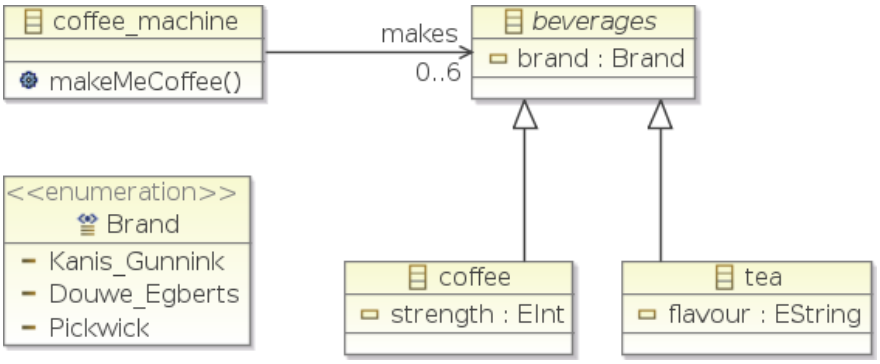
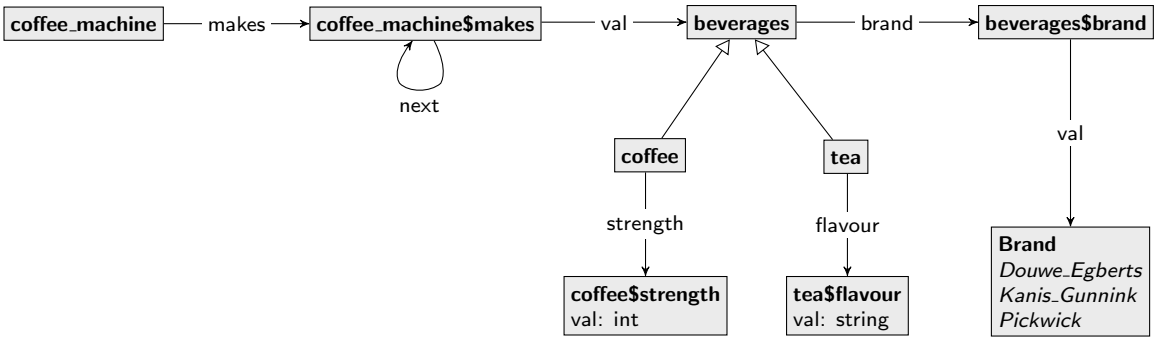


Connecting GROOVE to the world using XMI

Master thesis

Author:
Stefan Teijgeler

Supervisors:
Dr. Maarten de Mol
Dr. Ir. Arend Rensink
Dr. Ivan Kurtev



August 25, 2010

Preface

It has been a long road of almost 10 years to get to the point of graduation. The road was a tough one, and there have been many obstacles along the way. It has not been easy with, for example, the major changes to the study program, my lack of motivation after Inter-*Actief* and my unsuccessful first graduation project. However, I did get here, and I am proud to present my graduation thesis that now lies in front of you.

First of all I would like to thank my parents. These last few years have been a real trial for you, probably more so than for myself, the exception perhaps being the last few months. You have always supported me, even though I have not always given you reason to continue doing so. Thanks for everything, your ordeal is finally over!

Żabka, thanks for taking care of me! I would haved lived even more unhealthy, especially the last months, if it weren't for you. I know you would have liked to help me more, but that I have been keeping my work to myself. Still, thanks for the gestures. Now that I finally have some spare time again, it will be my turn for the long journey by train.

Next I would like to thank my supervisors. Maarten, for reading most of my written work and your constructive feedback. Arend, for your quick replies and fixes to GROOVE. Ivan, for your help on Ecore modelling and getting started with the EMF API. And all of you, thanks for working on such a tight schedule for me these last few weeks and the weeks before your holidays.

Finally, thanks to all of my friends. I have not been fun to be around with lately and I have not spent enough time with you. Now that I am finally graduating and this extremely busy period is over, it is time to make up for lost times. Prepare to go out for movies, games and drinks a lot!

The next step will be hunting for a job, but first a week or two of well deserved rest! That's all I have to say here. Happy reading!

Contents

Summary

Model driven engineering has become an important part of software design, allowing engineers to design more complex systems on a higher, abstract level with a visual notation. Model transformation is an important aspect of model driven engineering. Model-to-text transformations are used to generate code from a model, model-to-model transformations are used for model refactoring or refining, or combining parts of a system. Many approaches exist for modelling, like the standards set by MOF or EMF, and many approaches exist for model transformation. Some use a standard like QVT to define transformations, many use a custom approach that is often inspired by graph transformation.

GROOVE is a graph transformation, simulation and verification tool. In this work we made it possible to use GROOVE as a model transformation tool for Ecore models. Ecore models are imported as type graphs and instances of Ecore models as instance graphs. Instance graphs can be transformed to other instance graphs within GROOVE, and resulting instance graphs can be exported back to Ecore instance models. XMI was used as a standard to perform model exchange. Online experiences indicate that model exchange is not always possible between tools. We performed an extensive XMI compatibility research of various modelling tools to determine a set of tools for GROOVE to be compatible with. This resulted in our findings that model exchange between tools is rarely possible without errors, and that the exchange of Ecore models led to the best results.

In order to import Ecore models and instance models into GROOVE, we defined a mapping from elements of Ecore modelling to a graph representation. Other work has been done to represent UML or Ecore models as graphs, but approaches are often not complete or are not targetted to a specific graph transformation tool. We discussed all features of the Ecore metamodel, and our mapping to graphs for GROOVE supports all available elements that are relevant for instance models. We implemented a Java program as a package of the GROOVE project that can transform Ecore models and instance models to graph grammars for GROOVE with a type graph and instance graphs. It can also transform instance graphs in a graph grammar back to Ecore instance models, making GROOVE a model transformation tool for Ecore models with a formal basis in graph theory.

Chapter 1

Introduction

Model driven engineering

Model driven engineering has become an important part of software engineering with software systems that have become more and more complex. Models that visually describe the design of a system have allowed engineers to design such software systems on an abstract and intuitive level. One major player in model driven engineering is Object Management Group (OMG), providing standards for Model Driven Architecture (MDA) [29], Meta Object Facility (MOF) [28] and the Unified Modelling Language (UML) [33], which is defined by MOF. Another large player is the community driven Eclipse Foundation. They provide the Eclipse Modelling Framework [42] which consists of the metamodelling standard using Ecore, but also provide tools as part of the Eclipse Framework.

Model transformation is an important aspect of model driven engineering. Model-to-text transformations transform a model to a textual representation, usually program code. Model-to-model transformations can be used to define changes to a design on the abstract level, for example for program refactoring or refining, or combining parts of a system. There are several approaches for model transformation. OMG has proposed a standard for model transformations on MOF models, called Query / View / Transformation (QVT) [32], which is used by some tools [19, 23]. Other tools use a custom approach to specify model transformations [37, 8, 45, 24, 27, 44, 2, 20, 16], many of which are inspired by graph transformation concepts.

Graphs and GROOVE

Graphs are data structures that consist of a collection of objects and a collection of relations between these objects. Objects in graphs are called *nodes*, relations in graphs are called *edges*. Graphs are often used to model object oriented systems. Graph transformation tools can then perform graph transformations and verification of properties of a system by exploring the transition system.

Models that describe software systems, like UML class diagrams or Ecore models, have a strong visual similarity with graphs, consisting basically of boxes and relations with added features. Modelling standards, at least in the case of UML and Ecore, are defined textually without formal semantics. Graphs and graph transformations on the other hand have a strong formal basis, of which an extensive overview can be found in [14]. Work has been done to formalize visual modelling languages like UML in terms of graphs [5, 4, 21, 22, 35], but usually these approaches are incomplete or not targeted to the capabilities of a specific graph transformation tool. Without tool support, mappings of models to graphs remain an on-paper idea without practical applications.

GROOVE (GRaph-based Object Oriented VErification) is a tool set for modelling object oriented systems as graphs [36, 17]. It is being developed at the formal methods group of the University of Twente, with a first release in 2004. With it, users can create graphs and define graph transformation rules. GROOVE can then apply the rules in a predefined or arbitrary order, and explore the resulting labelled transition system. Properties of each state can be verified, systems can be simulated, and graphs can be transformed into other graphs which is essentially model transformation.

Interoperability of GROOVE

Even though there is a visual similarity of UML or Ecore models with graphs, there is no interoperability of GROOVE with modelling tools for these models. If GROOVE can be connected to UML or Ecore modelling tools and handle models created in those tools, GROOVE can be used a model transformation tool with verification and simulation features.

However, UML or Ecore models cannot be imported into GROOVE or exported from GROOVE, and these models cannot be represented as simple graphs in a one to one mapping of modelling elements to nodes and relations to edges. In order to extent GROOVE with interoperability with modelling tools, it must be possible to exchange models with these tools and to represent their models as graphs.

The goal of our work is to add interoperability with UML or Ecore modelling tools to GROOVE. In order to do this, it must be possible to import and export such models, and represent them as graphs. Mappings of UML or Ecore models to graph representations exist, but only partial mappings. Furthermore, these mappings are not targetted to the feature set of GROOVE in particular. Our goal is to support all features of a modelling standard in a graph representation of which everything must be supported by GROOVE.

Connecting GROOVE to the world

In order to add interoperability to GROOVE, it must be possible to exchange UML or Ecore models between modelling tools and GROOVE. We use XML Metadata Interchange (XMI) [30], which is an OMG standard for model exchange. It provides means to exchange any model information between tools,

like UML and Ecore models, and is supported by many modelling and model transformation tools [46, 45, 38, 27, 18, 42, 26, 25, 43, 34, 3, 15, 9]. There is a difference between models and diagrams: Models contain information about elements but no visual representation, diagram are visual representations of models. XMI only supports exchange of models. Diagrams have to be regenerated after imported a model using XMI.

With a compatibility experiment we identify which tools are compatible to each other with regard to model exchangeability. Even though many tools support XMI, there has not been any published work with an experiment to determine model exchangeability using XMI. Based on our experiment, we chose the modelling standard of Ecore to connect GROOVE.

The next step for adding interoperability to GROOVE is to define a complete mapping of Ecore models and instances of those models to graph representations that are supported by GROOVE. Ecore models are mapped to a type graph representation and instance models are mapped to an instance graph representation. For our mapping of Ecore models to graphs we use several principles as guidelines, similar to the ones in [21]. Instance graphs should be as simple and consistent as possible, but they should still support all features of Ecore models. If some property of a model element requires a larger subgraph to represent it, this larger subgraph is also used if the property is not enabled to keep a consistent representation. Type graphs should be as close as possible to instance graphs and the Ecore model it represents, instead of using counterintuitive type graphs.

To make our work applicable in practice, the mapping of Ecore models to graph representations must be used by a transformation tool. This tool should be able to take an Ecore model and instances of that model, and transform them to a type graph and an instance graph for each instance model. It should also be able to take a graph grammar from GROOVE and transform all valid instance graphs of that grammar to instance models.

Results

We compared the UML and Ecore model exchangeability using XMI of of various tools. The best results are achieved when exchanging Ecore models, therefore we chose to add interoperability with Ecore models to GROOVE.

We examined all elements that are part of the Ecore metamodel and can be used in Ecore models. All modelling elements are either supported and are represented in a graph, or have no impact on instance models and need no representation. Advanced features often not supported by other approaches, like multi-valued attributes and ordering of references, are supported by our approach.

A transformation tool has been developed as a package of the GROOVE project. It can transform Ecore instance models in both directions, from Ecore to GROOVE, and back to Ecore. With our mapping and transformation tool, we have added interoperability with Ecore modelling tools to GROOVE.

Structure of the thesis

The remainder of this thesis is structured as follows: chapter 2 contains our XMI compatibility research, and provides arguments for our choice to connect GROOVE to the world of Ecore. Then, in chapter 3 we introduce GROOVE and the features that are relevant for our work. In chapter 4 we present a thorough analysis of the Ecore metamodel and a complete mapping of all modelling elements to a graph representation. Example Ecore models are used to illustrate the mapping in actual models. Chapter 5 explains the usage and implementation of the Ecore to GROOVE transformation tool, provides an example of how to use GROOVE as a model transformation tool, and describes some features and strengths of using GROOVE. Finally, in chapter 6 we discuss related work and in chapter 7 we present our conclusions and discuss possible future work.

Chapter 2

XMI compatibility of modelling tools

2.1 Introduction

Many modelling tools support importing and exporting of models using XMI. XMI has been developed by OMG as a standard to enable exchangeability of models between tools. However, performance issues of rebuilding the original model when importing using XMI have been reported [27], and online experiences indicate that exchanging models is not always possible between tools. It seems that sometimes models cannot be imported at all, and in other cases they are imported but with errors. A research towards the compatibility of various tools for model exchangeability using XMI has not been performed or published.

In this thesis we extend GROOVE to enable importing and exporting XMI files. The goal of this chapter is to select a tool or a set of tools that we wish to connect to GROOVE using XMI for exchange of models. We determine which standards are used by various modelling tools and base an expected compatibility between the tools on this. We then present an experiment to verify the compatibility of these tools, where we will see that compatibility is not what it should have been in theory, confirming experiences found online.

In this work we use a difference between the notions of models and diagrams. Models contain the information of for example classes, relations and attributes but do not contain a visual representation. The diagrams in turn are visual representations of models or part of models. This difference needs to be made explicit, since XMI is meant to exchange only model information between tools, not the visual representation. UML and Ecore are, amongst other things, meant to visualize designs to improve the readability, making it it easier to understand and communicate about a design. Since diagrams are the visual representation of models, they are a crucial part of the modelling process. Diagrams can usually be regenerated in tools after importing a model, but larger models may lose a lot of visual expressiveness without the original diagrams. OMG presented a

standard to retain diagram information in XMI files, called UML 2.0 Diagram Interchange. Only one tool supported this, so we could not test the compatibility of the implementations of this standard between the tools. Ecore supports diagrams as separate XML files which refer to the associated Ecore models, so these are more easily exchangeable between tools.

In section 2.2 we show 9 different modelling tools that we considered, some are open source and free to use, some are licensed for which we used evaluation editions. In section 2.3 we explain and present the experiment we did, and finally section 2.4 contains the conclusions of this experiment and our choice which set of tools we wish to be compatible with.

2.2 Researched tools

As mentioned before, we selected 9 tools to be included in our experiment. This selection contains various commercial and licensed tools and some open source tools. They support different modelling standards and export and import formats. Below we list each of the selected tools and their supported features and standards, and the editions we used for our experiment. The standards supported by the different tools are usually found online, but at times the exact version supported is not mentioned. In these cases we used our own findings by studying the tool, and the native and exported file contents. A detailed study whether tools are fully compliant to the claimed standards is outside the scope of this thesis. Whether or not the standards are correctly implemented will show from the compatibility experiment.

Rational Software Architect. Rational Software Architect [18], part of Rational Rose, has been bought by IBM in 2004. It is a tool to model software architectures using UML 2.1. The tool has been redeveloped by IBM and it is now based on the Eclipse framework. It can be used either as a plugin to an existing Eclipse installation, or standalone. The tool supports importing and exporting of UML 2.1 models using XMI 2.1, and importing and exporting from and to Ecore models. The analysis of this tool has been done using the trial version of version 7.5.4 of Rational Software Architect Standard Edition.

Eclipse with EMF. The Eclipse Project is a pluggable development environment that comprises tools and runtime environments to model, develop and maintain software systems. The Eclipse Platform offers frameworks and services for plugins, which in turn determine how the Eclipse Platform should handle resources like Java files or UML models. Any resource can be handled by Eclipse if there is a plugin available for it. The Eclipse Projects were created by IBM in 2001. Since 2004 the project is hosted by the Eclipse Foundation, an independent non profit corporation, to allow Eclips to be further developed by a neutral, transparent and open community, for everyone to use. The Eclipse community now consists of individuals and companies contributing to the projects. The Eclipse Foundation employs a staff to provide services to the community, but does not employ any Eclipse developers.

In this thesis we use the Eclipse Platform Galileo 3.5.1 with the Eclipse Modelling Framework (EMF) 2.5.0 plugin [42], as well as the Ecore Tools plugin 0.9.0. The Eclipse modelling Framework makes Eclipse understand EMF models, and can for example connect Ecore models to Java generators. The Ecore Tools allow visually developing Ecore models and instance models. Ecore models are natively stored using XMI 2.0.

Modelio. Modelio is a tool to model software architectures [26]. It is being developed by Modeliosoft, a company located in France. The tool supports full UML 2.1 modelling, and supports features like code generation, consistancy checks, and traceability management. Modelio is based on Objecteering, a tool developed since 1991 to support object oriented programming and modelling. Objecteering has now been discontinued, and development by Modeliosoft is now fully dedicated to Modelio. There are three versions of Modelio available, the Free, Express and Enterprise Editions. Each of the editons supports importing of UML 2.1 models using XMI 2.1, but only the Enterprise Edition also supports exporting models using XMI 2.1. Because we need both XMI import and XMI export features, we use an evaluation edition of Modelio Enterprise Edition version 1.1.0.

Visio. Visio is part of the Office Suite developed by Microsoft [25]. It is a tool to create diagrams like UML diagrams. It is not intended to support developing software architectures. There is no seperation between the created diagrams and the underlying models. The goal of Visio is not to support the development cycles of software development, but only to visualize, explore and communicate complex data. A XMI export feature has been available for Visio 2007 since Service Pack 2 using Visual Basic code. For this work we used the latest version of Visio at the time, Visio 2007 Professional Edition with Service Pack 2. It supports diagrams using UML 1.4 and UML 2.0, and exporting using an unspecified version of XMI. No XMI import feature seems available for any edition of Visio.

ArgoUML. ArgoUML is a free and open source UML modelling tool that was first publicly released in 2002 [43]. It is not funded and is developed by volunteers, users of the tool are encouraged to contribute in some way. The latest released version is 0.28.1 and supports modelling using UML 1.4, importing of UML 1.3 and 1.4 models using XMI 1.0, 1.1 and 1.2, and exporting of UML 1.4 models using XMI 1.2. UML 2 and XMI 2 is not yet supported in the latest release. Preliminary support for UML 2.0 is being added to 0.29.1, but at the time of writing this was not yet ready for testing. ArgoUML also has support for code generation, OCL, exporting to image formats and multiple views, or diagrams, for a single model. For our work, we use the latest released version of ArgoUML, 0.28.1.

BoUML. BoUML is a free and open source tool for creating UML 2.1 models [34]. It is being developed by Bruno Pagès, and funded by voluntary donations

from users. It is multiplatform and runs under Windows, Linux and MacOS by using Qt. The tool is extensible in nature, and functionality can be extended by so called plug-outs. BoUML supports a wide range of standards and features using plug-outs, like code generation, reverse engineering and importing of UML models using XMI 2.0 and XMI 2.1, importing of Rational Rose projects, and exporting of UML models using XMI 1.2, XMI 2.0 and XMI 2.1. Only importing from XMI 1.2 is not possible. For our work we use the Windows build of BoUML 4.17.1, released in January 2010.

Artisan Studio. Artisan Studio is a development tool that allows users to model systems and software using industry standards like UML [3]. Artisan Studio is one of the software suites being developed by Artisan, founded in 1997 and located in the USA and the UK. It provides support for all phases of a development lifecycle, and provides consistency checks and traceability between models, code, documentation and test cases. Artisan Studio supports modelling using UML 2.1, and importing and exporting of UML models using XMI 2.1. We use the evaluation version of Artisan Studio 7.1 for our work.

Poseidon. Poseidon for UML is a UML modelling tool created by Gentleware AG [15]. Gentleware was founded in 2000 in Hamburg. They have a basis in the open source project ArgoUML described above, and offer free licenses to non commercial open source projects. The latest version, Poseidon for UML 8 beta, is closely tied together with Eclipse and is built on EMF, but not yet available for evaulating. There are several editions available of the latest release of Poseidon for UML, as well as an Eclipse plugin, Apolle for Eclipse, to create UML class models from Java code and to generate Java code from models. Poseidon for UML comes in four editions, the community, standard, professional and embedded edition. Each edition can be evaluated, and the community edition is freely available for open source projects. It however has limited import and export capabilities. For this work we use the evaluation edition of Poseidon for UML Professional 6.02. It supports modelling using an unspecified version of UML, which after closer inspection turns out to be UML 1.4 with some features from UML 2.0 added to it. Terms and elements from UML 1.4 and 2.0 are mixed together, for example Poseidon supports 8 diagrams including the collaboration diagram as defined in UML 1.4, activity diagram elements like the activity partition from UML 2.0, and supports the UML 2.0 Diagram Interchange standard which is also part of UML 2.0. Poseidon can import from Rational Rose projects and XMI 1.2 files, and export to Ecore and XMI 1.2. Poseidon for UML is also the only tool encountered that supports the UML 2.0 Diagram Interchange standard, which is surprising since this standard is actually defined for use in conjunction with UML 2.0 and XMI 2.0.

Borland Together. Borland Together is a visual modelling platform developed by Borland [9]. It is based heavily on the Eclipse platform, and is in fact a set of plugins for use in Eclipse. It allows modelling using UML 1.4 and UML 2.0, and supports model transformation using OCL and QVT, code generation, and compliance to other standards. Importing and exporting of models is possible using XMI 2.0 and XMI 2.1 for UML 2.0 and UML 2.1 models, and possible

using XMI 1.2 for UML 1.4 models. Borland Together can also import projects created in Rational Rose. Furthermore, Ecore models can be transformed to UML 2.0 models and vice versa using QVT model transformation. Borland provides templates for model transformation projects, however these will not be looked at in this thesis. The standard used for modelling is UML, however the underlying metamodel used is EMF Ecore. Borland Together also uses and helps to develop Eclipse plugins that are tied together to EMF, like the Graphical modelling Framework (GMF). For our work we use an evaluation of the latest version, which is Borland Together 2008 with Service Pack 1, released in July 2009.

	modelling standards	Import standards	Export standards
Rational	UML 2.1	XMI 2.1, Ecore	XMI 2.1, Ecore
Eclipse	Ecore	Ecore	Ecore
Modelio	UML 2.1	XMI 2.1	XMI 2.1
Visio	UML 1.4, UML 2.1	-	XMI
ArgoUML	UML 1.4	XMI 1.0, XMI 1.1, XMI 1.2	XMI 1.2
BoUML	UML 2.1	XMI 2.0, XMI 2.1	XMI 1.2, XMI 2.0, XMI 2.1
Artisan	UML 2.1	XMI 2.1	XMI 2.1
Poseidon	UML 1.4 (some 2.0 features)	XMI 1.2	XMI 1.2, Ecore
Borland	UML 1.4, UML 2.1	XMI 1.2, XMI 2.0, XMI 2.1, Ecore	XMI 1.2, XMI 2.0, XMI 2.1, Ecore

Table 2.1: Features of UML / Ecore modelling tools

Export \ Import	Rational	Eclipse	Modelio	Visio	ArgoUML	BoUML	Artisan	Poseidon	Borland
Rational									
Eclipse									
Modelio									
Visio									
ArgoUML									
BoUML									
Artisan									
Poseidon									
Borland									

Table 2.2: Expected export / import compatibility between tools. White cells indicate an expected incompatibility, grey cells indicate an expected compatibility, dark grey cells were not examined.

Table 2.1 summarizes the standards that are supported by the tools. For each tool the modelling standards that are used within the tool are listed, as well as the standards that can be used for exporting and importing models. The Ecore export and import standard is actually XMI 2.0, but it contains an Ecore model instead of a UML model. To make this more clear in the table, we named it Ecore instead of XMI 2.0.

We attempted to predict the exchangeability of models between tools by looking at the supported standards. Exchange of models from tool A to tool B in the case of UML models should be possible when two constraints are satisfied. First, tool A must use the same standard for exporting that B uses for importing. Secondly, both tools A and B must use the same standard for modelling. This is because tool B must know the model type and version of the model it wants to import. This is the case when both tools use the same standard for modelling. Ecore export and import formats are treated a little differently, we predict that whenever tool A can export using Ecore, and tool B can import using Ecore, it should be possible to exchange models from A to B, regardless of the used modelling standards of both tools. This is because whenever a tool supports exporting or importing to or from Ecore it transforms a UML model to or from an Ecore model. The used modelling standard by the tool thus becomes irrelevant. The exception of course is Eclipse which uses Ecore as the modelling standard. Table 2.2 contains our predictions. Vertically the tools that export a model are listed, horizontally the tools that import a model. The grey cells indicate that it should be possible to exchange a model from the vertical listed tool to the horizontal listed tool. It is clear that this table is not transitive, some tools do not support the same export and import standards, and Visio cannot import anything at all. In section 2.3 we extend this table with our actual findings.

2.3 Actual exchangeability of models

We performed an experiment to determine the exchangeability of models between tools, and to confirm the negative reports of XMI as the standard for model exchange. We created a simple coffee machine model (class diagram in UML terms or Ecore model in Ecore terms) in each tool. Figure 2.1 shows the model we created in Eclipse, but we created the same model in each tool. The model contains basic classes, an abstract class, inheritance, operations, attributes, an enumeration, composition relations and an association relation. In our opinion it is a fair set of the possible elements, and we can determine the exchangeability between tools based on this model to a reasonable extent. At the very least we can state that if this model cannot be exchanged, more complicated models can also not be exchanged.

The next step was to export the model in each tool using each version of XMI and Ecore the tool supports. Some tools offer various options and standards for exporting, so we ended up with at least one XMI or Ecore file for each tool we looked at. We then went through the tools in order, and attempted to import the exported files from each of the other tools in a new and empty project. Our results are ordered by the importing tool, so whether or not models can

be exchanged from tool A and B can be found at tool B in the sections below. Table 2.3 contains a summary of the results. As before, the grey cells indicate that we expect the tools to be compatible, but this time a check mark indicates that models can actually be exchanged. A cross indicates that the tools are not compatible and that we received some error after attempting to import. A question mark means we were unable to determine compatibility. Footnotes in the table provide some additional information about the compatibilities we found, like the used standards and whether models lose information after importing.

Export \ Import	Rational	Eclipse	Modelio	Visio	ArgoUML	BoUML	Artisan	Poseidon	Borland
Rational		✓[1]	✓[3]	✗	✗	✗	✓[35]	✗	✗
Eclipse	✓[1]		✗	✗	✗	✗	✗	✗	?
Modelio	✓[3]	✗		✗	✗	✓[34]	✓[35]	✗	✗
Visio	?	✗	?		?	?	?	?	?
ArgoUML	✗	✗	✗	✗		✗	✗	✓[24]	✓[25]
BoUML	✓[3]	✗	✗	✗	✗		✓[34]	✗	✓[25]
Artisan	✓[3]	✗	✗	✗	✗	✗		✗	✗
Poseidon	✓[146]	✓[16]	✗	✗	✗	✗	✗		✓[25]
Borland	✓[34]	?	✓[34]	✗	✓[24]	✗	✓[34]	✓[25]	

Table 2.3: Actual export / import compatability between tools. 1) Ecore export. 2) XMI 1.x export. 3) XMI 2.x export. 4) Some model information is lost. 5) Some model information is changed. 6) Only seperate packages instead of entire models.

