs.

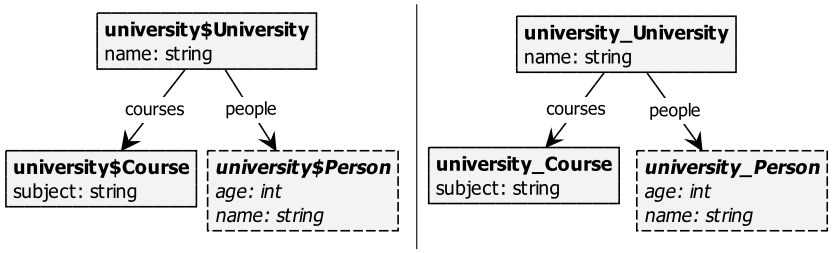


Figure 4.19: Effect of the `Names` option on labels in graphs. On the left the underscore separator is used, on the right the dollar sign.

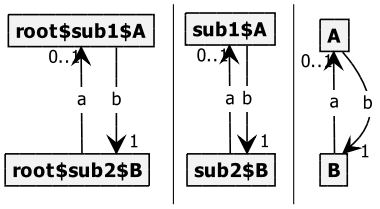


Figure 4.20: Effect of the `Namespaces` option on identifiers in graphs. The figure shows the `Full`, `NonAmbiguous` and `Flat` alternatives respectively.

- The `NullableClasses` option controls the creation of nodes for *nullable* classes. It provides three alternatives:
  - `All` : Every class will have an associated nullable class node, even if it is not referenced by any relation in the type model.
  - `OnlyUsed` : Only those classes for which the nullable class is referenced in the type model will have an associated nullable node.
  - `Never` : No nullable nodes are created at all, and no difference is made between proper and nullable classes. Additionally, the multiplicity for non-container references to nullable classes is set to 0..1 in GROOVE, rather than 1..1.

  In fig. 4.21 the effect of these three alternatives on the type graph is shown. Note that with each consecutive alternative the type graph becomes smaller.

- For *enumerations*, the `Enumerations` option controls how they are represented. The `Nodes` alternative uses nodes to represent enumeration values. The `Flags` alternative uses flags instead. An example of both alternatives is shown in fig. 4.22. If the latter alternative is selected, optionally extra constraint rules may be added (depending on other configuration options discussed later on). The default is to use nodes only for representing enumerations, where enumeration values

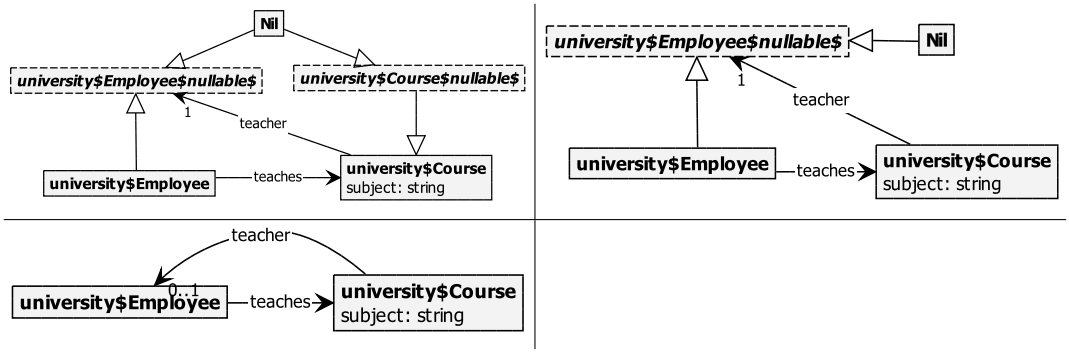


Figure 4.21: Effect of the `NullableClasses` option on type graphs.

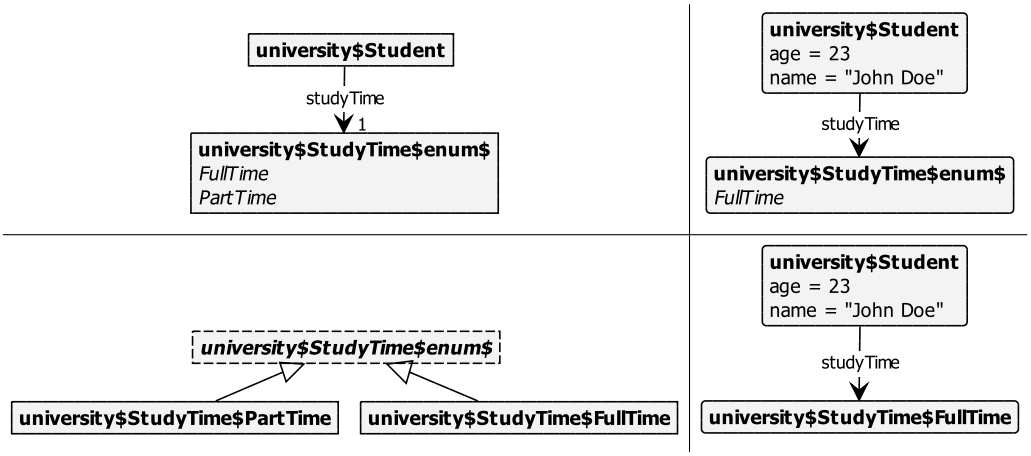


Figure 4.22: Type and instance graphs generated for enumerations using flags or nodes.

Figure 4.23: Rule graphs generated for enumerations using flags. The left rule checks if no flags exists, the right rule checks that two flags exist (also matches if more than two flags exist).

are represented by nodes which are a sub type of an (abstract) node representing the enumeration itself. When the `Flags` alternative is used, only one node is created for an enumeration, with flags for each of its values. When this alternative is used, additional constraint rules may be generated to enforce an enumeration value uses exactly one flag. Figure 4.23 shows an example of such rules. One rule checks the absence of all possible flags. If an enumeration node has no flags, this rule will match. The other rule uses a regular expression to match each possible pair of flags. If an enumeration node contains two or more of such flags, one of these pairs will match and consequently the entire rule will match.

- The `Fields` option controls the graph structures used to represent the various *fields* and their types. `OppositeEdges` controls whether or not opposite edges are drawn and required between fields that have been marked opposite. These edges are shown in fig. 4.8, and would not be present if the option was turned off. This does not influence the opposite constraint checks however. The `Containers` option controls the representation of container types in the graph. The `IntermediatePostfix` option allows the type of the container to be appended to the name of an intermediate node of that container if it exists. The postfixes used are `_Set`, `_Bag`, `_Ord` and `_Seq` for the setof, bagof, ordof and seqof containers respectively. This is only applicable if a meta graph is not used, otherwise the meta graph provides this information. The effect of this option is shown in fig. 4.24. The `Ordering` option controls the representation of ordering information

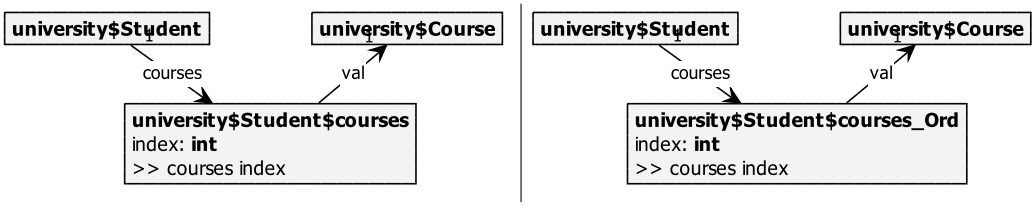


Figure 4.24: Effect of the `IntermediatePostfix` option. On the left the option is turned off, on the right it is turned on.

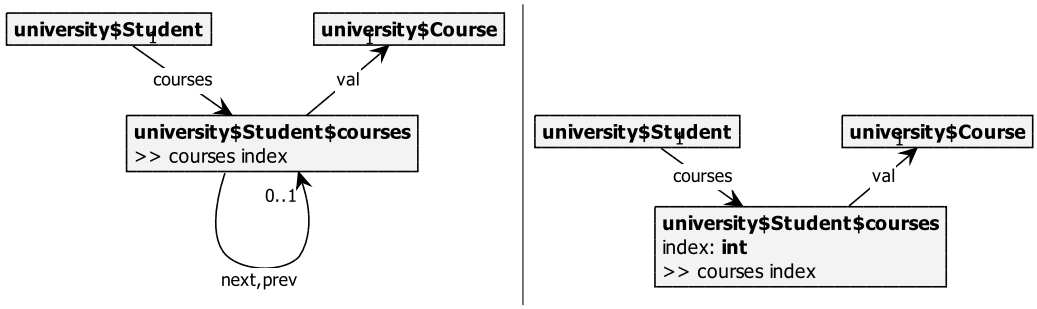
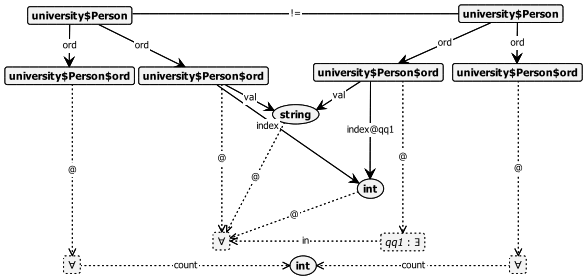


Figure 4.25: Effect of the `Ordering` option on intermediate nodes of ordered container types in type graphs. Note that the index of the values is hidden in the right graph, but internally still present.



When using next edges, not the index value, but the next element is compared for equivalency.

Figure 4.26: Constraint rule for the identity property, for an ordered container using next edges.

between container values. The `Representation` determines the graph structure used. Either an index value is used (the `Index` option), or edges between the nodes (the `Edge` option).

- If the `Index` option is used, an integer attribute is added to the node representing the container value which has an index value.
- If the `Edge` option is used, edges pointing to the next element are added to the graphs. When this option is used, the `Prev` option may additionally be used to add edges pointing to the previous element as well to the graph.

The effect of these alternatives is shown in fig. 4.25, which shows the resulting graph for the same ordered containers with both alternatives chosen in the configuration. The effect on instance graphs for the same containers is shown in fig. 4.27.

A constraint rule that is largely affected by this option is that of the identity property for ordered containers. The constraint rule created for indexed containers is shown in fig. 4.11, the constraint rule for values using next edges is shown in fig. 4.26. The `Placement` option controls what nodes are used to add the ordering information to. This may be the intermediate nodes themselves (the `UseIntermediate` alternative) or the value nodes when applicable (the `PreferValue` alternative). If the latter is not possible, intermediate nodes are used instead. For the ordering information to be placed on a value node, the following conditions must hold:

1. The type of the value node is a *Class*.
2. The name of the edge or attribute used to represent the ordering is not used by a *Field* of the class or any of its superclasses.
3. The value is never shared between any other ordered container.
4. The value is contained within a containment relation.

- The `IntermediateNodes` option controls the creation and usage of *intermediate* nodes in the graph for fields and container types. Three alternatives are possible:

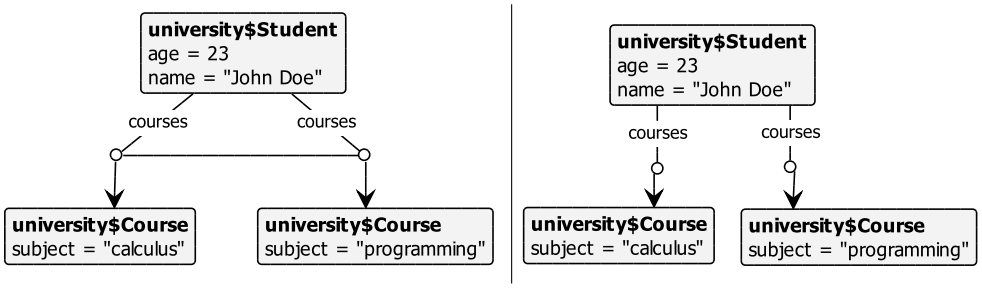


Figure 4.27: Effect of the `Ordering` option on intermediate nodes of ordered container types in instance graphs.

- `Always` : All fields use an intermediate node. This results in the largest graphs, but all fields will consistently have an intermediate node.
- `Container` : All container fields use intermediate nodes, even if not strictly required. Other fields may use an intermediate node when required, such as for opposite fields.
- `Required` : Intermediate nodes are only created when required.

The `Required` alternative depends heavily on other options, since they may or may not depend on the presence of intermediate nodes. An intermediate node is required when one or more of the following conditions hold:

1. It represents a container with non-unique values.
2. It represents an ordered container, and the ordering representation is not placed on the value of the container (the `PreferValue` option).
3. It is part of a field for which an opposite property exists, and the `OppositeEdges` option is enabled.
4. The container contains other containers.

The effect of the `IntermediateNodes` option is shown in fig. 4.28. Note that for the image associated with the `Required` alternative, the `PreferValue` option was used to allow the use of a direct edge. Furthermore, the relation between classes `F` and `G` was marked as containment to allow the importer to infer that a unique index for all `G` instances is possible.

- The `Default` option controls the use of *default values* in the model (the defaultValue property). The `UseRule` option creates a rule for the default value when enabled. This allows the actual value used to be tracked and exported later on, and may be used during model transformations. The `ApplyImport` options enables the application of the default value to actual instances when applicable during import. This ensures that values that are unset in the instance model will be assigned their default value when applicable.

- The second part of the configuration tree concerns the use of *properties* and *constraint rules*. The `Properties` option controls the generation of properties during export. Each property can be individually turned on or off. Turning a property off will stop the exporter from searching for a certain graph pattern or constraint to generate the property, which may help if the graphs contain structures which are not compatibly with the exporter. The `ConstraintRule` option allows the types of constraints rules generated and analyzed to be selected. Each option controls whether or not that kind of constraint rule is generated on import or searched for on export. Certain properties may only be expressed as constraint rules (such as identity and keyset), so turning these options off may cause this information to be lost.

- Finally, the configuration allows the *strings* used for the various graph structures and elements to be changed. This entails the names of edges such as used for opposites and ordering, but also names for container types and postfixes for enumerations and data types.

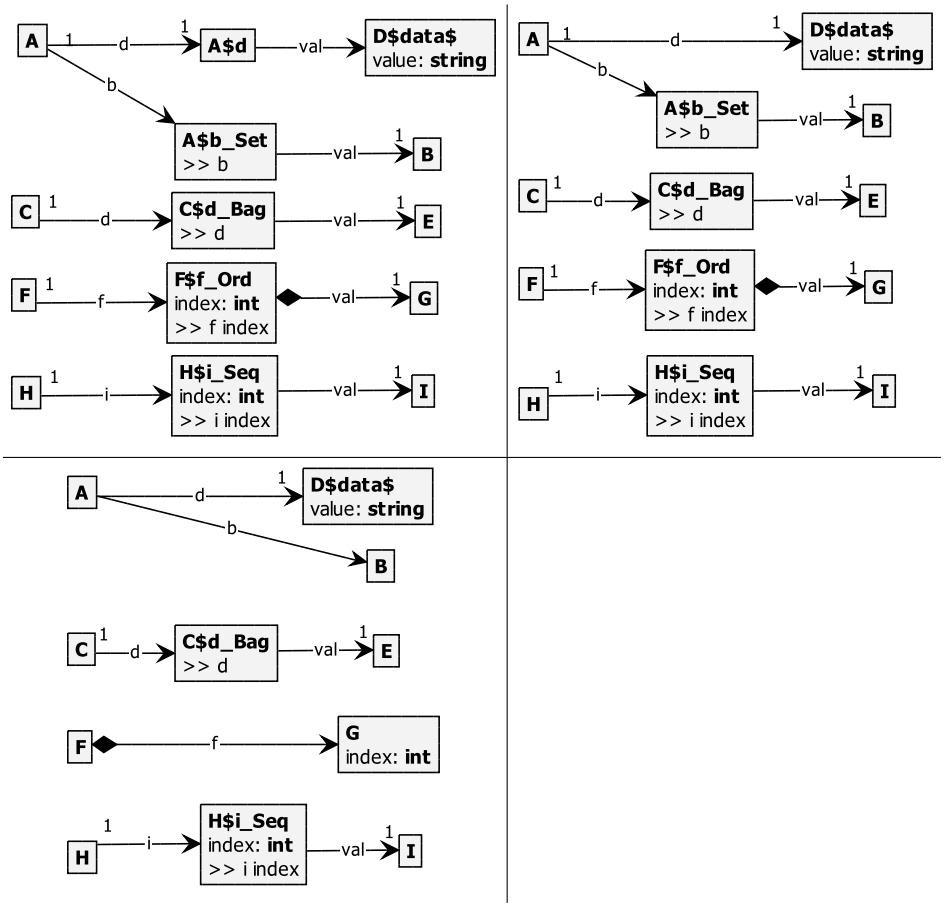


Figure 4.28: Effect of the `IntermediateNodes` option on intermediate nodes in type graphs. The effects of the `Always`, `Container` and `Required` alternatives are shown respectively.

An important aspect of the configuration is that it is required to use the same configuration for both exporting and importing the same model. If a different configuration is used, the exporter may incorrectly interpret an existing graph, resulting in no model or an incorrect model being generated. Therefore, the translator allows configurations to be stored and loaded as required, as is explained in section 6.1.3.

4.5 Overview

An overview of the mapping between the conceptual model and GROOVE is given in the following table. For each main concept in the conceptual model a short description is given on how it is represented in GROOVE graphs.

Feature	Representation
Namespaces	Type name in nodes with a separator
Classes	Type nodes in type graph
Enums	Type nodes in type graph, with a special enumeration postfix
Tuples	Type nodes in type graph, with a special tuple postfix
Data types	Using built-in data types
Custom data types	Type nodes in type graph, with a special data type postfix. Additionally a single string attribute is added
Inheritance	Subtype edges in the type graph
Attributes	An edge to the node representing the attribute type
Relations	An edge to the node representing the attribute type
Multiplicities	The `out` aspect for field edges
Abstract	The `abs` aspect for type nodes
Opposite	Using opposite edges between intermediate nodes, and/or an opposite rule.
Identity	An identity rule
Keyset	A keyset rule
Containment	The `part` aspect for edges
Default value	Directly inserting the value in instance graphs and/or a default value rule
Unique containers	A unique value rule, or connecting values directly by an edge, omitting an intermediate node
Ordered containers	A ordered value rule
Objects	Nodes in an instance graph
Object identities	Optionally by using the `id` aspect on nodes in the instance graph

4.6 Conclusions

This chapter describes the mapping between GROOVE and the conceptual model. The actual implementation is discussed in appendix C. This mapping answers the second part of the third research question, namely how the conceptual model can be mapped to GROOVE. In the following chapter, the mapping between the external languages and the conceptual model is described, which will answer the first part of this research question, solving the question entirely.

An important aspect of the mapping between the conceptual model and GROOVE is its configurability. The configurability provides a means to change the way the conceptual model is represented by graphs in GROOVE. This gives the user more control over the import and export process, and allows a choice to be made between more detailed graphs or simpler graphs. Implementation details of the configuration can be found in section 6.1.3.

Chapter 5 Mapping CM to Ecore, GXL and DOT

This chapter elaborates on the implementation of the mapping between the various graph languages and the conceptual model. The mapping is split in a mapping between the language and the type model, and between the language and the instance model. A separation is made between importing and exporting. Although ideally importing and exporting would be the exact opposite of each other, this is not always possible. In some cases the differences are minor, but in others the difference is major, such as for DOT.

5.1 Ecore

The mapping from an Ecore model to a type model is fairly straightforward. This is mainly because the conceptual model was designed with primarily Ecore in mind. Therefore, many of the concepts found in Ecore match closely to those found in the conceptual model, and vice versa, which makes the translation of them rather trivial. An Ecore document is translated to and from a `Resource` by the Ecore runtime. Such a `Resource` is added to a `ResourceSet`, which may contain multiple of such `Resources`. Such a `Resource` could be either an Ecore model, or an Ecore instance model. The use of a `ResourceSet` allows the various references between these models to be automatically resolved.

5.1.1 Importing type models

When importing an Ecore model, a new `ResourceSet` is created and the resource associated with the Ecore model is inserted. Then, all elements in this resource are traversed in order to translate it into a type model. Each element in Ecore is always contained within a package, class or data type. Thus, the following case distinctions are made:

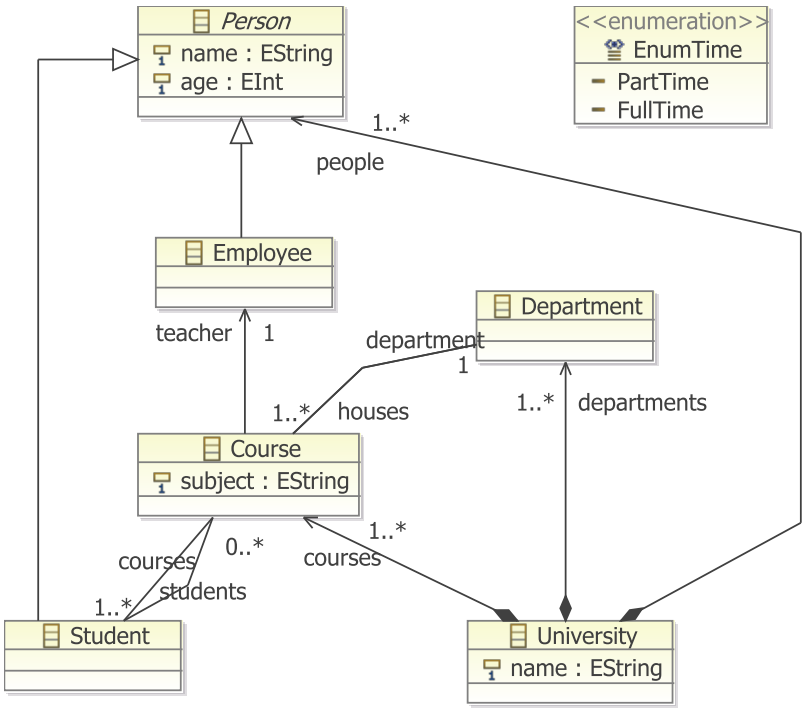
1. **EPackage**
2. **EClass**
3. **EDataType**
4. **EEnum** (as a special case of **EDataType**)

Other Ecore elements will always be contained by these elements (such as references and attributes).

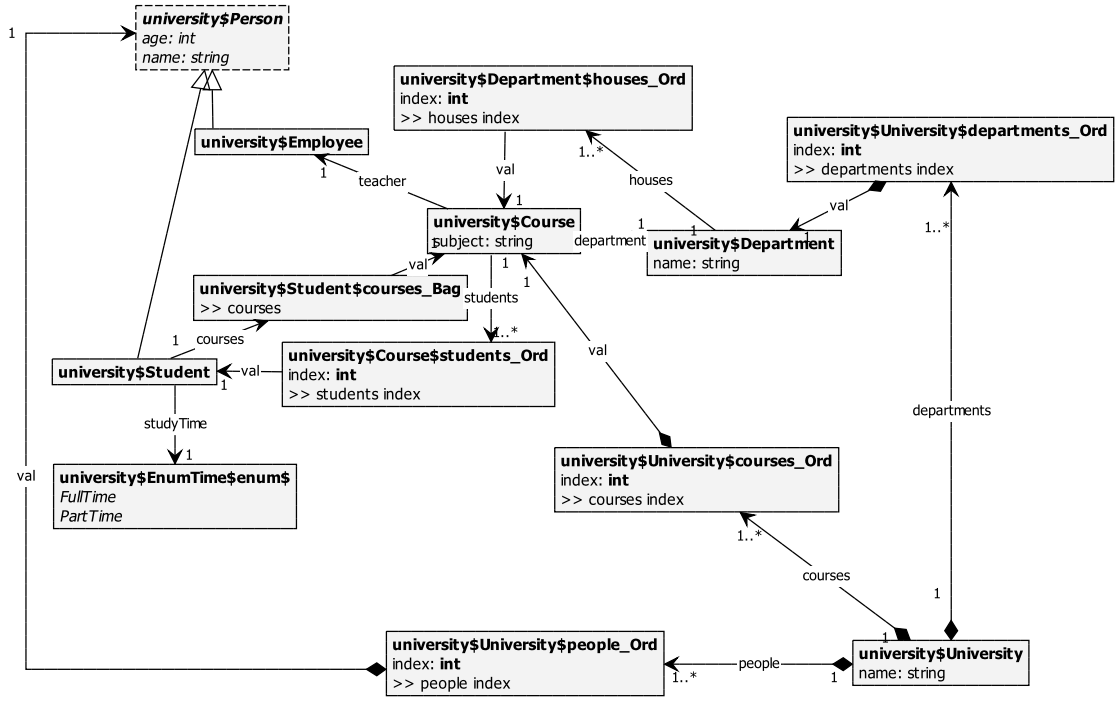
An example Ecore model and corresponding type graph are shown in fig. 5.1. It shows the use of various container formats, intermediate nodes, abstractness and enumerations. In order to simplify the type graph, opposite edges are hidden although they are used in the Ecore model.

EPackage

An **EPackage** is not directly mapped into the conceptual model. Rather, it will provide a namespace for the elements contained within the package. Furthermore, each package will be explicitly put into a registry of packages in the `ResourceSet`, so other models may reference these packages or elements

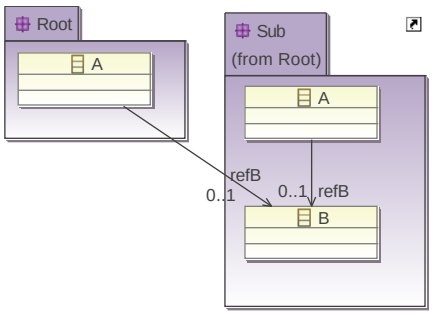


(a) Ecore type model

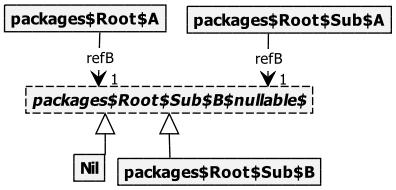


(b) GROOVE type graph

Figure 5.1: Example Ecore model and corresponding GROOVE type graph.



(a) Ecore model with packages



(b) GROOVE type graph with namespaces

Figure 5.2: Example of an Ecore type model with packages and correspodning GROOVE type graph.

contained within these packages (such as classes). The namespace of a package is based on its name, and the namespace of the parent package. A root package of an Ecore model will be contained in the root namespace of the conceptual model.

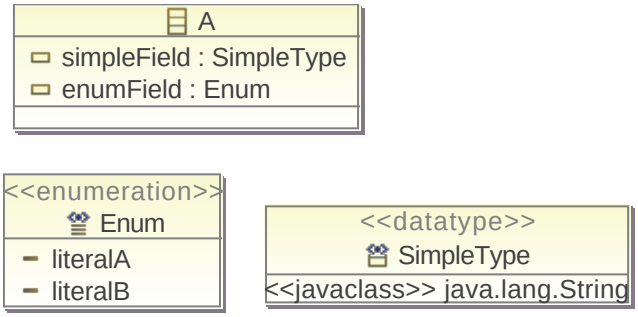
EClass

An **EClass** will be mapped to an *Id* in the type model, which is an element of $Class_{\mathrm{Tm}}$. This *Id* is based on the name of the class and the namespace provided by the package the class is contained in. If the **EClass** is abstract (or an interface), an abstract property is added to the type model bearing the *Id* of the class. For each superclass of the **EClass**, the corresponding class in the type model will be added to the Inh_{Tm} set paired with the current class in the type model.

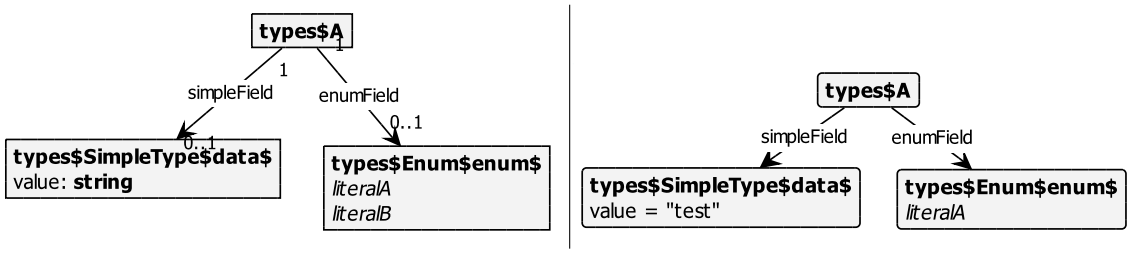
For each **EReference** and **EAttribute** contained within the **EClass**, a new element is added to $Field_{\mathrm{Tm}}$ in the type model, complete with the proper signature in $\mathrm{FieldSig}_{\mathrm{Tm}}$. The name of the field is determined by the name of the **EReference**/**EAttribute**, the multiplicity is determined by the lower bound and upper bound. The type of the field is determined by the type of the attribute (which will be the type created for the referenced **EDataType**) or reference (which will the class referenced by EClass). If the type is a class and the multiplicity is 0..1, then the type will be that of a nullable class. Otherwise, it will be of a proper class. If the upper bound is larger than 1, then a container type is generated. The type of the container is determined by the ordered and unique attributes of the **EReference**/**EAttribute**.

For **EAttribute**s, the default value is also taken into account. This will always be a literal string in an Ecore document, which will be translated to the $Constant_{\mathrm{Tm}}$ set in the type model, with the proper entry in $\mathrm{ConstType}_{\mathrm{Tm}}$ based on the type of the attribute. The actual value (which is already known by Ecore) will be added into the $\mathrm{DefaultValue}_{\mathrm{Im}}$ function, mapping the generated $Constant_{\mathrm{Tm}}$ element to the actual $Value_{\mathrm{Im}}$ element. Finally, a defaultValue property is added to the type model, referencing the **EAttribute** for which it is defined.

For **EReference**s it is possible to specify an opposite **EReference**. If this is the case, opposite properties will be added to the type model, referencing both **EReference**s. Since each reference is visited when translating the Ecore document, it is ensured that an opposite property is added for both directions. Additionally, an **EReference** may specify a containment relation. If that is the case, a containment



(a) Ecore type model with different types



(b) Corresponding GROOVE type and instance graphs

Figure 5.3: Imported Ecore class, enum and data type, and corresponding GROOVE graphs.

property is added to the type model as well. Finally, if the reference specifies a set of keys, then a keyset property is added, referencing the field generated for this **EReference**, as well as each field generated for the **EAttribute**s in the keyset.

Finally, an **EClass** can have one **EAttribute** designated as its identifying attribute. For this attribute an identity property is added to the type model, referencing the field created for that attribute.

EDataType

Ecore defines a set of built-in data types, as well as allowing the user to specify custom data types. The built-in data types are, where applicable, mapped to the data types in the conceptual models (booleans, integers, real numbers and strings). Other **EDataType** elements are mapped to an element in $UserDataType_{\mathrm{Tm}}$ in the type model. Just like a class, its *Id* based on the name of the data type and its containing package. The underlying Java class information is ignored. Any values for this data type will be treated as an opaque string.

EEnum

As a special case of **EDataType**, **EEnum** is mapped to an element in $Enum_{\mathrm{Im}}$ in the type model. Like classes and data types, the *Id* is based on its name and containing package. For each **EEnumLiteral** contained within the **EEnum**, an element is added to $EnumValue_{\mathrm{Tm}}$, based on the name of the literal (other information of the literal is lost).

Operations and generics

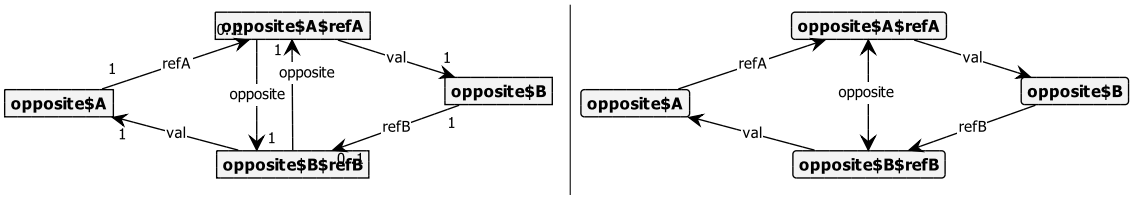
Operations and generics are not supported. Both concepts are not represented in the conceptual model. Operations and generics both have no influence on instance models. Because of this, the

choice was made not to include them, as they only apply to type models, which, considering from a GROOVE point of view, are more or less static and not relevant to model transformations.

Operations model executability, which does not apply to a single instance model (but rather the transformation thereof). Generics do not apply to instance models either, as the type of objects and values in an instance model is always set to a specific type, and not a generic type.

5.1.2 Importing instance models

(a) Ecore type model with opposite relations



(b) GROOVE type and instance graphs with opposite edges

Figure 5.4: Imported Ecore model showing opposite relations, and corresponding GROOVE graphs.

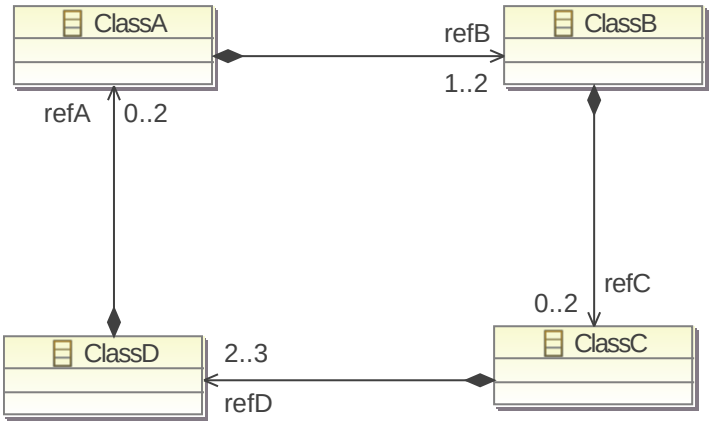
Ecore instance models are imported in a similar fashion to type models. First, an Ecore instance model document is loaded as a `Resource` and inserted into a `ResourceSet`. In order to be able to resolve all types, this `ResourceSet` must also contain an appropriate Ecore model, such as one loaded when importing a type model.

Each **EObject** in the instance `Resource` is translated to an element in $Object_{\mathrm{Im}}$. The type of the **EObject** will be an **EClass** (the `ResourceSet` will automatically resolve this), which is used to map to an element in $\mathrm{ObjectClass}_{\mathrm{Im}}$. The **EObject** furthermore has a unique location in the Ecore document, as it is contained by one other object (multiple objects contained by the same object are further distinguished by their index in the document). This location is used as an entry in $\mathrm{ObjectId}_{\mathrm{Im}}$.

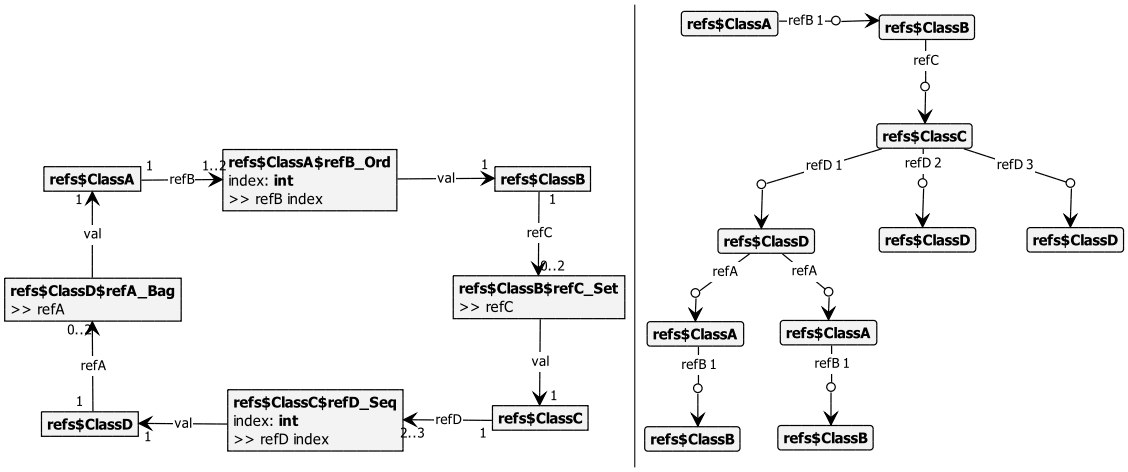
For each attribute and reference in the class that is the type of the object, a value entry is added to $\mathrm{FieldValue}_{\mathrm{Im}}$. The value is either another object for references, or a data type value (or enumeration value) for attributes. For attributes and references with an upper bound larger than 1, an appropriate $ContainerValue_{\mathrm{Im}}$ is used instead. For attributes, the value can either be of a built-in data type or a custom data type. Built-in types are translated accordingly, custom data type values are treated as strings (and are available as such in the Ecore instance model).

5.1.3 Exporting type models

Just as importing an Ecore model requires a `ResourceSet`, so does exporting it. When exporting a type model, first a `ResourceSet` is created, and an empty `Resource` is added to it. Next, all the elements of the type model are added into the `Resource` by translating them to their Ecore counterparts. This happens in a fashion similar to that of importing a type model, by iterating over the various elements in the type model, and translating them and any elements contained within them or referenced by them.



(a) Ecore type model with various relations



(b) GROOVE type and instance graphs with different containers

Figure 5.5: Imported Ecore model showing various relation types. On the left the resulting type graph, on the right an example instance graph.

Before any element in the type model can be translated, first a way needs to be defined to translate the namespaces in a type model to packages. This is done by walking through the namespaces of an *Id* of each element that has one, and creating an EPackage in the `Resource` for each namespace, making sure that the packages are properly contained within each other.

Type models are translating by visiting the classes ($Class_{\text{Tm}}$), enumerations ($Enum_{\text{Tm}}$), custom data types ($UserDataType_{\text{Tm}}$) and properties ($Prop_{\text{Tm}}$).

Classes

Classes are translated into **EClass** objects. The name of this **EClass** will be the name part of the *Id* that represents the class in the type model, and it will be added to the package that corresponds to the namespace of the *Id*. Next, each relevant element in Inh_{Tm} is translated into the **EClass** representing the superclass, which is added to the set of superclasses in the ECore model. Finally, each relevant $Field_{\text{Tm}}$ entry (with corresponding $\text{FieldSig}_{\text{Tm}}$ value) is translated into either an **EAttribute** or **EReference**. For container types, the appropriate unique and ordered attributes are set.

If the type of the Field is a Class or a container of a Class, it will be translated into an **EReference**. Otherwise, an **EAttribute** is created. For attributes, the type may be an **EEnum** (translated from the referred $Enum_{\text{Tm}}$ entry) or **EDataType** (if the type is one of the built-in data types or an entry in $UserDataType_{\text{Tm}}$).

Recursive containers are not supported by Ecore. Therefore, if a recursive container is encountered, a new **EClass** is generated, with one field that references the subcontainer. The field will have an unbound multiplicity (as subcontainers in the conceptual model do not enforce a multiplicity as well), and its uniqueness and ordering depend on the type of the container that it represents.

A special case is the translation of tuple types. Ecore has no notion of tuple types, so they are translated to a new **EClass** instead, with an **EReference** or **EAttribute** for each tuple element instead. A similar problem exists for recursive containers (that is, containers within containers). However, these occur infrequent enough not to be translated, and they will be ignored instead.

Data types and enumerations

The built-in data types can be mapped directly to the predefined data types in Ecore. The custom data types are mapped to new instances of **EDataType**. However, there is no information about the underlying Java class information in the conceptual model, so instead the `String` class is used, as it is guaranteed to be able to hold the string that represents values of this data type.

Enumerations are mapped to **EEnum**s. For each elements in $EnumValue_{\text{Tm}}$, a new **EEnumLiteral** is created contained by an **EEnum**, of which the literal name is as it occurs in the conceptual model.

Properties

Properties allow themselves to be translated fairly trivially. They are translated as follows:

- Abstract: The class referenced by the abstract property is set as abstract in Ecore.
- Containment: The relation referenced by this property is set to be a containment relation in Ecore.
- Identity: The attribute referenced by this property is set to the Ecore identity property of the class the attribute belongs to. Note that although the conceptual model allows multiple attributes in identity properties, only one can be set as the identity property in Ecore. Other attributes are simply ignored.

- Keyset: Each attribute referenced by the keyset property is added to the keys property of the **EReference** that is the translation of the relation referenced by the property.
- Opposite: Both **EReference**s, as translated from the relations of the opposite property, are marked as opposite of each other.
- DefaultValue: Ecore only allows attributes to have default values. Default values for references in the conceptual model are ignored. The $Constant_{\mathrm{Tm}}$ in the conceptual model is mapped directly to the literal string of the default value for the **EAttribute** that represents the field in the conceptual model, for which the property was added.

5.1.4 Exporting instance models

Exporting an instance model to an Ecore instance model document requires, just like with an import, a `ResourceSet` which contains the references to the required types. When exporting an instance model, all the elements in $Object_{\mathrm{Im}}$ are mapped to an **EObject**. This is done by using the Ecore factories of each **EClass** that is the translation of the corresponding $Class_{\mathrm{Tm}}$ entry from $\mathrm{ObjectClass}_{\mathrm{Im}}$. The $\mathrm{ObjectId}_{\mathrm{Im}}$ function is ignored, as Ecore does not support names for objects. The $\mathrm{DefaultValue}_{\mathrm{Im}}$ function is ignored as well, as the literal values for default values are used instead, and mapped from the type model.

For each entry in $\mathrm{FieldValue}_{\mathrm{Im}}$, the corresponding attribute or reference value in Ecore is set. The normal data type values are trivially translated into their Ecore counterparts. For custom data types, the string value can directly be used, as the corresponding **EDataType** always accepts a string literal as its value for storage. Enumeration values are created using the corresponding **EEnum**, which creates the correct value when given the literal from the conceptual model.

Just as in the type model, tuples in the instance model cannot directly be translated. Therefore, an instance is created of the **EClass** representing the tuple, and each value of the tuple in the conceptual instance model is translated into an attribute or reference value in the Ecore instance model. Recursive containers are treated similarly, with an instance being created for the class representing the recursive container. The values of the container are translated and assigned to the **EAttribute** or **EReference** of this container class representing the container values.

5.1.5 Issues

Because the conceptual model is mostly based on Ecore, there are not many issues when importing or exporting. The main issue is that some information is lost when importing, most notably the operations and generics information. In appendix B.1 some more details are given to how and why this information is lost. Most of the other problems occur during the export process, as in some cases the conceptual model is capable of representing information that Ecore is not. For example, recursive containers are a concept that Ecore does not support.

Information loss occurs also when importing data types and enumerations. In both cases some of the properties which are closely related to the underlying Java model are ignored. For data types, this is for example the Java class that would be used and for enumerations the value of the literal. The conceptual model would need to be modified to support this, but this would lead to a too close focus to the Ecore/Java models which introduces additional problems for other languages.

Finally, Ecore also supports the definition of interfaces, which are, in short, abstract classes with only abstract operations defined. Since operations are not supported, these classes are marked as abstract instead. Upon export, the fact that these were originally interfaces is lost. However, this only has a small impact on the model compared to the loss of other information.

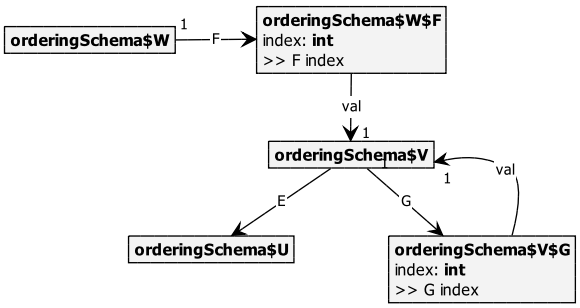


Figure 5.6: Simple GXL type graph with ordered edges.

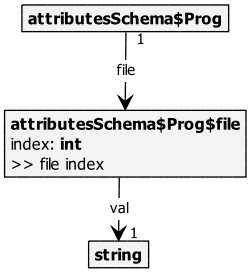


Figure 5.7: Simple GXL type graph with attributed nodes.

5.2 GXL

The mapping from GXL to the conceptual model has to deal with a few more incompatibilities than the mapping of Ecore, as the conceptual model is less closely related. In a GXL document, multiple graphs can be found. A graph is either a GXL type graph (which makes use of the GXL meta schema) or a GXL instance graph (which is typed by the aforementioned GXL type graphs).

Elements in GXL can be typed (and this is often required). These types refer to elements in other GXL documents (possibly the same document). In the GXL meta schema these types are often self-referential as the meta schema can describe itself. For instance graphs, these types refer to elements in the type graph, the type graph refers to the meta schema.

5.2.1 Importing type graphs

The GXL meta schema mainly consists of three parts: The graph part, the attribute part and the value part. The graph part describes the structure of the graphs it types, such as allowed nodes, edges and subgraphs (as well as relations). The attribute part describes the various possible (types of) attributes of elements in the graphs. Finally, the value part allows default values to be specified in a type graph for various attributes.

A type graph consists of a single actual graph with nodes and edges typed by the meta schema. Other elements are not allowed or ignored. Nodes specify the elements that are allowed in instance graphs, including attributes and edges. Edges specify various relations between, or properties of, such elements.

```
<gxl xmlns:xlink="http://www.w3.org/1999/xlink" xmlns="http://www.gupro.de/GXL/gxl
    -1.0.dtd">
  <graph id="ordering-schema" edgeids="true">
    <type xlink:href="../../../metaschema/gxl/gxl-1.0.gxl#gxl-1.0"/>
    <node id="orderingSchema">
      <type xlink:href="../../../metaschema/gxl/gxl-1.0.gxl#GraphClass"/>
      <attr name="name">
        <string>orderingSchema</string>
      </attr>
    </node>
```

Listing 5.1: Example GXL document header and graph node

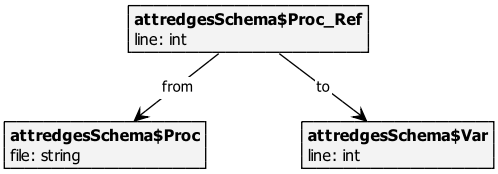


Figure 5.8: Complex GXL type graph with an attributed edge.

GraphClass

First, all **GraphClass** nodes are identified, and their hierarchy is computed. These nodes provide the namespaces of the elements in the conceptual model. The nodes and edges contained by the graphs these **GraphClass** nodes represent will have their corresponding identifiers in the conceptual model based on the name of the **GraphClass** node as their namespace. GXL does not support graphs directly containing other graphs, but rather nodes, edges and relations containing other graphs. When determining the namespace, these elements are ignored and only the name of the **GraphClass** nodes is taken into account.

NodeClass

NodeClass nodes are translated into classes in $Class_{\mathrm{Tm}}$. The related AttributeClass nodes are translated into the fields of such classes. The *Id* of the class is based on the name of the **NodeClass** and its containing **GraphClass**. If the **NodeClass** is attributed as abstract, a new abstract property is added to the type graph. A **NodeClass** may inherit from another **NodeClass** via the **isA** edge. If this is the case, the proper subclass relation will be added to Inh_{Tm} in the type model.

EdgeClass

EdgeClass nodes, and their subtypes **AggregationClass** and **CompositionClass**, are translated into either relations or fields. If the edge is a simple edge directed from one **NodeClass** to another **NodeClass**, then a new relation is added to $Field_{\mathrm{Tm}}$ based on the class *Id* of the source **NodeClass** with the type being that of the class *Id* of the target **NodeClass**.

It is possible that edges connect to other edges. Additionally, edges may be abstract, inherit from other edges or feature attributes. Such edges are promoted to classes in the conceptual model as well, as these features cannot be represented by relations. Such promoted classes will have two relations, namely that of the class they connect to and the class they originate from. If an edge is the source or target of such a promoted edge, it will be promoted to a class as well.

```
<edge id="c3" from="orderingSchema" to="W">
  <type xlink:href="../../../metaschema/gxl/gxl-1.0.gxl#contains"/>
</edge>
<node id="W">
  <type xlink:href="../../../metaschema/gxl/gxl-1.0.gxl#NodeClass"/>
  <attr name="name">
    <string>W</string>
  </attr>
  <attr name="isabstract">
    <bool>false</bool>
  </attr>
</node>
<edge id="c4" from="orderingSchema" to="E">
  <type xlink:href="../../../metaschema/gxl/gxl-1.0.gxl#contains"/>
</edge>
<node id="E">
  <type xlink:href="../../../metaschema/gxl/gxl-1.0.gxl#EdgeClass"/>
  <attr name="name">
    <string>E</string>
  </attr>
  <attr name="isabstract">
    <bool>false</bool>
  </attr>
  <attr name="isdirected">
    <bool>true</bool>
  </attr>
</node>
```

Listing 5.2: Example GXL nodes and edges

```
<node id="domainSeq">
  <type xlink:href="../../../metaschema/gxl/gxl-1.0.gxl#Seq"/>
</node>
<node id="domainString">
  <type xlink:href="../../../metaschema/gxl/gxl-1.0.gxl#String"/>
</node>
<edge id="e1" from="Prog" to="file">
  <type xlink:href="../../../metaschema/gxl/gxl-1.0.gxl#hasAttribute"/>
</edge>
<edge id="e2" from="file" to="domainSeq">
  <type xlink:href="../../../metaschema/gxl/gxl-1.0.gxl#hasDomain"/>
</edge>
<edge id="e3" from="domainSeq" to="domainString">
  <type xlink:href="../../../metaschema/gxl/gxl-1.0.gxl#hasComponent"/>
</edge>
```

Listing 5.3: Example of GXL attributes

If the edge is either an **AggregationClass** or **CompositionClass**, a new composite property is added to $Prop_{\mathrm{Tm}}$ featuring the class the edges connects from. If the edge was promoted, the property is based on the relation from the source class to the promoted class.

The multiplicity of the created field is determined by the **limits** property of the target edge. For promoted classes, the **limits** property of the source edge is also taken into account. Edges can be given an ordering. If this is the case, an ordered container type will be used for these edges. GXL always allows multiple edges of the same type to connect the same nodes, so the containers for relations in the type model are always of a non-unique type.

RelationClass

The **RelationClass** and **RelationEndClass** node types are ignored, as the conceptual model does not support relations.

AttributeClass

Almost all elements in a GXL type graph support the specification of attributes. However, these can only be mapped to the fields in the conceptual model, which only classes support. Therefore, attributes are only supported for the **NodeClass** types, and promoted **EdgeClass** types. **AttributeClass** nodes define attributes, which have a specific **Domain** indicating their type, and optionally a **Value** as a default value. The **Locator** domain is treated as a string as locators can always be represented as such. The **AtomicDomain** types (**Bool**, **Float**, **Int**, **String**) are mapped to the built-in types in the conceptual model. **CompositeDomain** types are mapped to container types for **Bag**, **Set** and **Seq**, and tuple types for **Tup**. Finally, for **Enum** domains a new enumeration is added to $Enum_{\mathrm{Tm}}$, with each contained **EnumVal** being translated to a corresponding $EnumValue_{\mathrm{Tm}}$ entry based on the value string. Note that in GXL, attributes have no multiplicity specification. Rather, the multiplicity of container types is always assumed to be 0..*, that of other types is 1..1.

If an **AttributeClass** has a default **Value**, it is translated to a $Constant_{\mathrm{Tm}}$ entry based on the value string, URI string or collection of those. The type in $\mathrm{ConstType}_{\mathrm{Tm}}$ is that of the **Domain** of the attribute. Instance models will have an actual $\mathrm{DefaultValue}_{\mathrm{Im}}$ entry based on the generated $Constant_{\mathrm{Im}}$ entry and the actual value it represents.

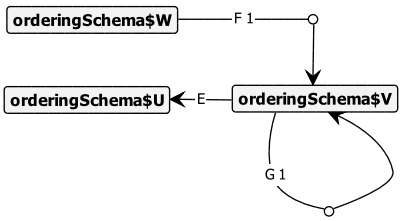


Figure 5.9: Simple GXL instance graph with ordered edges.

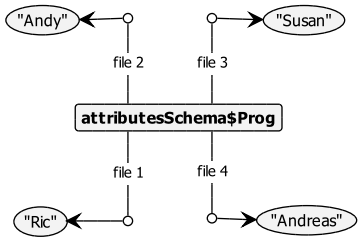


Figure 5.10: Simple GXL instance graph with attributed nodes.

Names

Names in the type graph are treated as being able to represent having a namespace as well, being separated by a period. The reason is that although on import a subgraph can represent a namespace, on export this is not always possible and namespaces are flattened to names instead, using the dot as a namespace separator (see section 5.2.3).

5.2.2 Importing instance graphs

Instance graphs are imported by iterating through all subgraphs and nodes contained by those subgraphs. Each node is typed by a **NodeClass** node in the type graph, which has been translated to a class in the type model. Edges between those nodes are typed by an **EdgeClass**, which is translated either as a field or a class if it was promoted.

First, all nodes that do not contain any subgraphs are traversed. For each such a node, a new element is added to $Object_{\mathrm{Im}}$, and the identifier of that node (if any is available in the GXL document) is used for $\mathrm{ObjectId}_{\mathrm{Im}}$. Next, for each outgoing edge that is of an edge type which was not promoted, an entry is added to $\mathrm{FieldValue}_{\mathrm{Im}}$. For edges that were promoted, a new object is added to $Object_{\mathrm{Im}}$, and an entry is added to $\mathrm{FieldValue}_{\mathrm{Im}}$ for both the source and the target. For each attribute of a node or promoted edge an entry is added to $\mathrm{FieldValue}_{\mathrm{Im}}$ as well, based on the actual value of the attribute.