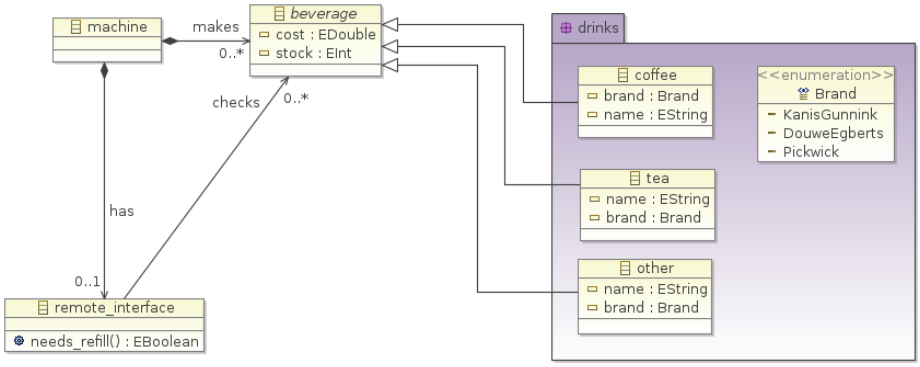


Figure 2.1: Simple model of a coffee machine that contains basic classes, an abstract class, inheritance, operations, attributes, an enumeration, composition relations and an association relation.

Rational Software Architect. Rational Software Architect can import models that were exported in Eclipse, Modelio, BoUML, Artisan, Poseidon and Borland Together. UML 2 models exported in Borland can be imported, however only models that were exported using the option XMI for UML 2.1, and labels on associations are missing in this case. UML 2 models exported using any other option cannot be imported, including the option to create XMI for UML 2.0 compliant to the OMG standard. This gives the impression that only XMI files that are not compliant to the OMG specification can be used to import models from Borland Together. It is not clear what the difference is between XMI files that are OMG compliant and XMI files that are not OMG compliant. Models from Poseidon can be imported by using the Ecore file format. However, much information is lost like association relations and operations. Also, Poseidon only supports exporting packages to Ecore, not entire models. So everything not in the exported package is lost, and this even includes subpackages. Entire models can still be exchanged by exporting and importing each package seperately.

Eclipse with EMF. Eclipse with EMF does not have an option to import or export models using XMI. The native storage format for Ecore models in Eclipse is XMI 2.0. However, Eclipse does not understand the XMI file formats that are exported by the other tools, so those files simply cannot be opened as models in Eclipse. This is because they contain UML models and not Ecore models. Rational Software Architect, Poseidon and Borland Together have the possibility to export models to the Ecore file format. The Ecore files created with Rational Software Architect can be opened in Eclipse and all model data is present. Poseidon can only export packages as Ecore, and subpackages are not included. So model data can be exchanged from Poseidon to Eclipse, but only partially. The model transformation that Borland Together offers to obtain Ecore files has not been tested in this work. It only provides a framework to define Ecore to UML transformations in, and it does not work out of the box.

Modelio. Modelio can import models that were exported in Rational Software Architect and Borland Together. XMI files from Borland can only be imported if they are created with the XMI for UML 2.1 option, and again labels on associations are missing. XMI files created in Borland using the other options are again not recognized. Models from BoUML cannot be imported, because of a data type conflict with the predefined data type "integer". Since basic datatypes will usually be used in UML models, we say that BoUML XMI files are not compatible. Interestingly, whenever Modelio cannot import an XMI file, it mentions that the file is not recognized as an Ecore file. This does not make sense, since Modelio should be looking for UML data within the XMI file, not Ecore data. Even when attempting to import an Ecore file created in Eclipse it says that it is not recognized as an Ecore file.

Visio. Visio 2007 SP2 has a VBA command that allows users to export UML models using XMI. However, we could not get it to work within our time frame. Reports online however indicate that the obtained XMI files are not compatible with many UML tools without tempering with the file. Further testing might

show that certain tools can import Visio XMI files, either with or without loss of information of the model, but we cannot be sure until this has been done.

ArgoUML. ArgoUML uses UML 1.4 as a modelling standard, and therefore cannot import any UML 2 models, regardless the version of XMI used. UML 1.4 models created in Borland and exported using the XMI for UML 1.4 options can be imported, but much information is lost, like labels and multiplicities on associations. None of the other models could be imported in ArgoUML. As a special note, BoUML and Poseidon do use XMI 1.2 which is supported by ArgoUML, but since the models themselves are UML 2 or contain some elements of UML 2 they cannot be imported. These results are expected, since ArgoUML is the only tool tested that only supports modelling using UML 1.4 and not UML 2.

BoUML. BoUML can only import XMI files created with Modelio. UML 2 models from Modelio can be imported, however all association relations are lost. Generalization and composition relations are still present in the imported model. BoUML expects XMI 2.1 files, and XMI 2.0 files cannot be imported. However, even XMI files that are XMI 2.1, from Rational Software Architect, Artisan Studio and Borland Together, give errors when attempting to import them.

Artisan Studio. Artisan Studio supports importing of XMI 2.0 files containing UML 2 models. It can import XMI files created with Rational Software Architect, Modelio and BoUML. It must be noted that whenever importing an XMI file into a project, the project must be reloaded for the imported elements to become visible, probably some bug is causing this. Models from XMI 2.1 files can be imported as well even though it expects XMI 2.0 files. However, in all cases problems occur, and models are changed after importing. After importing a model from Modelio, a class changed into an abstract class. After importing from BoUML, relations are in the wrong direction, return types of operations are lost and in some cases the data types of attributes are lost as well. The direction of relations can be solved by choosing to export for Eclipse in BoUML. The imported models from Borland Together seem most correct, and although things like association labels and data types are missing, nothing seems wrongly altered.

Poseidon. Poseidon can import XMI files from ArgoUML and the XMI 1.2 files from the UML 1.4 Borland project. This strengthens our beliefs that Poseidon uses UML 1.4 for modelling, as the only models that can be imported are also UML 1.4 models. Even so, some model information is lost after importing the ArgoUML XMI file in Poseidon. Data types from attributes, return types from operations, and enumeration literals are lost. Model information is also lost when importing the Borland UML 1.4 project, such as data types from attributes. Additionally, the directions of the association relations are reversed.

Borland Together. Borland Together offers the most options for exporting and importing models using XMI. However we have been unsuccessful in importing any XMI 2.0 or XMI 2.1 file into an empty UML 2.1 project. We then attempted to import an XMI file that was exported from Borland itself, which did work. We did manage to import XMI 1.x files from ArgoUML, BoUML using XMI 1.2 and Poseidon in an empty UML 1.4 project. The import from ArgoUML however did lead to some loss and change of model information; the return type of the operation is lost, as well as the generalization relations. The enumerated data type is changed into an empty class called string, and the data type of an attribute brand is changed from the enumeration to an integer. The import from BoUML looked slightly better, the generalization relations were still missing, the association relations are reversed, and the enumeration is changed into a class called enum, with the literals as attributes. Considering that Borland Together does not support enumerations in UML 1.4 models, this last bit seems acceptable. The imported Poseidon model is also changed in several ways, the association relation is missing, the custom data type is gone, all data types of attributes are changed to integer, an operation is added to a class that already has an operation, and the return type of that operation is missing.

2.4 Conclusion

The exchangeability of models between tools is poor at best. In many cases a tool fully fails to import models where it was expected to be possible based on the standards supported, and in many other cases models are only partially imported and some information from the original model is changed or lost. Our expectations turned out to be an overapproximation, nowhere did we find a compatibility that was not expected.

Especially Borland Together has been a disappointment. Of all the tools we looked at it supports most standards. However, compatibility to other tools is very poor, nowhere could models successfully be exchanged to or from Borland, it either was not possible at all or the models were changed in some way. Rational Software Architect on the other hand scored reasonably well, it could import all models it was expected to and in many cases nothing was lost. It is hard to say based on our results whether the lack of compatibility between tools can be contributed to either the exporting or the importing tool. Rational Software Architect is exceptionally good at importing models and Artisan Studio scored reasonable in this regard, Borland Together and Modelio perform better exporting than importing, but there is no line that can be drawn. It seems to be a hit and miss whether a model can be exchanged between two tools, and a compatibility based on the supported standards only provides a chance that it is possible, not a guarantee.

Most problems occurred when attempting to exchange a UML 2.1 model between tools using XMI 2.1. Problems with UML 1.4 and XMI 1.2 or with Ecore were less frequent, although our test set there was a lot smaller. With UML 1.4 and XMI 1.2, only exchanging models from Poseidon to ArgoUML did not work, but we contribute this to Poseidon which uses some parts of UML 2.0. And in the case of Ecore, exchanging models worked everywhere it was expected,

even though not without faults in the case of Poseidon to Rational. We think the problems that we encountered when exchanging UML 2.1 models using XMI 2.1 can be contributed to two things: the implementation of UML in tools, and the general and complex nature of XMI 2.1.

UML tools seem to have different interpretations of the UML 2.1 standard. The UML 2.1 standard is very large and consists of many different diagrams, more are even added each subversion. Different tools support a different amount of diagrams, use different names or terms for the same elements and we expect they interpret the standard of the diagrams as defined by OMG differently. This might not apparent to a user, but it may be a reason for the lack of exchangeability of models between tools.

The other reason, the general and complex nature of XMI 2.1, is an even bigger cause in our opinion. It is a standard that has the goal to allow exchange of *any* type of model between tools, not just UML models. It can be used to exchange any of the UML diagrams, but also anything else. So the type of model that is being exported using XMI 2.1 has to be present in the file as well, this includes a reference to the meta model of the encoded model. Since XMI 2.1 supports *any* meta model, an XMI 2.1 file can contain *any* data. In order to make this general nature possible, the standard is very complex. We think OMG has put the bar too high by attempting to define a standard that can do *anything*, and that this is the main reason why exchangeability of models between tools using XMI 2.1 is as poor as it is, especially in combination with UML 2.1, which in itself is a large and complex standard as well. To determine if there is truth in our claims however, further research into the standards and implementations of them is needed.

The Ecore standard for modelling is a lot simpler than UML, and it only supports models that can be compared to the UML class diagrams. The native file format of Ecore is XMI 2.0. However, since the Ecore standard is simple, the files are also simple in themselves and even humanly comprehensible, much unlike UML 2.1 files exported using XMI 2.1. Because model exchange of Ecore models led to the best results and the simpler XMI files, we choose to connect GROOVE to the world of Ecore.

The diagrams of Ecore models that are created by Eclipse are also in XML format in separate files, although not compliant to any official standard. This makes it possible to use model information as well as diagram information when importing an Ecore model in another tool than Eclipse. We could use model information to structure the graph representing the model, and layout this graph in a similar way as the original diagram of the model. Our work does not include the use of diagrams for the layout of the graphs, but it could be extended to do so.

Chapter 3

GROOVE

3.1 Introduction

GROOVE is a tool set for modelling object oriented systems as graphs, and perform transformations and verifications on those systems [36, 17]. GROOVE is the target graph transformation tool for our mapping of Ecore models to graphs, so graph representation choices must be supported by the current feature set of GROOVE. In this chapter we discuss the features of GROOVE that are relevant for our work. A complete list of features and explanations of them can be found in the GROOVE manual that comes with the tool [17]. We focus on graphs and graph transformations in GROOVE instead of the general definitions of graphs and graph transformations. Therefore we do not give formal definitions of the features we describe, but instead explain how they are used in GROOVE. However, formal definitions can be found in [14, 21] and other work.

In section 3.2 we explain graphs as they are used in GROOVE. In section 3.3 we explain type and instance graphs. In section 3.4 we explain graph transformations rules, and finally in section 3.5 we explain how type graphs, instance graphs and transformation rules are combined into graph grammars to be used in the GROOVE Simulator.

3.2 Graphs

GROOVE uses labelled and directed graphs. Nodes are represented by boxes, and edges by arrows between them. The edges are directed, from a source node to a target node, and are labelled. Self edges are edges with the same source and target node. Labels of these edges are treated as node labels, and we refer to them as such. They can be displayed either as a node label for a node or as a labelled self edge, but they are formally the same.

Node labels. There are two methods to indicate that labels must be node labels: by indicating that labels are types or that labels are flags. A node can be of at most one node type. Node type labels are displayed in bold. Nodes can also have flags, which are used to model properties of nodes. Nodes can have any number of flags. Flag labels are displayed in italics. Figure 3.1 shows a graph with two nodes. One has a node type label *Buffer* and one has a node type label *BufferSlot*. The *BufferSlot* also has a flag labelled *empty*.

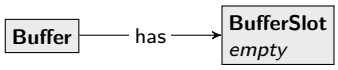
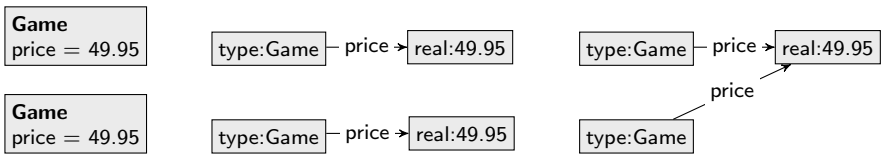


Figure 3.1: Two nodes, one of type *Buffer* and one of type *BufferSlot*. The *BufferSlot* is flagged to be empty.

Attributes. Nodes in graphs have unique identities, based on an internal numerical identifier. This means that even when labelled the same, two distinct nodes in a graph have a different identity. Data nodes, used for values of attributes, are an exception to this. Data nodes represent values that can be either of type string, integer, real or boolean. Data nodes are uniquely identified with their values. This means that if a graph has two data nodes with the same value, they formally represent the exact same node. Data nodes must be labelled with the value and the datatype prefixed to it. Data nodes are displayed as attribute values of other nodes. The label on the edge from the node to the data node represents the name of the attribute. Figure 3.2a shows a graph with two *Game* nodes. Both have an attribute *price* with value *49.95*. Figures 3.2b and 3.2c show the same graph in the editor. Since data nodes are uniquely identified by their value, the two data nodes with the same value are really the same node. Therefore both figures from the graph editor represent the exact same graph.



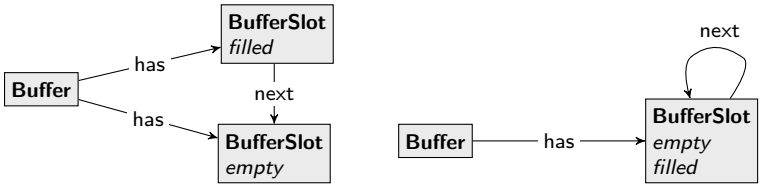
(a) Display view (b) Editor view (c) Editor view

Figure 3.2: Two *Game* nodes with the same price. The three figures are three representations of the same graph.

3.3 Type and instance graphs

Graphs in GROOVE can be typed by a type graph. A type graph defines valid instance graphs. A mapping of the nodes and edges of an instance graph to the nodes and edges of the type graph must exist in the form of a graph morphism [14, 21]. This concept is very similar to that of metamodelling, where a metamodel defines valid instance models. Figure 3.3 shows an instance and a type graph. The *Buffer* node in the instance graph is mapped to the *Buffer*

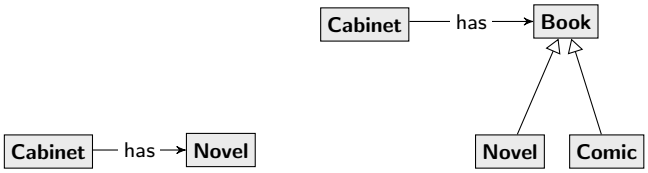
node in the type graph, and both *BufferSlot* nodes in the instance graph are mapped to the *BufferSlot* in the type graph. For the mapping of an edge, the mapping of the source and target node must be the same type graph nodes as the source and target nodes of the mapping of the edge. For the mapping of nodes and edges, the labels must be preserved as well.



(a) Instance graph (b) Type graph

Figure 3.3: An instance graph, typed by a type graph.

Node type inheritance. Nodes in type and typed graphs must always have a type, which means they must have a node type label. GROOVE supports node type inheritance in type graphs. Node types can be defined as subtypes of other node types. Subtypes are displayed with an inheritance edge. For an instance graph to be correctly typed by a type graph with node type inheritance, a clan morphism must exist from the instance graph to the type graph [4]. Subtypes inherit all labels and edges from their supertypes. This concept shows a strong resemblance to class diagrams, where a class can inherit from another class, in which case it inherits the list of attributes and associations of its supertype. In the type graph in Figure 3.4, a *Cabinet* contains *Books* that can either be *Novels* or *Comics*. The *Cabinet* in the instance graph only has one *Novel*, and is correctly typed by the type graph.



(a) Instance graph (b) Type graph

Figure 3.4: An instance graph, typed by a type graph with node inheritance.

Data nodes. Data nodes are used in instance graphs for values of attributes. Data types are used in type graphs to define the data types of attributes. The label of the edge to a data type node is the name of the attribute, as in instance graphs. Data types of attributes are displayed as the attribute name with a colon, and then the data type. Figure 3.5 shows an instance graph and a type graph. A *Game* has a price of data type *real*. In the instance graph the price of the *Game* is *19.95*.

(a) Instance graph (b) Type graph

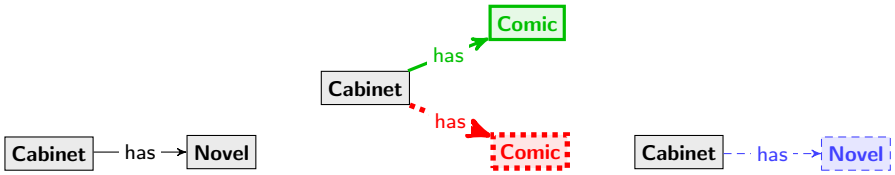
Figure 3.5: An instance graph and a type graph that defines the data type of an attribute.

3.4 Transformation rules

Graph transformation rules consist of a left-hand-side (LHS), a right-hand-side (RHS) and optionally a negative application condition (NAC). GROOVE uses a combined view for the LHS, RHS and NAC that consists of reader, eraser, creator and embargo nodes and edges. When type graphs are enabled in a graph grammar, transformation rules must also be typed. GROOVE uses the SPO (single pushout) approach, which in short means that edges that are left dangling after applying a rule are deleted as well [14].

Reader, eraser, creator and embargo elements. Reader elements are in both the LHS and the RHS or a rule. Reader elements are matched in a graph, but do not modify it. They are displayed as ordinary graph nodes and edges with solid black lines and text. Eraser elements are only in the LHS of a rule. Nodes and edges that are matched by eraser elements are deleted when a rule is applied. They are displayed as thin, dotted blue lines and text. Creator elements are only in the RHS of a rule. When a rule matches, these nodes and edges are created. Creator elements are displayed with thick, solid green lines. Embargo elements are only in the NAC of a rule. Embargo elements are forbidden to exist in a graph for a rule to be applicable. Embargo elements are displayed with thick, dotted red lines and text.

Figure 3.6 shows an instance graph and two transformation rules. The rule in Figure 3.6b matches a *Cabinet* node, and when one does not already exist, adds a *Comic* to the *Cabinet* when it is applied. The rule in Figure 3.6c matches a *Cabinet* with a *Novel*. When a match is found, the *Novel* is deleted from the graph when the rule is applied.



(a) Instance graph (b) New comic (c) Delete novel

Figure 3.6: An instance graph and a type graph that defines the data type of an attribute.

Wildcards. Wildcards are special edge labels that stand for any label. They are denoted with a *?*. An edge with a wildcard will match any edge of which

the source and target node also match. Wildcards can be guarded and named. Guarded wildcards are followed by a comma separated list between square brackets of allowed or disallowed labels for the wildcard to match. A disallowed list of labels must be prefixed with a ^. Named wildcards have a name that serves as a label variable. The name immediately follows the *?*. When a named wildcard is matched in a graph, the label that is matched is bound to the label variable. Different occurrences of wildcards with the same name must all match to the same labels. Named wildcards can also be used on creator edges as long as the variable is bound by another occurrence of a wildcard of this name. The rule in Figure 3.7 shows a rule that uses a bound and named wildcard, named *book*. It can only be bound to *novel* or *comic*. When applied, a new *Book* is created and the matched flag is copied to the new *Book*.

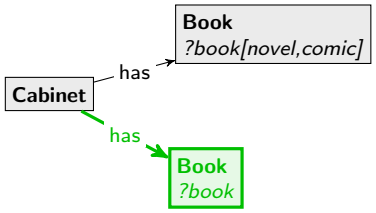


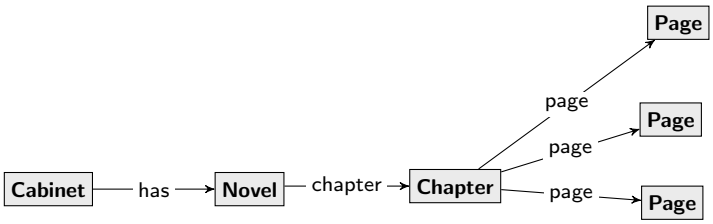
Figure 3.7: A guarded and named wildcard. When the rule is applied, a new flag is created on the new node that is the same as the flag that was matched by the wildcard.

Path expressions. GROOVE supports path expressions on edges in transformation rules. They can only be used on reader and embargo edges, not on eraser or creator edges. Path expressions are matched in a graph by a sequence of edges of which the labels form a valid work of the path expression. Path expressions are built from the operators in Table 3.1. Detailed explanations for all operators can be found in the GROOVE manual [17]. Figure 3.8 shows a rule with a simple path expression that matches a sequential composition of edges labelled *has* and *chapter*, and deletes a *Page* of the matched *Chapter*.

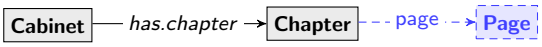
Expression	**Meaning**
label	Simple label
=	Empty path / equality of nodes
?	Wildcard, can be guarded and/or named
$R_1.R_2$	Sequential composition of R_1 and R_2
$R_1\|R_2$	Choice between R_1 and R_2
R^*	Zero or more repetitions of R
R^+	One ore mode repetitions of R
-R	Inversion of R (matches R when followed backwards)
!R	Negation of R (absence of a match for R)

Table 3.1: Operators for path expressions (from the GROOVE manual [17]).

Quantification nodes. Quantification nodes can be used to change sets of subgraphs at the same time, instead of only the direct match of the LHS of



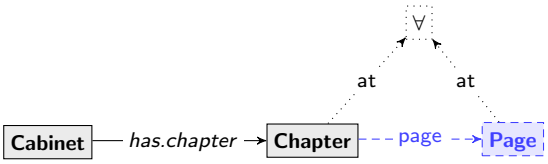
(a) Instance graph



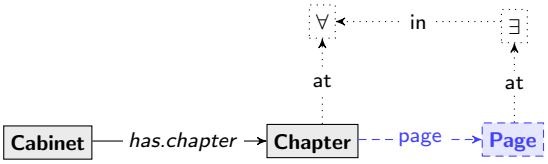
(b) Rule with a path expressions

Figure 3.8: A rule with a path expression that deletes a *Page* from a *Chapter*.

a rule. They are implementations for the universal and existential quantifiers from predicate logic. Quantification nodes can be nested, which is displayed by *in* edges. The top level quantifier must be a universal quantifier, and universal and existential quantifiers must alternate. Every nesting level can contain a sub-rule, every node of the sub-rule must have an *at* edge to the quantification node of its level. Figure 3.9 has two rules that use quantification nodes. The rule in Figure 3.9a matches all *Chapters* and *Pages*, and deletes all *Pages*. The rule in Figure 3.9b also matches all *Chapters*, but for each *Chapters* it matches and deletes only one *Page*. The universal quantifier also matches a graph if there are no *Chapters* to match. More examples and explanation can be found in the GROOVE manual.



(a) Rule to delete all *Pages* from all *Chapters*

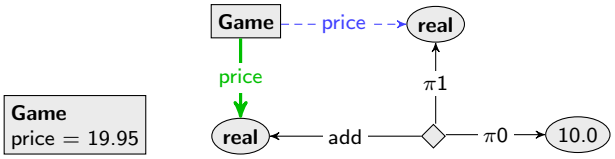


(b) Rule to delete one *Page* from each *Chapter*

Figure 3.9: Rules with quantifier nodes to delete *Pages* from *Chapters*.

Attribute operations. Attribute values can be manipulated with transformation rules by using operations. Operations consist of product nodes, argument edges and operator edges. Data nodes to be used in operations can either be a concrete value, or a data type if the concrete value is not known or has to be calculated. Figure 3.10 shows a rule that increases the price of a *Game* by *10.0*. The diamond shaped node is the production node, which has two ar-

gument edges and an operator edge. The argument edges are displayed as $\pi 0$ and $\pi 1$. The operator edge is displayed with an *add* label. The first argument of the operation is *10.0*, a concrete value of type *real*. The second argument is not a concrete value, but the *real* value of a *price* attribute of a *Game*. When applying the rule, the old value for the attribute is removed and it is set to the value of *10.0* plus the old value. A full list op operations can be found in the GROOVE manual [17].



(a) Instance graph (b) Increase price

Figure 3.10: A rule that increases the price of a *Game* by *10.0*.

3.5 Graph grammars

Using the GROOVE Simulator, users can load graph grammars that contain type graphs, instance graphs and transformation rules. Graph transformation, verification or simulation can then be done by applying the rules manually, by exploring the complete LTS (labelled transition system) or by using control programs to control the order in which to apply the rules. Figure 3.11 contains a screenshot of the GROOVE Simulator. The bottom left panel contains the graphs of the grammar and the top left panel contains the rules. The central panel consists of 5 tabs: the first shows the currently select graph, the second the currently selected rule. The third and fourth show the LTS and the control program, which are features that are not relevant for our work. The fifth tab shows the type graphs. Finally, the right panel shows the labels of the graph that is currently shown in the central panel.

Rule priorization. Graph transformation rules can be prioritized. If rules of a higher priority are applicable, then rules of lower priorities are blocked. They provide a simple method to schedule the applicability of rules. Rules are grouped in the Rules panel by their priority, where higher priority rules are placed above lower priority rules. Higher priority rules that do no modify a graph can for example be used to detect error states of a graph transition system.

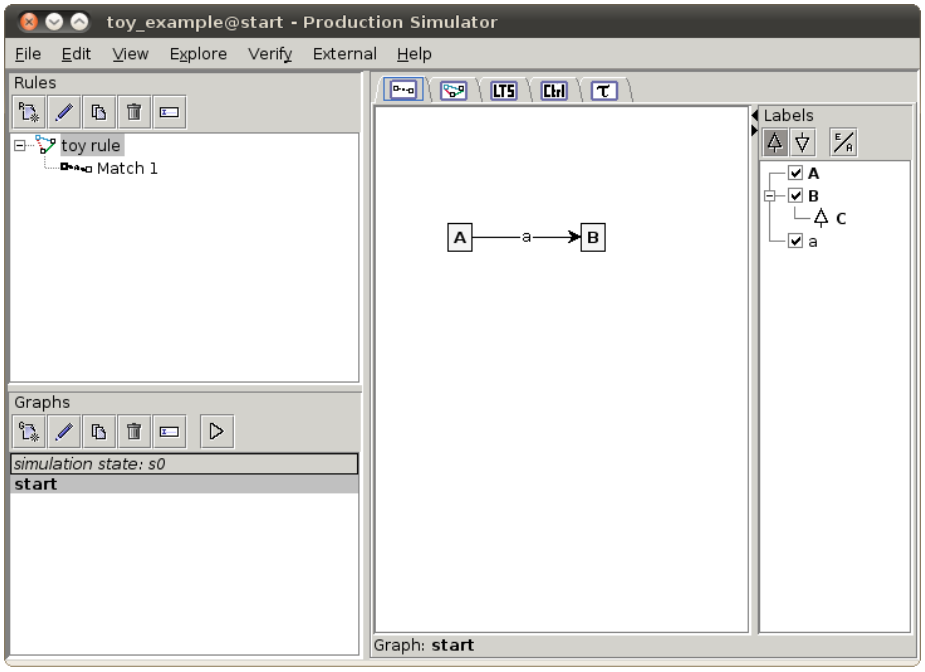


Figure 3.11: Main window for the GROOVE Simulator showing the Rules, Graphs, Central and Labels panels.

Chapter 4

Representing Ecore models as graphs

4.1 Introduction

Ecore modelling uses three layers of models. The Ecore metamodel is the top layer, an instance of the Ecore metamodel is referred to as an Ecore model, and an instance of an Ecore model is referred to as an Ecore instance model. This is visualised in Figure 4.1. Similar to MOF modelling, the layers can be referred to as M3 to M1. In this view, the real world can then be viewed as layer M0, being an instance of M1. In Ecore modelling, M3 is also sometimes referred to as the Ecore core model or the Ecore metametamodel, M2 as the Ecore metamodel and M1 as the Ecore model. However, we use the terms Ecore model for M2 and Ecore instance model for M1 because of the analogy to the terms of type graph and instance graph we map them to.

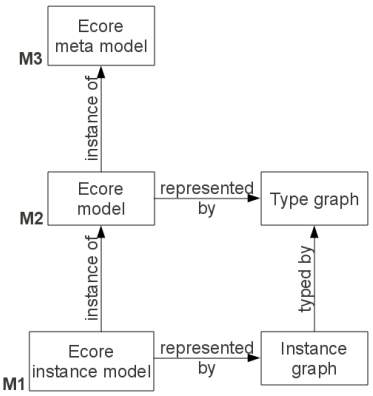


Figure 4.1: Three Ecore layers and their graph representation.

In order to transform Ecore models to graphs, we need to study the Ecore metamodel and determine how each modelling element can be represented in a graph. Our goal in this chapter is to represent Ecore models and instance models as graphs to be used in GROOVE grammars. More specifically, Ecore models will be represented as type graphs and Ecore instance models will be represented as instance graphs. The representation should make it possible to make changes to the instance graphs while remaining conform to the type

graph, and then transform the instance graph back to a valid Ecore instance model which in turn is conform to the Ecore model.

As a first step to determine how Ecore models and instance models can be represented as graphs, we take a detailed look at the Ecore metamodel in section 4.2. The Ecore metamodel is available as an Ecore model in the EMF package of Eclipse. Even though it is an Ecore model, it represents the Ecore metamodel. We inspect the elements of this metamodel which are the building blocks for Ecore models, the properties that can be set for each element, and the hierarchy and relations between the elements. For each element and property we explain what it implies and how it can be represented in a graph.

After looking at the building blocks in the Ecore metamodel, we turn our focus to actual Ecore models and instance models created in Eclipse. In section 4.3 we show examples of models and instance models created in Eclipse. Our goal here is to demonstrate the implications of our representation choices from section 4.2 for Ecore models and instance models. We do this by presenting graph representations of these example models.

Finally, we cannot enforce the validity of an instance graph as a representation of an Ecore instance model by just using type graphs. We need some other method to check the validity of instance graphs before or while transforming an instance graph to its instance model. We do this by means of constraints, which will be explained in section 4.4.

4.2 The Ecore metamodel

The Ecore metamodel contains all elements that can be used in Ecore modelling, and their relations to each other. Figure 4.2 shows the hierarchy of all Ecore elements. The green or dark grey elements are abstract and cannot be instantiated in Ecore models. They however have properties that are inherited by child classes. The yellow or light grey classes are elements that can be instantiated in Ecore models.

Table 4.1 shows an overview of all Ecore metamodel elements from the hierarchy view and their properties. These properties are attributes, operations, references or annotations. In Ecore models, reference properties refer to other elements in the model. Attribute properties can be set to for example a string or boolean value, and are independent of other elements in the model. Both of these type of properties are examined below, and when relevant for instance models we discuss how they are represented in graphs. Operation properties however define how to modify, delete or add elements or properties of instances of Ecore models, but are not part of instance models themselves. In instance models, the model editor is used to change the model and the operation properties are not used. Similarly, in the graph representation graph transformation rules are used to change the model. Operation properties are never part of the structure of instance models nor of the graph representation, so they have no graph representation and we will not discuss them. Finally, annotation properties are notes for some elements in the Ecore metamodel. They denote constraints for valid

Ecore models. However, we only aim to transform Ecore models and instance models to a graph representation. Whether or not an Ecore model is valid with regard to the constraints set in the Ecore metamodel is up to the used Ecore model editor, and we assume Ecore models to be valid. Annotation properties from the Ecore metamodel have no impact on instance models, so they do not need a graph representation, and we will not discuss them. However, when transforming the Ecore metametamodel as an Ecore model to a type graph, and an Ecore model as an instance model to an instance graph, we cannot guarantee in the instance graph that no constraints from the Ecore metametamodel are violated. Constraints in annotations have no semantics defined in the model, so they cannot be translated into constraints on instance graphs.

Representation examples are given for each element in section 4.3. Table 4.3 contains a list of all supported elements and properties, and which examples demonstrate them. Some elements or properties are represented by constraints that enforce valid instance graph. Constraints are discussed in section 4.4. Table 4.3 also lists the constraint examples that demonstrates constraints required for the listed elements and properties.

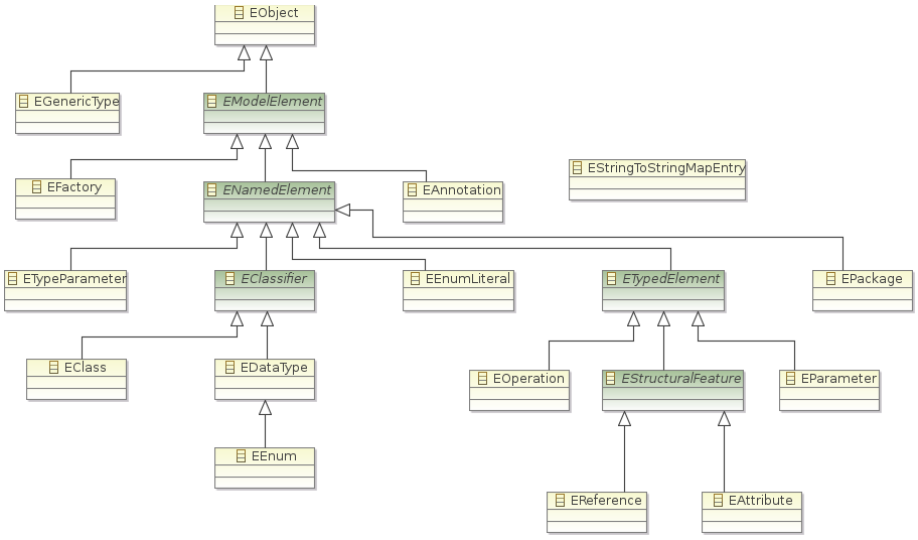


Figure 4.2: Ecore metamodel class hierarchy.

Table 4.1: Modelling elements and properties that occur in the Ecore metamodel along with how they are represented in a graph representation. The first column lists the elements from the Ecore metamodel. The second and third columns list the properties and their types for each element, in the case of dashes this row shows representation of the Ecore metamodel element itself. The fourth and fifth columns shows how Ecore metamodel elements and their properties are represented, Ecore model as type graph and Ecore instance model as instance graph. In the case of dashes there is no graph representation.

Ecore metamodel element	**Property name**	**Property type**	**Type graph rep.**	**Instance graph rep.**
EObject	-	-	-	-
EObject	eClass	Operation	-	-
EObject	eIsProxy	Operation	-	-
EObject	eResource	Operation	-	-
EObject	eContainer	Operation	-	-
EObject	eContainerFeature	Operation	-	-
EObject	eContainmentFeature	Operation	-	-
EObject	eContents	Operation	-	-
EObject	eAllContents	Operation	-	-
EObject	eCrossReferences	Operation	-	-
EObject	eGet	Operation	-	-
EObject	eGet	Operation	-	-
EObject	eSet	Operation	-	-
EObject	eIsSet	Operation	-	-
EObject	eUnset	Operation	-	-
EModelElement	-	-	- (abstract)	-
EModelElement	getEAnnotation	Operation	-	-
EModelElement	eAnnotations	Reference	-	-
ENamedElement	-	-	node type	node type
ENamedElement	name	Attribute	Label of node type	Label of node type

Continued on the next page

Table 4.1 – continued from the previous page

Ecore metamodel element	**Property name**	**Property type**	**Type graph rep.**	**Instance graph rep.**
ENamedElement	constraints	Annotation	-	-
EClassifier	-	-	- (abstract)	-
EClassifier	instanceClassName	Attribute	- (not used in instances)	-
EClassifier	instanceClass	Attribute	- (derived)	-
EClassifier	defaultValue	Attribute	- (derived)	-
EClassifier	instanceTypeName	Attribute	- (not used in instances)	-
EClassifier	isInstance	Operation	-	-
EClassifier	getClassifierID	Operation	-	-
EClassifier	ePackage	Reference	Prefixed to names	Prefixed to names
EClassifier	eTypeParameters	Reference	- (not used in instances)	-
EClassifier	constraints	Annotation	-	-
ETypedElement	-	-	- (abstract)	-
ETypedElement	ordered	Attribute	*next* edge	*next* edges + constraint
ETypedElement	unique	Attribute	-	Constraint
ETypedElement	lowerBound	Attribute	-	Constraint
ETypedElement	upperBound	Attribute	-	Constraint
ETypedElement	many	Attribute	- (derived)	-
ETypedElement	required	Attribute	- (derived)	-
ETypedElement	eGenericType	Reference	- (not used in instances)	-
ETypedElement	eType	Reference	*val* edge	*val* edge + constraint
ETypedElement	constraints	Annotation	-	-
EStructuralFeature	-	-	- (abstract)	-
EStructuralFeature	changeable	Attribute	-	Constraint
EStructuralFeature	volatile	Attribute	- (not used in instances)	-
EStructuralFeature	transient	Attribute	- (not used in instances)	-
EStructuralFeature	defaultLiteralValue	Attribute	-	Constraint

Continued on the next page

Table 4.1 – continued from the previous page

Ecore metamodel element	**Property name**	**Property type**	**Type graph rep.**	**Instance graph rep.**
EStructuralFeature	defaultValue	Attribute	- (derived)	-
EStructuralFeature	unsettable	Attribute	- (not used in instances)	-
EStructuralFeature	derived	Attribute	- (not used in instances)	-
EStructuralFeature	getFeatureID	Operation	-	-
EStructuralFeature	getContainerClass	Operation	-	-
EStructuralFeature	eContainingClass	Reference	-	Constraint
EStructuralFeature	constraints	Annotation	-	-
EGenericType	-	-	- (not supported)	-
EGenericType	eClassifier	Reference	-	-
EGenericType	eLowerBound	Reference	-	-
EGenericType	eUpperBound	Reference	-	-
EGenericType	eRawType	Reference	-	-
EGenericType	eTypeArguments	Reference	-	-
EGenericType	eTypeParameter	Reference	-	-
EGenericType	constraints	Annotation	-	-
EFactory	-	-	- (not supported)	-
EFactory	create	Operation	-	-
EFactory	createFromString	Operation	-	-
EFactory	convertToString	Operation	-	-
EFactory	ePackage	Reference	-	-
EAnnotation	-	-	- (not supported)	-
EAnnotation	source	Attribute	-	-
EAnnotation	contents	Reference	-	-
EAnnotation	details	Reference	-	-
EAnnotation	eModelElement	Reference	-	-
EAnnotation	references	Reference	-	-

Continued on the next page

Table 4.1 – continued from the previous page

Ecore metamodel element	**Property name**	**Property type**	**Type graph rep.**	**Instance graph rep.**
EAnnotation	constraints	Annotation	-	-
ETypeParameter	-	-	- (not supported)	-
ETypeParameter	eBounds	Reference	-	-
EEnumLiteral	-	-	Flag on EEnum	Flag on EEnum + constraint
EEnumLiteral	value	Attribute	- (not used in instances)	-
EEnumLiteral	instance	Attribute	- (not used in instances)	-
EEnumLiteral	literal	Attribute	label of flag	label of flag + constraint
EEnumLiteral	eEnum	Reference	- (opposite of *eLiterals*)	-
EPackage	-	-	prefixed to names	Prefixed to names
EPackage	nsURI	Attribute	- (not used in instances)	-
EPackage	nsPrefix	Attribute	- (not used in instances)	-
EPackage	getEClassifier	Operation	-	-
EPackage	eClassifiers	Reference	- (opposite of *ePackage*)	-
EPackage	eFactoryInstance	Reference	- (not used in instances)	-
EPackage	eSubpackages	Reference	Done automatically	Done automatically
EPackage	eSuperPackage	Reference	Done automatically	Done automatically
EPackage	constraints	Annotation	-	-
EClass	-	-	node type	Node subtype + constraints
EClass	abstract	Attribute	-	Constraint
EClass	interface	Attribute	- (not used in instances)	-
EClass	isSuperTypeOf	Operation	-	-
EClass	getFeatureCount	Operation	-	-
EClass	getStructuralFeature	Operation	-	-
EClass	getStructuralFeature	Operation	-	-
EClass	getFeatureID	Operation	-	-
EClass	eAllAttributes	Reference	- (derived)	-

Continued on the next page

Table 4.1 – continued from the previous page

Ecore metamodel element	Property name	Property type	Type graph rep.	Instance graph rep.
EClass	eAllContainments	Reference	- (derived)	-
EClass	eAllGenericSuperTypes	Reference	- (derived)	-
EClass	eAllOperations	Reference	- (derived)	-
EClass	eAllReferences	Reference	- (derived)	-
EClass	eAllStructuralFeatures	Reference	- (derived)	-
EClass	eAllSuperTypes	Reference	- (derived)	-
EClass	eAttributes	Reference	- (derived)	-
EClass	eGenericSuperTypes	Reference	- (not used in instances)	-
EClass	eIDAttribute	Reference	- (derived)	-
EClass	eOperations	Reference	- (not used in instances)	-
EClass	eReferences	Reference	- (derived)	-
EClass	eStructuralFeatures	Reference	Edge to feature	Edge to feature
EClass	eSuperTypes	Reference	Subnode types	Subnode types
EClass	constraints	Annotation	-	-
EDataType	-	-	node type	node type + constraint
EDataType	serializable	Attribute	- (not used in instances)	-
EEnum	-	-	node type	node type + constraints
EEnum	getEEnumLiteral	Operation	-	-
EEnum	getEEnumLiteral	Operation	-	-
EEnum	getEEnumLiteralByLiteral	Operation	-	-
EEnum	eLiterals	Reference	-	Constraint
EEnum	constraints	Annotation	-	-
EOperation	-	-	- (not supported)	-
EOperation	eContainingClass	Reference	-	-
EOperation	eExceptions	Reference	-	-
EOperation	eGenericExceptions	Reference	-	-

Continued on the next page

Table 4.1 – continued from the previous page

Ecore metamodel element	**Property name**	**Property type**	**Type graph rep.**	**Instance graph rep.**
EOperation	eParameters	Reference	-	-
EOperation	eTypeParameters	Reference	-	-
EOperation	constraints	Annotation	-	-
EParameter	-	-	- (not supported)	-
EParameter	eOperation	Reference	-	-
EReference	-	-	node type	Node subtype
EReference	container	Attribute	- (opposite of *eReferences*	-
EReference	containment	Attribute	-	Flag + constraints
EReference	resolveProxies	Attribute	- (not used in instances)	-
EReference	eKeys	Reference	-	Constraint
EReference	eOpposite	Reference	*opposite* edge	*opposite* edge + constraint
EReference	eReferenceType	Reference	*val* edge	*val* edge + constraint
EReference	constraints	Annotation	-	-
EAttribute	-	-	node type	node type
EAttribute	iD	Attribute	-	Constraint
EAttribute	eAttributeType	Reference	*val* edge	*val* edge + constraint
EAttribute	constraints	Annotation	-	-
EStringToStringMapEntry	-	-	- (not supported)	-
EStringToStringMapEntry	key	Attribute	-	-
EStringToStringMapEntry	value	Attribute	-	-

4.2.1 EObject

The *EObject* is the top level element in the Ecore metamodel. It can normally not be instantiated in Ecore models because there is no modelling element that can be a container for an *EObject*. An *EObject* also cannot be a container for other elements in an Ecore model. It is possible and allowed to create an Ecore model with an *EObject* as root element, however this Ecore model cannot have any other elements since *EObject* cannot contain anything, and would therefore have no meaning. Considering this and that *EObjects* cannot be contained by other elements in Ecore models, we can safely say that *EObjects* will not occur in any meaningful Ecore models, and therefore do not need a graph representation.

4.2.2 *EModelElement*

EModelElement is abstract so it cannot be instantiated in Ecore models. It has two properties, one containment relation *eAnnotations*, and an operation that acts as a *get*-method for all contained *EAnnotations*. *EAnnotations* do not need a graph representation as described in the respective section below, so the reference that contains them also does not need a graph representation.

4.2.3 EFactory

The *EFactory* is an element like *EObject* in the sense that it is not instantiated in an Ecore model, except in the case of a meaningless model with a single *EFactory* root element with no other element. An *EFactory* cannot be contained by any other elements, nor can it contain other elements. An *EFactory* is not used in Ecore modelling as an element to represent something of the world like other elements do.

Instead, an *EFactory* provides operations to create instances of non-abstract classes for an instance model. A singleton instance of *EFactory* is generated for every package in an Ecore model, with operations to create instances of classes and convert *EObjects* that represent *EDataType* values to and from strings. For example, the generated Ecore model instance editor uses the operations of this *EFactory* to create instances of Ecore model elements. Additionally, apart from using the Ecore model editor in Eclipse, Ecore models can also be created dynamically in Java programs by using the *EFactory* of the Ecore metamodel, since an Ecore model is an instance of the Ecore metamodel.

So *EFactories* and its operation properties do not need a graph representation since they will not occur in Ecore models to represent something.

4.2.4 *ENamedElement*

ENamedElement is an abstract element and has a *name* property. The name of an element is a string of characters. All elements that inherit from *ENamedElement* are identified within their container by their name. This container

is better known as a *namespace*, which is the context of the element. Many elements in Ecore modelling can in turn be a container for other elements. This means that for any given element, its namespace and name combine into the namespace of contained elements.

Any element of an Ecore model that needs a graph representation will be represented by a node type in both type graph and instance graph. The label of this node consists of the ***name*** with the namespace prefixed to it. Node types in type graphs must have unique labels and uniquely represent an element of an Ecore model. This is achieved by suffixing a *$* to the *names* of *EPackages* and prefixing a *$* to the *names* of other *ENamedElements*.

This is required because *EPackages* can contain both *EClassifiers* and *EPackages*. Both *EClassifiers* and *EPackages* can be the namespace of elements. We use the suffixed and prefixed *$* to distinguish namespaces that are *EPackages* from namespaces that are *EClassifiers*. Consider the example Ecore model in Figure 4.3. We have *ClassA* containing *has* twice, one where an *EClass* contains an *EReference*, the other where an *EPackage* contains an *EClass*. In our graph representation, the former *has* is represented by a node type labelled *ClassA$has*, the latter is represented by a node type labelled *ClassA$$has*. The double *$$* distinguishes *EPackages* containing *EClassifiers* from other cases, and this ensures that cases like *has* in Figure 4.3 get different and unique node types, which is required in type graphs.

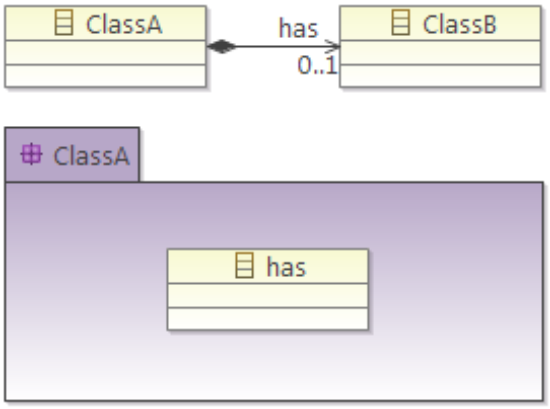


Figure 4.3: Example of an Ecore model with a possible conflict of *ClassA* containing *has*.

Ecore models have a root element, which is a container for all other elements. Since all elements are contained by this root element, its name is part of the namespace of every element. It does not contribute to the unique identification of an element and is therefore usually omitted in a representation of a model. The root element of an Ecore model is the root element of the XML file. We need this information when transforming an instance graph back to an instance model. However, we use the Ecore model itself to transform an instance graph to an instance model and have the information available. Therefore this information does not need a graph representation.

Finally, if from the earlier example *ClassA* would be contained only by the root *EPackage*, the node type that represents it would be *$ClassA*. In these cases we can remove the leading *$* from the node type since it does not act as a seperator anymore.

Examples for the *name* property of *ENamedElements* are listed in Table 4.3.

4.2.5 *EClassifier*

EClassifiers are either *EClasses* or *EDataTypes*, which are represented by a node type as defined for *ENamedElement*. *EClassifier* has one relevant property, namely the ***ePackage*** reference. This reference points to the package that an *EClassifier* belongs to. The package of an *EClassifier* is the context or namespace of this element. This is represented as defined for *ENamedElement* by prefixing the namespace to the name in the node type label.

EClassifier has other properties which do not need a graph representation. *instanceClass* and *defaultValue* are derived properties, which means they are derived from other elements or properties in the model and do not need an explicit representation. Furthermore there are two operation properties *isInstance* and *getClassifierID* which are not a part of instance models, a reference property *eTypeParameters* referring to *ETypeParameters* which are not supported as explained below, and the attribute properties *instanceClassName* and *instanceTypeName* which are only used for Java code generation when *EClassifiers* in the Ecore model represent some Java object.

Examples for the *ePackage* property of *EClassifiers* are listed in Table 4.3.

4.2.6 *ETypedElement*

ETypedElement is another abstract class and can either be an *EOperation*, *EParameter*, *EReference* or *EAttribute*. Since *EOperations* and *EParameters* are not instantiated in instance models and are therefore not relevant for a graph representation, we only need to concern ourselves with *EReferences* and *EAttributes*. There are five properties for *ETypedElement* that are relevant for instance models and their graph representation, *ordered*, *unique*, *lowerBound*, *upperBound* and *eType*.

Ordered is a boolean that indicates whether or not the order in which values occur is of importance. In a graph representation, we can impose an order by using *next* edges. Since edges can only link nodes and not other edges, *ETypedElements* must be represented by node types in graphs. The nodes representing *ETypedElements* can then be ordered by using *next* edges. In the type graph, nodes representing *ETypedElements* have a *next* self edge to indicate that nodes of this type can be both source and target of these edges. In an instance graph, when there are multiple instances of a single *ETypedElement*, these instances have *next* edges between them. By just typing a graph, we cannot enforce that instances are actually ordered. For example, instances can

have self edges or there can be loops of *next* edges. We use constraints to enforce that the placement of *next* edges imposes an actual ordering.

Unique is another boolean property, it indicates whether the same value may occur more than once. This is dependent on the type of the value. For example for *EReferences* there may not be multiple references to the same model element and for *EAttributes* with a *many* multiplicity the same integer, string or other value may not occur more than once. This cannot be represented in a type and instance graph setting and we use constraints to enforce this.

ETypedElements have a ***lowerBound*** and ***upperBound*** property for the multiplicity. These indicate the number of instances of values that are allowed in the instance model. *EReferences* often have a *zero-or-one* or *any* multiplicity, and *EAttributes* usually have just a *zero-or-one* multiplicity, but any boundaries are possible. There are ways to represent multiplicity in the type graph setting, but GROOVE does not support this. We use constraints to make sure the lower and upper bounds are not violated.

Finally as the name suggests, *ETypedElements* are elements of a specific type, the ***eType***. The *eType* of an *ETypedElement* refers to the *EClassifier* that represents the type of the element. However, not all *EClassifiers* are valid types for an *ETypeElement*. The valid types are bound by the *ValidType* constraint in the Ecore metamodel. The *eType* of an *EAttribute* must refer to an *EDataType* and the *eType* of an *EReference* must refer to an *EClass*. Since the allowed *eTypes* are different for each *ETypedElement* that can be instantiated, we deal with the representation at the dedicated subsections below.

The other four properties, *many*, *required*, *eGenericType* and *constraints*, are not relevant for instance models and their graph representations. *Many* and *required* are derived properties and do not appear in instance models, *EGenericTypes* that *eGenericType* refers to are not supported in instance models as explained in section 4.2.16, and the annotation property *constraints* also has no influence on instance models.

Examples for the relevant properties of *ETypedElements* are listed in Table 4.3, as well as examples of constraints on instance models.

4.2.7 *EStructuralFeature*

These elements are either *EReferences* or *EAttributes*. Two properties which are tied together, *changeable* and *defaultLiteralValue*, appear in instance models and are relevant in the graph representation, as well as the *eContainingClass* boolean property. When *changeable* is set to *false*, the value of the *EStructuralFeature* may not change in the instance model. The target of *EReferences* may not change after instantiating a model, and the value of *EAttributes* can only be the default value, which is set by the *defaultLiteralValue* property.

For consistency and applicability of graph transformation rules for the instance graph, we do not want to represent ***unchangeable*** *EStructuralFeatures* different from changeable *EStructuralFeatures*. This means that we do not enforce the values in the type graph. *EAttributes* are represented as an attribute

in GROOVE as explained later, and *EReferences* can target any instance of the type determined in the metamodel and type graph. To enforce that the value cannot change in the instance graph we use a constraint. The ***defaultLiteralValue*** is part of that constraint.

It must be noted that *EReferences* and *EAttributes* are represented in graphs by a node type labelled *name* with *namespace* prefixed. Since node types in type graphs must have unique labels, this could potentially result in a conflict if an *EReference* and *EAttribute* have the same name and are contained by the same element. However this cannot occur, since only *EClasses* can contain *EStructuralFeatures*, and *EClasses* have a constraint in the Ecore metamodel that all their features must have unique *names*.

An instance of an *EStructuralFeature* in an Ecore model is always contained by an *EClass* through the ***eContainingClass*** reference property, which is the opposite of the *eStructuralFeatures* reference property of *EClasses*. For a representation of an *EStructuralFeature* in the instance graph there must always be exactly one edge from the *container EClass*. This is enforced by a constraint.

The other properties do not require a graph representation. When set, *volatile* and *transient* have no impact on instance models, and *defaultValue* is derived from other properties. When an *EStructuralFeature* is set to *unsettable* methods are generated when generating Java code that support an *unset* state of variables, but this has no effect on instance models. The *derived* property indicates that values of *EStructuralFeatures* are derived, however they can still be set manually in instance models, so they have no impact on instance models nor their graph representation. The last three properties are two operation properties, *getFeatureID* and *getContainerClass*, and an annotation property, *constraints*, and hence are also not part of instance models.