Nodes that do contain subgraphs are not instantiated as objects. Rather, their subgraphs are explored and the nodes contained within it are translated. The node containing the graph is not instantiated as an object and the relation of containing a graph is not supported.

5.2.3 Exporting type graphs

Whereas the type graph importer allows the use of subgraphs by treating them as namespaces, the type graph exporter will not use the same approach. The reason is that when exporting a type model with multiple namespaces inside the root namespace (for example the *Id*s namespace1.class1 and namespace2.class1), actually two root graphs would be required in the associated exported instance

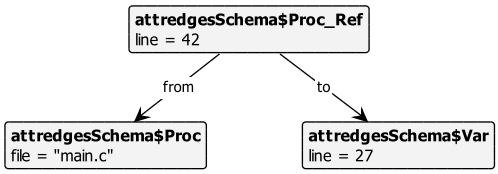


Figure 5.11: Complex GXL instance graph with an attributed edge.

models. However, a GXL graph allows only one such a root graph. Therefore, *Id*s are flattened into single strings which are used for the names of the exported elements (the importer has provisions to support both types of namespacing).

When exporting the type model, first all classes are exported as attributed nodes, with edges to other nodes for each relation. Thus, each element in $Class_{\mathrm{Tm}}$ is translated into a **NodeClass** node, with for each corresponding element in $Field_{\mathrm{Tm}}$ either an **AttributeClass** or an **EdgeClass** depending on whether the field specifies an attribute or a relation.

For attributes, the multiplicity is ignored, as GXL has no provisions for this concept for attributes. Additionally, GXL has no type for the ordof container type. This type is exported as a GXL **Seq** type, which allows for a wider range of values (they do not have to be unique as opposed to ordof values).

The built-in data types are mapped directly to their counterparts in GXL. For the custom data types the **String** type is used, as GXL has no direct provision for custom data types. This will cause the fact that this is a custom data type (with its own *Id*) to be lost, but the actual values can still be stored. Enumerations are mapped to the **Enum** data type in GXL, and **EnumVal** nodes for each literal value in the type model. Tuples are mapped either to **Tuple** attributes or nodes representing the tuple. If the tuple is used as an attribute and consists only of attribute types, then the tuple is translated to a GXL attribute. Otherwise, a node representing the tuple is created, with attributes and edges referencing the types the tuple consists of.

Not all the properties in the type model can be translated to GXL. The abstract property is translated to the **isabstract** attribute of the various class nodes. The containment property is translated by promoting the **EdgeClass** nodes corresponding to the containment relation to **CompositionClass** nodes. The default property is translated by creating the proper **Value** nodes representing the default value, and adding a **hasDefaultValue** edge to the **AttributeClass** node corresponding to the attribute, pointing to the default **Value** node. The identity, keyset and opposite properties are ignored, as they are not supported by GXL.

5.2.4 Exporting instance graphs

Instance models are exported by iterating over all $Object_{\mathrm{Im}}$s and creating nodes in the graph for each object. For each object, a new node is created typed by the **NodeClass** that is linked to the class as determined by $\mathrm{ObjectClass}_{\mathrm{Im}}$. For attributes in $\mathrm{FieldValue}_{\mathrm{Im}}$ entry, an attribute value is set for the node, based on the name of the field and its value. For relations, an edge is created, based on the **EdgeClass** (or **CompositionClass**) node in the type graph related to the corresponding relation.

For relations with container values, multiple edges are created to the corresponding nodes. If the container is ordered, then the edges will be given an ordering as well.

The built-in data types are exported as the corresponding GXL attributes. For custom data types, a string attribute is used. Enumerations in GXL instance graphs are represented by strings as well, and for this the string literal from the type model is used.

Attribute containers are exported to their GXL counterparts as well. The only exception is that of

the ordof container type, for which no analogue exists. Instead, a **Seq** attribute is used instead, as it preserves the ordering and only looses the fact that the container was originally for unique values only. Attribute tuples can be exported directly to GXL, other tuples are translated into nodes with values assigned for the components of the tuple.

5.2.5 Issues

The main problem when importing GXL is that edges may be attributed as well as connect to graph elements other than nodes (although they may not be mixed). This is handled by promoting such edges to nodes with a to and from edge, which does allow attributes, as well as inheritance and abstraction. However, when exporting back again, such a promoted edge is kept as a node and thus the exported graph will show a higher complexity. This is also the case for the imported type model, which has an extra class for each such an edge. An alternative would be to ignore the attributes of such edges and treat them as simple references instead. However, this does lose some information which may be significant and so the promotion solution was used instead.

GXL also supports the concept of relations, which are the equivalent of hypergraphs. This is also not supported by the conceptual model. A solution would be to promote these kind of elements as classes as well, with incoming and outgoing relations for each endpoint of such a GXL relation. However, such elements occur not very often and as such the current importer ignores them instead.

Finally, the last major incompatibility arises from the use of subgraphs. Subgraphs add an extra structure to a GXL graph, but the conceptual model does not support this. Rather, subgraphs are treated as namespaces, and elements contained within a subgraph will be translated to identifiers contained in this namespace. An alternative could be for example to create special classes in the type model that represent such subgraphs, containing all the elements of that subgraph. However, this adds a fair amount of complexity to the resulting type model and requires the exporter to be able to distinguish such classes from regular classes, which also adds complexity to the type model. Therefore, the more simple namespace approach was taken. Note however that when exporting, namespaces are not translated back into subgraphs, but rather flattened node names. This has been done to prevent issues that occur when multiple namespaces would required multiple root graphs to be created, which is not possible.

5.3 DOT

The DOT language can represent simple attributed graphs. Unlike Ecore or GXL, it does not make a distinction between type models and instance models. This creates a large gap between the conceptual model and DOT. Furthermore, DOT is intended for graph visualization and as such many of its features are directed towards changing the visual attributes of the graph, such as coloring and layout. In fig. 5.12 an example DOT graph is shown, for which a GROOVE representation is given in fig. 5.13.

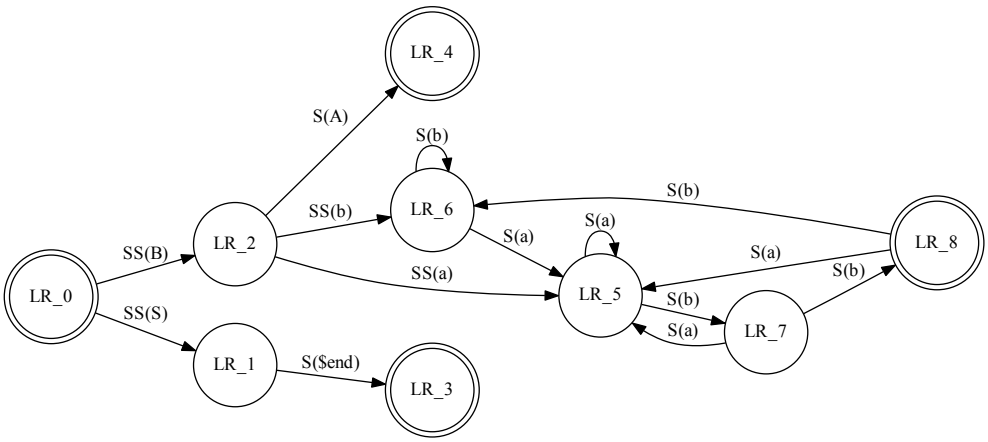
5.3.1 Importing type and instance models

As DOT does not make any distinction between instance and type graphs, importing a DOT graph as a type model does not make much sense. An approach to this could be to define a fixed set of semantics on DOT graphs and use that for defining type and instance graphs, but this would leave little room for many of the DOT graphs used.

The current approach is to import a DOT graph as an instance graph, and build a type model dynamically based on this graph. That is, the type model is generated on the fly when importing the instance model. Although the type model only reflects that graph specifically, it allows a typed graph to be

```
digraph finite_state_machine {
  rankdir=LR;
  size="8,5"
  node [shape = doublecircle]; LR_0 LR_3 LR_4 LR_8;
  node [shape = circle];
  LR_0 -> LR_2 [ label = "SS(B)" ];
  LR_0 -> LR_1 [ label = "SS(S)" ];
  LR_1 -> LR_3 [ label = "S($end)" ];
  LR_2 -> LR_6 [ label = "SS(b)" ];
  LR_2 -> LR_5 [ label = "SS(a)" ];
  LR_2 -> LR_4 [ label = "S(A)" ];
  LR_5 -> LR_7 [ label = "S(b)" ];
  LR_5 -> LR_5 [ label = "S(a)" ];
  LR_6 -> LR_6 [ label = "S(b)" ];
  LR_6 -> LR_5 [ label = "S(a)" ];
  LR_7 -> LR_8 [ label = "S(b)" ];
  LR_7 -> LR_5 [ label = "S(a)" ];
  LR_8 -> LR_6 [ label = "S(b)" ];
  LR_8 -> LR_5 [ label = "S(a)" ];
}
```

(a) Source of a DOT graph



(b) Rendering of a DOT graph

Figure 5.12: Example DOT graph, both as source and the generated image.

used in the conceptual model without having to generalize too far. Then, all nodes and edges in the DOT graph have a tailor made type in the type model, which can be kept as simple as possible. This leaves little room for other instance models given the type model, but may provide better models for analysis purposes.

When a DOT file is read, the graphs and subgraphs are mapped into namespaces, similar as is done for GXL. Thus, each (sub)graph becomes a namespace within the namespace of the parent graph, or the root namespace. Next, each node and edge is mapped to a *Class* and *Field*. The *Id* of the class is based on the namespace of the graph the node is contained within and the identifier of the node. This identifier may be either the identifier as it appears in the DOT file, or its label (which is an attribute of the node). For edges, the label is used as the name of a field for the class. If the edge has no label, it is mapped to a field named edge (optionally with an index appended if a node has multiple unlabeled outgoing edges).

Aside from the generated *Class* and *Field* in the type model, an *Object* is added to the instance model typed by the generated *Class*, together with a reference value to the object representing the target node of the edge represented by the generated *Field*.

Attributes for nodes are mapped to attributes in the *Class* of the type string, which is capable of representing all attributes as they appear as strings in the source DOT file. Attributes for graphs and edges (other than the label attributes) are ignored.

An alternative approach defines a fixed type model for DOT graphs, containing a class for nodes with fields for attributes and edges to other nodes. This is a very rough generalization, but keeps the resulting instance models simple. The type model could be extended by allowing edges to be attributed as well (which they often are in DOT graphs), but this causes each edge to have an associated $Object_{\text{Im}}$ as well, complicating the instance model. Because this generalisation adds such an amount of overhead in both the instance and type graph, this alternative has not been chosen. This means quite a bit of information is lost, but the overall structure of the (instance) graphs remains the same.

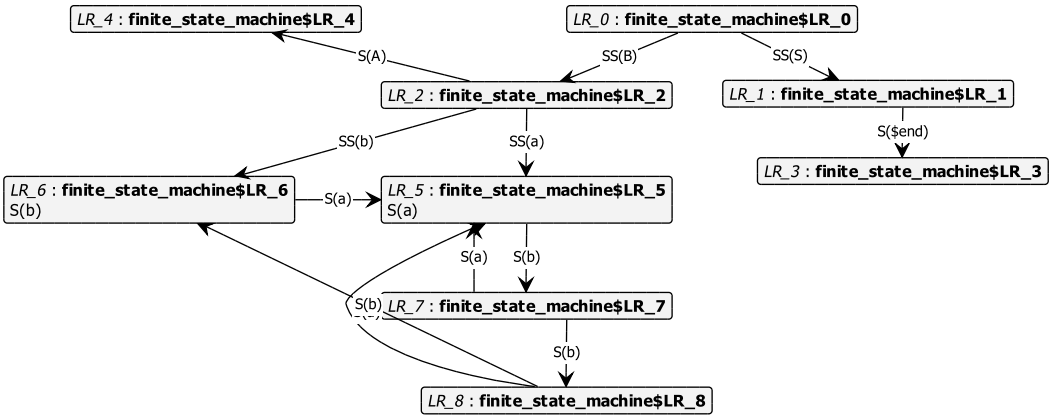
5.3.2 Exporting type models

When exporting a type model to DOT, most of the semantic information will be lost. However, the resulting graph can be made to give a graphical representation of the model similar to what GROOVE would show. Note, however, that since type models cannot be imported from DOT, there is a clear difference between importing and exporting, unlike the other languages which provide a more symmetric approach to importing and exporting.

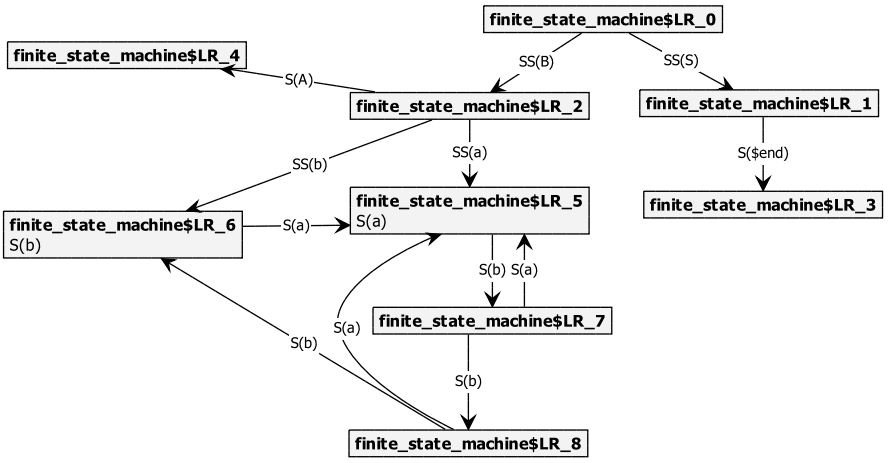
First, namespaces in the type model are exported as subgraphs in DOT. Unlike GXL, DOT is capable of using multiple root graphs, so this poses no problems. The classes in the type model are exported as nodes in the DOT graph. For each field, a label is added to the node indicating the name and type of that field. Furthermore, if there is a relation between classes in the type model, an edge is added to the DOT graph for that relation. Inheritance relations are exported as arrows which visually resemble those used in UML diagrams to indicate inheritance (which is an empty arrowhead).

Enumerations are also exported to nodes, with a text entry for each enumeration literal. Tuples are exported as nodes with markup which visually separates the various elements of the tuple. Containers are not supported, but field multiplicities are indicated by adding that multiplicity to the label (or edge) of that field. Other data types are not exported to nodes, but simply as labels with a textual representation of the type.

Of the properties, the identity, keyset, and opposite properties are not supported and will not be exported. The abstract property is represented by drawing the class node with a dashed outline. The containment property is indicated by drawing the edge of a containment relation with a diamond arrow tail. The defaultValue property is represented by adding a string representation of the value to the label of an



(a) Instance graph



(b) Type graph

Figure 5.13: Instance graph and type graph of the FSM DOT graph.

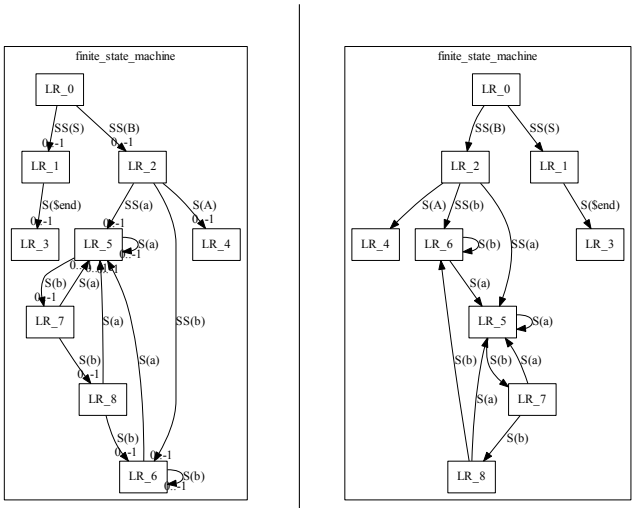


Figure 5.14: Exported type graph (left) and instance graph (right), rendered by DOT.

attribute with that default value.

The exported type models retain very little information. However, as DOT is primarily used to visually represent graphs rather than interpret them, this approach is still useful to be able to show graphs outside of GROOVE. Other applications would have no knowledge of any special structure imposed by a type model, so exporting them with some special structure only known to GROOVE would add very little benefit.

5.3.3 Exporting instance models

Instance models are exported by creating a node for each object, of which the name is either the name of the object as specified by the $\mathrm{ObjectId}_{\mathrm{Im}}$ function, or the type of the object if such a name is not defined. Like the fields of a class, the assigned field values are shown as labels in the object's node, and an edge is created for each relation between two objects.

As with the type model, namespaces in the instance model are translated to subgraphs, each bearing the name of each namespace. The same method is used as for type models.

5.3.4 Issues

As DOT is mainly intended for visualisation, there is no semantic model for the type and instance graphs. Thus, it is not possible to map the type and instance models from the conceptual model to DOT while keeping all these semantics. Thus, the choice has been made to import DOT graphs as instance graphs with an implicit type model and export them to graphs that visually represent them, rather that semantically. This means that the import and export are asymmetrical and with much loss of information.

5.4 Mapping overview

For each language and the conceptual model, a short overview is be presented which gives an indication of the various features that are supported by the importer/exporter. Since not every feature or element is supported, this overview will help indicate what information will remain and what will be lost when importing from and exporting to the various different languages.

5.4.1 Ecore

Feature	Mapping	Comment
EObject	*Object*	Objects from Ecore instance models are directly mapped to Objects in the conceptual model.
EFactory		**EFactory** is generally used only in dynamic models at runtime, which the exporter uses at runtime, but is not relevant to instance or type models directly.
EAnnotation		Annotations are not supported.
EPackage	*Id*	Packages are mapped to namespaces.
EClass	$Class_{\text{Tm}}$	The name of an **EClass** is mapped to a Class, various attributes are mapped to Field and Property.
EDataType	*DataType* and $CustomDataType_{\text{Tm}}$	Built-in data types are mapped to DataType. Other data types are mapped to CustomDataType with a string representation.
EEnum	$Enum_{\text{Tm}}$	Enumerations are mapped to Enum.
EEnumLiteral	$EnumLiteral_{\text{Tm}}$	Enumeration literals are mapped to EnumLiteral using only their string representation.
EAttribute	$Field_{\text{Tm}}$	Attributes are mapped to fields of a (container of) a data type.
EReference	$Field_{\text{Tm}}$	References are mapped to Field of a (container of) a class.
EOperation		Operations are not supported.
EParameter		Operation parameters are not supported.

5.4.2 GXL

Feature	Mapping	Comment
GraphClass	*Id*	A **GraphClass** defines a namespace.
RelationClass		Relations are not supported.
RelationEndClass		Relations are not supported.
NodeClass	$Class_{\text{Tm}}$	Nodes are mapped to Class.
EdgeClass	$Field_{\text{Tm}}$	Edges are mapped to fields. The type of the field is defined by the target of the edge.
AggregationClass	$Field_{\text{Tm}}$	Mapped like **EdgeClass**, but with an additional containment property.
CompositionClass	$Field_{\text{Tm}}$	Like **AggregationClass**.
AttributeClass	$Field_{\text{Tm}}$	Mapped to fields typed by a data type.
Locator	string	Locators are mapped to the string type.
Bool/Float/Int/String	*DataType*	Each of these data types is mapped to a matching built-in data type.
Enum	$Enum_{\text{Tm}}$	Enumerations are mapped to Enum.
EnumVal	$EnumValue_{\text{Tm}}$	Enumeration values are mapped to EnumVal.
Bag/Set/Seq/Tup	$Container_{\text{Tm}}$	Each of these composite types is mapped to a container type, or a tuple.
LocatorVal	$Constant_{\text{Tm}}$	Values in graph schemas are solely used for default values, and are mapped to Constant, ConstType, and DefaultValue.
BoolVal/FloatVal/IntVal/StringVal	$Constant_{\text{Tm}}$	Same as above.
BagVal/SetVal/SeqVal/TupVal	$Constant_{\text{Tm}}$	Same as above.

5.4.3 DOT

Feature	Mapping	Comment
Graph	*Id*	Graphs are mapped to namespaces.
Graph attributes		Graph attributes are not supported.
Node	$Class_{\text{Tm}}/Object_{\text{Im}}$	Nodes are mapped to classes in type graphs and objects in instance graphs.
Node attributes	$Field_{\text{Tm}}$	Node attributes are mapped to fields with a string type.
Edge	$Field_{\text{Tm}}$	Edges are mapped to fields typed by the class mapped to the target node. The label is used as the field name.
Edge attributes		Edge attributes are not supported (with the exception of the label, see above).
Default attributes		Default attributes (defined for nodes, edges and subgraphs) are not supported.

5.4.4 Conceptual model

A short overview of supported elements of the conceptual model by the various graph languages is given in table 5.1. A ✓means the feature is supported, otherwise it is either not supported at all or not without loss of information.

Feature	Ecore	GXL	DOT
Namespaces	✓	1	3
Classes	✓	✓	4
Enums	✓	✓	
Tuples		✓	
Data types	✓	✓	
Custom data types	✓		
Inheritance	✓	✓	
Attributes	✓	✓	5
Relations	✓	✓	6
Multiplicities	✓	✓	
Abstract	✓	✓	
Opposite	✓		
Identity	✓		
Keyset	✓		
Containment	✓	✓	
Default value	✓	✓	
Unique containers	✓	2	
Ordered containers	✓	2	
Objects	✓	✓	✓
Object identities		✓	

1. Namespaces in GXL are imported both using subgraphs and delimited names and exported just as delimited names.
2. For relations, uniqueness is always implied and non-unique relations are not supported. For attributes, ordered, non-unique containers are not supported.
3. Namespaces in DOT are represented using named subgraphs.
4. Nodes are treated as classes in DOT, with a field for each assigned attribute.
5. Attributes are imported as string types, and exported always using a string representation.
6. Edges between nodes are imported as simple, 1-valued relations.

Table 5.1: Supported features in various languages.

5.5 Conclusion

In this chapter, an overview has been given of the various mappings between the external languages and the conceptual model. These mappings provide the answer of the first part of the third research question, namely how the external languages can be mapped to and from the conceptual model. This, combined with the results in chapter 4, gives a complete answer for the third research question. Having determined how the conceptual model is defined and how it is mapped to GROOVE and the external languages, the next step is to implement this and make it usable in GROOVE itself. For this, chapter 6 will give a description of the actual implementation. Note that not every aspect of all the languages is supported. Some issues still remain, which have been covered for each language independently. Although this means that not every possible model of these languages can be imported without problems, most still can and the importer is still useful for most cases.

Chapter 6 Implementation and Validation

This chapter consists of two parts: first, a description is given of the actual implementation of the various components that are involved in the importer/exporter. Next, some experiments are performed to see how well the implementation performs and to verify its output. This chapter provides the answer to the last two research questions, namely how the importer and exporter can be implemented, and how its output can be validated.

The actual implementation is written in the Java programming language, since this is the language GROOVE is written in. This simplifies the integration of the importer/exporter. Furthermore, parsers for the external languages will also be required, and for many of these various Java libraries exist to interface with them.

6.1 Design

6.1.1 Conceptual model

The implementation of the conceptual model has been made to closely resemble the formal definition of it. This way, changes to the formal definition of the conceptual model can be integrated more easily into the implementation. For example, fig. 6.1 shows a (simplified) class diagram for the type model and its various types. As can be seen, a type model contains sets of classes, enumerations, custom data types and properties as does the type model in the formal definition. Classes, enumerations and custom data types inherit from an `Identifiable` interface, which indicates that the definition of these classes depends on an identifier, as their counterparts do in the formal definition.

An exception to this close resemblance is the use of default values. In the formal model, both the type and the instance model have provisions for the use of default values (the type model defines constants, which an instance model links to actual values). In the implementation the separation between constants and values is not required (and would actually complicate the implementation) and so the choice has been made to simply directly refer to a value for the defaultValue property.

An addition to the implementation of the conceptual model in respect to the formal model is the inclusion of names for the type and instance models. These names are mainly used to keep track of names of models imported from external languages or graphs from a GROOVE grammar, in order to be able to provide a (default) name for exported models or graphs.

6.1.2 Mapping

The mappings for the external languages and GROOVE have been set up similar to each other, using the conceptual model as a common interface. The mappings from the conceptual model to the external languages and GROOVE all use the same visitor pattern, which visits each element of the conceptual model. This allows similar patterns to be reused, and when a new element in the conceptual model is added or an existing concept is modified, the scope of the change is clearly defined in the implementation. When importing an external model to the conceptual model however, such a pattern cannot be used as each language has a different interface.

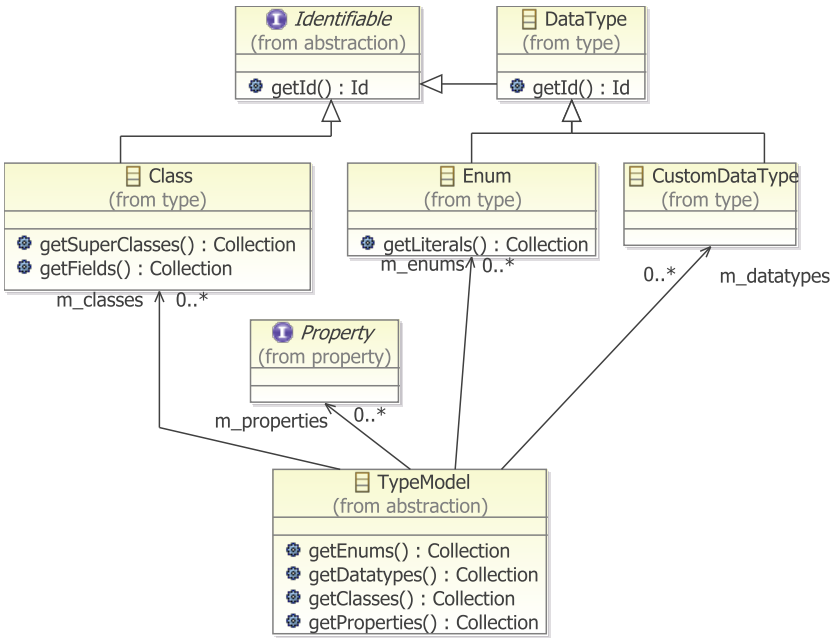


Figure 6.1: Part of the implementation of the type model.