Examples for the *changeable*, *defaultLiteralValue* and *eContainingClass* properties of *EStructuralFeatures* are listed in Table 4.3, as well as an example for constraints that are needed.

4.2.8 EClass

EClasses are the main building blocks of Ecore models. They have three properties that are relevant for instance models and their graph representation: *abstract*, *eSuperTypes* and *eStructuralFeatures*, and 19 that are not relevant. *Abstract EClasses* cannot be instantiated in instance models, and can only be inherited from. *eSuperTypes* is a reference with *many* multiplicity that refers to the *EClasses* an *EClass* inherits from. Finally, the *eStructuralFeatures* of an *EClass* are its*EReferences* and *EAttributes*.

EClasses are represented in type and instance graphs by node types, as inherited from *ENamedElement*. Ecore instance models have a ***root*** *EClass* element that directly or indirectly contains all other *EClasses* of the model through *containment EReferences*. The *root* can be any *EClass* of the Ecore model. We need to identify this in instance graphs for checking constraint violations. A singleton node type labelled *EClass* is added to a second type

graph, the Ecore type graph, that all *EClasses* are subtypes of. This node type has a *root* flag. In the instance graph only the *root EClass* must have this flag. Whether this is true is checked using a constraint.

EClasses have a hierarchy structure where classes inherit the list of attributes and relations from their supertype. In Ecore, ***eSuperTypes*** is a relation with *many* multiplicity that refers to the supertypes of an *EClass*. We use subtypes of node types to represent inheritance of *EClasses* in the graph representation. All features (*ETypedElements*) that are contained by an *EClass* are represented in the type graph at the node type representing this *EClass*, and can occur for any nodes that inherit from this node type in the instance graph. This exactly mimics the behaviour of inheritance in Ecore models. Multiple inheritance is also supported this way.

Abstract *EClasses* cannot be flagged as such in the type graph, since any classes that inherit from it would also inherit the flag that marks it *abstract* in GROOVE. Any node in the type graph can occur in the instance graph, so any nodes that represent *abstract EClasses* in the type graph, can also occur in the instance graph. A constraint is used to prevent instantiations of nodes that represent *abstract EClasses* in the instance graph.

eStructuralFeatures is a relation with *many* multiplicity that refers to all *EStructuralFeatures* that are contained by an *EClass*. *EStructuralFeatures* are represented in the type and instance graph representation by a node type, as inherited from *ENamedElement*, so we use edges from the *EClass* node type to each of the contained *EStructuralFeatures* node types to represent *eStructuralFeatures*. For node types in the type graph, the labels on outgoing edges must be uniquely named. *EClass* has a *UniqueFeatureNames* constraint, requiring that *names* of contained *EStructuralFeatures* are unique. We also use the *names* as labels on the edges representing *eStructuralFeatures*.

Of the properties that are not relevant, the attribute property *interface* denotes an *EClass* to be an interface. This implies that *EOperations* defined in an *interface EClass* must be implemented in *EClasses* that implement the *interface EClass*. Since *EOperations* do not occur in instance models, this property has no effect on instance models or graph representations, so it can be omitted. The *eOperations* and *eGenericSuperTypes* containment relations contain *EOperations* and *EGEnericTypes*. Both of those are elements not supported in graph representations, so the relations containing them do not need a graph representation. The other properties are operation properties, derived reference properties and an annotation property. All of those are not part of instance models, so they are not relevant for the graph representation.

Finally, any instance of an *EClass* must have have exactly one *container EClass*, except for one instance which is the *root EClass* of an instance model. Also, instances of *EClasses* may not form cycles with *containment EReferences*. These properties are not captured by any properties in the Ecore model, but must be enforced in an instance model, which is done using constraints. Examples for the representation and constraints of *EClasses* and their supported properties are listed in Table 4.3.

4.2.9 EPackage

Usually, all *EClassifiers* in Ecore models belong to an *EPackage* as explained above in the section about *EClassifiers*. Even when not apparently so, they belong to the root *EPackage* which is omitted in visual representations. *EPackages* have a name, and two other relevant properties, *eSubpackages* and *eSuperPackage*, and six that are not relevant.

Since *EPackage names* are already represented in graphs by prefixing *EClassifiers* that belong to an *EPackage* with their namespace, *EPackages* are not explicitly represented by node types. *EPackages* can be nested, an *EPackage* can have several sub-*EPackages*, and each *EPackage* apart from the root *EPackage* has a super-*EPackage*. Because the full namespace is part of the labels of *EClassifier* node types in the graph representation, all information of the *EPackage* and super-*EPackages* that an *EClassifier* belongs to is already encoded in the graph representation.

The *NsPrefix* and *NsURI* are not required in the graph representation to represent instance models. The *NsPrefix* is a string that indicates how elements of this *EPackage* are prefixed in the XMI serialization of an Ecore model, and the *NsURI*, usually an absolute URI, universally identifies the package. They are required to transform an instance graph to a valid instance model. However, this information can be acquired from the Ecore model when transforming back to an Ecore instance model and we do not need to represent this.

Examples for the representation of *EPackages* are listed in Table 4.3.

4.2.10 EReference

EReferences are association links between *EClasses*. They are always contained by (or originate from) an *EClass*, and their type (or target) is also an *EClass*. *EReferences* have a *containment* boolean property, can have an *eOpposite* reference property, always have exactly one *eReferenceType* and can have an *eKeys* reference property. Intuitively we would like to represent *EReferences* as simple edges from source to target, as they are represented visually in Ecore models. However, because *EStructuralFeatures* can be ordered, *EReferences* can have opposites, and we prefer a consistent representation, we need to represent *EReferences* as node types. These node types can then be ordered or have an edge that refers to its opposite. A preferred representation would be to use ordered edges [13] to represent ordering so we can use only edges to represent *EReferences*. This is however not supported by GROOVE, and it would still not allow representation of *opposite EReferences*. There are three more properties that do not require a graph representation.

The ***containment*** boolean indicates whether or not an *EReference* represents a composition relation. When set, this implies that the target *EClass* of the *EReference* is contained by the source *EClass*. In Ecore models, all *EClasses* apart from the root element must have a *container EClass*, or in other words have an incoming *containment EReference*. An *EReference* is represented by a node type, and this node type is a subtype of a singleton node type labelled

EReference in a second Ecore type graph. This *EReference* node has a *containment* flag in the type graph. In instance graphs, any node that represents a *containment EReference* must have this flag set, and other instances of *EReferences* may not have this flag set. This is required for detecting constraint violations, as explained in section 4.4.

EReferences are unidirectional. However, something that mimics bidirectionality can be achieved by using the ***eOpposite*** property. This relation property can refer to an *EReference* in the opposite direction of the former *EReference*, making them opposites of each other. Two *containment EReferences* cannot be opposites of each other, since that would violate a constraint that *containment EReferences* may not form cycles. Also, opposite *EReferences* in Ecore models must both have the *eOpposite* property refer to each other. If one *eOpposite* does not refer to the opposite *EReference*, a constraint of the Ecore metamodel is violated. *eOpposite* is represented in the type and instance graph by *opposite* edges in both directions between the node types that represent the *EReferences* that are opposites. Just a single edge in one direction would add inconsistency in the direction of the edge for graph transformation rules. Omitting one the *EReferences* from an opposite pair is also not possible, since both *EReferences* can still have a many-multiplicity and be ordered, as we will show in an example in section 4.3. When two *EReferences* are opposites, then the *eOpposite* property must be set for both *EReferences*, and they must refer to each other. Whether this is true in instance graphs is enforced with a constraint.

The reference property ***eKeys*** refers to *EAttributes* of the referenced *EClass* that uniquely identify a referenced instance. This means that the values of the set of *EAttributes* that are referred to must be unique for each instance of the *EClass* referred to by this *EReference*. This cannot be enforced by a type and instance graph setting and must be enforced with constraints. This is only possible when the referenced *EAttributes* are unique, not ordered and of supported *EDataTypes*, which are explained in section **??**. This property only puts a constraint on valid instance models, and since this is enforced with a graph constraint, *eKeys* needs no graph representation.

The ***eReferenceType*** is derived from *eType* and refers to the type of the *EReference*, which must be an *EClass*. In visual representations of Ecore models, this is the target of the arrow representing the *EReference*. In the type and instance graph representation of an *EReference*, the *eReferenceType* is represented by an edge labelled *val* from the node type representing the *EReference* to the node type representing the *EClass*. There must be exactly one *val* edge to represent the target *EClass* in an instance graph, and this is enforced by using a constraint.

Two properties are not relevant for a graph representation. The attribute property *resolveProxies* adds certain Java code optimizations and has no impact on instance models, the annotation property also does not impact instance models.

Examples for *EReferences* and their supported properties and required constraints are listed in Table 4.3.

4.2.11 EAttribute

EAttributes have an *eAttributeType* property that refers to the data type of this element. Furthermore, *EAttributes* have a boolean property *iD* that indicates whether an *EAttribute* uniquely identifies the containing *EClass* within its container, and one unsupported annotation property. A singleton node type labelled *EAttribute* is added to the Ecore type graph that all *EAttributes* are subtypes of.

The ***eAttributeType*** refers to the *EDatatype* of an *EAttribute* in Ecore models, and in instance models it refers to a concrete value of this *EDatatype* if it is serializable. It is derived from *eType* defined for *ETypedElement*, but may only refer to *EDataTypes* instead of *EClassifiers*. The graph representation for data types and concrete values is explained in section **??**. The *eAttributeType* reference property is represented in the type and instance graph as an edge labelled *val* from the node type representing the *EAttribute* to the node type representing the *EDataType*. If an *EDataType* is not supported in instance models and graph representation, the *val* edge is omitted and values of *EAttributes* cannot be represented. There must be either zero or one *val* edge to represent the target *EDataType* in an instance graph, this is enforced by using a constraint.

If ***iD*** is set, an *EAttribute* is used as a unique identifier for an *EClass*. This implies that in instance models, the concrete value of an *EAttribute* is unique for all *EClasses* that have the same container *EClass*. We cannot enforce this in a type and instance graph setting and must check whether this property holds by using constraints. This property is similar to *eKeys* of *EReference*, however *iD* marks an attribute to be the unique identifier for the *containment EReference* containing the *EClass* this *EAttribute* belongs to. The property *eKeys* can also be set for *non-containment EReferences*. Also, unlike *eKeys*, only a single *EAttribute* can be set to be the identifier. *eKeys* can refer to more than one *EAttribute* of an *EClass*.

Examples for *EAttributes* and their supported properties and required constraints are listed in Table 4.3.

4.2.12 EDataType

EDataTypes have a different representation in Ecore models and instance models. In Ecore models, the type of data is defined, in instance models an *EDataType* holds a variable value of this type. The difference in Ecore and instance models requires a specific representation for *EDataTypes*, different from *EClassifiers*. There is one attribute property that has some impact on instance models, *serializable*, but only for *EDataTypes* defined in the Ecore metamodel.

The Ecore metamodel defines a number of *EDataTypes* that can be used in Ecore models, for example *EString* and *EInt*. *EDataTypes* can also be defined by users in Ecore models. The reference property *instanceTypeName* determines the actual data type of an *EDataType* and valid values in instance models. For example, *EString* refers to the *java.lang.string* class and is used to represent

simple strings. Only *EDataTypes* that are *serializable* can be serialized to XML using XMI. Since the native storage format of Ecore is XML using XMI, only *serializable EDataTypes* can have values set in instance models, and therefore have a graph representation.

Name	**Serializable**	**Graph representation**
EBigInteger	yes	integer
EByte	yes	integer
EByteObject	yes	integer
EIntegerObject	yes	integer
ELongObject	yes	integer
EShortObject	yes	integer
EInt	yes	integer
ELong	yes	integer
EShort	yes	integer
EChar	yes	integer
ECharObject	yes	integer
EBigDecimal	yes	real
EDouble	yes	real
EDoubleObject	yes	real
EFloat	yes	real
EFloatObject	yes	real
EBoolean	yes	boolean
EBooleanObject	yes	boolean
EByteArray	yes	string
EDate	yes	string
EString	yes	string
EJavaClass$< T >$	yes	-
EJavaObject	yes	-
EDiagnosticChain	no	
EEList$< E >$	no	
EEnumerator	no	
EFeatureMap	no	
EFeatureMapEntry	no	
EMap$< K, V >$	no	
EResource	no	
EResourceSet	no	
ETreeIterator$< E >$	no	

Table 4.2: *EDataTypes* in the Ecore metamodel, whether they are serializable and how they can be represented in graphs.

The *EDataTypes* that are defined in the Ecore metamodel and are serializable can have values in instance models and a graph representation, since serialization methods are defined for them. However, user defined *EDataTypes* that are *serializable* cannot be supported, since serialization methods must be written in Java code. From here on we only discuss how to represent *EDataTypes* defined in the Ecore metamodel.

GROOVE supports attributes of type string, integer, real or boolean. If an

EDataType is serializable, it can be represented by either of these four attribute types, and will be represented in the type graph by a node of this type. The instance graph can then hold a variable value of this type. Table 4.2 contains a list of all *EDataTypes* that are defined in the Ecore metamodel and whether or not they are *serializable*. For the *serializable EDataTypes*, we show with which attribute type they are represented in GROOVE. There are two *serializable EDataTypes* that cannot have values in instance models or have a graph representation, *EJavaClass* and *EJavaObject*. *EJavaObject* depends on an *EGenericType* which is not supported. *EJavaObject* refers to *java.lang.Object*, which is a general *EDataType* that can refer to any Java object which in turn can be serializable when implementing the *Serializable* interface. These *EDatatypes* are not supported, since serialization methods are available for them in the Ecore model. When using *EDataTypes* that support values with a higher precision than 32 bits, like *ELong* or *EDouble*, the *algebraFamily* of the generated GROOVE grammar is set to *big*. GROOVE then supports *int* values or arbitrary precision and *real* values with the same precision as *java.math.BigDecimal*.

When transforming an instance graph back to an instance model, the Ecore model is used to determine if values in the instance graph are valid. Because of this, we do not have to know which *EDataType* is represented by an attribute in GROOVE, and the value can just be used. We use constraints to detect invalid values in the instance graphs. For example, an *EByte* is represented by an integer attribute in GROOVE, but valid values must be in range from -128 to 127.

Examples for *EDataTypes* and their supported properties and required constraint are listed in Table 4.3.

4.2.13 EEnum and EEnumLiteral

An *EEnum* is a specialization of an *EDataType*. It has an *eLiterals* containment relation, and through it contains any number of *EEnumLiterals*. These are the literals that are valid for the enumeration.

EEnums are represented by a node type since the *val* edge from an *EAttribute* that represents the *eType* must refer to a node. *EEnumLiterals* have just one property that is relevant for instance models and the graph representation, *literal*, a string that contains the value of this literal. The *EEnumLiterals* that are contained are represented in the type graph as flags, labelled with the ***literal*** string property of the *EEnumLiteral*. In the instance graph exactly one flag must be set for an *EEnum*, which is enforced with a constraint. Additionally, each *EEnum* in an instance graph must be the value of an *EAttribute*, so it must have exactly one incoming *val* edge.

The other four properties of *EEnum* are operation and annotation properties which need no further explanation. *EEnumLiteral* has three properties that are not relevant for instance models. The reference property *eEnum* is the opposite of *eLiterals*. Since *EEnumLiterals* are represented as flags on the *EEnum* referred to by *eEnum*, this needs no further representation. The attribute property *value* is the internal integer value used in Java programs to represent the

literals, but these do not occur in instance models. The other attribute property, *instance*, also does not occur in instance models.

Some other options to represent *EEnums* and *EEnumLiterals* exist, but are not preferred for various reasons. Firstly, the node type representing an *EEnum* can be omitted and flags representing *EEnumLiterals* can be added directly to the *EAttributes*. However, this introduces an inconsistency since there would be no *val* edge anymore from the *EAttribute* to represent its type. Another option is to represent *EEnumLiterals* as singleton node types that are subtypes of the node representing an *EEnum*. In instance graphs, the *val* edge from an *EAttribute* can then directly refer to the singleton node representing an *EEnumLiteral*. This however leads to dangling parts of a graph if an *EEnumLiteral* is not a value of an *EAttribute*, and in our representation everything must be contained by the *root* element of the graph. Finally, we could represent *EEnums* are string attributes in GROOVE, and enforce with constraints that the value of the string in the instance graph equals one of the *literal* strings of the *EEnumLiterals*. This is counter-intuitive since it no longer models *EEnums* as *EDataTypes* with a predefined set of allowed values.

Examples for *EEnums* and *EEnumLiterals* and their supported properties are listed in Table 4.3, as well as the required constraints to enforce correct instance models.

4.2.14 EAnnotation and EStringToStringMapEntry

EAnnotations are notes for any *EModelElements*. They can be used to add some textual information about elements in an Ecore model. An *EAnnotation* has a *source*, which is a string identifier that is typically an URI and uniquely identifies the type of *EAnnotation*. Furthermore, *EAnnotations* can refer to *EObjects* from a model, and contain any number of *EStringToStringMapEntries*, which are pairs of *EStrings*, key and value. For example, in the Ecore metamodel several elements have an *EAnnotations* with an *EStringToStringMapEntry* $\langle key, value \rangle$ pair of $\langle$"*constraints*", $\langle$*string with constraints*$\rangle\rangle$. For the *EAnnotation* of the *EClass* element, the value of *constraints* contains the *"InterfaceIsAbstract"* substring, which indicates that any interface *EClasses* must also be an abstract *EClass*. However, the semantics of *EAnnotations* and *EStringToStringMapEntries* in Ecore models cannot formally be specified within the model, and is left up to the user. They are not instantiated in instance models, so they do not need a graph representation.

4.2.15 EOperation and EParameter

EOperations can operate on *EParameters* that are passed to it and return some value of type *eType*. When generating Java code, each *EOperation* generates an empty method for which functionality needs to be added. As with operation properties, *EOperations* do not operate on instance models, but only function as methods in generated Java programs. For instance models, the model editor is used to make modifications, and in instance graphs, graph transformation rules

are used. Therefore *EOperations* need no graph representation. *EParameters* in turn are only used by *EOperations*, and hence also do not require a graph representation either.

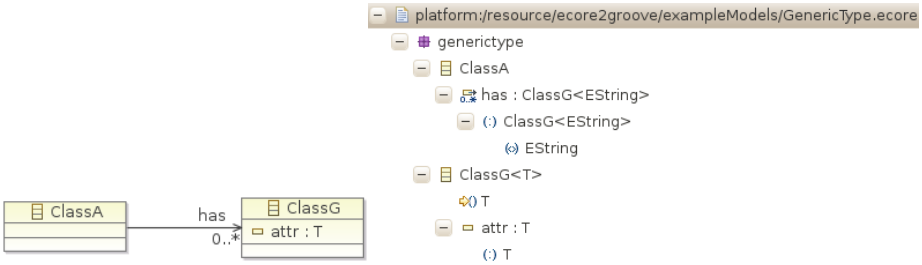
4.2.16 EGenericType and ETypeParameter

EGenericTypes are generic types that can be used by *ETypedElements* or for *eGenericSuperTypes* of *EClasses* when at modelling time the actual type is unknown. An *EGenericType* can either refer to an *EClassifier* from within the model or from the Ecore metamodel through the *eClassifier* relation, or it can refer to an *ETypeParameter* through the *eTypeParameter* containment relation. These are mutually exclusive, so only one reference can be set at a time.

When an *EGenericType* functions as the supertype of an *EClass* it always refers to an *EClassifier*. When it refers to an *EClass* of the Ecore model itself it is a normal supertype, and also referred to by a different property of the former *EClass*. In this case the *EGenericType* needs no graph representation since it is represented in a different way. An *EGenericType* contained by an *EClass* can also refer to *EClassifiers* from the Ecore metamodel. *EClasses* inherit from their supertypes, but if the supertype is not explicitly defined in the Ecore model there are no explicit properties to inherit. This means there is no point representing this in an instance model, nor in the graph representation of the instance model. The same reasoning is applicable to all cases where *EGenericTypes* refer to *EClassifiers*, so these cases never need a graph representation.

EGenericTypes can instead refer to an *ETypeParameter*. *ETypeParameters* can be declared by *EClasses* and *EOperations*, and when set it declares an *EClass* or *EOperation* to be generic. Such *EClasses* and *EOperations* require a type argument passed to it to indicate which type of *EObjects* it should operate on. *ETypeParameters* have an *eBounds* containment relation to indicate the bounds of the type arguments that are allowed, in which case the type argument must be a subtype of the set bound.

As an example of *EGenericTypes* and *ETypeParameters* consider Figure 4.4. In the visual diagram of the Ecore model nothing regarding *EGenericTypes* or *ETypeParameters* is visible, so instead we look at the treeview of the Ecore model. *ClassG* is a generic *EClass*, it declares the *ETypeParameter* T. The *EAttribute attr* is of generic type T, so *attr* refers to an *EGenericType*. Whatever type argument is passed to *ClassG* upon invoking an instance of it is the type of *attr*. In our example, *ClassA* can contain any number of *ClassG* through the containment relation *has*. The type of *has* is *ClassG*, but it also passes a type argument, *EString*. In this case any *ClassG* contained through *has* will have *attr* of type *EString*. This can in theory be represented, because *EAttributes* of type *EString* can be represented in a graph by using string attributes. However, in many cases a representation for values is not possible because the type of an *EAttribute* cannot be represented. In fact, *EAttributes* with a generic type cannot have a value in instance models in Eclipse, not even in our example where in theory it could be possible. Modelling using *EGenericTypes* and *ETypeParameters* is only used for generated Java code. Even when setting aside the instance model editor of Eclipse, such generic types cannot be serialized to XML using



(a) Visual Ecore model (b) Tree of Ecore model

Figure 4.4: Example of a generic *EClass ClassG* with *EAttribute attr*.

XMI either, because any value can be of any *EJavaObject*, which can be an *EClass* that cannot be serialized.

Ecore models with generic *EDatatypes* can still be transformed to a graphs representation, but as with *EDatatypes* that are not serializable, they cannot have values in instance graphs. Other *EGenericTypes* or *EGenericParameters* have no impact on instance models, and do not need a graph representation.

EGenericTypes and *ETypeParameters* in *EOperations* have not been discussed, but the principle is the same, plus that *EOperations* do not need a graph representation in any case. In short, *EGenericTypes* and *ETypeParameters* are not supported in instance models, so they do no need a graph representation. Even if a representation would be possible, it is not possible to exchange values of generic types in instance models to GROOVE using XMI.

4.3 Ecore models and representation examples

We have examined the Ecore metamodel and identified all elements and properties that are relevant for instance models. For each of these elements and properties we described how it is represented in a type and instance graph representation of the Ecore and instance model. We now demonstrate how these representation choices come together in a series of example models and their graph representations. For each of the concrete classes in the Ecore metamodel (*EClass*, *EPackage*, *EDataType*, *EEnum*, *EEnumLiteral*, *EAttribute*, *EReference*) we show one or more examples of an Ecore and instance model, along with their type and instance graph representation. Table 4.3 shows an overview of all modelling elements and properties we identified in section 4.2, and in which examples they are shown, together with the constraints in section 4.4 for these elements and properties.

Ecore metamodel element	**Element property**	**Representation examples**	**Example constraints**
ENamedElement	name	Figure 4.5, 4.6, 4.11, 4.12	-
EClassifier	ePackage	Figure 4.11	-
ETypedElement	ordered	Figure 4.8, 4.10, 4.13	Figure 4.21
	unique	Figure 4.8, 4.13	Figure 4.19, 4.18
	lowerBound	Figure 4.6, 4.13	Figure 4.20
	upperBound	Figure 4.6, 4.13	Figure 4.20
	eType	Figure 4.5, 4.6, Figure 4.12	Figure 4.17
EStructuralFeature	changeable	Figure 4.14	Figure 4.30
	defaultLiteralValue	Figure 4.14	Figure 4.30
	eContainingClass		Figure 4.24
EEnumLiteral	-	Figure 4.16	-
	literal	Figure 4.16	-
EPackage	-	Figure 4.11	-
	eSubPackages	Figure 4.11	-
	eSuperPackage	Figure 4.11	-
EClass	-	Figure 4.5	Figure 4.23, 4.24
	abstract	Figure 4.5	Figure 4.25
	eStructuralFeatures	Figure 4.5	-
	eSuperTypes	Figure 4.5	-
EDataType	-	Figure 4.8, 4.12, 4.13, 4.14, 4.15	Figure 4.31
EEnum	-	Figure 4.16	Figure 4.33
	eLiterals	Figure 4.16	Figure 4.32
EReference	-	Figure 4.6, 4.7, 4.8, 4.9	-
	containment	Figure 4.7	Figure 4.24
	eKeys	Figure 4.9	Figure 4.28
	eOpposite	Figure 4.7, 4.10	Figure 4.27
	eReferenceType	Figure 4.6	Figure 4.17
EAttribute	-	Figure 4.12, 4.13, 4.14, 4.15	-
	iD	Figure 4.15	Figure 4.29
	eAttributeType	Figure 4.12	Figure 4.17

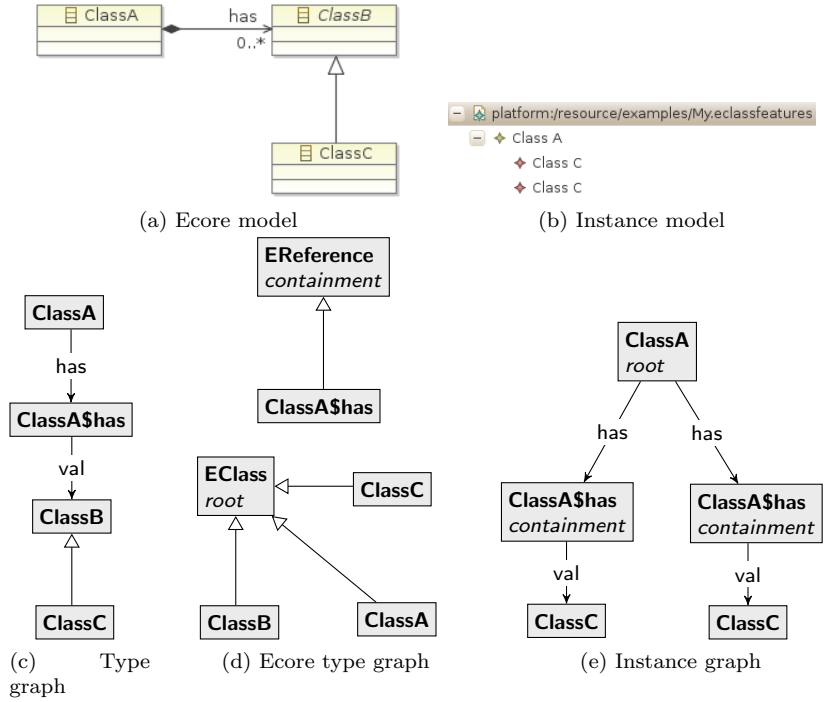
Table 4.3: Mapping of Ecore modelling elements and properties to representation examples and constraints.

4.3.1 EClass

In total, *EClasses* have five relevant properties as determined in section 4.2, *name*, *ePackage*, *abstract*, *eStructuralFeatures* and *eSuperTypes*. *ePackage* refers to the *EPackage* that contains an *EClass*, but this is shown later in the example for *EPackages*. The other four properties are demonstrated below.

Figure 4.5 shows an Ecore model with three *EClasses*. *ClassA* contains any number of *ClassB*, which is *abstract*. The *eSuperType* of *ClassC* is *ClassB*, or in other words *ClassC* inherits from *ClassB*. In the instance model there is one *ClassA* which contains two instances of *ClassC*.

There are two type graphs in the Figure, one for the representation of the model itself, and the *Ecore type graph* to indicate which nodes represent *EClasses* and *EReferences*. GROOVE merges these internally by mapping same types onto each other. In future examples we only speak of the type graph that represents the model and omit the *Ecore type graph*, unless it is relevant for the example. In the type graph representation of the Ecore model the *EClasses* are represented by node types labelled *ClassA*, *ClassB* and *ClassC*, which are the *names* of the *EClasses*. These node types are subtypes of the *EClass* node type to be able to match them in constraint rules. *ClassB* appears in the type graph, even though it is *abstract* and may not be instantiated in a valid instance graph. *ClassA* contains an *EReference* that is referred to by *eStructuralFeature*. The



(a) Ecore model (b) Instance model (c) Type graph (d) Ecore type graph (e) Instance graph

Figure 4.5: Example representation of an *EClass* to demonstrate *name*, *abstract*, *eStructuralFeatures* and *eSupertypes*.

edge representing the *eStructuralFeature* is labelled the same as the *name ESTructuralFeature* that is being referred to, *has* in this case. Finally, *eSuperType* of *ClassC* is represented by an inheritance edge, which implies that *ClassC* is a subtype of *ClassB*.

In the instance graph we see one *ClassA* instance and two *ClassC* instances. The instance of *ClassA* is the root of the model, and therefore is flagged as such. If a *ClassB* would be instantiated in the instance graph, it would violate a constraint described in section 4.4. An example of this is given in Figure 4.25.

Each instance of an *EClass* in the instance graph, except for one, must have exactly one *container EClass*, which is the *EClass* containing said *EClass* through a *containment EReference*. The exception to this is the *root EClass*. This is required because of the XML serialization format of Ecore models and instance models, which are structured like trees. This is enforced through a constraint, of which an example is given in Figure 4.24.

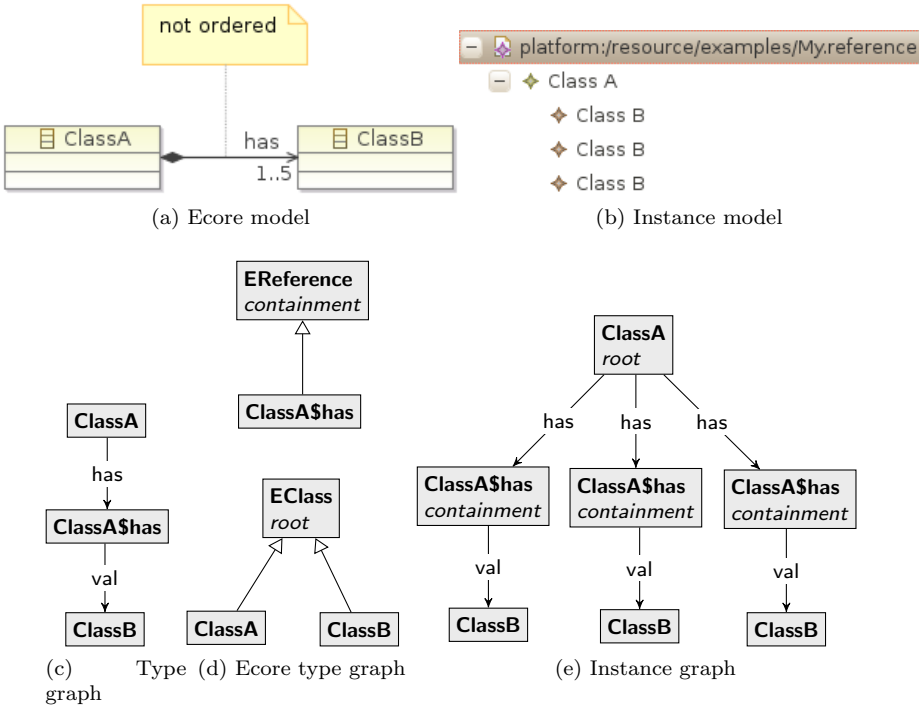
Finally, instances of *EClasses* may not form cycles with *containment EReferences* in an instance graph, which is enforced by a constraint. An example of a violation of this is given in Figure 4.23.

4.3.2 EReference

EReference has 10 properties in total that are relevant for instance models. We show these properties using four examples. The first example shows the usage of *name*, *eReferenceType*, *lowerBound* and *upperBound*. The second example shows the properties *eOpposite* and *containment*. The third example shows the usage of *ordered* and *unique*, and the fourth example shows *eKeys*. Two more properties, *unchangeable* and *defaultLiteralValue* are defined for *EStructuralFeatures*, but are only of relevance to *EAttributes*. *EReferences* cannot have literal values, and hence cannot have a *defaultLiteralValue*. *EReferences* that are *unchangeable* cannot change what they refer to after they have been initialized, but this is used at runtime in Java programs and has no meaning in instance models which are static. Finally we show an additional, more complicated example to demonstrate why an *EReference* with an *opposite EReference* cannot be omitted from the graph representation.

Figure 4.6 contains an Ecore model of *ClassA* which can contain one to five instances of *ClassB* through the *containment EReference has*. The *EReference* in this example is not ordered, which means that the order of instances of *ClassB* in the instance model is not relevant. In the instance model there are three instances of *ClassB*.

There are two type graphs again, one that represents the Ecore model, and one to denote which elements are *EClasses* and which are *EReferences*. The *EReference* is represented by a node type labelled the *name* of the *EReference* with *$* and the namespace prefixed to it. The namespace of an *EReference* is the name of the *container EClass* and its namespace prefixed to it. The namespace of *ClassA* in turn is the root of the Ecore model, so it is omitted in the representation. The *eReferenceType* of the *EReference* is represented



(a) Ecore model (b) Instance model

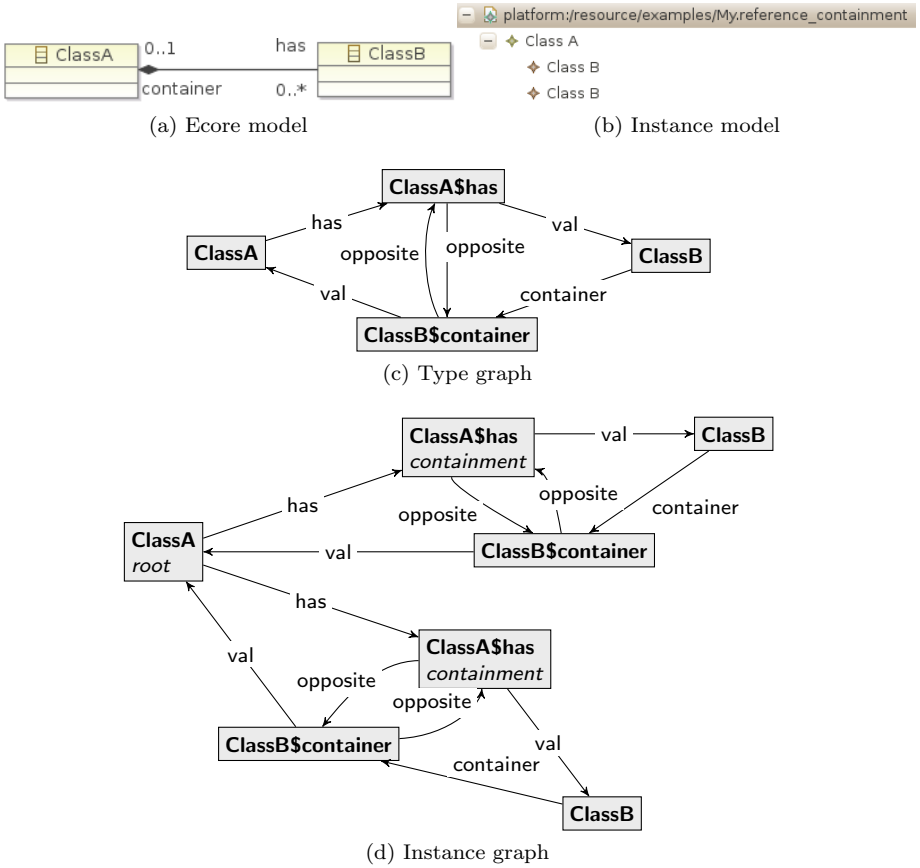
(c) Type graph (d) Ecore type graph (e) Instance graph

Figure 4.6: Example of an *EReference* representation demonstrating the *name*, *eReferenceType*, *lowerBound* and *upperBound*.

by an edge labelled *val* and refers to the target *EClass*. In the Ecore type graph we see that both *ClassA* and *ClassB* are subtypes of the *EClass* node type and therefore represent *EClasses*. *ClassA$has* is a subtype of node type *EReference*, so therefore represents and *EReference*. In the graph representation of the instance model there are three instances of the *EReference*, one for every instance of *ClassB*. They represent *containment EReferences*, so they have a *containment* flag to indicate this. The presence of these flags is enforced by a constraint. Whether or not the *lowerBound* and *upperBound* are violated cannot be checked by typing a graph and is done using constraints, of which an example is given in Figure 4.20. Additionally, there are constraints that there must be exactly one outgoing *val* edge and exactly one incoming edge from the *container EClass* for every instance of *ClassA$has* in the instance graph. A violation of these constraints is demonstrated in the example in Figure 4.17.

Figure 4.7 contains another example Ecore model. In this example there are two *EReferences*, the *containment EReference has*, and its opposite *EReference container*. The *EReference container* has a multiplicity of zero or one. It can be set to any multiplicity like any *EReference* but this would have no meaning. An *EClass* can only have one container, so there is only one valid target for this *EReference*. In the instance model there are two instances of *ClassB*.

In the type graph there are now two node types representing the *EReferences*. There is an edge labelled *opposite* to denote the opposite of an *EReference*. There are two *opposite* edges between each pair of *opposite EReferences*, one



(a) Ecore model

(b) Instance model

(c) Type graph

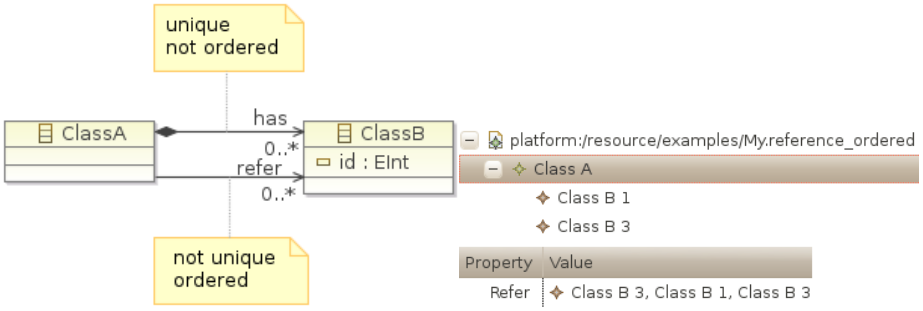
(d) Instance graph

Figure 4.7: Example representation of *EReferences* demonstrating *opposite* and *containment*.

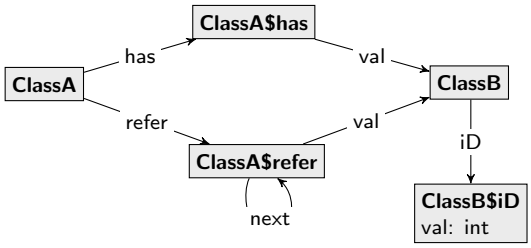
in each direction. In the instance graph there are four nodes that represent *EReferences*, of which the *containment EReferences* have a *containment* flag as inherited in the Ecore type graph. We use a constraint to enforce that *opposite* edges properly refer to the *opposite EReference*. This is demonstrated in Figure 4.27.

Figure 4.8 shows an *EReference* that is not unique and ordered. There are two *EReferences* in this example. *ClassA* can contain any number of *ClassB* through the *EReference has*, and additionally has an *EReference refer* that can refer to any number of *ClassB*. A single *ClassB* can be referenced to several times by *refer*, and the order is also of importance in instance models. The Ecore model contains an *EAttribute* to clarify the order in the graph representation of this example, but this is explained later. In the instance model there are two instances of *ClassB*, and the *EReference refer* refers to one of these instances twice.

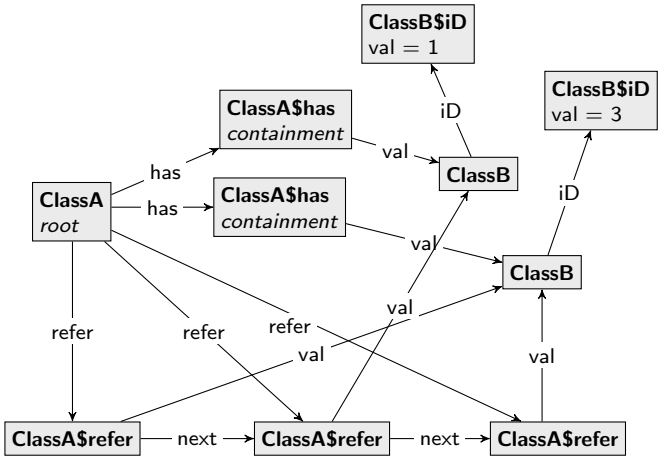
The type graph represents the *EReference refer* by adding a *next* edge to indicate the order in instance graphs. The non-uniqueness does not need a



(a) Ecore model (b) Instance model



(c) Type graph

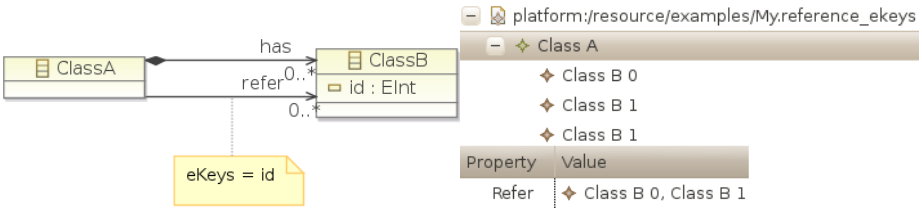


(d) Instance graph

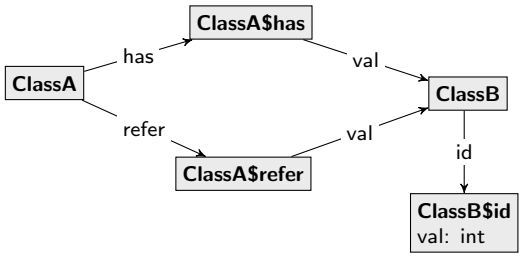
Figure 4.8: Examples representation *EReferences* that demonstrates *ordered* and not-*unique*.

special representation, but instead to check whether an *EReference* is unique requires us to use a constraint, as demonstrated in Figure 4.18. In the instance model we see three nodes to represent the *EReference refer*, ordered with *next* edges. The *EReference* first targets the *ClassB* with *iD 3*, then *iD 1*, and then *iD 3* again, as in the instance model.

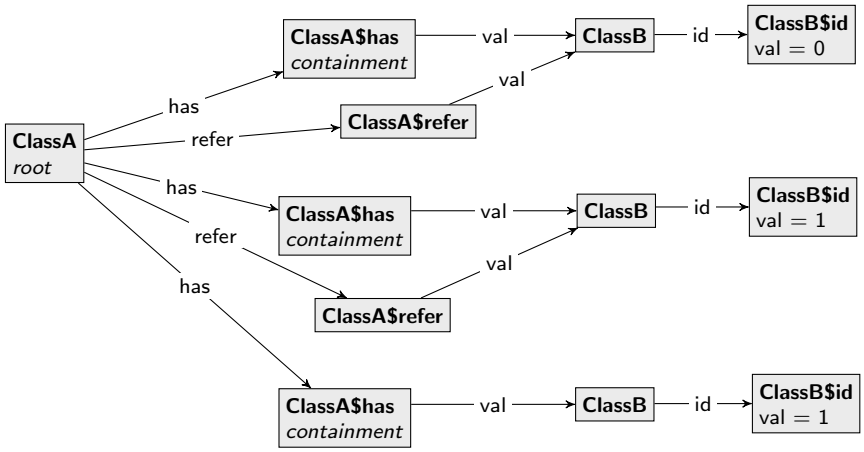
Figure 4.9 shows an example of *eKeys*. *ClassA* contains any number of *ClassB* through *has*, and additionally refers to any number of *ClassB* through *refer*. The *eKeys* property of *refer* refers to the *EAttribute id* of *ClassB*. In the



(a) Ecore model (b) Instance model



(c) Type graph



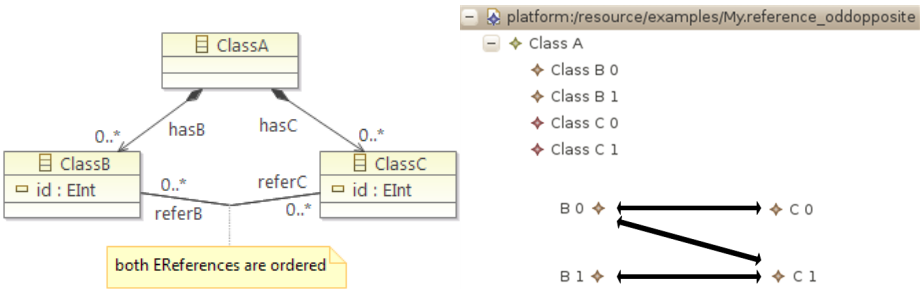
(d) Instance graph

Figure 4.9: Example representation of an *EReference* to demonstrate *eKeys*.

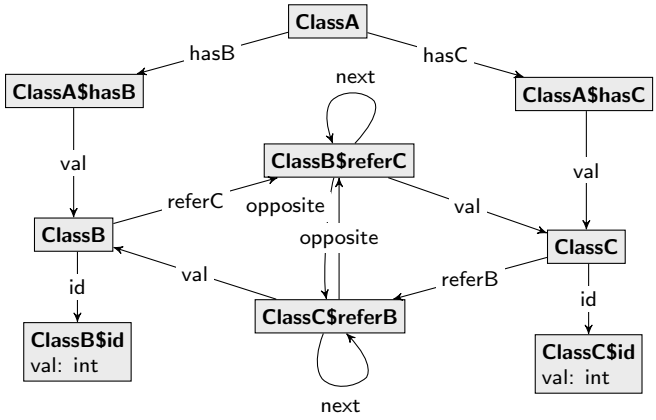
instance model there are three instances of *ClassB*, one with *id 0* and two with *id 1*. The instance of *ClassA* refers to two of these instances of *ClassB* through *refer*. Referring to the third *ClassB* as well is not allowed, since then the value of *id* no longer uniquely identifies a referred instance of *ClassB* in the case of *id 1*.

The graph representation shows nothing new, *EReferences* are represented as before. A third instance of *refer* in the instance graph referring to the third *ClassB* is not allowed as described above, but this can not enforced with a type graphs. Constraints are needed to detect cases that violate the *eKeys* property, as shown in the example in Figure 4.28.

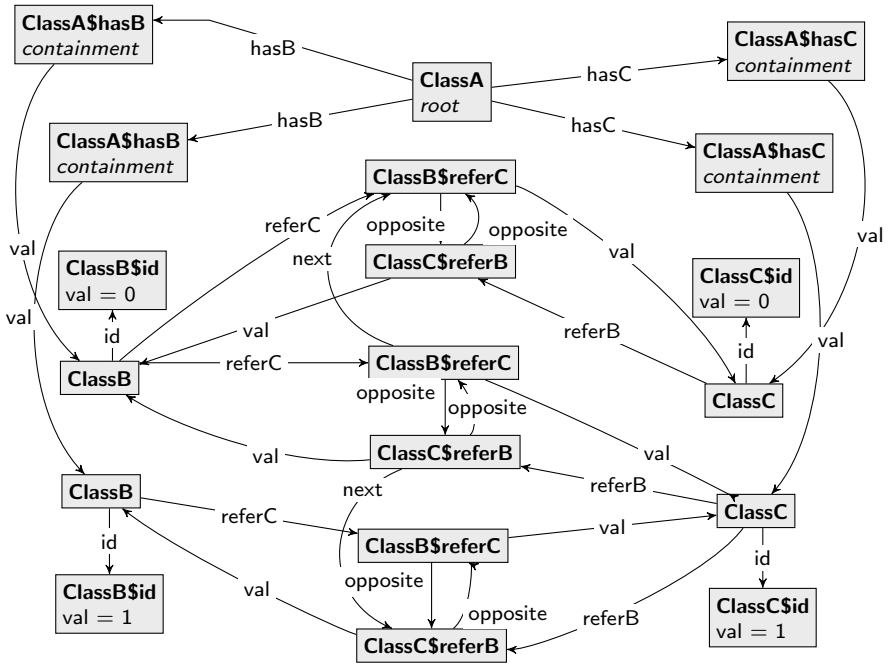
Figure 4.10 shows a more complex example. *ClassA* can contain any num-



(a) Ecore model (b) Instance model



(c) Type graph



(d) Instance graph

Figure 4.10: Example representation showing that *opposite EReferences* both need to be represented in graphs.

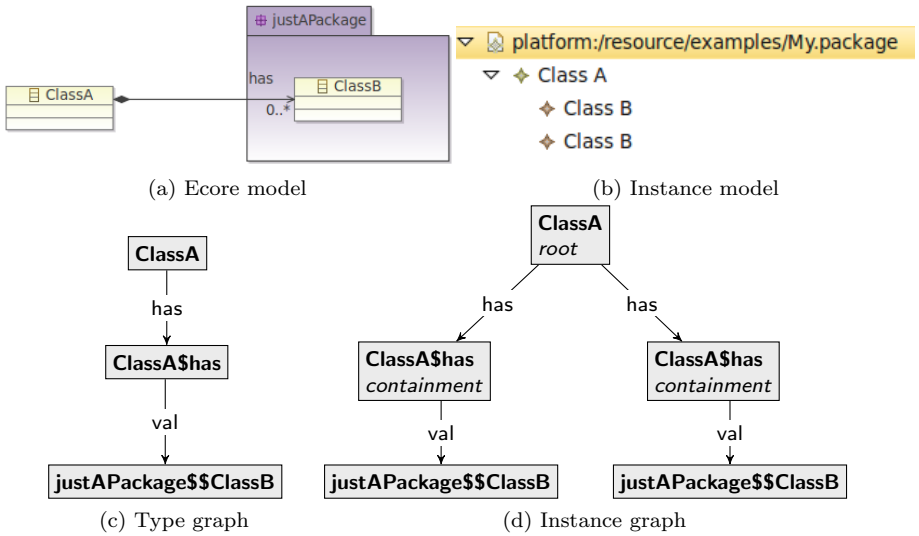
ber of *ClassB* and *ClassC*, *ClassB* refers to *ClassC* with *referC*, and *ClassC* refers to *ClassB* with *referB*. The *EReferences referB* and *referC* are opposites. The *EAttributes id* are used to identify different instances of *ClassB* and *ClassC*. In the instance model, there are two instances of both *ClassB* and *ClassC*. The figure also shows how the instances of *ClassB* and *ClassC* refer to each other with *referB* and *referC*, like *ClassB 0* refers to both instances of *ClassC*, and the opposite *EReferences* refer back to *ClassB 0*. The point here is that an *opposite EReference* must refer back, but may also refer to other instances; for example *ClassC 1* refers back to *ClassB 0*, but also refers to *ClassB 1*. Additionally, *referB* and *referC* are both ordered. *ClassB 0* refers to *(ClassC 1, ClassC 0)* in that order. *ClassC 1* refers to *(ClassB 0, ClassB 1)* in that order.

The type and instance graph representations introduce nothing new. However, because *referB* and *referC* are both ordered, we cannot omit one of these *EReferences* from the graph representation, even though they are opposites. If *referC* would be omitted from the representation the order of *ClassB 0* referring to *(ClassC 1, ClassC 0)* would be lost. The same happens when omitting *referB* from the representation, the order of *ClassC 1* referring to *(ClassB 0, ClassB 1)* would be lost.

4.3.3 EPackage

There are three properties of *EPackage*, *name*, *eSubPackages* and *eSuperPackage*. **Figure 4.11** shows an example of an *EPackage*. *ClassA* can contain any number of *ClassB*, which is in a different package, named *justAPackage*. In the instance model there are two instances of *ClassB*.

In the type and instance graph the label of the node type that represents



(a) Ecore model (b) Instance model

(c) Type graph (d) Instance graph

Figure 4.11: Example representation of an EPackage to show representations of *name*, *eSubPackages* and *eSuperPackage*.

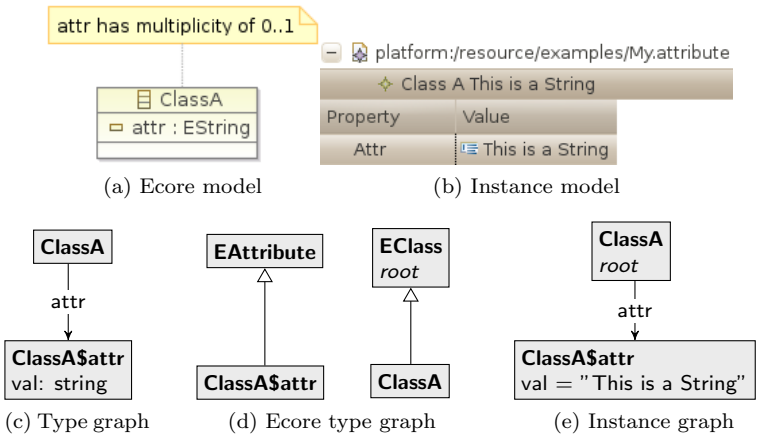
ClassB consists of two parts, the namespace part *justAPackage$*, and the *EClass name* part *$ClassB*. The namespace here consists of the *name* of the *EPackage* with *$* suffixed to it. The double *$$* shows a distinction between *EPackages* and an *EClassifier*, here *ClassB*. Any names separated by *$* before the *$$* are *EPackage names*, where for a given *EPackage name* the *name* before it in the string is its *eSuperPackage* and the *name* after it is a *eSubPackage*.

4.3.4 EAttribute and EDataType

EAttributes have 9 properties that are relevant in instance models and the graph representation, *EDataTypes* have just two. Because *EAttributes* and *EDataTypes* are usually used together in Ecore modelling we combine them in our examples. We use four examples to show how *EAttributes* and *EDataTypes* are represented in graphs. The first example shows the *name* property of both *EAttributes* and *EDataTypes* and the *eAttributeType* of *EAttributes*. The second example shows the *ordered*, *unique*, *lowerBound* and *upperBound* properties of *EAttributes*. The third example shows the *changeable* and *defaultLiteralValue* properties and the final example shows the *iD* property. *EDataTypes* also have the *ePackage* property for which we do not give a separate example. Representation of *EPackages* has been shown in the previous section.

Figure 4.12 contains a single *EClass* that contains an *EAttribute* named *attr*. This *EAttribute* is of *EDataType EString*. In the instance model the value of the instance of *attr* has value *This is a String*.