All mappings make use of a Java `Resource` interface, which allows the generated models and graphs to be collected and saved in one go. For example, the GROOVE mapping contains a `GrooveResource` class using this `Resource` interface. This class is responsible for adding all the graphs generated by the GROOVE mapping to the current grammar. The advantage of the `Resource` interface is that for languages that can store a relation between models, the `Resource` can collect all the necessary models and create these relations. Additionally, a `Resource` allows a many-to-one or one-to-many relationship to exist between models and their files (for example, GXL can store both instance graphs and type graphs in a single file).

Ecore/GXL/DOT

The mappings from the external languages to the conceptual model all consists of components to individually translate type and instance models. For Ecore and GXL, this leads to four components (import and export for type and instance models), but for DOT only three such components exist (import and exporter for instance models, but only an exporter for type models). This is due to the fact that there is no mapping for DOT to type models.

GROOVE

The GROOVE mapping translates type models to and from type and rule graphs, and instance models from instance graphs. To this end, four components have been defined for both exporting and importing. These components handle four different types of graphs individually, and are shown in section 6.1.2.

The importer and exporter make use of different graph types in GROOVE. For a short overview of these graphs, see table 6.2. These graphs provide a stacked model of information richness. Each graph type may be build on top of another graph, deriving information from its structure. For example, the default graph may specify edges labeled as `sub:`, which an aspect graph will interpret as an inheritance edge. A type graph may then use these inheritance edges to build the actual inheritance relations between nodes.

Component	Description
Type graph	This component translates between type models and their direct type graph representations.
Meta graph	This component translates between type models and their related meta graphs when enabled via the configuration (see section 4.2.2).
Constraint graph	This component provides the mapping between the constraints for the type model and their corresponding rule graphs.
Instance graph	This component provides the mapping between instance models and their direct instance graph representations.

Table 6.1: Overview of different graph components used by thew GROOVE mapping.

Graph type	Description
Default graph	Most basic graphs, used to store graphs in the file system. Supports simple labeled nodes and edges.
Aspect graph	Based on Default graphs, features different aspects such as flags and type labels.
Type/Host (instance) graph	Based on aspect graphs, these graphs are the closest to the actual type and instance graph models used by GROOVE. Feature information such as inheritance, multiplicities and abstractness.

Table 6.2: Graph types in GROOVE.

The importer generates aspect graphs for GROOVE as its result. These aspect graphs can be seen as an intermediate form between the actual type, instance and rule graphs (and their internal models) and the basic graph structures that are used to store the graphs (which are simple nodes and edges). These aspect graphs provide elements for the different aspects such as flags and type labels (on top of nodes and edges), but do not actual model information such as subtyping and multiplicities. Of course the labels required to represent this model information can be added to the aspect graph.

The importer cannot create the type and instance graphs directly, as an aspect graph is required to build these. However, using an aspect graph has the advantage that multiple representations can be used to model the same information, which makes it easier to change this if so required.

When exporting, the converter reads the grammar and finds the graphs related to the graph selected for export. When a type graph is exported, the exporter looks for an additional meta graph, if the configuration requires so, and any related rule graphs which may provide additional model constraints. When exporting an instance graph, the exporter will look for the related type graphs, as well as the related rule graphs in addition to the selected instance graph.

The exporter uses the names of the graphs to identify their relationships and possible roles. When exporting a graph, its namespace is used to identify other related graphs. For example, if the current selected graph is named `models.university.TypeModel`, then only those graphs are considered that have a name in the namespace `models.university`. Next, graphs that have a name ending in `meta` are considered to be meta graphs. It is up to the configuration whether or not they are required. Finally, if multiple possible matches exist for the various roles, the user is asked to select those graphs that need to be exported.

Integration

The converter has been integrated into GROOVE in various places. GROOVE provides a common interface for all importers and exporters, to which the converter has been added. This interface provides the name of an external file to the import interface, upon which the import operation can be

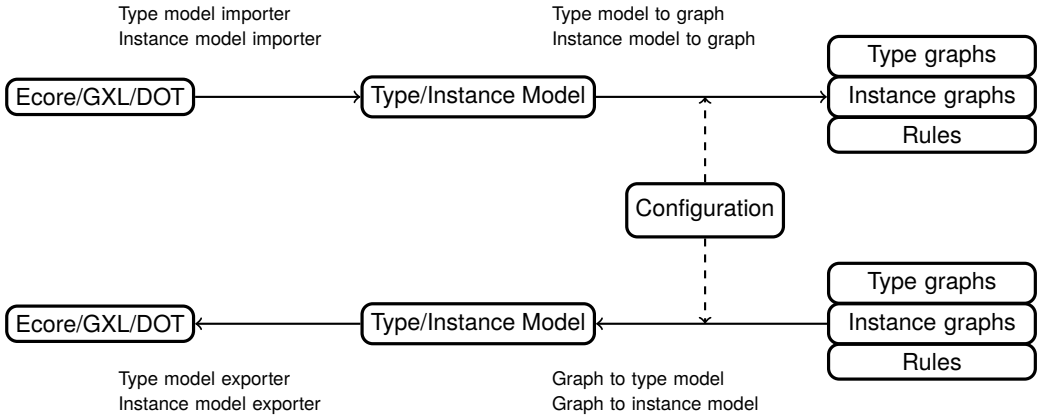


Figure 6.2: Overview of conversion process.

performed. The export interface provides the graph that is to be exported and the name of the file that it is to be exported to. This interface provides the connection between the user interface and the converter, and from here all operations start.

When exporting, the GROOVE host graphs and type graphs are used, as opposed to the aspect graphs. These graphs correspond to the actual models used in the formal description of GROOVE type and instance graphs. As such, information such as inheritance, multiplicity and abstractness is represented by these models. These are the most information rich graph structures which makes exporting easier to accomplish, since the information does not have to be derived manually, as would be the case for aspect graphs.

6.1.3 Configuration

The converter has been designed in such a way that the configuration should both be able to be read and modified by the underlying Java implementation, as well as being able to persist it in a GROOVE grammar. Furthermore, the configuration should be editable by the user, so a GUI has to be created as well. In order to be able to do all of this, the configuration has been implemented using various XML techniques, which allow themselves to be parsed and stored in multiple ways.

The configuration itself is stored in an XML document. There are many parsers for these kind of documents, including various XML libraries in Java. As such, XML parsing could be added to the converter without requiring any additional libraries. In order to make use of the XML documents for configuration, various other libraries have been used for various steps in the import/export process.

First, an XML Schema [10] has been created describing the configuration itself. An XML Schema is a document that describes a certain type of XML document, much like a type model in the conceptual model describes an instance model. The XML Schema imposes a certain structure on the allowed XML document and various constraint. This XML Schema allows other applications to determine the structure of any XML document that describes a configuration.

Based on the XML Schema, a set of Java classes has been generated using the Java Architecture for XML Binding (JAXB [22]). This allows any configuration file to be loaded into Java as a set of Java objects, giving easy access to the various parts of the configuration. These Java objects (possibly modified) can be exported back to XML as well. This way, the configuration can be loaded and stored by the importer/exporter.

To make the configuration accessible to the user, a GUI is generated from the XML Schema as well. This way, if the configuration structure is changed, a new GUI can easily be generated. In fact, the GUI

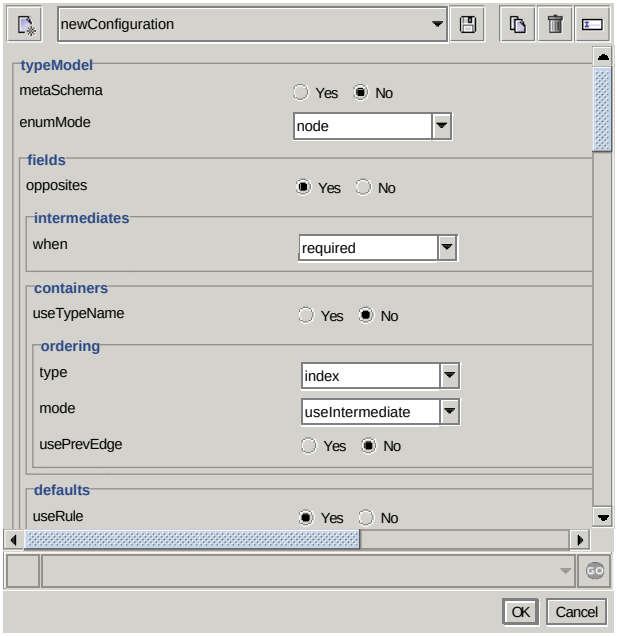


Figure 6.3: A generated configuration GUI.

can be generated at runtime, adding no extra cost to the developer. In order to do so, a new library is used which is capable of parsing an XML Schema and generating a GUI from it. Two alternatives have been identified during the development of the converter:

1. JaxFront: [34] This library can generate a basic GUI for any XML Schema, and optionally allows a special GUI description file to be loaded to change the default GUI, which allows for more advanced interfaces. The main strength of this library lies in its extensiveness. However, this does make the library quite heavy and only a fraction of the functionality is used.
2. Xample: [18] This library can generate a simple tree for the various XML elements which may be modified by the user. Unfortunately, the library is quite old and incompatible with other libraries used. Therefore, this is not a viable alternative.

An example of a generated configuration dialog is shown in fig. 6.3, which shows radio buttons for options that may be toggled on or off, and a drop down list of possible values for options that consist of alternatives.

6.1.4 GROOVY

In order to prototype the converter and allow quick access to its various parts, GROOVY [35] was used to invoke the converter. The GROOVY language is a superset of Java, which may be compiled at runtime rather than during the compilation of GROOVE itself. Thus, Java code invoking the converter could be written without having to completely recompile GROOVE. Additionally, any exceptions or errors can be caught during the script's execution, allowing for a safe continuation of the execution of GROOVY when such an event occurs in the converter.

GROOVY was added to GROOVE by means of an extra panel in the simulator with a text editor and the ability to select a GROOVY script and execute it. Figure 6.4 shows this panel being used.

One of the main advantages of including a scripting language in GROOVE is the ability to quickly pro-

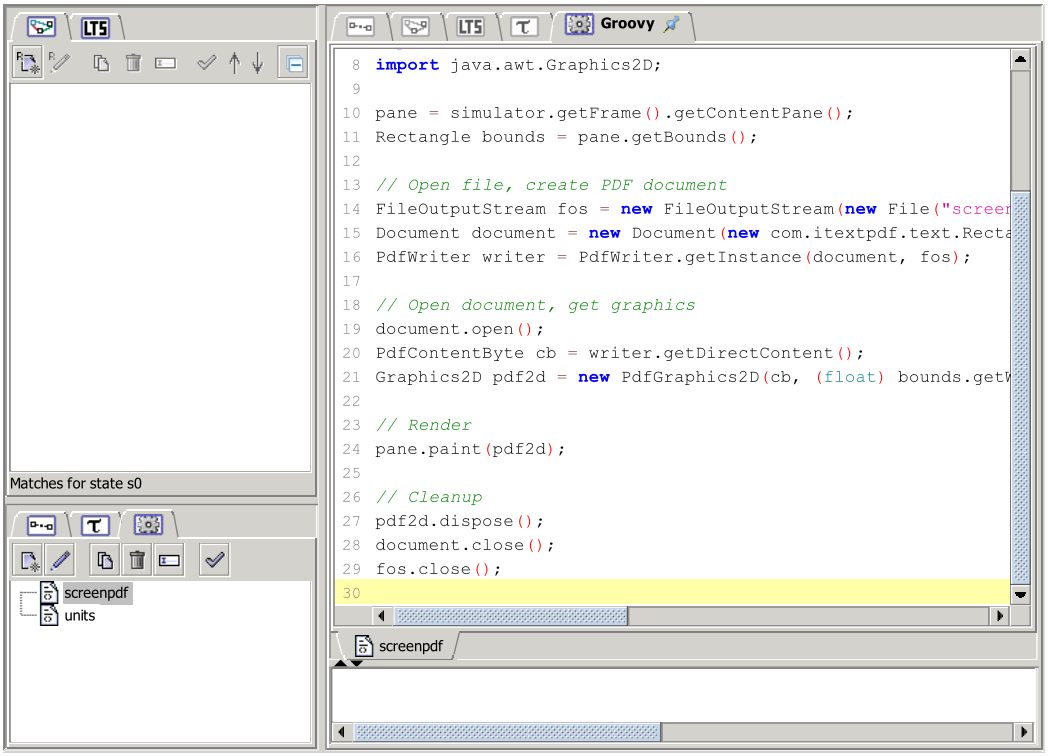


Figure 6.4: The GROOVY panel in GROOVE.

totype new features or testing current functionality. However, if GROOVY were to be more closely integrated into GROOVE, other possibilities would include the capability of executing arbitrary scripts based on events in the GROOVE simulator. For example, before a rule is applied, a callback into GROOVY could perform various checks, allowing a more fine grained control over the rule application process (and even the state space exploration strategies).

6.2 Validation

In this section, some experiments and tests are performed on the implementation, as well as a review of the implementation itself. The experiments give some insight in the usability of the implemented importer and exporter, concerning the amount of nodes it can generate and the time it takes for import and export various models of different sizes. The tests check that the various concepts from the conceptual model are supported by the various external languages. Finally, the code for the implementation is analyzed to check if the decision for use a conceptual model helped to improve the implementation.

6.2.1 Testing concept support

To verify that the import and export capabilities that were created do indeed work, some tests were performed to check that various features of the conceptual model are retained when exporting and importing. To this end, some artificial type and instance models were created, each of which contains a concept from the conceptual model that is tested. This list of concepts is identical to the lists used in section 5.4.

Since these tests only consider the importing and exporting of the conceptual model to the external languages, the configuration does not have to be considered.

- Class: This tests consists of a type model which contains a single class, and an instance model containing an object which is typed by this class.
- Enum: This test consists of a type model with a single class and a single enumeration, with the class having an attribute referencing the enumeration, and the enumeration having a single literal. An instance model is also included with a single object assigned an enumeration value.
- Data type: This test contains a single class with a single attribute, which is of a custom data type. It is combined with an instance model containing an object that is assigned a value for the custom data type.
- Inheritance: This test tests the inheritance concept, using a type model with 4 classes, which form a diamond shaped inheritance relation and define a single attribute which is inherited. An instance model is provided which contains an instance of each class, with each (inherited) attribute assigned a value.
- Attributes: This test checks the various data types. It consists of a type model with a single class having an attribute for each of the built-in data types, as well as an instance model with an object that is assigned a value for each of these attributes.
- Container types: This test contains a type model with classes defining relations using the four different container types. An instance model has objects for each class with values assigned for each relation.
- Recursive containers: This test contains a type model with a class that has a relation using recursive containers. An instance model provides objects with values assigned for these recursive containers.
- Tuples: This test contains a type model with three different kinds of tuples. These are tuples with only data types, tuples with only class types and tuples with these types mixed. The reason that these tuples are included is that in the case of GXL, a difference is made between these tuple types. Therefore, to test this all types are included. An instance model is provided which contains class instances with all values for all these tuples.
- Abstract property: This test consists of only a type model which contains a class and an abstract property for this class.
- Containment property: This tests consists of a type model with a relation between two classes and a containment property for this relation. It also contains an instance model with objects for these classes and values assigned for the relation, as the containment property changes the behavior of the Ecore import and export.
- Identity property: The test for the identity property contains a type model with a class with an attribute that has an identity property. No instance model is used.
- Keyset property: The test for the keyset property contains a type model with two classes, one of which has a relation to the other class, and the other class has an attribute. The relation and attribute are used by a keyset property. No instance model is used.
- Opposite property: This test checks if the opposite property can be exported and imported. It consists of a single type model with two classes and two opposite relations. An instance model contains multiple instances of each class with opposite relations to other instances.
- Default property: This test checks the defaultValue property. Only a type model is used, with a single class with an attribute that has a defaultValue property associated. An instance model is not used, as this property only applies during the instantiation of the instance model.

- Namespaces: This test checks the use of namespaces. A type model provides different classes each in a different namespace, with relations between them, and an instance model with an object for each of these classes (and values for the relations).

For each external language, a table is provided which gives an overview of the results of these tests. It gives the name of each test and whether it showed no information loss, some information loss or complete information loss. The information loss of a test is based on whether the concept that is being tested is retained. If the concept is successfully exported and imported, the information loss is marked as a none. If the concept remains only in certain situations, the information loss is marked partially. If the concept does not remain at all, the information loss is complete. Note that in some cases a concept may successfully be imported *or* exported, but not both. In that case the test's information loss still marked as complete.

Ecore

Concept	Information loss			Remark
	None	Some cases	All	
Class	✓			
Enum	✓			
Data type	✓			
Inheritance	✓			
Attributes	✓			
Container types	✓			
Recursive containers			✗	Classes are used to represent recursive containers.
Tuples			✗	Tuples are exported as classes.
Abstract property	✓			
Containment property	✓			
Identity property	✓			
Keyset property	✓			
Opposite property	✓			
Default property	✓			
Namespaces	✓			

GXL

Concept	Information loss			Remark
	None	Some cases	All	
Class	✓			
Enum	✓			Enums are only included if they are referenced to by an attribute.
Data type			✗	Data types are implemented as classes.
Inheritance	✓			
Attributes	✓			
Container types		✓		Only set and ord types are retained. Non-unique containers are not supported (they are exported as unique).
Recursive containers		✓		Recursive containers are supported for attributes. For relations, they are exported as classes.
Tuples		✓		Tuples with relations are implemented as classes, tuples with only attributes are exported as tuples.
Abstract property	✓			
Containment property			✗	
Identity property			✗	
Keyset property			✗	
Opposite property			✗	
Default property	✓			
Namespaces	✓			Subgraphs are imported as namespaces. The namespaces are exported as delimited names.

DOT

Concept	Information loss			Remark
	None	Some cases	All	
Class	✓			
Enum			✗	
Data type			✗	
Inheritance			✗	
Attributes		✓		Only string attributes are supported.
Container types		✓		Only set containers are supported.
Recursive containers			✗	
Tuples			✗	
Abstract property			✗	
Containment property			✗	
Identity property			✗	
Keyset property			✗	
Opposite property			✗	
Default property			✗	
Namespaces	✓			Namespaces are imported and exported as subgraphs.

6.2.2 Space performance

Since one of the desired features of the implementation was to be able to configure the generated graphs in GROOVE, the configuration allows various parts of the conceptual model to be represented in different ways, some more verbose than others. The effects of this are visible in tables 6.3 to 6.5, which show a comparison between two different configurations. The differences between these configurations are as follows:

Option	Big configuration	Small configuration
Meta graph	Yes	No
Enum	Node	Flag
Opposite edges	Yes	No
Intermediates	Always	Required
Ordering	Edge	Index
Prev edge	Yes	Node
Nullable	All	Node

For all three implemented external languages, some example models were collected and imported using the different settings. The tables refer to a number of elements for each of these models, which is defined as follows:

- For Ecore, the number of elements is the count of **EClass**, **EEnum** and **EDataType** instances. Attributes and relations are not counted.
- For GXL, the number of elements is defined as the number of **NodeClass** and **EdgeClass** (and derivatives) elements in all graphs of the imported model.
- For DOT, the number of elements is defined as the number of nodes in the DOT file.

For each model, the number of generated graph nodes and edges in GROOVE is shown for both the small and big configuration. What these tables show is that the configuration indeed has a big influence on the number of generated nodes and edges (which, depending on the model, may be up to two or three times as much for the big configuration compared to the small configuration). Another observation that can be made is that for GXL and DOT, the number of generated nodes with the small configuration is close to the number of elements in the source model. However, for Ecore this number can be much higher. One explanation for this is that in Ecore relations often allow multiple identical values, necessitating the use of intermediate nodes, whereas with GXL and DOT this is often not the case.

6.2.3 Time performance

The main purpose of these tests is to find out how well the implementation can handle very large models. The models that were generated for this test are very simple, but very large. The model size indicates the number of classes and relations that were generated. Each class was connected to three different classes by a single relation, with the model size being the total number of classes. For instance models, one instance of each class is generated, with a single value assigned for each relation of that class.

These tests were performed on an 3.00 GHz Intel Core 2 Duo CPU, with 4GB of RAM running Windows 7, using Sun Java version 1.6. The Java heap size was limited to 256MB, which is the default setting. Higher values were not used, as most users are not likely to adjust this value when running GROOVE, allowing the tests to more closely resemble what would happen on a normal installation.

Model name	#elements	#rules	Graph size big configuration		Graph size small configuration	
			Nodes	Edges	Nodes	Edges
Ecore.ecore	52	111	141	292	67	127
Security.ecore	9	32	40	85	19	34
University.ecore	7	19	32	58	15	22
type.ecore	3	2	10	11	4	3
JavaVMTypes.ecore	47	60	158	305	75	124
TCS.ecore	80	85	298	566	100	222
binding.ecore	50	42	154	238	54	90
XtextTest.ecore	21	57	130	252	51	103
Salesmodel.ecore	10	37	48	101	25	39
Domainmodel.ecore	8	12	30	51	14	19

Table 6.3: Ecore import resulting graph sizes.

Model name	#elements	#rules	Graph size big configuration		Graph size small configuration	
			Nodes	Edges	Nodes	Edges
hypergraphSchema.gxl	3	0	7	6	3	0*
attredgesSchema.gxl	3	2	13	15	5	5
undirectedSchema.gxl	3	1	10	10	4	3
attributesSchema.gxl	1	1	5	6	3	2
hypergraph2Schema.gxl	2	0	7	7	3	2
crossingSchema.gxl	6	6	21	26	7	6
simpleExampleSchema.gxl	4	4	17	22	6	8
hierarchicalGraphSchema.gxl	16	9	36	51	12	17
complexExampleSchema.gxl	7	8	24	33	9	11
orderingSchema.gxl	6	5	10	16	5	5

* In the case of the small configuration, the amount of edges is 0 since hyperedges are ignored by the mapping. For the big configuration, the edges are used for node attributes.
Examples retrieved from `http://www.gupro.de/GXL/examples/index.html`

Table 6.4: GXL import resulting graph sizes

Model name	#elements	#rules	Graph size big configuration		Graph size small configuration	
			Nodes	Edges	Nodes	Edges
NodeEdge.viz	23	11	40	22	29	11
switch.viz	25	80	144	160	64	80
inet.viz	24	51	76	104	24	52
transparency.viz	50	100	150	200	50	100
world.viz	49	49	97	98	48	49
unix.viz	41	49	90	98	41	49
root.viz	1054	4103	8177	8206	4074*	4103
datastruct.viz	13	30	56	60	26	30
process.viz	10	13	23	26	10	13
sdh.viz	23	131	206	262	75	131
crazy.viz	41	243	478	486	235	243
fsm.viz	9	14	23	28	9	14
hello.viz	2	1	3	2	2	1
fdpclust.viz	6	4	11	8	7	4
test.viz	21	20	46	42	25	21
cluster.viz	10	15	27	30	12	15
profile.viz	61	145	266	290	121	145
softmain.viz	213	673	1290	1346	617	673

* In this specific model, each node has four attributes, which explains the fourfold increase in size for the small configuration (eightfold for the big configuration due to the intermediate nodes for attributes).

Examples retrieved from `http://www.graphviz.org/Gallery.php`

Table 6.5: DOT import resulting graph sizes

Each test was performed by exporting and importing the same models 5 times, taking the mean average of the resulting times. The times are divided into the following columns:

1. Export TM: Performing the mapping from a type model in the conceptual model to the target language.
2. Export IM: Performing the mapping from an instance model in the conceptual model to the target language.
3. Save: Storing the actual generated elements in the target language. This all occurs within the external libraries used, or within GROOVE.
4. Load: Loading the generated models back in. The models are not transformed yet, but either loaded from disk or from the GROOVE grammar.
5. Import TM: Performing the mapping from a loaded model to a type model in the conceptual model.
6. Import IM: Performing the mapping from a loaded model to an instance model in the conceptual model.

The model size column indicates the size of both the type model and instance model generated (as the number of classes and objects respectively). Though it is not rare for instance models to have such a large number of objects, the type model is completely artificial, as in most cases the number of elements remains in the hundreds. Therefore, it is not expected that such large models are used during normal use of the importer/exporter.

GROOVE

The test results for the import and export of the type and instance models to and from GROOVE graphs are shown in table 6.6. When considering the translation of the conceptual models from and to GROOVE graphs, it is clear that the most time is spent saving the generated graphs into the grammar. The time it takes to process all these graphs can range from a few seconds to (in extreme cases) minutes. Trying to insert even larger graphs will cause the application to run out of memory. Because of this, no data is available for the translation of GROOVE graphs back to the conceptual model for models with a size larger than 1600. However, at this point the time required to translate these graph has increased so much that the exporter has become unusable from an UI perspective.

Profiling shows that much time is needed to parse the generated type graphs, with the instance graph being parsed much faster. Considering the generated type models are much larger than what is used in practice, the actual import and export times may considerably less than what these tests show (the tests can be considered a worst case).

Ecore

Table 6.7 shows the tests results for the Ecore importer and exporter. The Ecore translation performs fairly well. The timings of the mapping operations themselves stay well within a second and appear to increase linearly with the model size. However, the saving and loading of the Ecore models to and from disk takes much longer. Past 3200 elements, the save times exceed 2000ms, indicating there is a noticeable delay. However, profiling shows most of the time required to save the models is spend saving the type model. The instance model takes significantly less time to load.

A special case is the model with 25600 elements, which shows an almost four-fold increase in save and load time. This is most likely because the amount of available memory becomes very little, causing the Java VM to spend much time garbage collecting.

Model size	Export TM	Export IM	Save	Load	Import TM	Import IM
50	0.42	0.36	1547.04	8.46	0.48	1.59
100	0.81	0.56	1976.91	20.08	0.78	2.70
200	1.62	1.10	3197.63	49.18	1.20	4.18
400	3.03	2.24	5355.28	134.97	2.44	8.43
800	5.96	4.56	10 261.25	515.73	4.97	17.44
1600	12.63	9.61	23 092.81	1889.05	10.83	37.08
3200	113.93	111.63	-	-	-	-
6400	349.92	219.57	-	-	-	-
12800	719.79	2273.45	-	-	-	-
25600	2153.37	11 661.61	-	-	-	-

Table 6.6: Time required (in ms) to parse and generate models for GROOVE graphs.

To give an indication of how much influence the size of the type model has on performance, another test was performed with the Ecore exporter and importing using a smaller type model. The results of this test are presented in table 6.8. The type model had a fixed size of 200 classes, whereas the instance model had a variable number of instances as described by the model size column. As the results show, the performance improves dramatically, with the save operations now taking less than half a second (as opposed to half a minute or more).

GXL

For GXL, the tests results are shown in table 6.9. From the three external languages, GXL performs the worst. The main bottleneck lies with the actual serialisation of the GXL documents. Although generating the GXL models takes little time, the actual XML serialization takes a very long time, more than a minute for a model with 6400 classes. At this point memory will run out as well, causing further tests to fail. A solution could be to use a different serialization library, or perhaps a more extensive library such as Ecore uses, which may use more semantic information to improve performance. Loading GXL documents however goes much faster, but since this depends on saving documents (otherwise they do not exist), no results are available for model with a size larger than 6400.

An interesting observation is that for both GXL and GROOVE the same library is used to save GXL data, as GROOVE uses a subset of GXL to store its graphs. Improving the GXL serialisation should help both GROOVE and GXL in terms of performance.

DOT

Table 6.10 gives the tests results for the DOT importer and exporter. Considering performance, the DOT language performs better than Ecore and GXL. The DOT exporter is fairly fast, which is not too surprising as it is very simple and looses much of the information from the conceptual model. The import shows comparable timings.

6.2.4 Code analysis

In order to give some indication of how the conceptual model has affected the implementation, a few metrics were collected on the code of the implementation. One of the goals of the conceptual model is the reduction of effort required to add new languages to import and export. This effort can be expressed in multiple ways, including size of the code base and the complexity of the code. For this, the following metrics were collected:

Model size	Export TM	Export IM	Save	Load	Import TM	Import IM
50	0.28	0.26	8.59	3.35	0.44	0.31
100	0.58	0.51	15.48	6.74	0.87	0.63
200	1.01	1.03	38.12	13.11	1.73	1.28
400	2.02	2.00	101.97	32.18	3.52	6.12
800	4.09	4.02	380.15	94.63	6.89	6.41
1600	8.02	8.24	891.03	291.43	21.38	11.34
3200	15.96	18.05	2611.68	1033.83	42.69	24.23
6400	36.16	42.38	9768.01	3835.37	58.70	73.14
12800	70.40	138.19	35 011.38	14 608.08	154.03	140.95
25600	147.53	307.65	136 967.33	63 244.26	274.77	359.52

Table 6.7: Time required (in ms) to parse and generate models for Ecore models.

Model size	Export TM	Export IM	Save	Load	Import TM	Import IM
50	1.05	2.56	13.94	23.79	1.81	0.85
100	1.15	0.88	7.30	14.91	1.60	0.76
200	1.03	1.19	7.72	15.53	1.53	1.17
400	1.03	1.27	7.86	16.31	1.52	1.60
800	1.01	1.91	9.29	16.53	1.52	3.03
1600	1.02	3.19	12.42	19.25	1.52	5.97
3200	1.01	5.77	20.77	23.04	1.53	16.56
6400	1.02	10.92	45.34	38.01	1.66	24.79
12800	1.04	21.64	127.85	57.35	1.82	78.02
25600	1.02	46.16	433.66	91.22	1.80	149.11

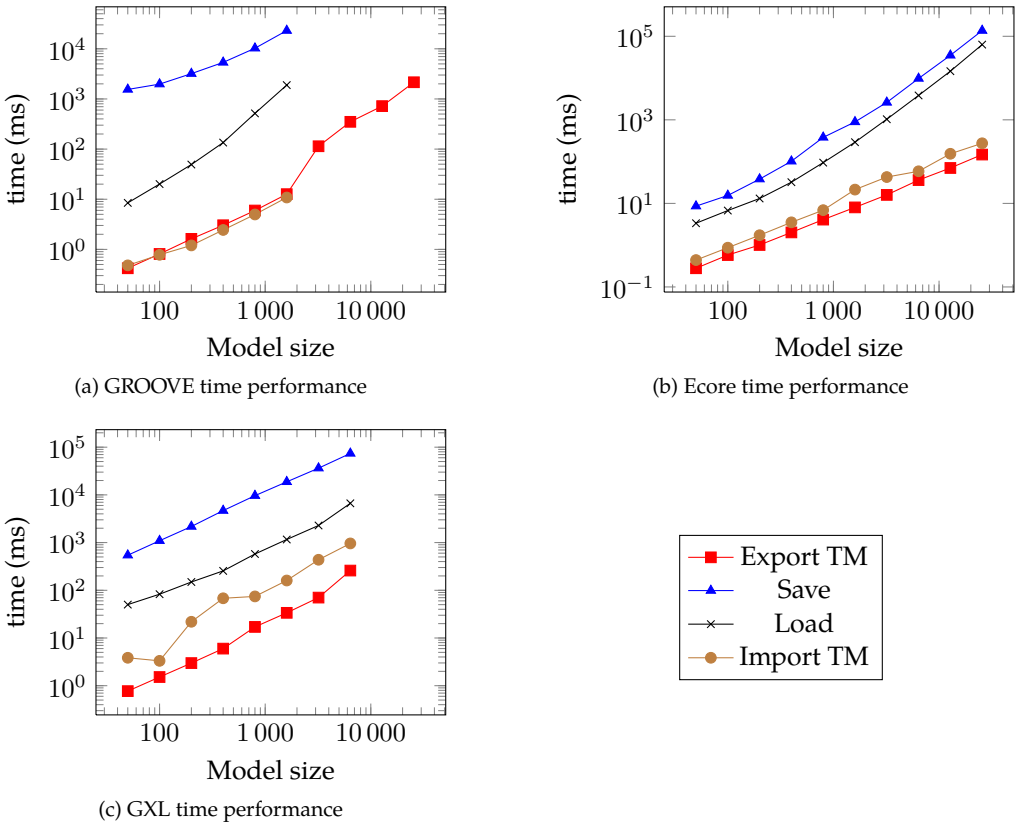
Table 6.8: Time required (in ms) to parse and generate models for Ecore models, using a small type model.

Model size	Export TM	Export IM	Save	Load	Import TM	Import IM
50	0.77	0.13	543.69	50.13	3.85	0.54
100	1.52	0.25	1093.52	82.82	3.32	0.92
200	2.98	0.49	2171.88	149.22	21.88	2.02
400	5.99	1.03	4683.00	255.10	68.02	4.05
800	17.07	2.10	9627.21	576.25	74.41	7.95
1600	33.69	4.90	18 772.88	1161.09	160.33	95.90
3200	70.30	10.40	36 303.81	2282.32	436.47	363.73
6400	259.13	22.99	73 957.47	6656.07	960.06	3319.48
12800	-	-	-	-	-	-
25600	-	-	-	-	-	-

Table 6.9: Time required (in ms) to parse and generate models for GXL graphs.

Model size	Export TM	Export IM	Save	Load	Import IM
50	0.17	0.25	2.83	2.50	0.20
100	0.32	0.53	5.73	4.79	0.39
200	0.61	0.98	14.32	7.82	0.76
400	1.21	1.95	34.50	15.14	1.51
800	2.36	5.17	70.84	26.11	3.33
1600	6.15	9.71	145.50	73.24	7.43
3200	9.31	38.62	290.34	145.85	27.95
6400	19.34	76.10	573.63	266.01	53.15
12800	45.29	163.45	1153.93	573.36	71.70
25600	107.71	337.53	2305.49	1692.64	923.71

Table 6.10: Time required (in ms) to parse and generate models for DOT (instance) graphs.



(a) GROOVE time performance

(b) Ecore time performance

(c) GXL time performance

Figure 6.5: Time performance of GROOVE, Ecore and GXL mappings.

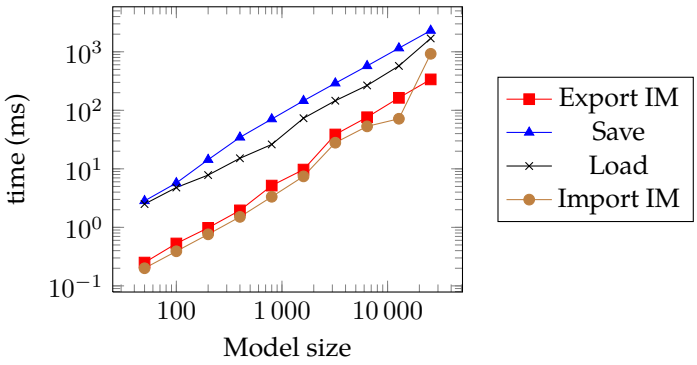


Figure 6.6: Time performance of the DOT mapping.

1. Method lines of code (MLOC) : This is the number of lines of code contained within methods. Since most of the functional code in an implementation resides in methods (as opposed to field declarations, comments and Java import statements), this metric gives a good indication of the amount of code written that is actually used.
2. Total lines of code (TLOC) : This is the total number of lines of code in the implementation, including all lines of code not included by the MLOC metric.
3. Mean McCabe cyclomatic complexity [27] : This metric measures the number of linearly independent paths through the source code. It gives an indication of the complexity of the code, with higher numbers indicating more complex code.

The results of these metrics can be found in table 6.11. The first two columns show the MLOC metric both in absolute lines of code and in percentage. The second column shows the TLOC metric, and the last column the means McCabe cyclomatic complexity. The mean cyclomatic complexity is calculated by taking the cyclomatic complexity of each method in the unit referred to be the table, and taking the mean average of all these values.

The table has been divided into sections. Each section may be divided up into subparts which shows the metrics for those parts. Note that these parts do not have to cover the entire section, sometimes various implementation details such as utility functions are not included by these parts, but are taken into account for the entire section. The 6 sections in the table are defined as follows:

1. The first section shows the metrics for the implementation of the conceptual model. It is further divided up into the implementation of the type model, instance model and the properties used by the type model.
2. The second section shows the metrics for the Ecore importer and exporter. It is further divided into im- and exporters for the type and instance models. These correspond to section 5.1.
3. The third section gives information about the GXL implementation of the im- and exporter just as for the second section, corresponding to section 5.2.
4. The fourth section gives the metrics for the DOT im- and exporter, corresponding to section 5.3. As DOT has no explicit importer for type models, it has no entry for a type model importer.
5. The fifth section shows the metrics for the GROOVE mapping. It consists of the importers and exporters for the type and instance models, but it also shows the importer and exporter for the meta graphs and the constraint graphs (rules).
6. The final section shows the metrics for the implementation of the configuration, which is used by the GROOVE graph importer and exporter. This includes the implementation of the configuration editor and the management of configuration files.

Part	MLOC	MLOC %	TLOC	Mean McCabe
Conceptual model	275	4.43	538	1.83
Type	252	4.06	563	1.13
Instance	130	2.09	321	1.43
Prop	68	1.09	229	0.22
Ecore	773	12.44	1176	3.69
import type	206	3.32	281	4.41
import instance	136	2.19	191	1.89
export type	264	4.25	382	2.73
export instance	127	2.04	199	2.10
GXL	1598	25.72	2197	4.24
import type	464	7.47	584	7.58
import instance	163	2.62	216	4.86
export type	388	6.24	525	3.34
export instance	135	2.17	213	2.93
DOT	507	8.16	769	3.42
import inst	106	1.71	153	4.67
export type	136	2.19	194	3.80
export inst	151	2.43	225	2.87
GROOVE	2519	40.54	3445	4.34
import type	258	4.15	352	4.15
import meta	186	2.99	250	4.23
import constraint	490	7.89	628	3.70
import instance	233	3.75	325	3.83
export type	348	5.60	427	10.92
export meta	86	1.38	125	6.17
export constraint	490	7.89	628	3.70
export instance	233	3.75	325	3.83
Configuration	541	8.71	791	3.42
Total	6213		8916	

Table 6.11: Collection of various code metrics of the implementation of the importer and exporter.

Looking at the MLOC metric, it is clear that the most code resides in the GROOVE graph importer and exporter. This is to be expected, as this part is what is configurable, which increases the amount of code required to handle all possible cases. Furthermore, the GROOVE importer and exporter do not map a type or instance model to a single graph, but multiple graphs of different types. This increases the amount of code required as well. The conceptual model itself has a relatively small amount of code (approximately 4%), which gives an indication that its introduction did not require much effort implementation wise.

The various importers and exporters for the external languages are all considerably smaller than the GROOVE importer and exporter, although the GXL mapping, most notably the type model importer, is still fairly large. This is paired with a relatively high cyclomatic complexity of the GXL type importer. This can be attributed to the fact that the GXL library used to import GXL graphs does not have any provisions for handling the GXL meta schema. For example, inheritance has to be resolved by the importer itself while, for example, the Ecore library does this transparently. However, these high values only apply to the type model importer for GXL, the other importers and exporters are not as complex.

Looking at the overall picture, it is clear that, implementation wise, the amount of effort required to implement a mapping expressed in size and complexity of the code shows some significant differences. The DOT mapping is the smallest of all, which can be explained by the fact that the DOT language itself does not support many of the concepts from the conceptual model and is in itself fairly simple. DOT is followed by Ecore, which matches the closest to the conceptual model and has a fairly extensive library that handles much of the work necessary to interpret its models. Next comes GXL, which, as mentioned before, is fairly large. However, this can be attributed to the fact that it supports a fair amount of concepts but its library is not as extensive as that of Ecore. Finally, the GROOVE graph mapping is significantly larger in terms of size (40% compared to 25% for GXL, which comes second) and has a relative high cyclomatic complexity.

It appears that the size of the mappings for external languages fluctuates with the complexity of the language combined with the extensiveness of the library used for that language, but still remains under the size of the mapping between the GROOVE graphs and the conceptual model. This fact indicates that most of the work in adding a mapping for an external language resides in parsing the information in that language, rather than the translation to the conceptual model. This gives a careful hint that the conceptual model has indeed helped reduce the effort to add other external languages to the importer and exporter.

6.3 Conclusions

This chapter has shown how the implementation was made, and has shown some insights in its performance. The conceptual model has been used as an intermediate layer between the external languages and GROOVE, with the configuration affecting the translation between the conceptual model and GROOVE. Thus, the first part of this chapter answers the fourth research question. The test results from section 6.2.1 are in line with the expected results from section 5.4. The timings from section 6.2.3 indicate that the implementation is efficient enough to be considered usable. However, the actual saving and rendering of the graphs after importing takes a considerable amount of time and memory, which prevents large models from being imported into GROOVE completely. These tests answer the fifth research question, validating and benchmarking the implementation. Finally, an analysis of the implementation indicates that the use of a conceptual model has helped reduce the effort required to make the implementation, though no definitive conclusion can be made about this.

Chapter 7 Conclusion

In this chapter an overview is given of the findings of this thesis. First, a summary is presented of this work. Then, the work is evaluated and the research questions are answered. This evaluation is followed by a number of possible improvements to the current results.

7.1 Summary

The main goal of this thesis was to improve the support for importing and exporting external graph languages to and from GROOVE. This improvement consisted of increasing the number of languages supported by the GROOVE importer and exporter, as well as adding the ability to configure the import and export process. This goal has been achieved by creating a new importer and exporter using an intermediate conceptual model, which allows various concepts from external languages to be abstracted away, as well as provide a single mapping between itself and GROOVE that can be configured.

The conceptual model has been created based on a number of concepts that may be found in a selection of external languages. These concepts have been formalized in a type model and an instance model. This formalization gives a clear definition of the concepts that are supported. The distinction between the type and instance levels allows for a closer mapping to GROOVE and various modeling languages, including Ecore and GXL. There is a clear separation between these two levels, and care has been taken that the type model does not refer to the definitions of the instance model, allowing a type model to be defined without any instance model, but an instance model is always typed by a single type model.

Following the definition of the conceptual model, mappings were defined between the conceptual model and three external languages: Ecore, GXL, and DOT. These mappings are defined based on the various concepts or elements that can be found in these languages. The mappings are not complete; not every concept of each language is covered by the mapping to the conceptual model, and not every concept from the conceptual model is covered by the mapping to any of the external languages. In some cases this has been mitigated to some degree by using different concepts to represent these missing concepts. However, this is not always possible or desirable. For the remaining gaps, information is simply lost when translating between the conceptual model and the external language. However, care has been taken to try and include the most commonly used concepts in the mappings, allowing most models to be imported and exported without loss of information.

GXL and Ecore can describe type models and instance models. In contrast, DOT is not a modeling language and therefore has no notion of these concepts. The inclusion of DOT has been used as an exercise to see how well a non-modeling language could be added to the conceptual model. The resulting mapping shows a big difference with those of Ecore and GXL. When importing DOT, only instance models are created based on the DOT input, with a corresponding type model being derived for this specific instance model. When exporting an instance or type model, no actual semantic information is exported, but rather a DOT representation that graphically mimics these models. This approach has been chosen as DOT is intended for graph visualization, rather than modeling.

A single mapping has been defined between GROOVE and the conceptual model, which is configurable. This is the most complex mapping, as its translation involves a bigger difference in expressiveness and is configurable. A formal definition of GROOVE type and instance graphs has been given, which allows the mapping between GROOVE and the conceptual model to be described at a formal level. However, no formal definition has been given for rule graphs, which have been used to

implement various constraints. Rule graphs have been described by their semantics, discussing the subset of features used by the constraint graphs.

Type models in the conceptual model are mapped to type graphs in GROOVE, with additional rule graph for various constraints in the type model that cannot be expressed by a type graph alone. Instance models are translated to instance graphs with no additional graphs needed. The mapping allows an optional meta graph to be generated, which may simplify the type graph but introduces an extra graph. Since the mapping is configurable, different representations are possible for the various concepts in the conceptual model. To be able to keep track of the configuration that applies to a specific graph, multiple configurations can be stored into and loaded from a single grammar.

The conceptual model and the mappings have been implemented in Java, the language that was also used to implement GROOVE. The implementation of the conceptual model follows the formal definition as close as possible, in order to make it easier to maintain it. One of the design goals was to make the conceptual model extensible. Following the formal definition more closely allows the implementation to follow changes in the formal definition more easily.

The configuration has been designed with the same principle in mind, although rather than a formal specification, XML Schema was used. XML Schema allows a GUI for the configuration editor to be automatically generated, as well as automatically generate an interface for Java to read and write the XML files created by the configuration editor.

Some tests have been performed to validate the resulting implementation as well. Three aspects of the implementation have been validated: the generated model files, the performance of the importer/exporter and the written code. For the generated output, test were performed to check that the support for various features matched the predictions of the mappings, by means of checking whether or not information was lost for various concepts. The results correspond to these predictions, indicating the implementation for the mapping follows the specification. Furthermore, the effect of the configuration on the size of the generated graphs has been tested, by comparing the size of the generated graph for the same model with different configurations. This effect is predictable, with the configuration capable of creating a significant difference is graph sizes. The difference between model sizes and graph sizes varies depending on the structure of the model, with the graph generally being around a factor two larger than the original model with the configuration set for smaller graphs.

The performance tests show that there is a linear relation between the size of the model and the time it takes to translate the model. The mapping process itself performs well, however the entire process shows a significant drop in performance for larger (type) models. This can mainly be attributed to the retrieval and storage of the generated models and graphs. However, experiments indicate that this is most likely the results of using very large type models, which in practice do not occur often. Smaller type models with (still) large instance models perform better, at least in the case of Ecore.

Analyzing the written code gives an indication that the use of the conceptual model has allowed some of the code to be reused for the various mappings, and move some of the complexity of mapping to external languages to the mapping between GROOVE and the conceptual model.

7.2 Evaluation

In the introduction, several questions have been asked that guided this research. Based on the results of the previous chapters, the answers to these questions can now be given:

"What are suitable languages for importing and exporting into and from GROOVE?"
This question has been answered by evaluating a set of (graphical) languages against a set of criteria, such as concreteness and uniqueness. Based on these criteria, a subset of these languages was selected for inclusion in the im- and exporter (namely Ecore, GXL and DOT).

"How can different languages be covered while reusing as much code as possible?"
The key to this answer lies with the conceptual model. By using the conceptual model as an intermediate between the various external languages and GROOVE, an interface can be created that each external language may use when mapping to the conceptual model. Additionally, the larger gap in expressiveness lies with the mapping between the conceptual model and GROOVE, ensuring the more complex translation operations only need to be performed once when mapping between GROOVE and the conceptual model. Finally, using the conceptual model, the configuration only has to be applied to this mapping between GROOVE and the conceptual model, preventing the other mappings from requiring extra code for the configuration.

"How to map the languages from and to the conceptual model, and GROOVE from and to the conceptual model?"
These mappings have been defined in chapters 4 and 5. In essence, each feature from the external languages is considered independently and is either included in the mapping or ignored. This inclusion is based on how close the feature matches to a concept (or concepts) in the conceptual model, the influence of the feature on the various models, and how often it is used in the external language. If a feature is not included in the mapping, then a model making use of this feature will lose information either when importing or exporting.

"How can the import and export facilities be implemented?"
The implementation has been created using the Java language, and can roughly be defined by the implementation of the conceptual model, the various mapping of the external languages to the conceptual model, the mapping between the conceptual model and GROOVE, and finally the implementation of the configuration. The implementation of the conceptual model has been kept as close to the formal definition as possible, which allows a change in the formal definition to be easier reflected in the implementation, for example, when the conceptual model gets extended. The configuration makes use of an XML Schema, which is used to generate the configuration editor GUI, as well as the binding between Java and the XML documents that are the actual configurations. This binding is then used by the mapping between GROOVE and the conceptual model to read the configuration settings and adjust the mapping accordingly.

"How can the results be validated?"
The results are validated by the output, expressed as information loss, model and graph sizes, performance expressed in time to translate models, and finally an analysis of the written code. The validation of the output gives some indication of the supported language features of the mappings, as well as the size of the generated graphs, which influences the readability and maintainability of these graphs. The performance is evaluated by means of the time taken to translate generated models to and from the external languages and GROOVE. This time is measured both for the time spent performing the actual mapping, and the time taken to generate the models or graphs by either external libraries or GROOVE itself. Finally, the code base is analysed by means of the number of lines of code written, and the complexity of the code as expressed by the McCabe cyclomatic complexity.

The main question asked at the beginning of this thesis was: *"How can the current import and export capabilities in GROOVE be improved?"*. Using the results of the preceding chapters, this question can now be answered. The improvement that is the subject of this question has been defined as the inclusion of more languages to the importer and exporter, as well as the possibility to configure the import and export processes. This improvement has been realized by the inclusion of two new languages, namely GXL and DOT, with the use of a conceptual model. This same conceptual model allows a single mapping to be defined between itself and GROOVE, which has been extended with a configuration that affects all the import and export operations between GROOVE and the external languages.

7.3 Future work

Although the implemented importer and exporter allow GROOVE to use Ecore, GXL and DOT models and graphs, further enhancements and changes can still be made, both in the implementation and the formal definitions. Although the current exporter and importer are able to handle most models, improvements can give more accurate results, or improve performance.

7.3.1 Command line

Although the main use case for the current importer and exporter assume the GROOVE GUI is used, it would be beneficial to add a command line interface as well, such as provided by the previous Ecore importer and exporter. One of the current issues with the graphical interface is that it takes a considerable amount of time to load a visual representation of graphs which have a large size. Using the system from the command line allows the visualization to be bypassed, giving the opportunity to work with more and larger graphs. Additionally, multiple models can be converted in a batch process, allowing multiple models to be translated without manual intervention.

An additional advantage of such a command line interface would be the ability to directly translate external models from one language to another using the conceptual model as an intermediate, without having to go through a GROOVE graph representation. This allows for a faster translation in situations where a model simply has to be represented in a different language, without requiring modification.

7.3.2 Extend CM

One of the problems of the current conceptual model is that it does not capture all the concepts from each language. Future work may add additional concepts to the conceptual model, or improve the current mappings. This allows these currently unsupported concepts to be integrated as well, preventing loss of information upon translation. For example, the current conceptual model is not capable of handling hypergraphs (relations in GXL), or attributed references and attributes. Adding these concepts would improve the support for the GXL and DOT languages.

7.3.3 Alternative CM

A more structural problem of the conceptual model lies with the fact that it is entirely geared towards models based on a type and instance system. Although this fits well with modeling languages such as GXL and Ecore, for other graph languages such as DOT this model is less suited. It would be worthwhile to see how well a different conceptual model would work geared more towards (attributed) graphs instead, resembling for example the GXL graph model.

7.3.4 Library performance

Experiments show that some of the external libraries used to store and load external models perform badly with large (type) models. This may be mitigated to some degree by using different libraries. For example, for storing and loading GXL, a simple XML layer is used without much knowledge of GXL itself, but which is very generic. Using a more dedicated library may allow knowledge of GXL to be used which can speed up the storing and loading of graphs.

Bibliography

[1] C. Amelunxen, A. Knigs, T. Rtschke, and A. Schrr. Moflon: A standard-compliant metamodeling framework with graph transformations. In *Model driver architecture - Foundations and applications: Second European conference, volume 4066 of lecture notes in computer science (LNCS)*, pages 361–375. Springer-Verlag, 2006.

[2] T. Arendt, E. Biermann, S. Jurack, C. Krause, and G. Taentzer. Henshin: advanced concepts and tools for in-place emf model transformations. In *Proceedings of the 13th international conference on Model driven engineering languages and systems: Part I*, MODELS'10, pages 121–135, Berlin, Heidelberg, 2010. Springer-Verlag.

[3] ATLAS group et al. Km3. `http://www.eclipse.org/gmt/am3/km3/doc/KernelMetaMetaModel%5Bv00.06%5D.pdf`, accessed Monday 12th November, 2012.

[4] AT&T Bell Laboratories. DOT. `http://www.graphviz.org/content/dot-language`, accessed Monday 12th November, 2012.

[5] Borland. Borland together. `http://www.borland.com/us/products/Together/`, accessed Monday 12th November, 2012.

[6] A. Boronat, J. Cars, and I. Ramos. Algebraic specification of a model transformation engine. In L. Baresi and R. Heckel, editors, *Fundamental Approaches to Software Engineering*, volume 3922 of *Lecture Notes in Computer Science*, pages 262–277. Springer Berlin Heidelberg, 2006.

[7] A. Boronat and J. Meseguer. An algebraic semantics for mof. *Formal Aspects of Computing*, 22:269–296, 2010.

[8] U. Brandes, M. Eiglsperger, I. Herman, M. Himsolt, and M. Scott Marshall. GraphML progress report (structural layer proposal), 2002.

[9] U. Brandes, J. Lerner, and C. Pich. GXL to GraphML and vice versa with XSLT. *Electronic Notes in Theoretical Computer Science*, 127(1):113 – 125, 2005. Proceedings of the International Workshop on Graph-Based Tools (GraBaTs 2004).

[10] C. E. Campbell, A. Eisenberg, and J. Melton. Xml schema. *SIGMOD Rec.*, 32(2):96–101, June 2003.