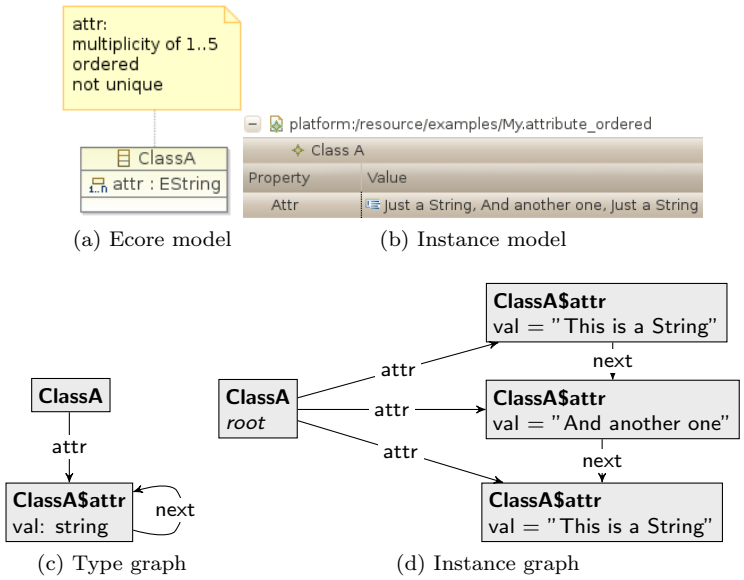
There are two type graphs. In the Ecore type graph, the *Class$attr* is a subtype of the *EAttribute* node type to denote this node represents an *EAttribute*. In the type graph representation of the Ecore model, the *EAttribute* is represented by a node type labelled *ClassA$attr*, a concatenation of the namespace and the *name* of the *EAttribute* with the leading *$* omitted. The *name* of the *EDataType* of *attr* is *EString*. In the mapping in Table 4.2 we see that this is



(a) Ecore model (b) Instance model

(c) Type graph (d) Ecore type graph (e) Instance graph

Figure 4.12: Example representation of an *EAttribute* and an *EDataType* to demonstrate their *name* and the *eAttributeType*.



(a) Ecore model (b) Instance model

(c) Type graph (d) Instance graph

Figure 4.13: Example representation of an *ordered* and non-*unique* *EAttribute* with *lowerBound 1* and *upperBound 5*.

represented in the type graph as a node with type *string*. This node is then referred to by a *val* edge, which represents the *eAttributeType* property of *attr*. In GROOVE attribute nodes are visualized different from regular nodes, as an attribute of a node instead of as a separate node. There is just one instance of *attr* in the instance model, and this is represented by a single node of type *ClassA$attr*. The *val* edge in GROOVE now refers to a string value, *This is a String*. Constraints are used to enforce that for each instance of *ClassA$attr* there is exactly one incoming edge from its *container EClass*, as well as no more than one outgoing *val* edge. A violation is demonstrated in the example in Figure 4.17.

The next example in **Figure 4.13** shows an Ecore model with an *EAttribute* *attr* with *lowerBound 1* and *upperBound 5*. Additionally, this *EAttribute* is *ordered* and *not unique*. The instance model has three string instances for *attr*, and one string occurs twice. The *order* of *Just a String*, *And another one*, *Just a String* is also of relevance in this example. An *EAttribute* with multiplicity else than *0 .. 1* behaves like a *Collection* of values as in java, or more specifically like in Table 4.4.

EAttribute	**Java Collection**
Not unique and not ordered	java.util.*Collection*
Not unique and ordered	java.util.*List*
Unique and not ordered	java.util.*Set*
Unique and ordered	-

Table 4.4: Behaviour of *EAttributes* with multiplicity other than *0..1*

The type graph has the same representation as the previous example, but

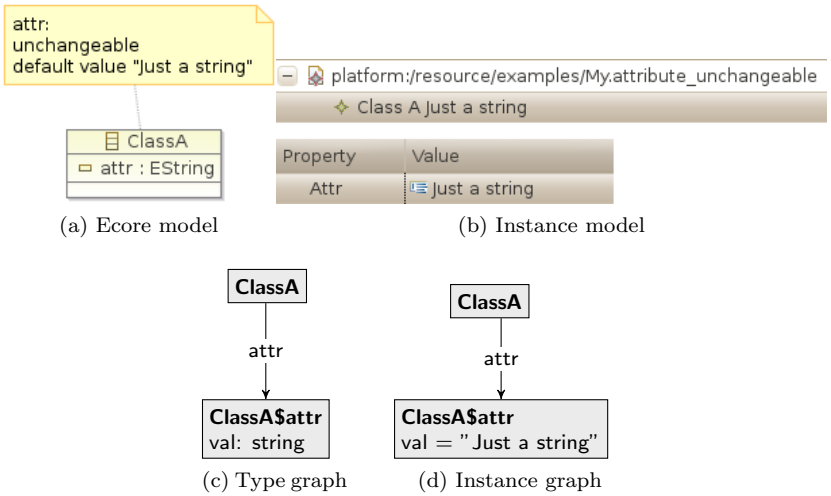
with an added *next* edge to represent the order of instances in the instance graph. The instance graph has three nodes of type *ClassA$attr* to represent the three string instances in the instance model. Each of these nodes has a GROOVE string attribute to represent the value of the string in the instance model. Multiplicity is not supported by GROOVE, so we need to enforce a valid number of instances in the instance graph with a constraint, as in Figure 4.20. The string values are *ordered* by ordering their containers, the nodes representing instances of the *EAttribute*, with *next* edges. We cannot enforce proper placement of the *next* edges by typing the graph, so we need a constraint again, as shown in Figure 4.21. Finally, the values of *attr* in the instance model are not *unique*, but this needs no special representation in the instance graph. As with the *unique* property for *EReference*, we only need to check uniqueness by use of constraints. This is demonstrated for *EAttributes* in Figure 4.19.

The third example shows an *EAttribute attr* with the *changeable* property set to false. The *defaultLiteralValue* of *attr* is *Just a string*. This is shown in **Figure 4.14**.

We want *unchangeable EAttributes* to behave the same as ordinary *EAttributes* in the graph representation. Because of this, the representation is as in the example of Figure 4.12. In the instance graph the value of this *EAttribute* may only take the value of *This is a String*. This is enforced with a constraint, as shown by an example in Figure 4.30.

The final example in **Figure 4.15** shows the *EAttribute attr* with *iD* set to *true*. The instance model has two instances of *ClassB* with different values of *iD*, *0* and *1*. Unique values of *attr* are required for every instance of *ClassB* contained by the same instance of *ClassA*, because it is set to be the *iD*.

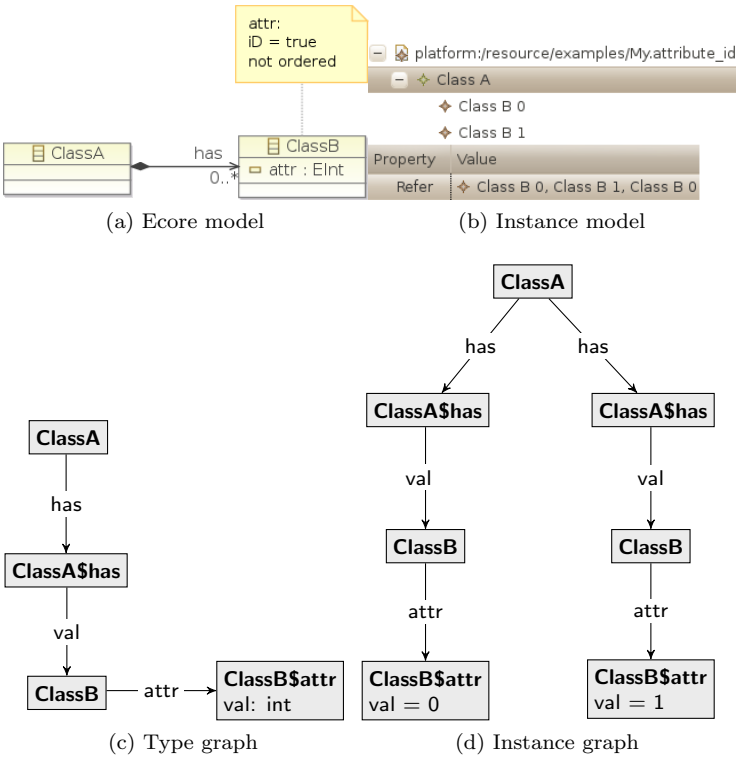
The type and instance graph representation introduce nothing new. Whether or not the values of *attr* are indeed unique for every *ClassB* contained by the



(a) Ecore model (b) Instance model

(c) Type graph (d) Instance graph

Figure 4.14: Example representation of *EAttribute* that is not *changeable* and *defaultValue* set to *Just a string*.



(a) Ecore model (b) Instance model

(c) Type graph (d) Instance graph

Figure 4.15: Example representation of an *EAttribute* that is set to be the *iD*.

same container must be enforced with a constraint, demonstrated by an example in Figure 4.29.

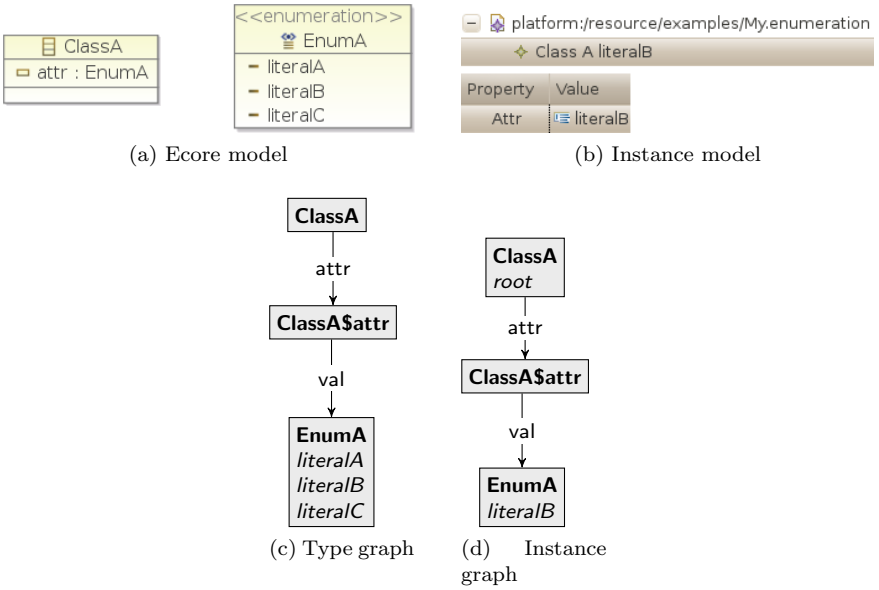
Figure 4.8 shows another example that shows the use of *EDataType*, this time named *EInt*. It is represented in the type and instance graph the same way as for an *EString*, except that the attribute in GROOVE is of type *int* as found in Table 4.2. *EDataTypes* can have a smaller bound of valid values than what is allowed in GROOVE for integer or real values. Valid values are enforced with a constraint. A constraint violation with an invalid value for an *EDataType* is shown in the example in Figure 4.31.

4.3.5 EEnum and EEnumLiterals

EEnums are special *EDataTypes* that have a *name* and *ePackage* property and additionally an *eLiterals* reference property. *EEnumLiterals* only have a *name* and *literal* property. The *ePackage* property of *EEnums* is not shown in an example here, it has been explained above for *EPackage*.

Figure 4.16 shows an Ecore model that contains an *EEnum EnumA* and has three *EEnumLiterals* referred to by *eLiterals*. The *EAttribute attr* of *ClassA* is of type *EnumA* and can only take the literal values of the *literals* of *EnumA*. In the example instance model, the value of *attr* is *literalB*.

EEnums are represented in the type and instance graph by a node type labelled with its namespace and *name*, just *EnumA* in this case. Its *EEnumLiterals* are represented as flags for this node type, one flag in the type graph for each *EEnumLiteral* in the Ecore model. These flags are labelled with the *literal* value of the *EEnumLiterals*. The *names* of *EEnumLiterals* are not represented because *EEnumLiterals* are referred to in the XMI serialization of Ecore instance models by their *literal* value and not their *names*. In instance graphs, only a single flag for each instance of an *EEnum* may be set, and violations are detected by a constraint, as in the example in Figure 4.32.



(a) Ecore model (b) Instance model

(c) Type graph (d) Instance graph

Figure 4.16: Example representation to demonstrate the *name* and *eLiterals* of an *EEnum*, and the *literal* of *EEnumLiterals*.

4.4 Constraints

By only typing an instance graph it is not always guaranteed that an instance graph correctly represents an Ecore instance model. We mentioned several times in sections 4.2 and 4.3 that constraints are needed to enforce that instance graphs are valid representations of Ecore instance models. We use graph transformation rules in GROOVE to detect violations of constraints. We refer to these graph transformation rules as constraint rules. The constraint rules do not change anything, but whenever there is a match of a constraint rule in an instance graph, a constraint is violated and the instance graph does not correctly represent an instance model. Graph transformation rules to transform an instance graph defined by users can be blocked by using a higher priority for constraint rules. This is useful to block a transformation process once an invalid instance graph is detected. After transforming an instance graph, it must be verified that no constraint rule has a match in the instance graph to be able to transforming it back to an instance model.

First, we briefly list the constraint rules we identified previously, and then we explain them in more detail with examples. In Table 4.3 we have listed for each Ecore element and property the examples that demonstrate their representation, but also which constraint rule examples demonstrate the constraint rules required for them. The constraint rule examples are not necessarily directly applicable for the representation examples in the table, but only show how constraint rules can be constructed.

Several Ecore modelling elements and properties require constraint rules, as mentioned in sections 4.2 and 4.3. Table 4.5 lists all constraints for representations of Ecore modelling elements and properties. The constraint rules that are used to detect violations of these constraints are explained below, using examples.

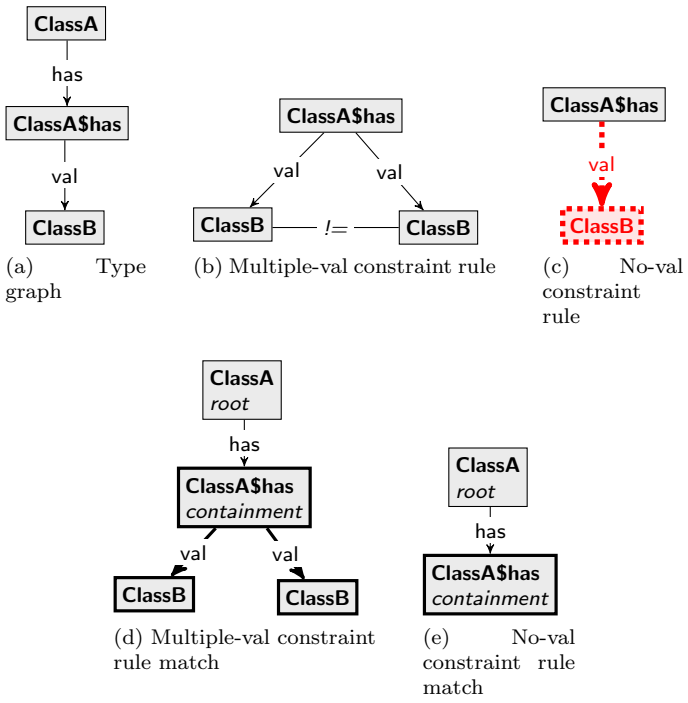
Ecore element	**Property**	**Constraint in instance graph**	**Examples**
ETypedElement	eContainingClass	Exactly one ⟨*name*⟩ edge from the container *EClass*.	Figure 4.17
ETypedElement	eType	≤ 1 outgoing *val* edges.	Figure 4.17
ETypedElement	eType	Only for *EReference*: *not 0 val* edges.	Figure 4.17
ETypedElement	unique	Instances with the same *container EClass* may not have the same *val* edge target.	Figure 4.18, 4.19
ETypedElement	lowerBound	Number of instances $\geq$ *lowerBound*.	Figure 4.20
ETypedElement	upperBound	Number of instances $\leq$ *upperBound*.	Figure 4.20
ETypedElement	ordered	Next edges connect and order all instances.	Figure 4.21
EClass	root	There must be exactly one *EClass* instance that is marked *root*.	Figure 4.22
EClass	cyclicity	Instances of *EClasses* may not cyclically contain each other.	Figure 4.23
EClass	container	An instance of an *EClass* must have exactly one container *EClass*, except for the root *EClass* which may not have one.	Figure 4.24
EClass	abstract	*abstract EClasses* may not be instantiated.	Figure 4.25
EReference	containment	Instances of *containment EReferences* must have a *containment* flag, other *EReferences* may not.	Figure 4.26
EReference	opposite	*EReferences* with an *opposite EReference* must have an outgoing and incoming *opposite* edge to and from the opposite *EReference*.	Figure 4.27
EReference	eKeys	Referred *EAttributes* must have unique concrete values.	Figure 4.28
EAttribute	iD	Concrete values must be unique for *EAttributes* of an *EClass* with the same *container EClass*.	Figure 4.29
EAttribute	unchangeable	Value must be equal to *defaultValue*.	Figure 4.30
EDataType	values	Some *EDataTypes* must be in a specific range of valid values.	Figure 4.31
EEnum	literals	Exactly one literal flag per instance of an *EEnum*.	Figure 4.32
EEnum	value	*EEnums* in instance graphs must be the value of a *EAttribute* and have exactly one incoming *val* edge.	Figure 4.33

Table 4.5: The constraints of instance graph representation of properties of Ecore elements.

4.4.1 ETypedElement

Even though in the type graph representation there is an outgoing *val* edge for each *ETypedElement*, we cannot enforce in the instance graph that there must be exactly one outgoing *val* edge for each instance of an *EReference*. For the representation of each instance of an *EAttribute* in the instance graph, there must be zero or one outgoing *val* edges. Zero is allowed for *EAttributes* because their *EDataType* may not be serializable, in which case there is no outgoing *val* edge. In **Figure 4.17** we show example constraint rules to enforce the number of *val* edges. The *multiple-val* constraint rule matches violations of the constraint that there may not be more than one outgoing *val* edge. The rule matches the instance graph in Figure 4.17d with the *ClassA$has* node type that has two outgoing *val* edges. In this example the *multiple-val* constraint rule is demonstrated for representations of *EReferences*, but for representations of *EAttributes* we need the same constraint rule. The *no-val* constraint rule matches violations where an instance of an *EReference* in an instance graph has no outgoing *val* edge, as shown in the instance model in Figure 4.17e.

A similar constraint is that each instance of an *ETypedElement* must have exactly one incoming edge from the node type representing the *container EClass*, labelled the *name* of the *ETypedElement*. The constraint rules to detect violations are similar to the constraint rules detecting violations that there must be



(a) Type graph

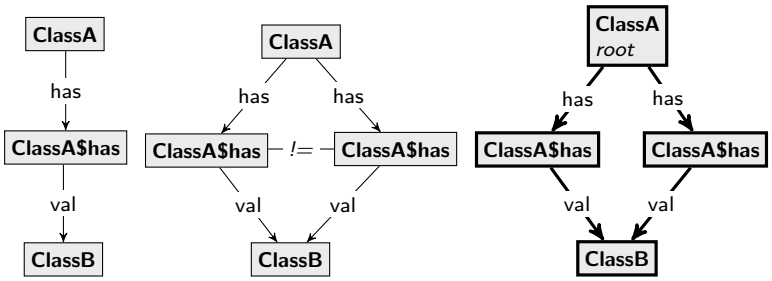
(b) Multiple-val constraint rule

(c) No-val constraint rule

(d) Multiple-val constraint rule match

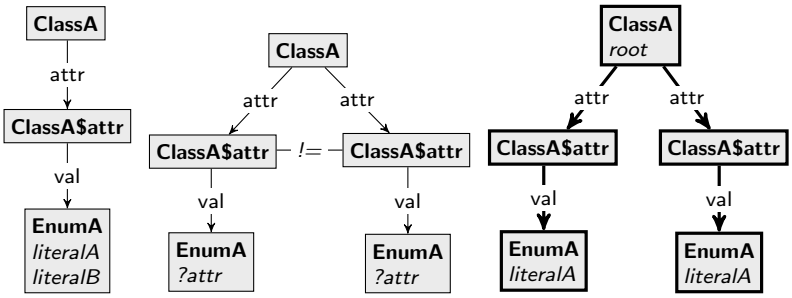
(e) No-val constraint rule match

Figure 4.17: Example of a constraint rule detecting an *ETypedElement* with two outgoing *val* edges, and a constraint rule detecting there is no *val* edge for an *EReference*. Constraint rules for the incoming *name* edge are done in the same way.



(a) Type graph (b) Unique constraint rule (c) Unique constraint rule match

Figure 4.18: Example of a constraint rule detecting two non unique values of an *EReference*, a violation if the *EReference has* is set to be unique.



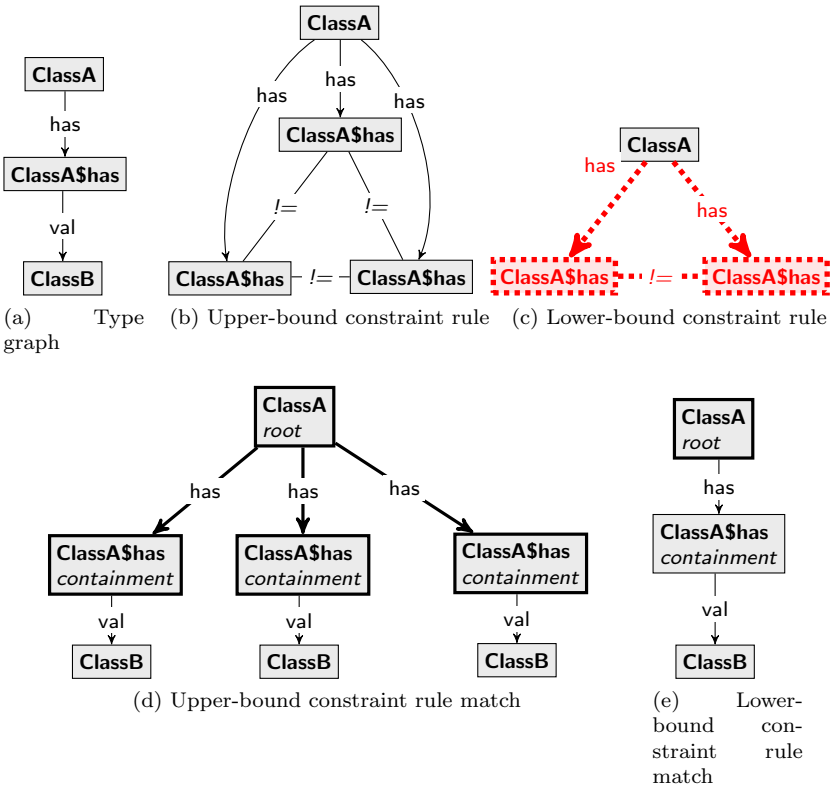
(a) Type graph (b) Unique constraint rule (c) Unique constraint rule match

Figure 4.19: Example of a constraint rule detecting two non unique values of an *EAttribute* of type *EnumA*, a violation if the *EAttribute attr* is set to be unique.

exactly one outgoing *val* edge. We do not give another example to demonstrate this.

When *ETypedElements* are set to be unique in the Ecore model, then values in instance models must be unique. Node types representing instances of an *ETypedElement* with the same *container EClass* may not have outgoing *val* edges to the same nodes, which can either be a representation of an *EClass* or of an *EDataType*. Consider the example in **Figure 4.18** for a *unique EReference has*. The constraint rule looks for two distinct node types labelled *ClassB$has* with the same *container* and the same *eReferenceType*. This rule matches the violation in the instance graph. For *EAttributes* with *eAttributeType* being a *serializable EDataType*, the rule looks the same.

However, when an *EAttribute* is of an *EEnum* type the constraint rule is different. When such an *EAttribute* is set to be unique, the node types that represent an *EEnum* value may not have the same flag to represent the same literal value. An example constraint rule to detect non-unique values of an *EAttribute* of *EEnum* type is given in **Figure 4.19**. *ClassA* has an *EAttribute attr* of type *EnumA*. The constraint rule compares the flag of the *EnumA* values



(a) Type graph

(b) Upper-bound constraint rule

(c) Lower-bound constraint rule

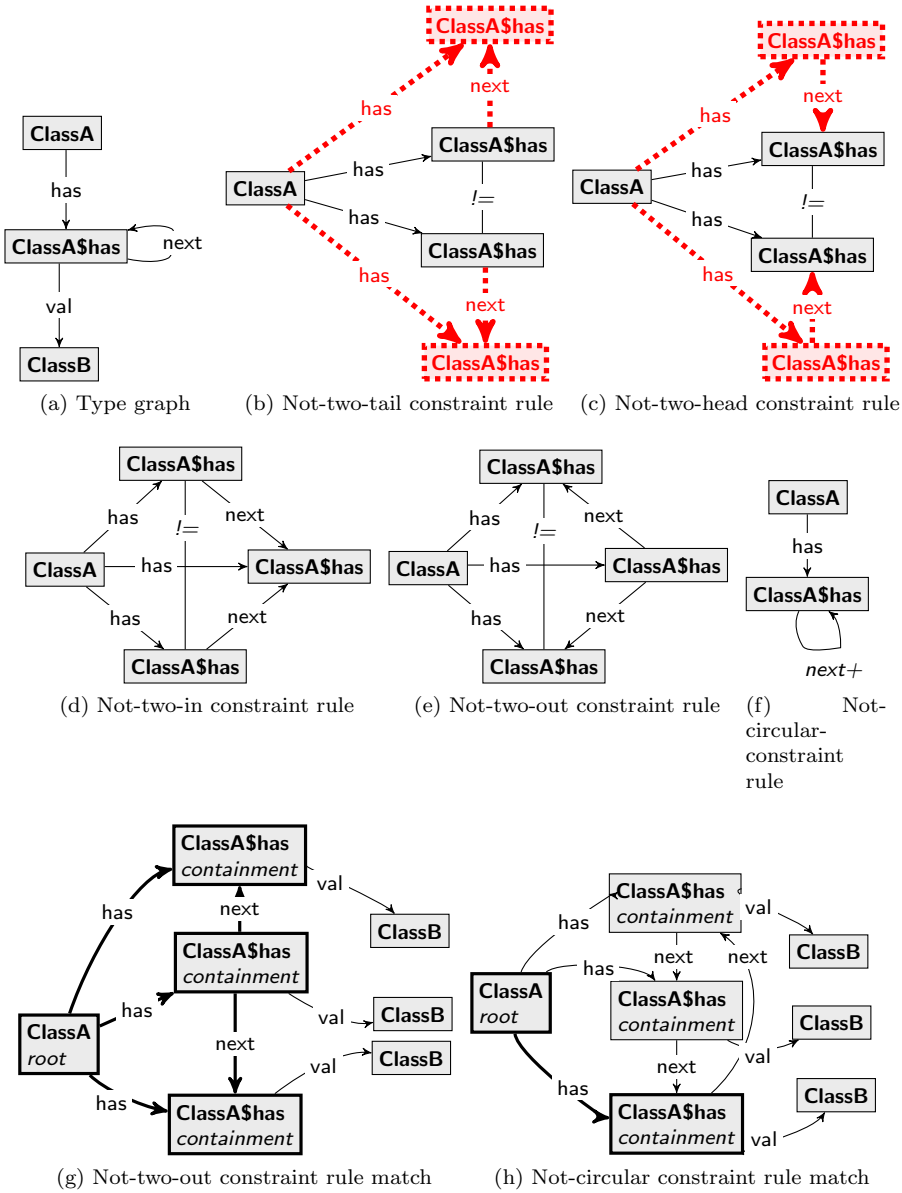
(d) Upper-bound constraint rule match

(e) Lower-bound constraint rule match

Figure 4.20: Examples of two constraint rules, one detecting violation of the *upperBound* and one detecting violation of the *lowerBound*. The *EReference has* has multiplicity *2..2*, so one or three instantiations are both violations.

using wildcards. If they are the same, then the rule matches and a violation is found. The instance graph in this example shows a violation, and the constraint rule matches it.