[11] D. T. Chang and E. Kendall. Metamodels for rdf schema and owl. In *Proceedings of the First International Workshop on the Model-Driven Semantic Web (MDSW 2004*, 2004.

[12] M. Clavel, F. Durn, S. Eker, P. Lincoln, N. Mart-Oliet, J. Meseguer, and J. F. Quesada. Maude: Specification and programming in rewriting logic, 2001.

[13] Cytoscape. Cytoscape. `http://www.cytoscape.org/`, accessed Monday 12th November, 2012.

[14] K. Czarnecki and S. Helsen. Feature-based survey of model transformation approaches. *IBM Systems Journal*, 45(3):621 –645, 2006.

[15] J. de Lara and H. L. Vangheluwe. Using AToM^3 as a Meta-CASE environment.

[16] Eclipse community. Ecore. `http://www.eclipse.org/modeling/emf/`, accessed Monday 12th November, 2012.

[17] J. Ellson, E. R. Gansner, E. Koutsofios, S. C. North, and G. Woodhull. Graphviz and dynagraph - static and dynamic graph drawing tools. In *Graph drawing software*, pages 127–148. Springer-Verlag, 2003.

[18] F. Golubov. Xample. http://www.felixgolubov.com/XMLEditor/, accessed Monday 12th November, 2012.

[19] GROOVE. About GROOVE. http://groove.cs.utwente.nl/about/, accessed Monday 12th November, 2012.

[20] GROOVE. Download GROOVE. http://groove.cs.utwente.nl/downloads/groove/, accessed Monday 12th November, 2012.

[21] M. Himsolt. GML. http://www.fim.uni-passau.de/fileadmin/files/lehrstuhl/brandenburg/projekte/gml/gml-technical-report.pdf, accessed Monday 12th November, 2012.

[22] JAXB. http://jaxb.java.net/, accessed Monday 12th November, 2012.

[23] F. Jouault and I. Kurtev. Transforming models with ATL. In *Satellite Events at the MoDELS 2005 Conference*, volume 3844 of *Lecture Notes in Computer Science*, pages 128–138, Berlin, 2006. Springer-Verlag.

[24] Kermeta. Kermeta architecture. http://www.kermeta.org/docs/fr.irisa.triskell.kermeta.documentation/build/html.chunked/KerMeta-Manual/ch03.html#section_architecture.link, accessed Monday 12th November, 2012.

[25] I. Kurtev. State of the art of qvt: A model transformation language standard. In Andy Schrr, Manfred Nagl, and Albert Zndorf, editors, *Applications of Graph Transformations with Industrial Relevance*, volume 5088 of *Lecture Notes in Computer Science*, pages 377–393. Springer Berlin Heidelberg, 2008.

[26] M. Lawley and J. Steel. Practical declarative model transformation with tefkat. In *Proceedings of the 2005 international conference on Satellite Events at the MoDELS*, MoDELS'05, pages 139–150, Berlin, Heidelberg, 2006. Springer-Verlag.

[27] T.J. McCabe. A complexity measure. *Software Engineering, IEEE Transactions on*, SE-2(4):308 – 320, dec. 1976.

[28] Microsoft. Microsoft visio. http://office.microsoft.com/en-us/visio/, accessed Monday 12th November, 2012.

[29] P.A. Muller, F. Fleurey, and J.M. Jézéquel. Weaving executability into object-oriented meta-languages. In *International Conference on Model Driven Engineering Languages and Systems (MoDELS), LNCS 3713*, pages 264–278. Springer, 2005.

[30] Object Management Group. Ocl. http://www.omg.org/spec/OCL/2.2/, accessed Monday 12th November, 2012.

[31] J. Punin and M. Krishnamoorthy. XGMML. http://www.cs.rpi.edu/research/groups/pb/punin/public_html/XGMML/, accessed Monday 12th November, 2012.

[32] A. Rensink. The GROOVE simulator: A tool for state space generation. In J. L. Pfaltz, M. Nagl, and B. Böhlen, editors, *Applications of Graph Transformations with Industrial Relevance (AGTIVE)*, volume 3062 of *Lecture Notes in Computer Science*, pages 479–485, Berlin, 2004. Springer-Verlag.

[33] A. Rensink. Nested quantification in graph transformation rules. In A. Corradini, H. Ehrig, U. Montanari, L. Ribeiro, and G. Rozenberg, editors, *Graph Transformations (ICGT), Natal, Brazil*, volume 4178 of *Lecture Notes in Computer Science*, pages 1–13, Berlin, September 2006. Springer-Verlag.

[34] Smart GUI Solutions. Jaxfront. http://www.jaxfront.com/, accessed Monday 12th November, 2012.

[35] SpringSource and the Groovy Community. Groovy. `http://groovy.codehaus.org/`, accessed Monday 12th November, 2012.

[36] Stanford Center for Biomedical Informatics Research. Protégé. `http://protege.stanford.edu/`, accessed Monday 12th November, 2012.

[37] S. Teijgeler. Connecting GROOVE to the world using XMI, August 2010.

[38] W3C. IsaViz. `http://www.w3.org/2001/11/IsaViz/`, accessed Monday 12th November, 2012.

[39] A. Winter, B. Kullbach, and V. Riediger. An overview of the GXL Graph Exchange Language. In *Revised Lectures on Software Visualization, International Seminar*, pages 324–336, London, UK, 2002. Springer-Verlag.

[40] yWorks. yEd. `http://www.yworks.com/en/products_yed_about.html`, accessed Monday 12th November, 2012.

Appendix A Implementing the conceptual model

This appendix describes the Java implementation of the conceptual model as it is used by the importer/exporter. This implementation closely resembles the formal definition of the conceptual model, though there are some differences. The goal of this implementation is to provide an intermediate layer between the graph grammar in GROOVE and the various documents of the external graph languages. The implementation has been written in Java for the same reasons stated in chapter 6, namely to ease the integration in GROOVE, with the various mappings, and with other tools.

A.1 Names and identifiers

Names in the conceptual model are represented by a `Name` class in Java. It contains a `String` which represents the literal name that is used in the type and instance model. The namespaces and identifiers are represented by the `Id` class in Java. An `Id` consists of a `Name` instance, and another `Id` which represents the namespace of the `Id`. The root namespace is modeled by a static `Id` instance named `ROOT` (which refers to itself for its namespace).

In order to link the identities used in the conceptual model to the various Java classes, an interface `Identifiable` is used to specify that a certain class is associated with an `Id`. This interface can then be used to mark all classes that represent the various elements in the conceptual model that are part of the *Id* set.

A.2 Data types

Data types are represented by an abstract `DataType` class, which is inherited by concrete classes for the various data types in the conceptual model. The boolean, integer, real and string types are mapped to the `BoolType`, `IntType`, `RealType` and `StringType` classes respectively. The `DataType` class itself uses the `Identifiable` interface, as data types in the conceptual model are all mapped to an *Id* in the root namespace (bool, int, real and string). The `DataType` class itself derives from an abstract `Type` class, which is used as a base class for all types in the conceptual model. A class diagram for the `DataType` class is provided in fig. A.2.

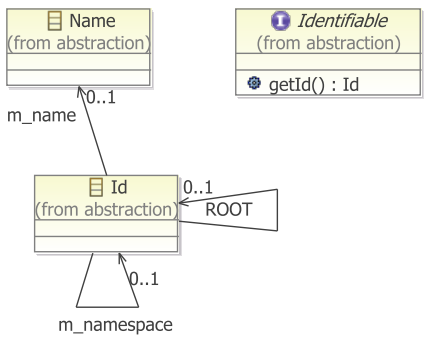


Figure A.1: Names and identifiers.

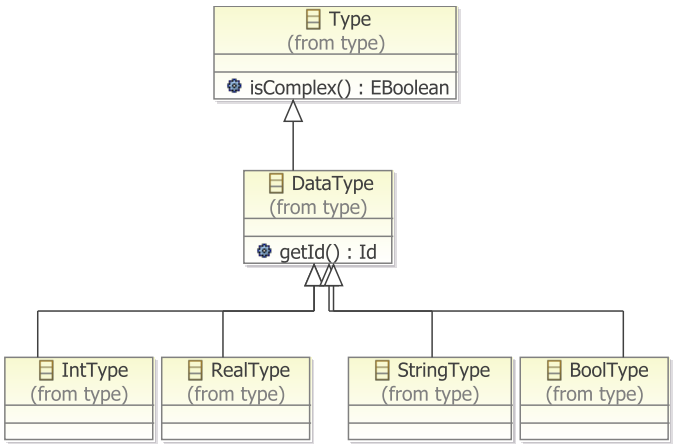


Figure A.2: Java classes for various (data) types.

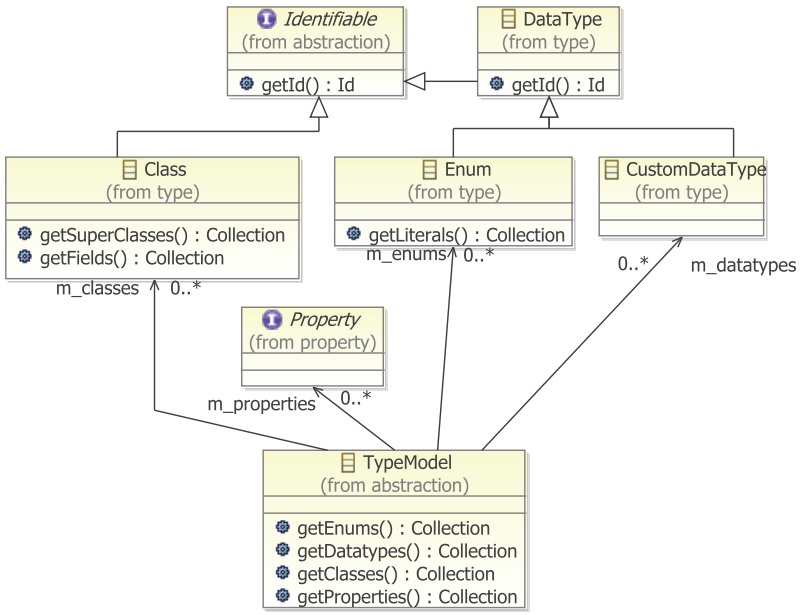


Figure A.3: The Java classes for the type model.

A.3 Type model

The type model is mapped to a `TypeModel` Java class, which contains all the entities that are relevant to the type model. In the conceptual model, a type model is specified as a tuple $\mathrm{Tm} = \langle \mathit{Class}, \mathit{Enum}, \mathit{UserDataType}, \mathit{Field}, \mathrm{FieldSig}, \mathit{EnumValue}, \mathit{Inh}, \mathit{Prop}, \mathit{Constant}, \mathrm{ConstType} \rangle$. The `TypeModel` class only has 4 fields, namely a set of classes, enumerations, custom data types and properties, representing the *Class*, *Enum*, *UserDataType* and *Prop* elements in the type model tuple respectively. All the other elements in the tuple are indirectly defined through the classes that represent the fields in the `TypeModel` class.

A.3.1 Types

The types in the type model are all represented by the abstract `Type` class. Concrete types in the implementation inherit from this class, and may refer to other types for instance to define relations. The `Type` class identifies two kinds of types: simple and complex. Complex types are build from other (complex or simple) types, such as tuples and containers. Simple types are defined on their own.

The classes in the type model are represented by a `Class` Java class. A class in the type model consists of an element in the *Class* set in the type model, the set of related *Fields* in the type model and the *FieldSig* function. All this information is represented by the `Class` Java class in combination with the Java `Field` class. The `Class` class uses the `Identifiable` interface which provides the relation between the `Class` class and the element in the `Class` set in the type model that is represented. The fields related to the class are represented by the `Field` class, which provide both the name of the field (and thus the corresponding element in the *Field* set in the type model) and the signature of the field (corresponding to the *FieldSig* function in the type model). This representation consists of a `Name` field, a `Type` field, and two integer fields that define the lower and upper bound. The `Class` class represents both the *Class* set in the type model (by means of the `Identifiable` interface) and the *ClassType* (by inheriting from the `Type` class). An instance of this class can therefore represent a *nullable* or a *proper* class. This is represented by automatically creating two instances of `Class` for each class in the conceptual model, one representing the nullable class and one representing the proper class. The proper class is automatically defined as being a sub-class of the nullable class. For every `Class`, the functions `getNullableClass` and `getProperClass` allow the corresponding class to be retrieved. The *Inh* relation in the type model is modeled by a field in the `Class` class which contains all its superclasses. As `Class` actually models the class types, this directly defines the subtype relations based on the inheritance.

The enumerations in the type model are represented by the `Enum` class, which, just like `Class`, implements the `Identifiable` interface and inherits from the `Type` class. Thus this class provides an implementation for the *Enum* set in the type model. The *EnumValue* set is represented, for each enumeration, by a field in the `Enum` class which is a set of names for each enumeration value.

The complex types in the type model (containers and tuples) are modeled by the `Container` and `Tuple` classes respectively. The Java `Container` class has a field referencing the type of the elements the container may contain (which may be of another complex type). The type of a Java `Container` is represented by a field of a Java `ContainerType` enumeration. This enumeration contains the values `ContainerType.BAG`, `ContainerType.SET`, `ContainerType.SEQ` and `ContainerType.ORD`, which represent the bagof, setof, seqof and ordof containers respectively. Another possibility would have been to use Java subtypes to define the various types of containers (for example, a `BagContainer` class), but this would mean that the logic for these containers would either have to be duplicated (for example, both seqof and ordof containers are ordered) or abstracted up to the point where a subtype of the `Container` class would add little value. *Tuples* are mapped to the `Tuple` Java class, which has a single field which is a list of other types. Just as with containers, these types may be complex and recursive.

As the data types are also part of the set of types for a type model, they also extend the `Type` class. Thus, just like classes and enumerations, the fact that they are represented as an element in the `Id` set is modeled by the `Identifiable` interface, and they are used as types in the type model by extending the `Type` class.

Finally, the properties in a type model are represented by the `Property` interface. The actual properties themselves are modeled by the `AbstractProperty`, `ContainmentProperty`, `IdentityProperty`, `KeysetProperty`, `OppositeProperty` and `DefaultValueProperty` classes. Each class defines its own set of fields which relate to the property. For instance, the `AbstractProperty` class contains a field for the class type that this property relates to. Likewise, the `KeysetProperty` has a field for the relation (`Field`) the keyset property relates to, and the set of attributes (also `Fields`) that define the keyset.

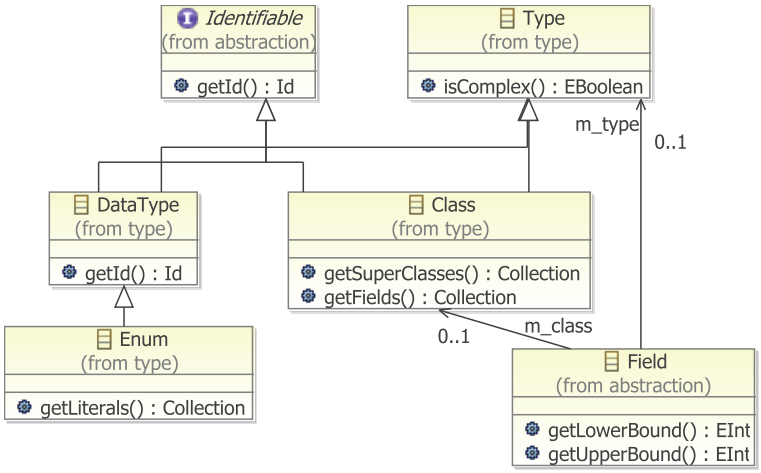


Figure A.4: The class and enum types.

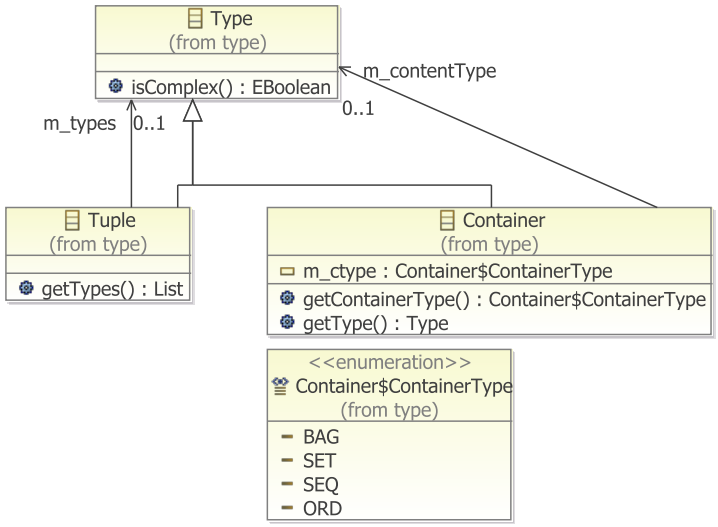


Figure A.5: The complex types.

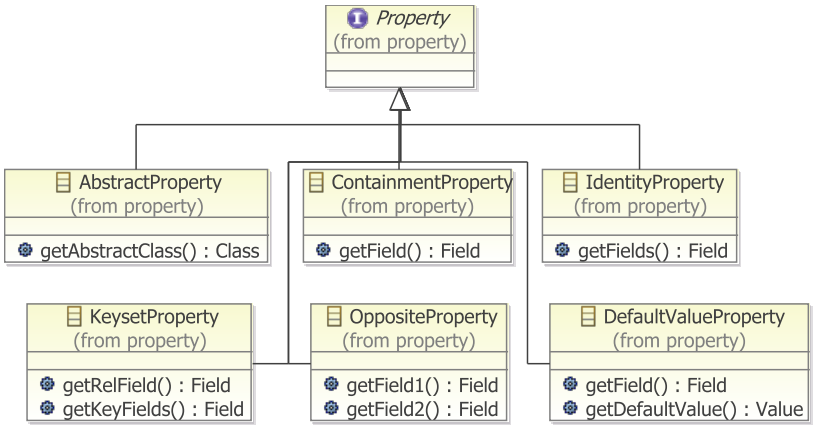


Figure A.6: The properties.

The `DefaultValueProperty` Java class represents both the *Constant* set and the $\mathrm{ConsType}$ function in the type model. In the Java implementation, both this set and function, combined with the *DefaultValue* function in the instance model, are consolidated into this single class to represent the default values for a given field.

A.4 Instance model

The instance model is mapped to the `InstanceModel` class. In the conceptual model, the instance model is a tuple $\mathrm{Im} = \langle \mathit{Object}, \mathrm{ObjectClass}, \mathrm{ObjectId}, \mathrm{FieldValue}, \mathrm{DefaultValue} \rangle$. The Java implementation contains a set of objects and a reference to the type model this instance model is an instance of. Figure A.7 shows an overview of the Java classes for the instance model.

Objects in the instance model are mapped to the `Object` class in the Java implementation (not to be confused with the system class `java.lang.Object`). This class provide fields for the name of the object (representing the *ObjectId* function in the model), the class (type) of the object (representing the $\mathrm{ObjectClass}$ function) and any field values assigned (representing the $\mathrm{FieldValue}$ function).

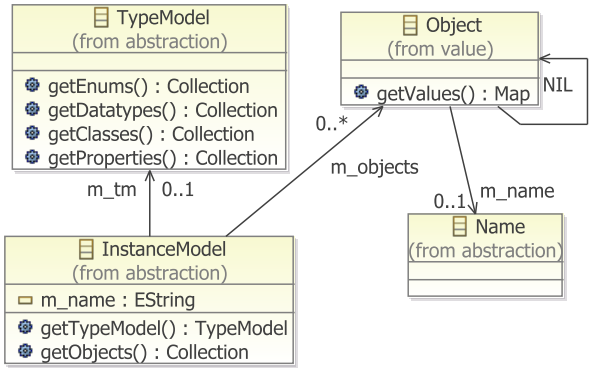


Figure A.7: The instance model.

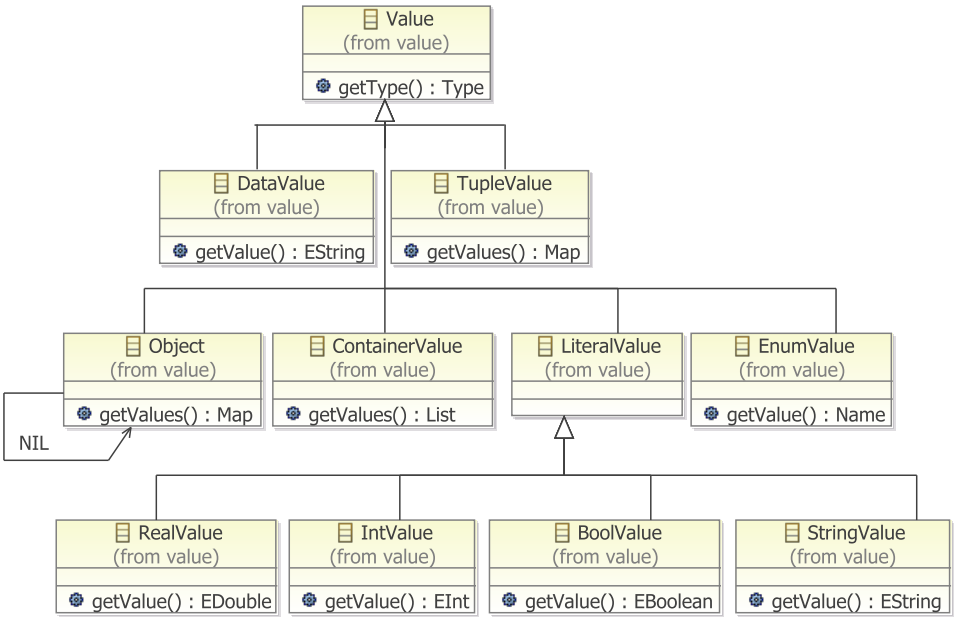


Figure A.8: Value types used within the instance model.

Appendix B Implementation of the graph language mappings

In chapter 5 an overview was given of the implementation of the mapping between the conceptual model and the graph languages. This appendix gives a more detailed description of this mapping, which is closer to the actual implementation.

B.1 Mapping Ecore-CM

As was mentioned previously, Ecore models can be compared to the type model, and instance models derived from these Ecore models can be seen as instance models in the conceptual model. Both the Ecore models and instance models are well defined by the underlying formats (such as XMI) and the specifications of Ecore itself. In fact, large parts of Ecore models can be modeled by Ecore itself (various constraints being the notable exception, this would require a more expressive language such as OCL [30]).

In [37] an extensive description is given about the capabilities of Ecore. This appendix will give a description on how the new converter will map Ecore models and instance models to the conceptual model, and what limitations apply.

B.1.1 Type model

In some cases, some attributes of an Ecore element are marked as 'derived'. This means that the value of the attribute is derived from some other attributes in the model. An example of this is the **ePackage** attribute, which is specified by the **EClassifier** element (and used by the various elements that inherit from it). The value of this attribute is the exact opposite of the **eClassifiers** attribute of an **EPackage** element, which specifies all elements contained within that **EPackage**. Therefore, mapping the **eClassifiers** attributes will automatically provide a mapping for the **ePackage** attribute (albeit in this case the reverse). Thus an explicit mapping for the **ePackage** attribute is thus not needed.

EObject

EObject is the top level class in the Ecore model. It does not define any attributes or relations by itself and does normally not occur in any Ecore model. Therefore, it does not have to be mapped to any of the concepts.

EGenericType

EGenericType is not supported by the conceptual model, as the conceptual model does not support generic types. Since the actual instance models always contain objects of a concrete class, generic classes are not required: The required concrete class can be created upon import (see also [37] for further details on generics).

ETypeParameter

Used in combination with **EGenericType**, and for the same reasons also not required.

EOperation

Models operations for classes in Ecore. As operations are not supported in the conceptual model (and have no influence in instance graphs), they are not mapped.

EParameter

EParameter models parameters for operations modeled by **EOperation**. As operations are not supported, **EParameter** is neither and will not be mapped.

EFactory

EFactory does not require a representation in the formalization, as it is only used to instantiate Ecore objects during runtime.

EAnnotation

The **EAnnotation** class inherits from **EModelElement**, and represents annotations in Ecore models. It is not modeled by the conceptual model, and annotations in an Ecore model are ignored.

EStringToStringMapEntry

This Ecore class is not supported by the conceptual model. They are primarily used for annotations, which are also not supported.

EPackage

An **EPackage** is mapped to a namespace in the conceptual model. The names of the elements inside the **EPackage** will be within this namespace. An **EPackage** has the following attributes:

- **eAnnotations**: Models annotations. This is not mapped to the conceptual model.
- **name**: The name of the package. This, together with the name of all superpackages, will provide the *Namespace* for all contained classes (see **eClassifiers**).
- **nsURI**: Not used, used for XML serialization.
- **nsPrefix**: Not used, used for XML serialization.
- **eFactoryInstance**: Not used, provides the factory for creating object instances for this package.
- **eClassifiers**: The classifiers contained in this package. The names of these classifiers will be prefixed with the *Id* of this package.
- **eSubpackages**: The packages contained within this package. The *Id* of these packages use the *Id* of this package as a *Namespace* (see **name**).

- **eSuperPackage**: Derived, opposite of **eSubpackages**.

Thus, based on these attributes, an **EPackage** will be mapped to a certain *Id* based on its name and any *Id* of a super package, if such a package exists. This *Id* will then be used as the namespace of any *Id* of either a mapped **EClassifier** or **EPackage** contained within this package.

EClass

EClass represents classes in the Ecore model, and is directly related to the *Class* set in a type model. It inherits from the abstract classes **EClassifier**, **ENamedElement** and **EModelElement**. Therefore, **EClass** provides the following relations and attributes:

- **eAnnotations**: Models annotations. This cannot be represented in the conceptual model, although the importer might add them as commentary in the imported graphs.
- **name**: The name of the class. This is the name part of the *Id* of the class in the conceptual model (the namespace is provided by the packages that contain the **EClass**)
- **instanceClassName**: Name of the Java instance class this object represents. Not used in the conceptual model.
- **instanceClass**: Java class this object represents. Not used in the conceptual model.
- **defaultValue**: Default value in Java of the Java class this object represents. Not used in the conceptual model.
- **instanceTypeName**: Name of the Java type this object represents. Not used in the conceptual model.
- **ePackage**: Derived, opposite of the **EPackage eClassifiers** relation. Used to determine the *Id* of this class.
- **eTypeParameters**: Not used, see **ETypeParameter**.
- **abstract**: The abstract property of a class is modeled by the abstract property in the conceptual model. If a class is abstract, this property is added to the type model for that class.
- **interface**: Property indicating the class is an interface. This property is not present in the conceptual model, but it can be modeled with the abstract property (as interfaces only provide operation definitions and thus cannot be instantiated).
- **e(All)SuperTypes**: Relation to other classes that this class inherits from. This is modeled by the *Inh* set. The 'All' variant provides every supertype, not only the direct supertypes. This is modeled by the $\sqsubseteq_{\mathrm{Tm}}$relation, which is transitive.
- **e(All)Operations**: Relation to operations this class has. This is not modeled by the conceptual model.
- **e(All)Attributes**: Reference to all the attributes of this class. This is modeled by the *Field* set for this class. The 'All' variant also provides the attributes of superclasses.
- **e(All)References**: Reference to all the references of this class. This is modeled by the *Field* set for this class. The 'All' variant also provides the references of superclasses.
- **e(All)StructuralFeatures**: Reference to all Ecore structural features of this class, which is the entire *Field* set for this class.
- **eIDAttribute**: Reference to the attribute which provides an identifier for this class. This is modeled by the identity property.