The *lowerBound* and *upperBound* properties indicate the multiplicity of an *ETypedElement*. For each *container EClass*, a number of instances of an *ETypedElement* lower than *lowerBound* or higher than the *upperBound* is a violation of the Ecore model. The number of node types in an instance graph cannot be enforced by typing a graph, so we need constraint rules to detect an invalid number of instances. Consider the example in **Figure 4.20**. The Ecore model has a multiplicity of *2..2* for the *EReference has*, but this is not visible in the type graph representation. Figure 4.20d and 4.20e show two instance graphs, one with three instances of *ClassA$has* and the other with just one instance. Both instance graphs violate the multiplicity of the Ecore model. The upper-bound constraint rule detects when an instance of *ClassA* has *upperBound+1* distinct instances of *ClassA$has* (three in the example), and this rule finds the violation in the instance graph. The lower bound constraint rule detects when an instance of *ClassA* does not have at least *lowerBound* distinct instances of *ClassA$has* (two in the example). It matches the instance of *ClassA* in the instance graph, which indeed violates the *lowerBound* since it has only one



(a) Type graph (b) Not-two-tail constraint rule (c) Not-two-head constraint rule

(d) Not-two-in constraint rule (e) Not-two-out constraint rule (f) Not-circular-constraint rule

(g) Not-two-out constraint rule match (h) Not-circular constraint rule match

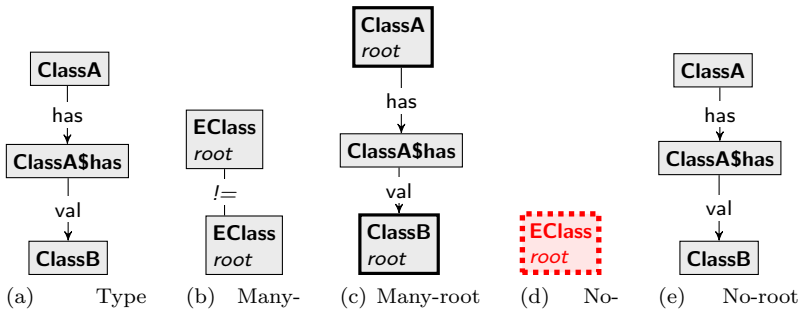
Figure 4.21: Example of constraint rules detecting invalid placement of *next* edges that represent the order of sibling instances of an *ETypedElement*.

instance of *ClassA$has*.

To represent an order of instances of *ETypedElements* in an instance graph, *next* edges are used. However, it cannot be enforced by typing that *next* edges between sibling *ETypedElements* are placed correctly and represent a deterministic ordering of the instances. Correct placement of *next* edges entails that all sibling elements must be connected through a sequence of *next* edges, starting from the head and ending at the tail, with no additional *next* edges. To detect a violation of this constraint, five constraint rules are needed, of which examples are shown in **Figure 4.21**. This example has five constraint rules and two matches of the constraint rules in instance graphs. The constraint is violated if there is more than one head or tail element, which indicates that *next* edges are missing or placed in the wrong direction. The instance graph in Figure 4.21g shows an example of this; the top and bottom instance of *ClassA$has* both only have an incoming *next* edge and no outgoing *next* edge, so this graph has two tail elements which is matched by the *not-two-tail* constraint rule. Next, the constraint is also violated if some instance of *ClassA$has* has two outgoing or two incoming *next* edges. The instance graph also is an example of this, since the middle instance of *ClassA$has* has two outgoing *next* edges, and is matched by the *not-two-out* constraint rule. An instance graph with three instances of *ClassA$has* with an additional *next* edge from the head to the tail element would also be matched by the *not-two-out* and the *not-two-in* constraint rules. Finally, *next* edges must form a non circular sequence of instances of *ClassA$has*. The instance graph in 4.21h shows an example of a circular sequence, and the *not-circular* constraint rule matches this violation. This constraint rule looks for instances of *ClassA$has* that refer to themselves through a sequence of one or more *next* edges. Instances of *ClassA$has* with a *next* self edge are also matched by this rule.

4.4.2 EClass

Instance models must have exactly one root *EClass* element. In instance graphs the *EClass* instance that represents the root must have a *root* flag, which is



(a) Type graph (b) Many-root constraint rule (c) Many-root constraint rule match (d) No-root constraint rule (e) No-root constraint rule match

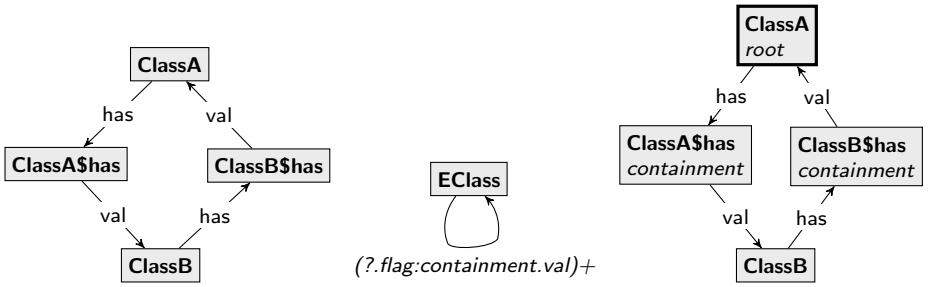
Figure 4.22: Example of constraint rules that detect when there is not exactly one root element in an instance graph .

inherited from the *EClass* node type the *EClass* is a subtype of. Consider the example in **Figure 4.22**, it shows two violations. Figure 4.22c shows an instance model that has two *root EClasses*, and the instance model in Figure 4.22e has no *root EClass*. *The many-root* constraint rule matches the former violation, the *no-root* constraint matches the latter.

Cyclicity is the phenomenon that an instance of an *EClass* contains itself through a series of *containment EReferences*. This is not allowed, but cannot be prevented by typing a graph. **Figure 4.23** shows an example where in the type graph *ClassA* has a *containment EReference* to *ClassB*, and *ClassB* has a *containment EReference* to *ClassA*. In the instance graph both *ClassA* and *ClassB* contain an instance of an *EClass* that is also its container *EClass*, which is a violation. The constraint rule looks for instances of *EClasses* that have a *(?.flag:containment.val)+* path to themselves. Each step of this path first matches any edge which will match the <*name*> edge, then the flag which matches an instance of a *containment EReference*, and then a *val* edge. Each step can reach an instance of an *EClass* through a *containment EReference*, so if an instance of an *EClass* can reach itself through one or more of these steps, it is cyclically contained and the constraint is violated. Finding violations of this constraint is not possible without being able to match an instance of a *containment EReference*, which is the reason the flags are needed.

In Ecore instance models, every instance of an *EClass* must have exactly one *container EClass*, except for the root element of a model which may not have a *container EClass*. This constraint can be split in three parts, the first part being that an instance of an *EClass* apart from the *root EClass* must have a *container EClass*, the second that no instance of an *EClass* may have more than one *container EClass*, the third that the *root EClass* may not have a *container EClass*. Examples for constraint rules for this constraint are demonstrated in **Figure 4.24**.

The first part of the constraint, that an instance of an *EClass* that is not root must be contained by a *container EClass*, is violated in the instance graph in Figure 4.24c. It shows an instance of *ClassB* that does not have a container, since the node type *ClassA$refer* does not represent a *containment EReference*.

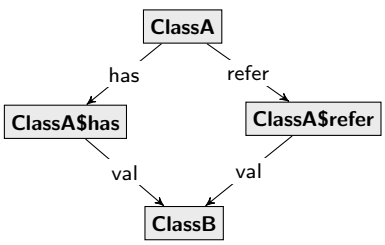


(a) Type graph (b) Cyclicity constraint rule (c) Cyclicity constraint rule match

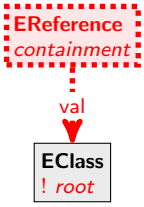
Figure 4.23: Example of a constraint rule that detects instances of *ClassB* that are cyclically contained by itself, a violation.

The *no-container* constraint rule finds instances of an *EClass* that are not root and have no incoming *containment EReference*. It matches the violation in the instance graph.

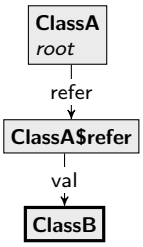
The *many-container* constraint rule detects violations of the second part of the constraint. No instance of an *EClass* may be contained by more than one container. This property is called the *unsharedness* property, since an element may not be shared by more than one container element. The instance graph in Figure 4.24e has a violation, the instance of *ClassB* has two container *EClasses*, even though this happens to be the same *ClassA* instance. The constraint rule



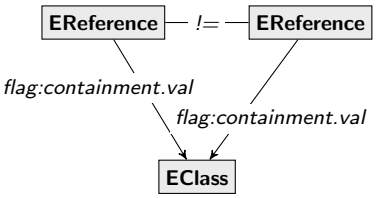
(a) Type graph



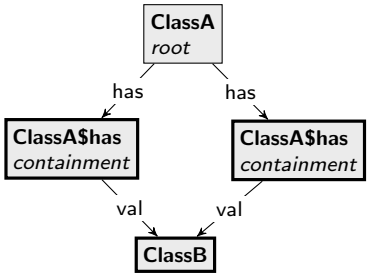
(b) No-container constraint rule



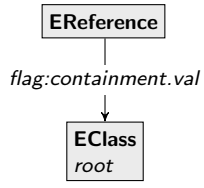
(c) No-container constraint rule match



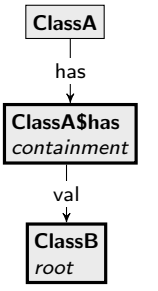
(d) Many-container constraint rule



(e) Many-container constraint rule match



(f) Root-container constraint rule



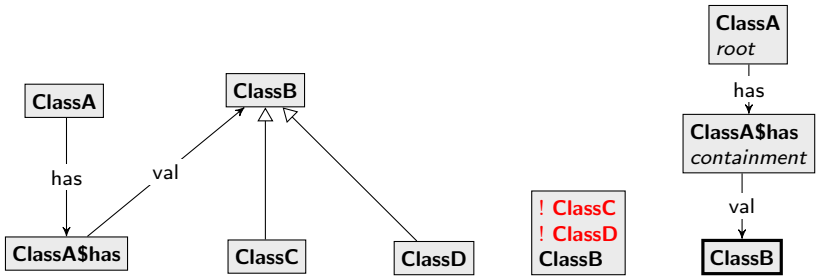
(g) Root-container constraint rule match

Figure 4.24: Examples of constraint rules that detect when an instance of an *EClass* does not have exactly one container, or in case of the root *EClass* has one container.

looks for two distinct *EReference* instances that can reach an *EClass* through the path *flag:containment* and then *val*. The flag matches *containment EReferences* and the *val* matches the edge connecting the *EReference* to the *EClass*. More intuitively would be to add the flag to the *EReference* node types and only label the edges with *val*. However this is not possible, since in the type graph, there is no *val* edge between the types nodes labelled *EClass* and *EReference* and the rule would not be typed correctly. When using a regular expression, this is not required as long as some subtypes of *EClass* and *EReference* have this *val* edge.

The *root-container* constraint rule is similar to the *many-container* constraint rule. It now looks for an instance of an *EClass* that is root, and also has a container. The regular expression used for this is the same as for the *many-container* constraint. The instance graph in Figure 4.24g shows a violation, since the instance of *ClassB* is marked to be root, but it is contained by the instance of *ClassA*.

There may not be instantiations of *abstract EClass* in an instance model, and hence not in the instance graph representation of instance models. However, node types representing *abstract EClasses* occur in the type graph and can therefore also occur in instance graphs. Constraint rules are needed to detect violations where node types representing *abstract EClasses* occur in instance graphs. An example constraint rule is given in **Figure 4.25**. *ClassB* represents an *abstract EClass* even though this is not visible in the type graph, and *ClassC* and *ClassD* are not *abstract* and are subtypes of *ClassB*. The instance graph shows a violation, since there is an instance of *ClassB*. Looking for a match of *ClassB* is not enough, since such a rule would also match any subtypes of *ClassB*. The constraint rule looks for a match of *ClassB*, but not any of its direct subtypes. This constraint rule matches the violation in the instance graph, but would not find a match in an instance graph with an instance of *ClassC* or *ClassD* instead of *ClassB*.



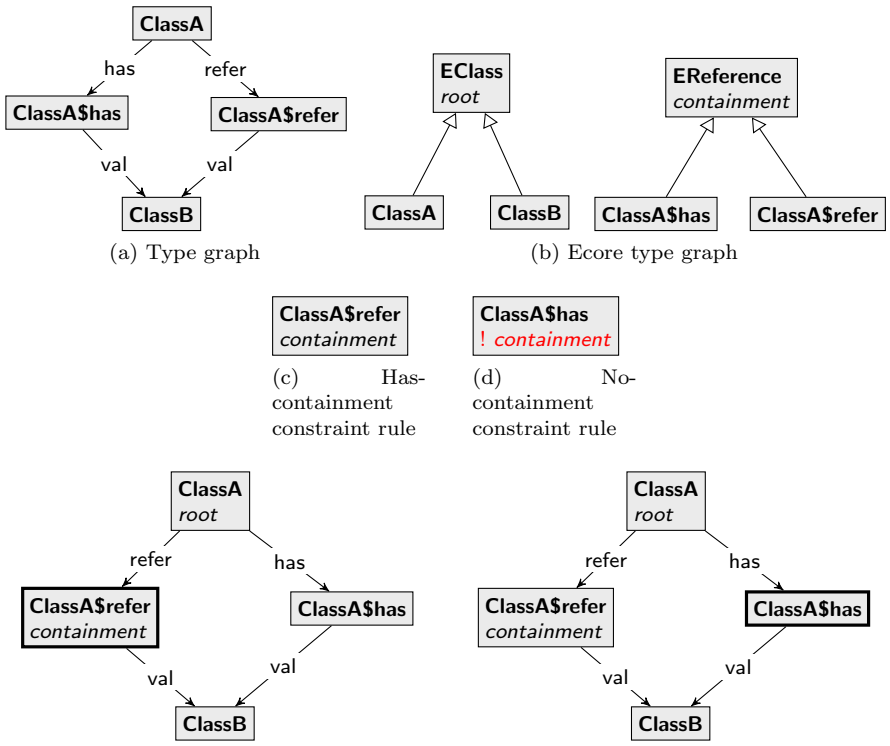
(a) Type graph (b) Constraint rule (c) Constraint rule match

Figure 4.25: Example of a constraint rule that detects instantiations of the *abstract EClass ClassB*.

4.4.3 EReference

EReferences are subtypes of the node type labelled *EReference*. This *EReference* node type has a *containment* flag. Every instance of an *EReference* that represents a *containment EReference* must have this flag in an instance model, and other *EReferences* may not. Since all *EReferences* are subtypes of *EReference* in the type graph and therefore can have the *containment* flag in an instance graph, we use a constraint to enforce that only representations of *containment EReferences* have it. **Figure 4.26** demonstrates violations of this constraint. There are two constraint rules, the *has-containment* constraint rule and the *no-containment* constraint rule. The *has-constraint* rule is required for each *non containment EReference* in the Ecore model, and it finds instances in the instance graph that do have a *containment* flag. Figure 4.26e shows such a violation, and that it is matched by the *has-containment* constraint rule. The *no-containment* constraint rule in the example finds instances of the *containment EReferences has* that do not have a *containment* flag, like the match in Figure 4.26f.

When an *EReference* has an *opposite EReference* in the Ecore model, then



(a) Type graph

(b) Ecore type graph

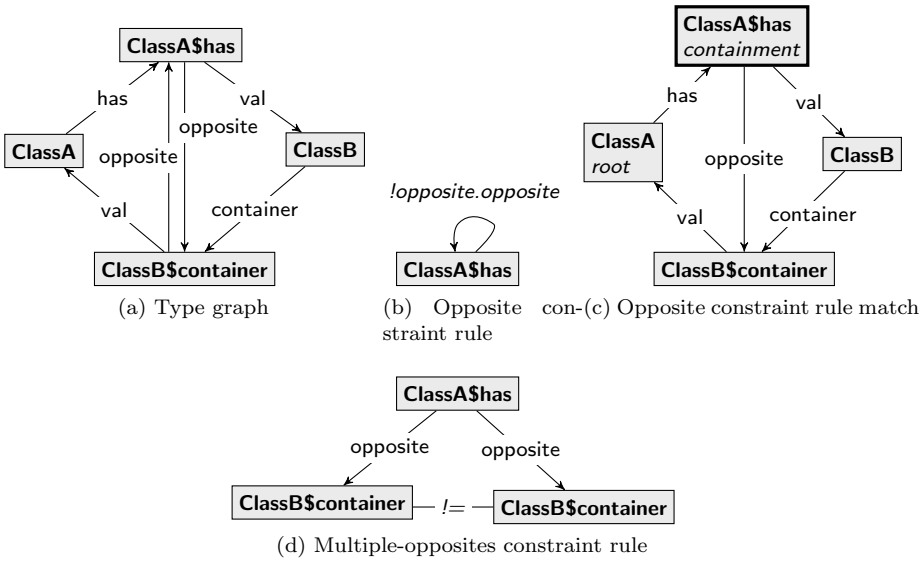
(c) Has-containment constraint rule

(d) No-containment constraint rule

(e) Has-containment constraint rule match

(f) no-containment constraint rule match

Figure 4.26: Example of constraint rules that detect incorrect placement of *containment* flags in an instance graph. *ClassA$has* should have a flag and *ClassA$refer* should not.



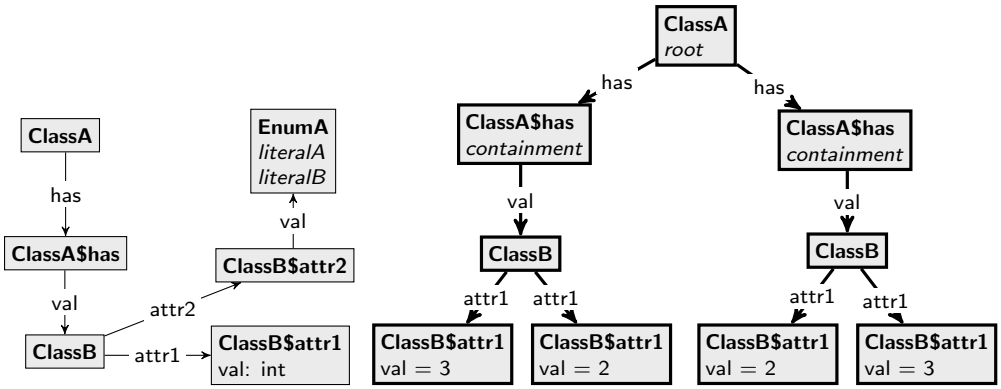
(a) Type graph (b) Opposite straint rule con- (c) Opposite constraint rule match

(d) Multiple-opposites constraint rule

Figure 4.27: Example of constraint rules that detects an instance of the *EReference has* without *opposite* edges or with more than one outgoing *opposite* edge.

every instance of this *EReference* must have an opposite in the instance model. This means that in the instance graph representation every node type representation of an *opposite EReference* must have an outgoing *opposite* edge. Additionally, since the *opposite* property is symmetrical, there must be *opposite* edges in both directions in the instance graph. **Figure 4.27** shows an example of a violation of this constraint. *ClassA$has* and *ClassB$container* represent *opposite EReferences*, and there is one instance of each of them in the instance graph. There must be an *opposite* edge in both directions, but one is missing. The constraint rule detects instances of *ClassA$has* that do not have a sequence of two *opposite* edges to itself. Since self edges labelled *opposite* are not possible due to typing, the sequence has to be to an instance of *ClassB$container* and back. When there is either no outgoing or no incoming *opposite* edge, this constraint rule will match, detecting the violation. The *multiple-opposites* constraint rule in Figure 4.27 detects instances of *ClassA$has* that has two or more outgoing opposites. There must be exactly one opposite. This constraint rule detects violations. An example match is not given, but this rule is similar to the *multiple-val* constraint rule.

The *eKeys* property of *EReferences* can refer to several *EAttributes* of the *EClass* that is the target of the *EReference*. The combined values of these referenced *EAttributes* in an instance model or graph must uniquely identify a referenced instance of the *EReference*. **Figure 4.28** shows an example; in the type graph we see that *ClassB* has two *EAttributes*, *attr1* and *attr2*. These two *EAttributes* are set to be the *eKeys* of the *EReference has*, although this is not apparent in the type graph. In the instance graph, the values of *attr1* and *attr2* are the same for each instance of *ClassB$has*. *attr1* has many-multiplicity, and



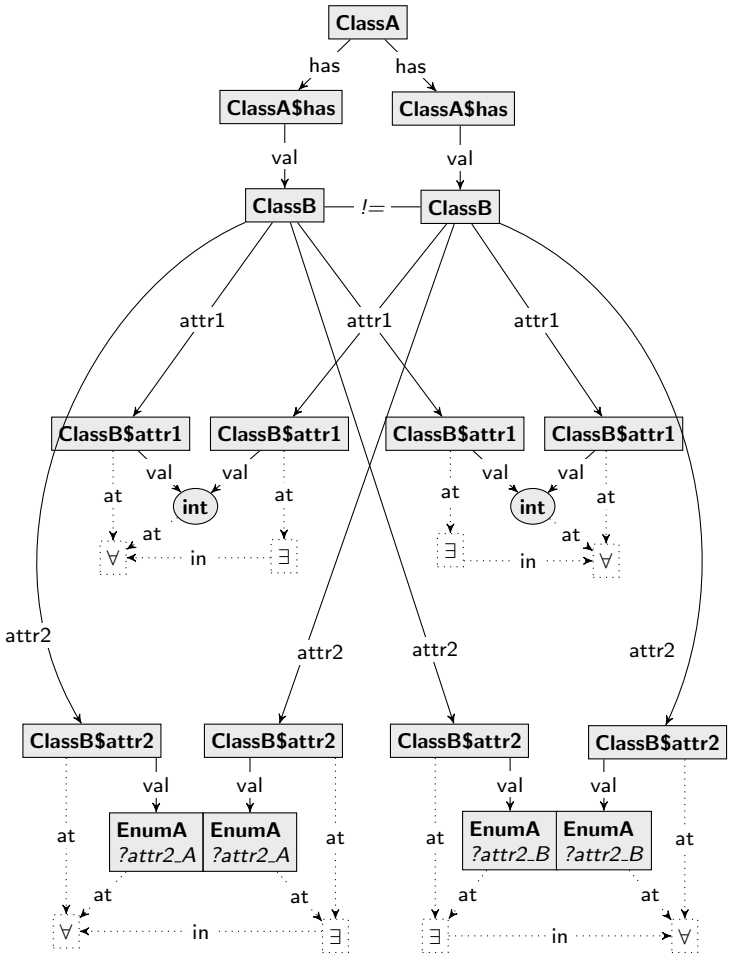


Figure 4.28: Example of a constraint rule that detects invalid values of *EAttributes attr1* and *attr2* that are set to be the *eKeys* of *EReference has*. Both *EAttributes* are unique and not ordered.

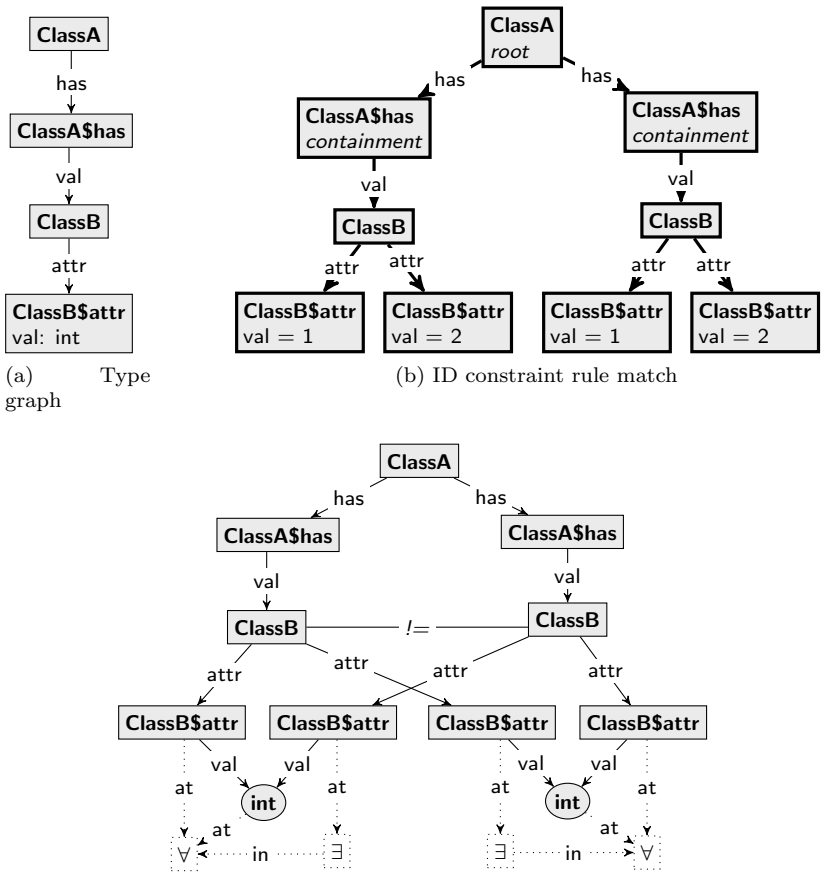
its values for both instances is the collection {2,3}. *attr2* is of an *EEnum* type, and in the instance graph it has no value for both instances of *ClassA$has*, which also counts as being the same. For every two distinct target *ClassB*instances of sibling *EReference has* instances, the constraint rule compares the values of *attr1* and *attr2*. For each *EAttribute*, the constraint rule checks if all values of that *EAttribute* from one instance of *ClassB* also occur in the collection of values of that *EAttribute* for the other instance of *ClassB*, and vice versa. If this is the case, then both collections of values for the *EAttribute* are the same, and if this is the case for all *EAttributes* then a violation is found and the constraint rule matches it. Matching same values of an *EEnum* is done using wildcards, as shown before in 4.19.

In order to detect violations with constraint rules, all *EAttributes* that *eKeys* refers to must be serializable, unique and unordered. If either of these properties is false for any of the *EAttributes*, then detecting violations is done by the transformation tool when transforming an instance graph back to an Ecore instance model. First, when an *EAttribute* is not serializable, values are not supported in the graph representation. This means that we cannot compare values, and cannot determine if this *EAttribute* is part of a unique identification of an instance of an *EReference*. Next, if an *EAttribute* is not unique, then values occuring multiple times cannot be identified. If the value of *attr1* of one instance of *ClassB* would be {2,2} and of another instance {2}, the constraint would still match it because all values of one collection occur in the other collection, even though the collections of values are not the same. Finally, it is not possible in GROOVE to compare ordered lists of values of variable length, so when an *ordered EAttribute* is part of the *eKeys* property, a constraint rule for the *eKeys* property is not supported.