- **e(All)GenericSuperTypes**: Not used, as the conceptual model does not model generics (See **EGenericType**).

An **EClass** is thus mapped to various elements in a type model. The *Id* generated for this **EClass** will be based on its name and containing package, and becomes an elements of the *Class* set. The abstract and interface properties determine if an abstract property is added to the type model bearing the *Id* of this class. This means that for interface classes in Ecore, the conceptual model will use abstract classes instead. Thus, when exporting the type model back to Ecore again, this information is lost and the class is mapped to an abstract class instead. However, as interfaces do not allow for attributes and relations but abstract classes do, no further information is lost. The **eSuperTypes** property is mapped to elements in the set *Inh*, where for each class in this relation, a pair is added to *Inh* with the *Id* of this class and the super class. For each element in the **eStructuralFeatures** property, an entry to the *Field* set is added using the *Id* of this class. For the exact representation of these structural features, see appendix B.1.1. Finally, the **eIDAttribute** property is mapped to an Identity property in the conceptual model if it has been set. If so, the *Field* generated for the references **EAttribute** and the *Id* of this classes will define this identity property.

EDataType

This class in the Ecore model is used to represent built-in data types, user defined data types, as well as a super type for **EEnums**. It supports the following properties:

- **eAnnotations**: Models annotations. This cannot be represented in the conceptual model, although the importer might add them as commentary in the imported graphs.
- **name**: The name of the data type, which for user defined data types will be mapped to an *Id* based on this name and the namespace of the containing package.
- **instanceClassName**: Name of the Java instance class this object represents. Not used in the conceptual model.
- **instanceClass**: Java class this object represents. Not used in the conceptual model.
- **defaultValue**: Default value in Java of the Java class this object represents. Not used in the conceptual model.
- **instanceTypeName**: Name of the Java type this object represents. Not used in the conceptual model.
- **ePackage**: Derived, opposite of **EPackage eClassifiers**.
- **eTypeParameters**: Not used, see **ETypeParameter**.
- **serializable**: Indicates that the data type can be serialized to XML. This is not directly related to the conceptual model, but non-serializable data types cannot be imported or exported.

As mentioned, Ecore has a set of built-in data types. The follow list gives an overview of these data types and the types they are mapped to in the conceptual model.

- **EBoolean**, **EBooleanObject**: These types are mapped to the boolean data type.
- **EBigDecimal**, **EBigInteger**, **EInt**, **EIntegerObject**, **ELong**, **ELongObject**: These integer types are mapped to the integer data type.
- **EDouble**, **EDoubleObject**, **EFloat**, **EFloatObject**: These types are mapped to the real data type.

- **EChar**, **ECharacterObject**, **EString**: These types are mapped to the string data type.
- **EByte**, **EByteArray**, **EByteObject**, **EDate**, **EDiagnosticChain**, **EEList**, **EEnumerator**, **EFeatureMap**, **EFeatureMapEntry**, **EJavaClass**, **EJavaObject**, **EMap**, **EResource**, **EResourceSet**, **ETreeIterator**: These are either complex data types, or unserializable data types and thus are not supported.

User defined data types are mapped to the *UserDataType* set, using the *Id* generated for this data type. Note that the actual Java instance class name will be lost upon import, but the data type itself may still be referenced.

EEnum

Inherits from **EDataType**. Represented by an element in the *Enum* set. Since it inherits from the **EDataType** class it supports the same properties. However, they are all ignored with the exception of the name and the contained **ELiterals**.

- **name**: The name of the enumeration, which will be mapped to an *Id* based on this name and the namespace of the containing package.
- **eLiterals**: Literals that belong to this enumeration. Represented by the *EnumValue* set for this enumeration.

Thus, each **EEnum** is mapped to an element in *Enum* based on the generated *Id*. An entry is added to *EnumValue* based on the *Id* of this enumeration and the contained **EEnumLiterals**.

EEnumLiteral

Defines a literal for an enumeration (**EEnum**). These are mapped to elements in the *EnumValue* set. It supports the following properties:

- **eAnnotation**: Not used.
- **name**: Name of the literal, as it appears in *EnumVal*.
- **value**: Not supported, the conceptual model does not support a separate definition of values for enumeration literals.
- **instance**: Not supported, this represents the Java instance.
- **literal**: String representation. This is not used in the conceptual model, but rather the **name** (which most often is identical).
- **eEnum**: Derived, opposite of **eLiterals** of **EEnum**.

For each **EEnumLiteral**, an entry is added to *EnumValue* based on the *Id* of the containing **EEnum** and the name of the **EEnumLiteral**, which is mapped directly to an entry in the set *Name*.

EReference/EAttribute

References and attributes are mapped to the *Field* set in the type model, based on their containing class. Each reference and attribute specifies an element in the *Field* set and the FieldSig function.

- **eAnnotations**: Models annotations. This cannot be represented in the conceptual model.
- **name**: The name of the reference. This is the name part of the *Field* descriptor.
- **ordered**: Only relevant if the type will be mapped to a container. If this is true, the container is ordered and thus either of type ordof or seqof.
- **unique**: Only relevant if the type will be mapped to a container. If this is true, the container contains only unique items and thus either of type setof or ordof.
- **lowerBound/upperBound**: Specifies the multiplicity of the reference, which is the *Mult* element of the *Field* descriptor.
- **many**: Derived from **upperBound** (true if upperBound ¿ 1).
- **required**: Derived from **lowerBound** (true if lowerBound ¿ 0).
- **eType**: Type of the reference. Modeled by the *Type* element in the *Field* descriptor of this reference.
- **changeable**: Modeled by the readonly property as its inverse.
- **volatile**: Not used, indicates whether a field will be generated in an implementation class in Java.
- **transient**: Not used directly, indicates the value will be persisted when saving the model. If a reference or attribute is transient, the value of it will not be persisted in an instance model and as such it will not occur when importing such an instance model. Normally these values are created during runtime.
- **defaultValueLiteral**: Modeled by the defaultValue property in combination with an element in the *Constant* set in the type model, and an element in the $DefaultValue_{\text{Im}}$ set in an instance model.
- **defaultValue**: Derived from **defaultLiteralValue** and represents the actual Java object for the default value.
- **unsettable**: Not supported by the conceptual model. Indicates the reference/attribute can be 'unassigned'.
- **derived**: Not used. Means derived from other attributes/references.
- **eContainingClass**: Opposite of **eStructuralFeatures** in **EClass**.

An entry is added to the set *Field* based on the *Id* of the containing class (**eContainingClass** property) and the name of this attribute/reference. The **lowerBound/upperBound** and **eType** properties determine the matching entry that is added to FieldSig. The pair **lowerBound/upperBound** is mapped directly to an entry in *Mult*. The class, enumeration or data type referenced to by the **eType** property is mapped to an element in *Type*. For classes, this is a nullable class if **lowerBound** equals 0 and **upperBound** equals 1, otherwise it is a proper class. If the **upperBound** is larger than 1, the type of the field (and as mapped to FieldSig) will be wrapped in a container type (an element from *Container*). This container will be setof if both **unique** is true and **ordered** is false, bagof if both properties are false, ordof if both properties are true and seqof if **unique** is false and **ordered** is true. The FieldSig entry will then be based of the selected element in *Mult* and the selected element in *Type*.

References in Ecore also have a set of properties not found in attributes. Properties specific for references:

- **container**: Derived, value based on the containment property of an **eOpposite**, if it exists.
- **containment**: Modeled by the containment property.

- **resolveProxies**: Not used, proxies are not used in the conceptual model.
- **eKeys**: references a set of **EAttributes** in the target **EClass** and is modeled by the keyset property.
- **eOpposite**: Modeled by the opposite property.
- **eReferenceType**: Derived from **eType** and not used directly.

Of these properties, the **containment**, **eKeys** and **eOpposite** properties are relevant to the mapping. If the **containment** property is true, a containment entry is added to the set *Prop* in the type model, based on the element in *Field* generated for this reference. If **eOpposite** is set, an opposite entry is added to the set *Prop* in the type model, based on the element in *Field* generated for this reference, and the element in *Field* for the reference as referenced by **eOpposite**. Finally, the **eKeys** attribute, if set, creates an entry keyset in the set *Prop*. This entry is based on the entry in *Field* generated for this reference, and the set of elements generated in *Field* generated for each **EAttribute** referenced by **eKeys**.

Likewise, attributes have a few properties not found on references. Properties specific for attributes:

- **iD**: Specifies that the attribute is the identifier of the containing class. This is modeled by the identity property, as discussed for **EClass**.
- **eAttributeType**: Derived from **eType** and not used directly.

No extra property of **EAttribute** is used by the conceptual model, so only the previous mentioned properties are used.

B.1.2 Instance model

Ecore instance models consist of objects. When serialized to XML (using the XMI standard), these objects are represented as elements in the XML document. Attributes and references are then defined as either XML attributes (possibly referring to other elements in the XML tree) or child elements for containment relations. As each element in the XML document has to be contained by another elements (with the exception of the root elements), the importer should find all those objects which are not contained by another object by means of a containment relation. These objects are then added to the Ecore `Resource`, which ensures that all objects are contained by that `Resource`

Objects are represented by elements in the set $Object_{\mathrm{Im}}$. Within the XML document, objects are referred to by a path, which is unique for each object (as each object is contained by at most one other object, although it may be referenced by multiple others). This may be mapped to an element in *Id*, thus providing an element in the $\mathrm{ObjectId}_{\mathrm{Im}}$ function.

XML attributes and child elements in the document provide the values for all the fields of objects. Attribute values may be either literal values, which are already represented by the $Value_{\mathrm{Im}}$ set for the data types or enumerations, or references to other elements in the XML document, in which case they will be treated as a value for an element in the $Object_{\mathrm{Im}}$ set. For reference or attributes that refer to multiple values, the element mapped to in $Value_{\mathrm{Im}}$ will be retrieved from the set $ContainerValue_{\mathrm{Im}}$. The *Type* of the *Field* as determined by the reference or attribute in the type model will determine that actual type of the *Container* for the $ContainerValue_{\mathrm{Im}}$. Each element in $Value_{\mathrm{Im}}$ of the reference or attribute will be appended to the list of values of the $ContainerValue_{\mathrm{Im}}$, maintaining the order as it was found in the Ecore instance model (even if the order may be arbitrary for unordered types). For user defined data types, the values are mapped to elements in the set of strings $\mathbb{S}$, as they are always serialized as strings. These strings will be handled as opaque in the conceptual model and will be exported back again verbatim.

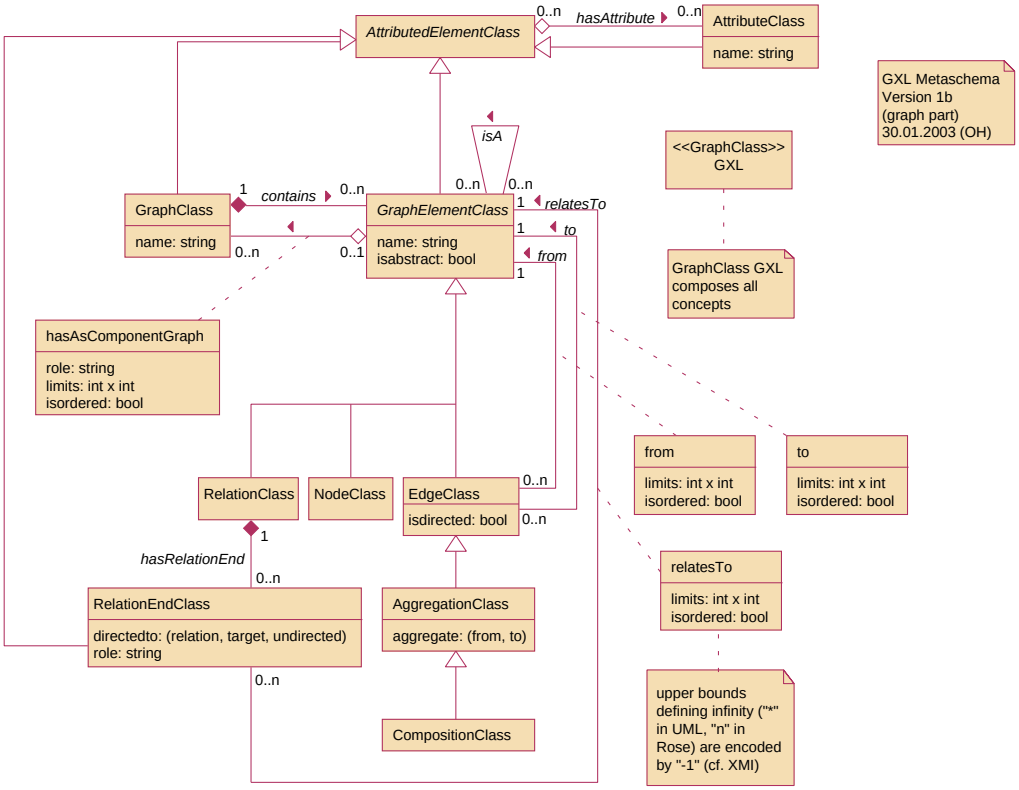


Figure B.1: Graph part of the GXL meta schema.

The importer will define the contents for $DefaultValue_{\mathrm{Im}}$ based on the given type model. This process will be based on the serialization functionality of the Ecore model, where the default value in the type model will be deserialized into a value in the set $Value_{\mathrm{Im}}$.

B.2 Mapping GXL-CM

In GXL, a single document can provide multiple type models and multiple instance models. These type models and instance models are provided by 'graph' elements in the document. More specifically, a graph in a GXL document is either an instance graph, or a graph describing one or more type graphs. Although it is possible for these elements to recursively contain other graphs (embedded by nodes, edges or relations), this is not always supported by the importer. A type model may contain subgraphs, but these are ignored. However, it is allowed for multiple graphs to be defined by a type model, which may be defined by special nodes (see the following explanation of type models). Subgraphs in instance graphs are allowed, and are treated as providing the namespace for an object. The actual interpretation of these subgraphs is explained in the following sections for the type model and the instance model.

The type and instance models are linked by the use of the **type** attribute (which is also used by the type model itself to refer to the meta schema). The **type** attribute of, for example, a node or edge in an instance graph refers to a single node, or edge, in a type graph. Likewise, elements in a type graph have a **type** that refers to elements from the meta schema.

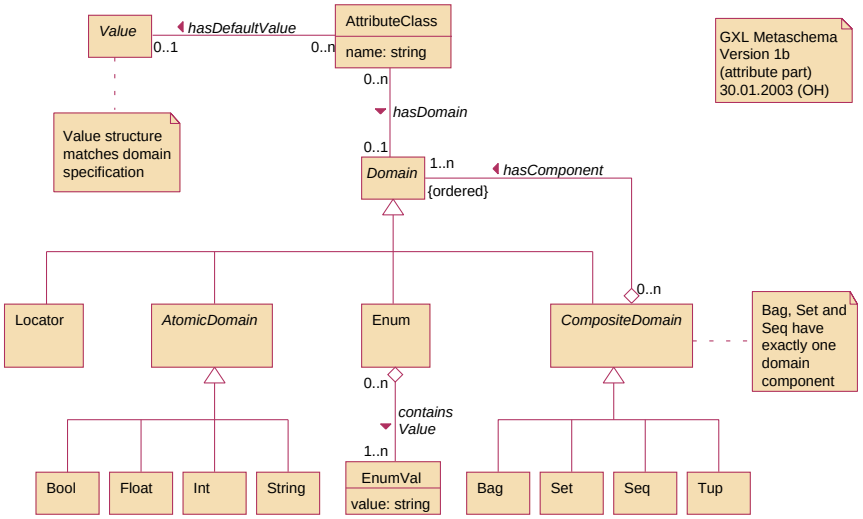


Figure B.2: Attribute part of the GXL meta schema.

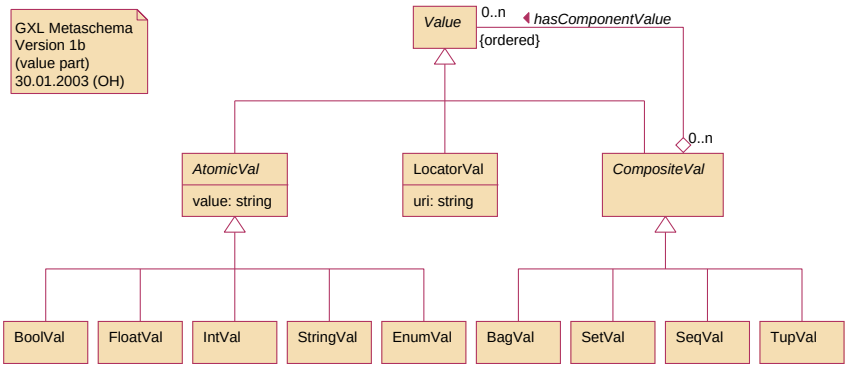


Figure B.3: Value part of the GXL meta schema.

B.2.1 Type model

A GXL type model always references the GXL meta schema. In the meta schema, nodes represent the various elements which will be mapped to the conceptual model, and edges represent various relations between these elements. The nodes and edges themselves may be attributed as well.

The meta schema specifies various nodes representing the actual elements that exist in a GXL model. These nodes may be abstract, in which case they are not mapped directly, but do specify various attributes and relations for subtypes. A graphical representation of this meta schema may be found in fig. B.1.

The following sections describe the mapping for the various parts of the meta schema, which may occur in a GXL type graph.

GraphClass

This node is used to represent a graph in an instance model, which is represented by using a namespace bearing the name of the graph that is defined by the **GraphClass** node. This name is derived from the **name** attribute. Graph attributes are not supported. **GraphClass** nodes always have a **contains** relation with all the nodes, edges and relations that are defined to be in this graph. This containment relation defines this graph to be the namespace of the contained nodes. If a **GraphClass** node is contained within another **GraphClass** (indirectly by a **NodeClass**, **EdgeClass** or **RelationClass** node), then the containing **GraphClass** will be used as the namespace of this **GraphClass**. Otherwise, the namespace will be the root namespace. The name of the graph element that contains the subgraph by the **hasAsComponentGraph** will not be used in the namespace.

RelationClass, RelationEndClass

These nodes are used to represent relations in a GXL model, which is not supported by the conceptual model. Hence, these node types are not supported.

NodeClass

This node is used to represent actual nodes in a GXL model. These nodes are mapped directly to classes in the conceptual model. The **name** attribute is used as the name of the class, and the graph that contains the **NodeClass** is used as its namespace. Attributes of this node are mapped as *Fields* in the conceptual model, of which the type is the type of the attribute (see **AttributeClass**) and the name is the **name** of the attribute. The abstract attribute is mapped to the abstract property in the conceptual model. The type model may also specify **isA** edges, which are mapped to the *Inh* set in the conceptual model.

EdgeClass

This node is used to represent actual edges in the type model. These edges are mapped to relations in the conceptual model as *Fields*. The **to** edge is used to define the type of the *Field*, the **from** edge is used to define the class that contains the field. GXL supports **limits** on *both* sides, however the conceptual model does not (only on the **to** side). Hence, the limits on the **to** side are directly mapped, the limits on the **from** side are only used for 'complex' edges (see below). If the upper bound of the limit of the **to** side is larger than 1, then the type of the generated field will be a container type. Otherwise, if the lower bound is 0, it will be that of a nullable class. A multiplicity of 1..1 is mapped to a proper class.

Edges are allowed to have attributes in GXL, as well as being abstract and inheriting from other edges. Since references in the conceptual model do not support any of this, these properties are either ignored or the edge is promoted to a class. If an edge is promoted to a class, it is mapped to the set of classes in the conceptual model, with two fields for the **to** and **from** edges, each with the multiplicity set to the **limits** attribute of each target. The attributes, abstract property and inheritance are mapped as with NodeClass nodes. These edges are then referred to as 'complex' edges.

Since GXL allows edges to connect any graph elements (nodes, edges and relations), not all **EdgeClass** nodes can be mapped, as only relations between classes are allowed. Thus, **EdgeClass** nodes connecting relations cannot be mapped, and **EdgeClass** nodes connecting other **EdgeClass** nodes can only be mapped if these targeted **EdgeClass** nodes are mapped to classes.

AggregationClass, CompositionClass

These nodes are specialisations of **EdgeClass** and behave the same. However, they additionally specify composition which is mapped to the containment property. The **aggregate** attribute of these nodes specifies the direction of the composition.

AttributeClass

This node is mapped to a *Type* in the conceptual model. An **AttributeClass** node may have an edge pointing to a *Value* as its default value by a **hasDefaultValue** edge. This is mapped to a *Constant* in the conceptual model, with a corresponding entry in $\mathrm{ConstantType}$ which is the type of the attribute. The value itself is placed in the $\mathrm{DefaultValue}$ function in the conceptual model, and a defaultValue property is created in the type model. The type of the attribute is specified by the **Domain** pointed to by the **hasDomain** edge. A graphical overview of attribute nodes can be found in fig. B.2.

Domain and subtypes

These node classes represent the actual types in the conceptual model apart from the classes. **Bool**, **Int**, **Float** and **String** are mapped to their data type counterparts in the conceptual model (**Float** being mapped to real). The **Locator** type is mapped to string, as the conceptual model does not support this type directly. However, locators can generally be represented by string without loss of information (but with a change in semantic meaning).

The **Bag**, **Set** and **Seq** classes are mapped to containers in the conceptual model of which the type is bagof, setof or seqof respectively. The **Tup** type is mapped to a tuple in the conceptual model. The contained types are recursively defined by the **hasComponent** edge or edges (multiple edges can be used in the case of a **Tup** node).

The **Enum** node is mapped to an *Enum* in the conceptual model. Enumerations in GXL do not have their own name. However, they are defined by a node which is referenced to by a unique ID, which can be used to define the name of the enumeration. The **Enum** node contains **containsValue** edges pointing to **EnumVal** nodes. These nodes represent the values of the enumeration, and are mapped to the $\mathrm{EnumVal}$ function in the conceptual model. The **value** attribute of these nodes is used as the name of these enumeration values.

B.2.2 Instance Model

A GXL graph that is typed by a GXL type graph may be mapped to an instance model in the conceptual model. This relation is defined by the **type** attribute of the graph, which should point to a

GraphClass node in a type graph. The actual graph then consists of nodes, edges and relations. Since the conceptual model does not support relations, they are ignored.

Nodes are mapped to *Objects* in the conceptual model. The *Class* that is the type of the object is determined by the **type** of the node. This type should point to a class in the conceptual model, via the mapping of a GXL type graph to the conceptual model. The *Id* of the *Object* is determined by the **ID** attribute of the node. The attribute values of the node are mapped to the field values in the conceptual model for these attributes.

Edges are mapped either to *Field* values, or *Objects*. This depends on the type of the edge. If the edge is not complex, it will be used as a field value for the object that represents the node that is the source of the edge. The edge type determined the field that is assigned a value. It is possible for multiple edges of the same type to exist. In this case, the value will be part of a container value.

If the edge is complex, it is mapped to an *Object*, of which the type is that of the class that was generated for the edge. The attribute values of the edge (if any) are mapped to the field values of the edge *Object*. The source and target nodes of the edge are the values for the **from** and **to** fields generated for this edge.

B.3 DOT-CM

DOT as a language does not provide semantics as detailed as Ecore and GXL. Rather, DOT itself is defined by an EBNF grammar (see listing B.1 and [4]), with only a few predefined constructs. A DOT file consists of the definition of a graph, which may contain node definitions, edge definitions (which may implicitly define nodes) and subgraphs (which may recursively contain other graphs). Furthermore, graphs, nodes and edges may be attributed. The meaning of these attributes is not defined by the language itself, but by the tools that read DOT files. However, many attributes are shared between these tools with the same meaning. An example of this is the **label** attribute, which defines the textual label associated with an element in the graph.

The DOT language also defines the use of default attributes. Within a graph, a default value may be assigned for the graphs, nodes, and edges that follows the specification of this default. The use of these default attributes is not supported. Although they relate to the defaultValue property in the conceptual model, their semantics differ too much under the current interpretation of DOT graphs.

Edges may be defined as being directed (when the graph is specified to be a directed graph defined by the **digraph** keyword) or undirected (for plain graphs, defined by **graph**). The importer and exporter only consider directed graphs, as all relations and references in the conceptual model have a direction. Furthermore, nodes may specify ports in DOT. Ports identify certain points on a single node that an edge may connect to. This is not supported by the importer and exporter as well, and thus node ports are ignored.

B.3.1 Type models

Type models can only be exported. A DOT file can only be imported as an instance model, for which the type model will implicitly be created. Section 5.3 describes how a type model is exported. As the process itself is fairly simple, this description is very close to the actual implementation.

Exporting

When a type model is exported, the following steps are taken:

1. For each class, a node is created with a label that contains all the fields of which the type is a data type. For fields that are typed by other classes, an edge is created instead. Based on the namespace of the class, a subgraph is created representing that namespace. If the class has an abstract property, the node is drawn with a dashed outline.
2. For each superclass, an edge is created between the two nodes representing the superclass and the subclass, in a style resembling that of UML inheritance relations.
3. For enumerations, a new node is created of which the label contains all the enumeration values.
4. For tuples, a record type node is created, which visually distinguishes the various elements of the tuple. Primitive types in the tuple are embedded in the node label, other types are referenced to by an edge.
5. Edges created for field that have a containment property are drawn with a diamond shaped tail.

B.3.2 Instance models

Instance models can both be imported from and exported to DOT. When an instance model is imported, a type model is implicitly created as well. When exporting an instance model, no relation is made to the type model of that instance model in the resulting DOT graph. Rather, the graph will stand on its own.

Importing

The importing process consists of the following steps:

1. An instance model is created for the graph, with the graph providing the root namespace.
2. Each node in the graph is visited. A new class is created for the node in the implicit type model, and a new object is created of that class in the instance model. The name of the class is based on the label of the node or, barring this, the ID of the node. Next, for each attribute assigned to the node explicitly, a new field is added to the implicit class of a string type. In the instance graph, the value of the attribute is assigned to this field.
3. Next, each subgraph is visited. This happens recursively, with each subgraph providing a new namespace.
4. Finally, all the edges of the graph are visited. Each edge creates a field in the implicit type model, of which the type is the class of the node the edge points to. Visiting the edges as the last step ensures the node classes and objects are created before the relations between these classes and objects.

Exporting

Exporting an instance model is fairly trivial. The export is performed exactly as is described in section 5.3. This results in a DOT graph that is not suitable for interpretation, but allows the instance model to be graphically visualized using the various tools available that can parse DOT files and generate images from them.

```
graph     : [ strict ] (graph | digraph) [ ID ] '{' stmt_list '}'
stmt_list : [ stmt [ ';' ] [ stmt_list ] ]
stmt      : node_stmt
          | edge_stmt
          | attr_stmt
          | ID '=' ID
          | subgraph
attr_stmt : (graph | node | edge) attr_list
attr_list : '[' [ a_list ] ']' [ attr_list ]
a_list    : ID [ '=' ID ] [ ',' ] [ a_list ]
edge_stmt : (node_id | subgraph) edgeRHS [ attr_list ]
edgeRHS   : edgeop (node_id | subgraph) [ edgeRHS ]
node_stmt : node_id [ attr_list ]
node_id   : ID [ port ]
port      : ':' ID [ ':' compass_pt ]
          | ':' compass_pt
subgraph  : [ subgraph [ ID ] ] '{' stmt_list '}'
compass_pt: (n | ne | e | se | s | sw | w | nw | c | _)
```

Listing B.1: DOT language grammar

The export process of an instance model is very similar to that of a type model:

1. All objects are enumerated. A node is created for each object in a (sub)graph based on the namespace of the class that types the object. For each assigned field value, either the label of the object is appended with this field value (for data types and enumerations), or an edge is created to another node representing the value.
2. For tuples, a node is created similar to one created in the type model.

Appendix C Implementation of the GROOVE mapping

This appendix describes the import and export process of GROOVE graphs in more detail. When a type or instance model is imported to or exported from a GROOVE grammar, various graphs may be involved, which are used by different steps. The import process happens as follows:

1. A type model is exported to a type graph. As much information as possible from the type model is added to the graph.
2. If enabled, a meta graph is generated as well, providing more detailed information about the type model.
3. Information not representable by the type and meta graphs may be added by means of rule graphs. If they are enabled, they will be added at this point.
4. Next, instance models can be imported as instance graphs. An instance model is represented by exactly one instance graph, which is typed by the previously created type graphs.

Exporting graphs happens in a similar manner:

1. First, the meta graph is analyzed. Some node type information is retrieved from the graph, but no type model is created yet. Note that this is an optional step.
2. The actual type graph is exported into a type model. The information from the meta graph may be used to aid this process.
3. When applicable, rule graphs are analyzed next. These may be used to further enrich the type model, for instance by adding extra properties.
4. Finally, instance graphs may be exported to instance models. This requires the mapping information from the type model export.

C.1 Type graphs

C.1.1 Import

Importing a type model into a GROOVE type graph consists of a number of (recursive) steps, each of which creates some nodes or edges. The procedure works as follows:

1. A new empty graph is created for the type model. Each class in the type model is enumerated.
2. When a class is being enumerated, a new node is created for that class. If the class being referenced is a nullable class, an abstract nullable node is created. Otherwise, a concrete proper node is created. For the nullable node an inheritance edge is created between that node and the global Nil node, as well as the proper node representing the same class.
3. For a proper class, the node for each superclass in its inheritance relation is retrieved (recursively) and an inheritance edge is created. Next, each field of the class is enumerated, and the node representing its type is retrieved. Between the class node and the field node an edge is created with the name of the field and its multiplicity.

4. Fields are represented by the node representing their type. This may be a node for a single type directly, or an intermediate node. This depends on the configuration.
5. For types with an intermediate node, the intermediate node is created, with a type name based on the name of the node referencing it. The intermediate node will point to a node representing its underlying type by means of a single value edge. Additionally, it may be assigned a next (and optionally prev) edge, or an index value.
6. Primitive data types are represented by primitive nodes. This applies to booleans, integers, real numbers and strings.
7. Tuples are represented by a single node, which has an edge pointing to each type element of the tuple.
8. Containers are represented by an intermediate node.
9. Custom data types are represented by a single node with a string attribute.
10. Next, the enumerations and custom data types referenced by the type model but not related to any class are enumerated, thus ensuring all types are added to the graph.
11. Finally, the properties of the type model are visited and handled as follows:
    (a) Abstract properties are represented by adding the abstract modifier to the nodes representing the proper class that is abstract.
    (b) Containment properties are represented by adding the containment property to edges for fields. If the field is represented by a type directly, the containment property is added to the field edge. If the field is represented by an intermediate node, the containment property is added to the value edge of that node.
    (c) The Opposite property is represented by two opposite edges between the intermediate nodes representing the opposite fields. Note that this has to be enabled by the configuration. The importer will ensure that for opposite fields an intermediate node will always be used.
    (d) The identity, keyset and defaultValue properties are not represented by the type graph, but by rule graphs.

C.1.2 Export

Exporting a type graph tends to be more complicated then importing one, as the concept that a node represents may have to be determined by looking at its neighbors or other contextual information. To aid in this, certain postfixes are used which are appended to type names, as well as the optional meta graph.

When a meta graph is used, it is parsed first as it unambiguously defines the type of each node that is added to the meta graph. The meta graph is parsed by first identifying the meta nodes which represent classes, enumerations, data types, intermediate nodes, and containers. Since the names of these nodes are always predetermined, this is trivial.

Next, the nodes representing elements from the conceptual model are identified by looking at their inheritance relation in the graph. When a node inherits from a meta node, it is interpreted to be of that class of nodes. Thus, a node inheriting from the meta class node will be treated as a node representing a class in the type model.

Next, the actual type graph is analysed. This happens in two phases. First, each node is classified using the same system of classes as the meta graph. Then the actual type model is created by walking through all these nodes.

The first phase occurs in several iterations:

1. The names of all unidentified nodes are checked for certain postfixes. This will be used to identify enumerations and custom data types. When configured as such, this will also be used to identity intermediate nodes and container types.

2. All the edges between the unidentified nodes are checked to see if they point to an intermediate node. Such nodes can be recognized by the fact that the name of the source node is the namespace of the target node.

3. In the final iteration, the rest of the nodes are classified. If the node inherits from an enumeration node, it is classified as an enumeration value. Primitive type nodes and the Nil value node are identified by their type. The rest of the nodes are assumed to be class nodes.

Next, the second phase will create the actual type model based on the information collected thus far. Again, this happens in several iterations:

1. Classes, enumerations (though not their values) and data types are instantiated in the type model for nodes classified as such. For classes, a check is made if the node is abstract, in which case the Abstract property is created as well.

2. The class inheritance relation is built from the inheritance relation between all the (proper) class nodes.

3. Enumeration values are instantiated and added to their respective enumerations.

4. Intermediate nodes are resolved to their types recursively (as an intermediate node may point to another intermediate node). The type information is attached to these nodes, but not yet added to the type model. If possible, the container type is determined here as well for intermediate nodes representing containers.

5. Tuple nodes are mapped to a tuple type, and the type of the target of each outgoing edge is added to the tuple.

6. Field edges are enumerated. For each edge a field is added to the class represented by its source node, of which the type is that of the type of the target node. The edges are checked for the containment property as well, for which the Containment property is added to the type model if present.

After the second phase, the structure of the type model is complete. However, some constraints may still have to be applied, either as a property or by means of changing the container type. This is done by analyzing the rule graphs in the grammar as show in appendix C.3.

C.2 Instance graphs

C.2.1 Import

Importing instance models happens in a fashion similar to type models. The objects in the instance model are enumerated and nodes representing those objects are created, with relations between them and other values being added as edges.

The import process follows the following steps:

1. The objects in the instance model are enumerated one by one.
2. Each enumerated object is represented by a node in the instance graph, which is created when the object is enumerated.
3. For each object, all the assigned field values are enumerated. In addition, a defaultValue property may apply to one of the fields of the object's class. If no value is assigned and the configuration is set to assign default values if applicable, the default value will be considered to be assigned as well. The node representing the value is retrieved and an edge is created between the value (which may in fact be another object) and the current object.
4. For values of primitive types, a primitive value node is created.
5. For values of a custom data type, a node representing that data type is created with a string attribute assigned containing the string literal that represents the data value.
6. For tuples, a tuple node is created with an edge pointing to each value in the tuple.
7. For containers, multiple nodes are created, one for each contained value. Thus, a container is not actually represented by a single node, but by many. If a reference is created to a container node, an edge is created for each container value.
8. Finally, if there exists an Opposite property for two fields, and the configuration is set to use opposite edges, these edges are added to the instance model as well.

C.2.2 Export

Exporting an instance graph to an instance model happens in two steps. First, all the object nodes are identified. Then, the assigned field values are analyzed and assigned to those objects. This process depends on the type information which is retrieved from a type graph which is exported to a type model. By exporting a type graph first, the node types are linked to types in the type model which allows the instance graph exporter to easily identify the various nodes it contains.

The process to export an instance graph works as follows:

1. All the object nodes are identified. Object nodes are recognized by being of a type that is directly related to a class in the type model. If the object node has an identity assigned to it, it will be assigned to the object in the instance model as well.
2. For each object node, each possible field value assigned is checked. If there exists an outgoing edge representing such a field, the target of the edge will be the value of that field.
3. Primitive data type nodes are treated as simple primitive values.
4. For custom data type nodes, the assigned string attribute will be used to store its string representation.
5. For intermediate nodes, a check is made if it represents a container or simply a field with a single value. In case of the latter, the intermediate node will represent the value it points to. For the first, it will represent the collection of values determined by its outgoing edges. Additionally, for ordered containers the ordering is determined. When the ordering is based on an index, the index values are sorted in ascending order. When edges are used, the order is based on the sequence of these edges.

Note that for all these steps, the information from the type graph is necessary to identify the type of each node.

C.3 Rule graphs

When importing a type model, various constraint rule graphs may be created. These rule graph can represent some information from the type model that it cannot represent by itself. Additionally, these rules help keep the instance models in the grammar consistent with the type model, preventing the creation of invalid models.

When a type model is exported, these rule graphs may be used again to retrieve the information they represent. This could either be a property in the type model, or some extra information for various types (such as container ordering and uniqueness). This requires the type graph to be exported to a type model first, so that the information from that conversion may be used to identify the various types of nodes in these rule graphs as well.

C.3.1 Opposite rules

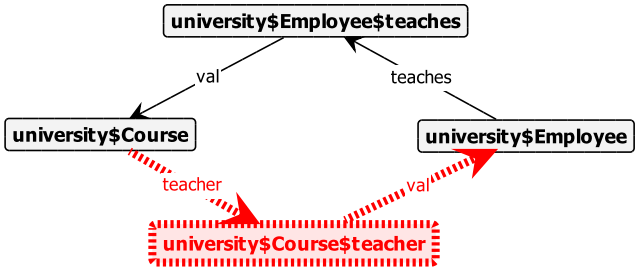


Figure C.1: Example opposite rule graph.

The opposite rule graph represents the opposite property and serves a dual purpose. First, it specifies which two relations are opposite, and puts a constraint check on instance graphs that this property holds. The structure of the rule consists of two nodes representing a class, with in between them edges (and possibly intermediate nodes such as in the example) representing the opposite relations. Thus, the information of an opposite property is fully present in the rule (both relations are uniquely identified). Alternatively an opposite edge may be present between two intermediate nodes in the type graph. However, it can be configured not to be used, in which case the rule graph can be used to retrieve the information instead. Second, the rule graph also adds a NAC to one of the relations. This way, a constraint is added, which is matched when two nodes exist in an instance graph, which have one relation between them but not the opposite. Opposite properties, and thus the opposite rules, always come in opposite pairs such that the constraint is checked in both directions at all times.

C.3.2 Unique rules

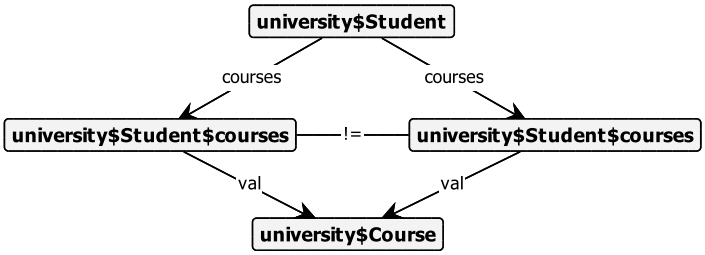


Figure C.2: Example unique rule graph.

The unique rules are used both to determine whether or not a container is unique (if this information is not stored in the type graph) and to enforce this uniqueness. These rule graphs are only used to represent unique containers that use intermediate nodes, otherwise this information is implicitly present and enforced (no two identical edges can connect the same pair of nodes in GROOVE). Unique rules uniquely identify the intermediate nodes of unique containers, such that the information is accurately represented. The constraint is enforced by checking if no two identical nodes are present in the container, or, in the case of ordered containers, no two nodes with the same index. This is not the exact same as the definition of equivalency in the conceptual model (see definition 15), as these rules do not check equivalency for recursive containers (containers contained within other containers). This would make the rules significantly more complex while such constructions rarely occur. Therefore, only single node values are checked.

C.3.3 Ordered rules

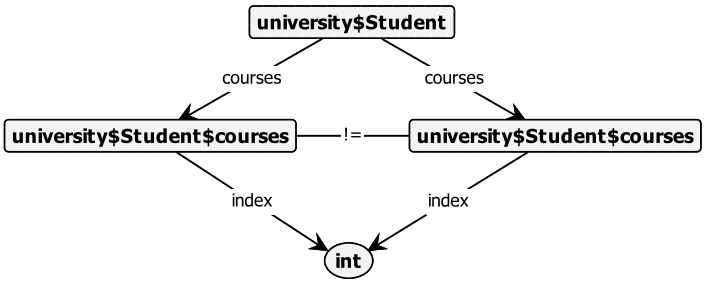


Figure C.3: Example ordered rule graph.

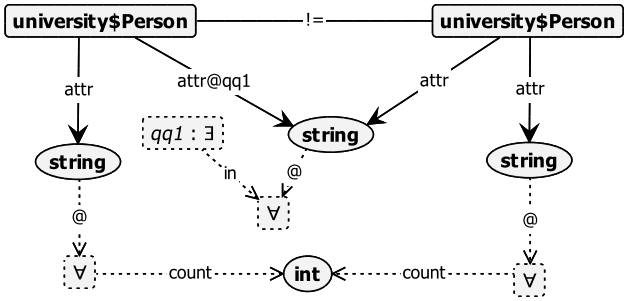
The ordered rules represent the fact that a certain container is ordered. Its purpose is only to enforce that elements of such containers are always well ordered in instance graphs. Since ordering can be represented in two ways (using an index or using a next edge), two types of rules are present. The first type, for use with indices, simply checks of no two elements in the same container have the same index value. If this is the case, the elements are well ordered since each index value is either larger or smaller than any other index value. The second type, for next edges, checks only if there are no more than one node in the container that have no incoming next edge. This is sufficient, as the edge multiplicities in the type graph ensure that each node has either zero or one outgoing next edge. Only the last node in the container is allowed to have no outgoing next edge, as otherwise the constraint rule would match (since there would be two nodes with no incoming next edge).

C.3.4 Enum

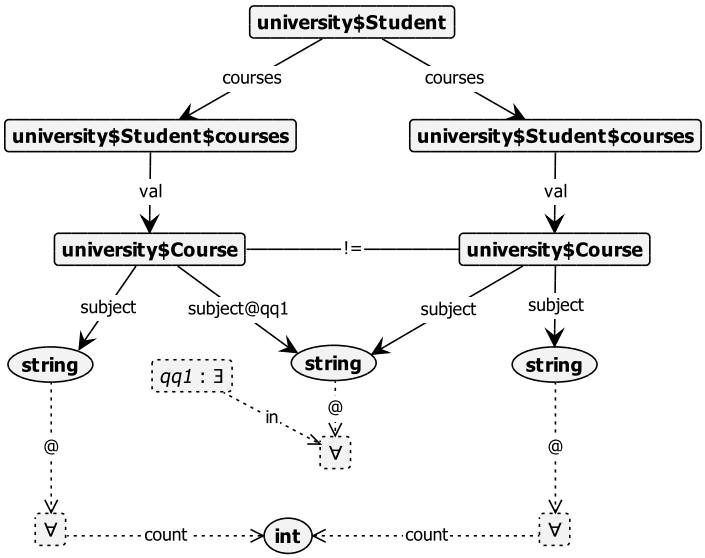
Figure C.4: Example enum rule graphs.

Constraint rules are also added for enumerations when they are represented using flags. These rules ensure that enumeration nodes in instance graph have exactly one flag set indicating which enumeration value they represent. To this purpose, two rules are created for each enumeration. One rule matches nodes with no flags set, the other rule matches nodes with two flags set (thus also matching nodes with three or more flags set).

C.3.5 Identifier/Keyset



(a) Identity rule



(b) Keyset rule

Figure C.5: Example identity and keyset rule graphs.

The identity/keyset rules are used both to store the identity and keyset properties and to enforce these constraints. The identity and keyset rules are mostly similar. Both rules check for a given pair of class nodes whether or not a set of fields have the same values. Fields are considered to have the same value if

- for fields with a single value, both values are equal (which translates to using the same node);
- for fields with multiple unique and unordered values, both instances have the same amount of values for that field and for each value in one instance there is an identical value in the other instance;
- for fields with multiple ordered values, both instance have the same amount of values for that field, and each pair of subsequent values is identical.

For the same reasons that uniqueness rules do not check recursive containers, identity and keyset rules do not check these either (additionally, Ecore does not have this notion and is the only language currently supporting these constraints).

Appendix D Graph languages overview

This appendix provides a short overview of each graph language that has been considered in the preliminary research (except those described in section 2.1), giving a short description about its intended purpose, capabilities and available tools. At the end of this appendix a series of criteria is given that has been used to select the languages for importing/exporting.

D.1 XMI

XMI is intended to be an exchange language for metadata using XML (See also [37]). It is commonly used to represent UML models, although other meta models can be used as well (see section 2.1.1). XMI in itself is not a model/meta model language, but is an exchange format for meta models that can be expressed in Meta-Object Facility (MOF). MOF is designed to be a standard for model driven engineering, and is defined by the OMG.

As XMI is just an exchange format, it is not suitable to be imported/exported directly. Rather, a language that makes use of XMI would be suitable for importing/exporting. An example of this is the Ecore language, as well as various UML diagrams (see appendix D.2).

D.2 UML

UML is a general purpose modeling language. It consists of a set of graphical notations that are used for modeling object-oriented software. It defines a multitude of diagrams depicting various aspects of object oriented software. One of the ways to encode it is using XMI, using the MOF.

UML captures virtually all modeling concepts, which means that it supports all the Ecore model concepts. The downside is that it makes the UML/XMI combination fairly complex, as is discussed in [37]. However, a subset of UML may be supported, for example only the class diagrams and just the parts of it that encompass the Ecore model.

Many editors for UML exist, each with their own way of encoding the data, though XMI can be used. However, as previous work has shown this is not always compatible. Examples are Borland Together [5] and Microsoft Visio [28].

D.3 GML/XGMML

GML (Graph Modeling Language) and the derived XGMML (eXtensible Graph Markup and Modeling Language) are both languages for modelling graphs. GML is text based and XGMML is XML based. XGMML allows for arbitrary attribute data, such as RDF descriptions (see also appendix D.5).

GML is intended as a common file format for storing graphs. Its main features are platform independence, ease of implementation, support for arbitrary data structures and flexibility. It is represented in ASCII, which is easily handled by various standard routines (although limiting the possible characters used). Although GML can be used to represent graphs (supporting graphs, nodes and edges), it can also be used to represent other data, such as structures [21].

XGMML is based on GML. XGMML is intended to be a format to exchange graphs between different graph authoring tools. It can be combined with other markup languages for adding additional data to graphs, nodes or edges [31]. XGMML supports graphs, subgraphs, nodes and edges and hypergraphs. Additionally, attributes can store extra data for elements, possibly using other markup languages.

Neither GML nor XGMML provide a model syntax, but rather a direct graph syntax (although strictly speaking, GML could be used with a custom model language). This can be used to specify models such as in Ecore, but would not be interoperable with other tools. Alternatively, a graph could be specified with the same structure as an Ecore model, with Ecore meta data as attributes.

GML can be edited using yEd [40], although for instance UML entities do not retain much of their properties. Cytoscape [13] is a tool that can store its data in XGMML. It was specifically designed for working with molecular data structures, but can now be used for analysis and visualization of complex networks.

D.4 GraphML

GraphML is an XML based language, resembling GXL. It is intended to describe the structural properties of a graph, with an extension mechanism for adding application-specific data [8]. It supports subgraphs, hypergraphs and references to external data. In addition, it supports locator tags for external references, and ports for nodes (which can be used as an edge's source or target).

GraphML is the main format of the yEd editor [40]. It can be used to store certain UML model information, such as classes and simple relations. The structure of GraphML is very similar to GXL. In fact, it can be translated to and from GXL, as shown in [9], although there are slight differences. From a conceptual point of view this makes GraphML a less interesting target for importing when GXL is already supported, although extending an importer to GraphML should be fairly trivial.

D.5 RDF/OWL

D.5.1 RDF

RDF (Resource Description Framework) is used to represent information on the Web, in particular metadata of web resources. It is intended to be machine readable, rather than human readable. It allows machine-understandable information to be exchanged without loss of meaning.

The model of RDF is based on triples, consisting of a subject, a predicate and an object, which define a statement about a resource. A predicate defines a relation between the subject and the object. RDF does not define any data types by itself, but uses the standard XML schema 2 as a recommendation. A description of the RDF metamodel can be found in [11].

RDF can be edited with the IsaViz RDF editor [38] which is developed by the W3C and can read and write RDF/XML (XML serialized RDF).

D.5.2 OWL

OWL (Web Ontology Language) is an ontology language for the semantic web, based on Description Logics. An OWL ontology allows a reasoner to infer information from a given set of axioms about the world the given ontology describes. It is a based on the open world assumption, where the inability to derive a fact does not mean that it is false (closed world assumption) but rather unknown. OWL defines three sublanguages:

- OWL Lite: OWL Lite is intended as a simple sublanguage, for basic classification and constraints.
- OWL DL: OWL DL is a computational complete and decidable sublanguage (that is, all conclusions can be derived and can be computed in finite time) with maximum expressiveness within these constraints. OWL DL supports all the OWL language constructs, although with a few extra restrictions.
- OWL Full: OWL Full is the full OWL language which allows the most expressiveness. It does not give any computational guarantees, but allows any RDF vocabulary to be described.

A description of the OWL metamodel can be found in [11].

OWL is very closely related to RDF(S). RDF(S) is used to define the schema of a model. It allows to specify classes and relationships, and makes use of typing and subtyping. OWL adds a semantic layer on top of RDF(S). It provides functionality to add meaning to the schema. For example, it allows to make statements about the equivalency of classes, or allow facts to be derived from other facts. Multiple versions of OWL exist, most notably versions 1.0, 1.1 and 2.0.

An editor for OWL ontologies is Protégé, which is developed by the Stanford Center for Biomedical Informatics Research [36]. It can be used to define ontologies, as well as invoke a reasoner and exporting to various formats, including the (common) RDF based syntax.

D.6 KM3 (Kernel MetaMetaModel)

KM3 is a language to write metamodels [3]. It has a simple, straightforward syntax which can be serialized into XMI. It is based on Ecore and it can be transformed from and to Ecore models. It does not define a syntax for model instances, although KM3 metamodels are themselves instances of the metamodel of KM3 itself.

As it is similar to (and based on) Ecore, it supports the same concepts, such as packages, classes, attributes, references and primitive data types. It is therefore not very interesting to support the KM3 language in an importer that can also handle Ecore files, as they are conceptually the same.

D.7 Kermeta (structural)

Kermeta is a modeling language that is based on the EMOF specification [29]. Its architecture is divided in a structural and behavioral package. For this research, the structural aspect is the most interesting, as the behavior architecture is used to describe executability, which is not relevant for type and instance models. The structural package can be split in a named view for named elements and a typed view [24]. It can be used as an object-oriented, a model-oriented, and an aspect-oriented language. Kermeta is designed to add executability to object-oriented model languages.

Like KM3, Kermeta is a way of textually representing Ecore models, and as such does not show different kinds of concepts that are relevant. The main exception is the addition of the concept of executability, but this does not affect model instances and as such is not required for an importer in GROOVE, just like operations in Ecore (see [37]).

Kermeta can be edited in Eclipse using a plugin, which allows it to be converted from/to Ecore models. As with KM3, this makes Kermeta a less interesting target for importing.

D.8 Selection criteria

From the languages described above, a selection has to be made. For this, a set of criteria is defined. These criteria are:

- Common usage: If a language is commonly used, it is more worthwhile to add it to the importer/exporter. Languages that are not used much would not add much to the interoperability of GROOVE with other tools, since not many tools support the language in the first place. If a language is not commonly used, it will not be included.
- Existence of a self describing meta model (meta schema): If a language supports a meta schema, it will be easier to reason about the meaning of a structure in that language, making importing and exporting that language easier. This criterion provides more of a suggestion, it does not provide a definitive choice for inclusion of the language. Such a meta schema is usually described in the modeling language itself, providing the top level in a modeling hierarchy (as the meta model that describes the meta schema would be the meta schema itself). Examples of this are the MOF and the GXL meta schema.
- Concrete notation: A language that provides its own fully concrete notation allows the scope of the language to be bounded. Those that do not do this would be much harder to support, as the same concept can be expressed in many different ways. Rather, more concrete sublanguages are more suitable. For example, XMI allows a very broad scope of meta models to be expressed in XML, as long as they are based on the MOF. This makes it hard to write an importer/exporter, as it would become too much general-purpose. Ecore, on the other hand, provides a more concrete implementation of XMI, which makes implementing it much more manageable. Therefore, if a language presents itself as a 'carrier language', it will not be included (the sublanguages provide a more suitable target).
- Similarity to other languages: In some cases, languages are very similar and might only differ in syntax, but not in expressiveness or meaning. An example of this is the KM3 language, which is based on Ecore and defines a textual format with the same meaning. In some cases, converters for these languages exists that can translate from one language to the other without any loss of meaning. Including such a language in the importer/exporter would not add any new insights. Therefore, if a language is similar enough to another language, it will not be included.