4.4.4 EAttribute

When an *EAttribute* is set to be the *iD* of an *EClass*, then values of the *EAttribute* uniquely identify instances of this *EClass* within its *container EClass*. Consider the example in **Figure 4.29**. In this case it means that values of *ClassB$attr* must be unique for instances of *ClassB* with the same *container ClassA*. The instance graph in the example violates this, because the value of both instances of *ClassB$attr* is the collection {1,2}. The constraint rule is very similar to the constraint rule for *eKeys*, except that only one *EAttribute* of an *EClass* is set as the *iD*. It matches the violation in the instance graph. If the *EAttribute* would be of an *EEnum* type, then wildcards would be used as shown in Figure 4.28 and 4.19. Additionally, as with the constraint rule for *eKeys*, a constraint rule is only supported if the *EAttribute* that is the *iD* is serializable, unique and not ordered.

When the *changeable* property of an *EAttribute* is set to *false*, it may not have a different value as its *defaultLiteralValue* in an instance model and instance graph representation. An example of a violation is given in **Figure 4.30**. Even though not represented in the type graph, *attr* is set to *unchangeable* and its *defaultLiteralValue* is *12*. In the instance graph however the value is *10*, which is a violation. The constraint rule compares the value of the *ClassA$attr*

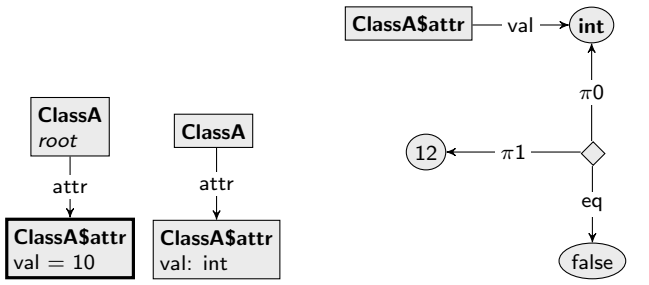


(a) Type graph

(b) ID constraint rule match

(c) ID constraint rule

Figure 4.29: Example of a constraint rule that detects a violation of the valid values of *ClassB$attr*, which is set to be the *iD* of *ClassB* and must uniquely identify an instance of *ClassB* within its container *EClass ClassA*. The *EAttribute attr* has a many-multiplicity.



(a) Unchangeable constraint

(b) Type graph match

(c) Unchangeable constraint rule

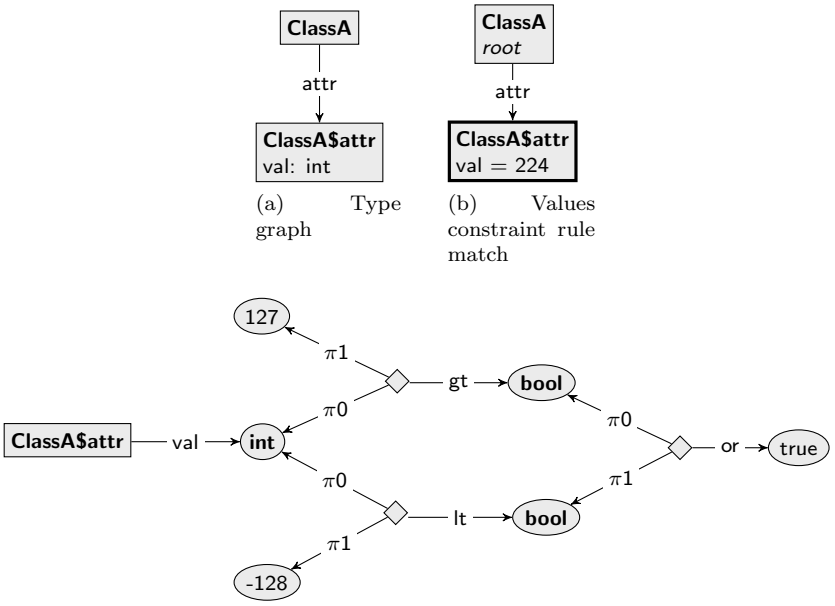
Figure 4.30: Example of a constraint rule that detects a violation of an invalid value of the *EAttribute attr*. It is *unchangeable* and has a *defaultValue 12*.

to *12*, and when they are unequal a violation is found and this instance of *ClassA$attr* is matched, as seen in the example.

4.4.5 EDataType

EDataTypes are represented in the graph representation by attributes in GROOVE, which can be either an integer, real, boolean or string. However, the range of valid values in the instance graph can be larger than the range of valid values in the Ecore model. Constraints are used to enforce that concrete values in the instance graph are within the range of valid values. Table 4.6 lists which *EDataType* representations need constraints to enforce that values in the instance graph are valid. When the big algebra in GROOVE is required for an Ecore model, more *EDataTypes* need constraints for valid values. Now consider the example in **Figure 4.31**. The node type *ClassA$attr* represents an *EAttribute* of type *EByte*, which is an 8-bit signed integer. In the instance graph the value of this *ClassA$attr* is *224*, which exceeds the range of valid values, and is therefore a violation. The constraint rule consists of three comparisons. The first tests if the value is higher than the upper limit of the range of valid values, $val > 127$ in this example. The second tests if the value is lower than the lower limit of the range of valid values, $val < -128$ in this example. If either of these unequalities is *true*, checked in the third comparison, then a violation is found and this constraint rule matches.

For the instance graph representation of an *EDate* a specific string format is used with the year, month, day, hours, minutes, seconds, milliseconds and timezone. We cannot check in GROOVE if a string value representing an instance of an *EDate* is correctly formatted. Additionally, when the big algebra family is enabled for a graph grammar, valid values for *EFloat* and *EDouble* must be checked



(a) Type graph

(b) Values constraint rule match

(c) Values constraint rule

Figure 4.31: Example of a constraint rule that detects invalid values of an integer attribute that represents and *EByte*. Valid values of an *EByte* are -128 .. 127.

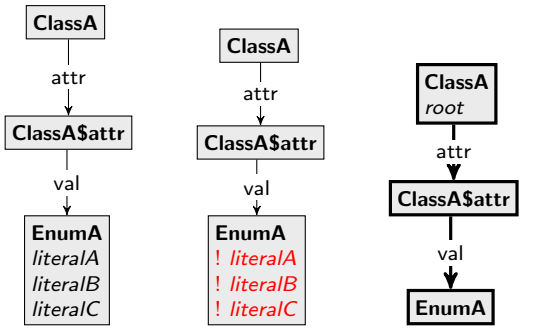
since their bounds of values are smaller than that of *java.lang.BigDecimal*. Because of the complex nature of minimum and maximum values of decimal values, this cannot be done with constraint rules. These constraints are instead checked by the Java program when transforming an instance graph back to an instance model.

***Default* algebra family**	***Big* algebra family**
EByte	EByte
EByteObject	EByteObject
EShort	EShort
EShortObject	EShortObject
EChar	EChar
ECharObject	ECharObject
EDate	EDate
	EInt
	EIntegerObject
	ELong
	ELongObject
	EFloat
	EFloatObject
	EDouble
	EDoubleObject

Table 4.6: *EDataTypes* for which valued values in the graph representation are bound by constraints in the cases of the *default* and *big* algebra family.

4.4.6 EEnum

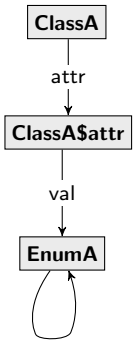
EEnums are special *EDataTypes* that can be the type of *EAttributes*. A value is represented by an instance of the node type that represents the *EEnum* with a flag that represents the *EEnumLiteral* value of the *EEnum*. Each instance of an *EEnum* in the instance graph represents one value for an *EAttribute*, and must therefore have exactly one flag representing the *EEnumLiteral* value. We cannot enforce by typing that there must be exactly one flag for each instance of an *EEnum*. The example in **Figure 4.32** shows two violation examples. The instance graph in Figure 4.32c has an instance of *EnumA* that has no flag to represent a literal value. The *no-literal* constraint rule looks for an instance of an *EnumA* that has no literal flag, and it matches the violation in the instance graph. Next, the instance graph in Figure 4.32e shows another violation, since it has two literal flags *literalA* and *literalC*. The *many-literals* constraint rule looks for instances of *EnumA* that has two flags using a regular expression. It matches when there is a sequence of two different flags on an instance of an *EnumA*. When an instance graph has any two or moreflags on an instance of *EnumA*, as in the example, the rule matches.



(a) Type graph

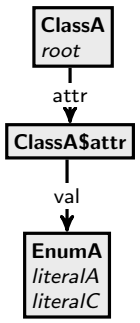
(b) No-literal constraint rule

(c) No-literal constraint rule match



flag:literalA.(flag:literalB|flag:literalC)|flag:literalB.flag:literalC

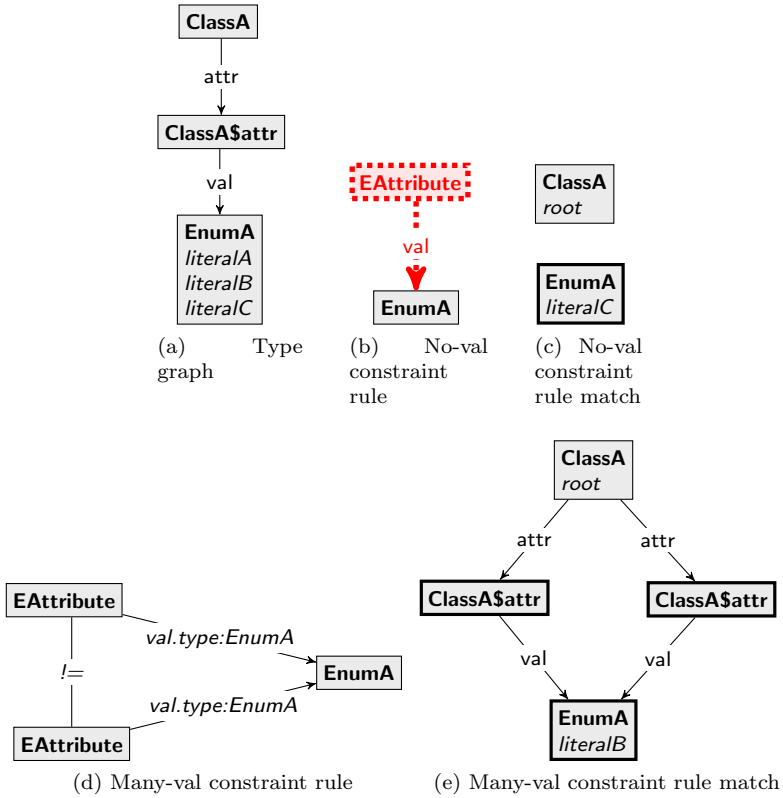
(d) Many-literals constraint rule



(e) Many-literals constraint rule match

Figure 4.32: Examples of two constraint rules, the first detects a violation that an *EEnum* representation has no flag to represent a literal value, the second that it has two flags that represent literal values.

EEnums are *EDataTypes* and must the value of exactly one *EAttribute*. Therefore, they must have exactly one incoming *val* edge in instance graphs. **Figure 4.33** shows two constraint rules that detect violations. The *no-val* constraint rule matches instances of an *EnumA* that is not the value of an *EAttribute* and it matches the violation in Figure 4.33c. The *many-val* constraint rule matches instances of *EnumA* that is the value of more than one *EAttribute*, also a violation. A path expression is used to match a *val* edge from any *EAttribute* node type, since *EAttribute* node types themselves have no outgoing *val* edge. Without the path expression, this rule would not be typed correctly. It matches the violation in Figure 4.33e.



(a) Type graph

(b) No-val constraint rule

(c) No-val constraint rule match

(d) Many-val constraint rule

(e) Many-val constraint rule match

Figure 4.33: Examples of two constraint rules, that detect violations of *EEnums* in instance graphs that do not have exactly one incoming *val* edge.

4.5 Conclusion

In this chapter we defined a mapping of Ecore models to type graphs and instance models to instance graphs. The graph representations are fully supported by GROOVE. Instance graphs can be transformed to other instance graphs in GROOVE, and if the resulting instance graph is typed correctly and does not violate any constraints, it is guaranteed to validly represent an instance model of the original Ecore model.

For our mapping, we examined all modelling elements and their properties and relations in the Ecore metamodel. These elements are the building blocks of Ecore models. The validity of an instance graph as a representation of an instance model can not be enforced by just using type graphs. When type graphs cannot be used, we defined constraints that must hold for an instance graph to be a valid representation. Detecting constraint violations is done using constraint rules. These are graph transformation rules that match constraint violations in a graph, but do not alter the graph. We have given graph representation examples and constraint rule examples for all supported modelling elements.

Using our mapping, it is possible to transform instance models to a graph representation and back to an instance model without losing any information. Additionally, the graph representations are fully supported by the feature set of a specific graph transformation tool, GROOVE. This has not been accomplished before.

There are a few limitations to some supported elements. Values of *EDatatypes* can only be supported in instance models when they can be serialized and defined in the Ecore metamodel. Other *EDatatypes* cannot be serialized using XMI, therefore the XML file will not contain values for these *EDatatypes* and we cannot support them. This is a limitation of the Eclipse Modelling Framework and not of our approach.

Some *EDatatype* have a smaller bound of valid values than what can be represented by integer or real attributes in GROOVE. Constraint rules are used to detect invalid values of these *EDatatypes*. However, *EDate* is an *EDatatype* with a complex string representation, of which valid values cannot be enforced within GROOVE. Additionally, when the big algebra family is used in GROOVE, constraints are also put on the valid values*EFloat* and *EDouble*. However, they have complex range of valid values of which violations cannot be detected with constraint rules. For these cases, invalid values are detected by the transformation tool.

Another limitation is with the *iD* and *eKeys* properties. If they target multi-valued *EAttributes* that are ordered and/or not unique, we cannot detect violations in GROOVE using constraint rules or typing. Violations are instead checked by the transformation tool when transforming an instance graph back to an instance model.

Three related pairs of elements are not supported, either because they have no impact on instance models or because they cannot be supported. The unsupported pairs of elements are *EOperations* and *EParameters*, *EAnnotations* and *EStringToStringMapEntries*, and *EGenericType* and *ETypeParameter*.

EOperations can operate on *EParameters* passed to it and optionally return some value. They are not instantiated in instance models nor do they operate on instance models, and only function as methods in generated Java programs. They are not part of the structure of instance models, and therefore do not need a representation. *EParameters* are only used by *EOperations* and therefore also do not require a representation.

EAnnotations are used to add textual information to elements of a model. *EStringToStringMapEntries* are used by *EAnnotations* as key and value pairs

of strings. They are not instantiated in instance models, and do not require a graph representation. Using *EAnnotations* it is possible to add constraints to an Ecore model to put a bound on valid instance models. However, such constraints have no semantics, and can therefore not be translated to constraint rules.

Finally, *EGenericTypes* and *ETypeParameters* can define *EClasses*, *EOperations* or *EAttributes* to be generic. Generic *EClasses* do not impact instance models, and *EOperations* are not supported in general, so in those cases they need no graph representation. For generic *EDatatypes*, in some cases it could theoretically be supported, of which we have given an example in section 4.2. However, serialization of values of generic *EAttributes* is never possible with EMF. If serialization using XMI is not possible, it cannot be supported.

Chapter 5

Ecore2groove transformation tool

In this chapter we describe *ecore2groove*, a transformation tool to perform the transformations described in chapter 4. This tool can perform 4 transformation tasks:

- Transform Ecore models to GROOVE type graphs.
- Transform Ecore models to GROOVE contraint rules.
- Transform Ecore instance models to GROOVE instance graphs.
- Transform GROOVE instance graphs back to Ecore instance models.

Ecore2groove is written in Java and has been tested to work in Windows and Linux. The EMF plugins for Eclipse and GROOVE are both written in Java and provide methods to interact with Ecore models and graphs. By using these we can focus on implementing the transformation itself and do not have to concern ourselves with the file formats. Ecore models and instance models are object oriented. Elements like classes can contain other elements like classes or attributes, or refer to other elements. An in-memory structure to represent this can efficiently be made using an object oriented programming language, and then iterating over the elements and the relations between them to transform a model in a graph representation. Another logical approach would be to use XSLT [48] to define the transformation of Ecore models to graphs, since both are stored as XML files, and XSLT is a W3C recommendation to transform XML documents into other XML documents. However, XSLT is very suited to transform XML documents with a tree structure, but not models stored as XML files because of the heavy use of cross references [41]. Model transformations specified using XSLT would have a very complicated composition of recursion, and would be very inefficient because of hopping through XML files looking for values of references.

Section 5.1 explains how the tool should be used, which arguments it expects and what output it will generate. Section 5.2 explains the implementation of the transformation tool and how the four transformations tasks are done. Section 5.3 demonstrates the tool by using an example Ecore and instance model, and then applying graph transformation rules on the graph representation of the instance model.

5.1 Tool usage

The transformation tool is created as a subpackage of GROOVE, and available from the subversion repository of GROOVE [17]. When running the tool without arguments, it outputs a list of arguments it expects to perform a transformation task. Figure 5.1 shows a diagram of the tool, and which inputs produce which outputs.

For an Ecore model and a list of zero or more Ecore instance models, it will generate a GROOVE grammar that contains both type graphs, constraint rules and one instance graph for each Ecore instance model. This transformation can be invoked by running the tool with two arguments or more. The first argument must be an Ecore model and the last argument must be the location of a GROOVE grammar, which will be created if it does not already exist. GROOVE requires this directory to end in *.gps*. There can by any number of arguments in between, each of which should be the location of an instance model of the Ecore model. Each of these instance models will be transformed to an instance graph which will be put in the GROOVE grammar. The name of the instance graph will be the same as the file name of the instance model. When a GROOVE grammar already exists, any instance graphs that are created but were already in the grammar will be overwritten. Other instance graphs are not touched. All type graphs in the grammar that already exist are deleted, and new type graphs are created. Furthermore, all constraint rules that were in the grammar will be deleted and replaced with newly generated constraint rules. This is to make sure that constraint rules for elements that are no longer in the model or no longer valid for model elements do not remain in the grammar. Any graph transformation rules that are not constraint rules, ie. do not start with *"constraint - "*, will not be changed or deleted. The line below shows the Linux command to transform an Ecore model and instance models to a GROOVE grammar.

```
./ecore2groove.jar <Ecore model> <Ecore instance models> <GROOVE grammar>
```

For a GROOVE grammar and an Ecore model, the tool will generate one instance model for each graph in the grammar. This transformation is invoked by running the tool with three arguments. This first argument must be the GROOVE grammar that serves as input, the second argument must be an Ecore model and the third argument must be a directory location where the generated instance models should be stored. The type graph in the GROOVE grammar must be a representation of the Ecore model, and all instance models

must be typed correctly and not violate any constraints. If these properties hold, then for each instance graph in the GROOVE grammar an Ecore instance model with the same name is generated and stored into the specified directory. The line below shows the Linux command to transform instance graphs in a GROOVE grammar to Ecore instance models.

```
./ecore2groove.jar <GROOVE grammar> <Ecore model> <Instances location>
```

5.2 Tool implementation

This section will explain the structure of the *ecore2groove* transformation tool, and the algorithms for each of the four transformations steps. Figure 5.2 shows an Ecore model of *ecore2groove*. This model only shows how the different classes interact, and not internal behaviour and private methods.

There are six connected classes, and one other class *GraphLabels* that only provides static methods used by the other classes. *Transform* contains the *main* method, which decides with transformation has to be done, based on the command line arguments. This class initializes the *ModelHandler* and passes the location of the Ecore model to its constructor. After the *ModelHandler* has been initialized and when transforming from Ecore to GROOVE, *Transform* initializes a *TypeGraphRep* for the representation of the Ecore model, a *ConstraintRules* which will create the constraint rules to put in the GROOVE grammar, and for each Ecore model an *InstanceGraphRep* to represent it. These representations are then written to the GROOVE grammar. When transforming from GROOVE to Ecore, after initializing the *ModelHandler*, *Transform* will only initialize the *InstanceModelRep* for each graph in the GROOVE grammar. These will be the Ecore instance model representations of an instance graphs.

The *ModelHandler* loads the Ecore model that was passed to its constructor when it is initialized. It then iterates over the elements of this Ecore model, and

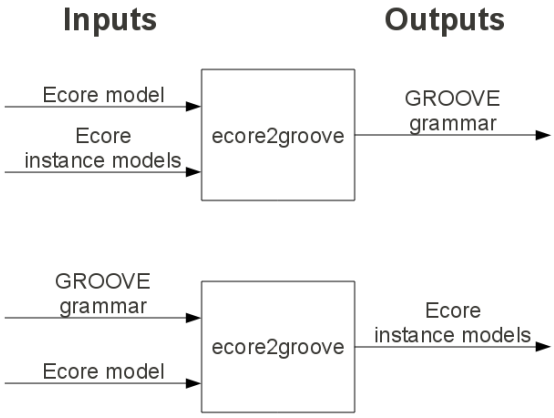


Figure 5.1: Diagram of the ecore2groove tool and which inputs and outputs are supported.

fills the *Vector* attributes with the respective elements from the Ecore model. It provides a method *loadInstance* to load an instance model, a method *createModel* to create a new instance model and a method *saveModel* to store an instance model. When loading an instance model, the *ModelHandler* iterates over its elements and fills the *iClasses* attribute with *EObjects* that are instances of *EClasses*. Finally, there are two string attributes *eClassType* and *eReferenceType*. They default to *type:EClass* and *type:EReference* to be used as node type labels in the Ecore type graph, but when these names are not safe, underscores are added until they are. They are safe when there is no element in the Ecore model that has the same name as the string part after *type:*.

When initialized, the *TypeGraphRep* retrieves all Ecore model elements from the *ModelHandler* that were put in the *Vectors*. It then generates the *typeGraph* and *ecoreTypeGraph*. The code for doing this is explained in subsection 5.2.1. The class provides get methods for the type graphs.

ConstraintRules has a set for *DefaultGraphs* which are the constraint rules, and a map from *DefaultGraphs* to *EStrings*. This map is to add names to constraint rules to be used in the GROOVE grammar. When the class is initialized, it iterates over the *Vectors* of elements of the Ecore model and creates constraint rules where needed. Code for this is also shown in section 5.2.1.

Next, and still part of the transformation from Ecore to GROOVE, is the *IntanceGraphRep*. This class is initialized by *Transform* for every instance model that a graph representation has to be created for. The *ModelHandler* is passed to its constructor after *Transform* has already loaded an instance model by invoking the *loadInstance* method. *InstanceGraphRep* can retrieve all elements from the instance model that represent instances of *EClasses*, and generate a *DefaultGraph* that represents the instance model. The implementation for this is explained section 5.2.2. The class provides a get method for the graph representation.

When transforming from GROOVE to Ecore, an instance of *InstanceModelRep* is initialized by *Transform* for every instance graph in the GROOVE grammar. The *ModelHandler* as well as the instance graph is passed to its constructor. A new instance model is created by invoking the *createModel* method of the *ModelHandler*. The instance graph is parsed and instances of elements are created by using the *Factory* methods of the *EPackages* that the model elements belong to. The algorithm for this transformation is explained in detail in section 5.2.3.

Finally, the *GraphLabels* class provides static methods that generate strings that are used as labels of node types in graph representations. There is a method for each type of Ecore element, and two methods that omit the *type:* part of the string so it can be used in the name of constraints. The *getLabel* method for *(EDataType, EObject)* generates a label string for a value of an *EDataType*, to be used in an instance graph. This is different from generating a label string for an *EDataType* itself which is used in a type graph.

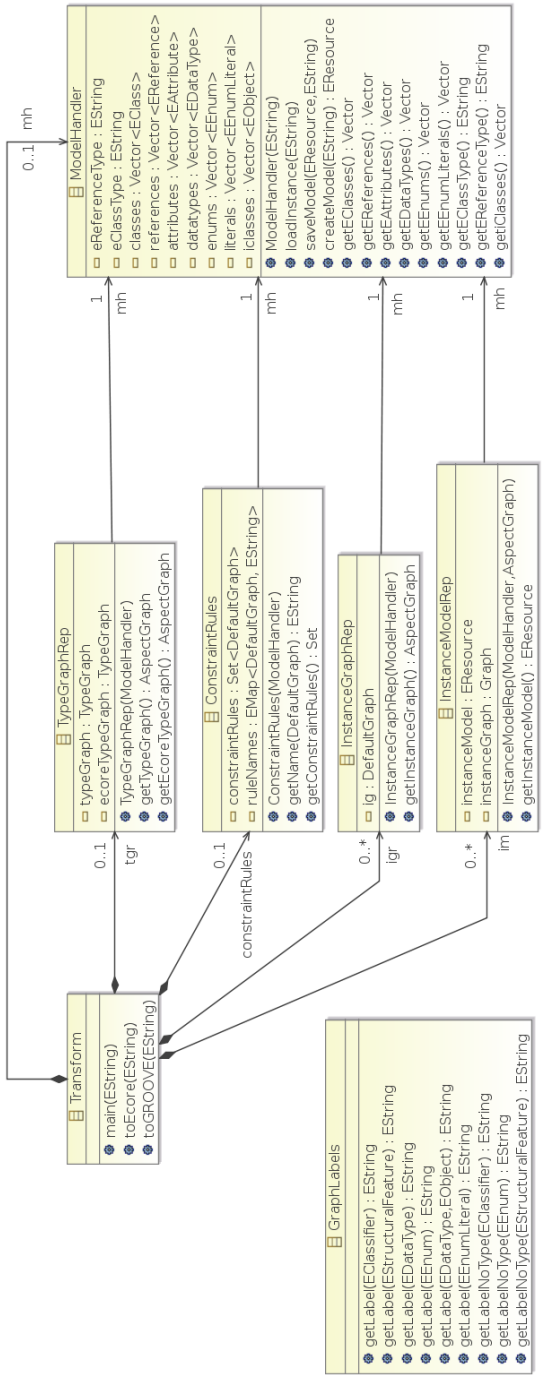


Figure 5.2: Ecore model of the ecore2groove transformation tool.

5.2.1 Ecore model to type graph

In this section we explain the three parts of the transformation process from an Ecore model to a GROOVE type graph. First, how to load an Ecore model, then how to create the type graph, and finally how to generate the required constraint rules. Each time we give a fragment in pseudo code to assist in the explanation. The pseudo code parts are simplified and human readable representations of the actual Java code, and are used to illustrate the flow and algorithms of the *ecore2groove* tool.

First, lets look at how to load an Ecore model. Listing 5.1 shows in pseudo code how this is done. All Ecore models and instance models are *Resources*, and a *ResourceSet* is used to contain *Resources*. We create and empty *ResourceSet*, and then a new *Resource* into which we load the Ecore model. We iterate over all *EPackages* of the *Resource* and register them in the *ResourceSet*, and also also find the root element of the Ecore model. *EPackages* need to be registered in the *ResourceSet*, so the *ResourceSet* can find the *EPackage* when loading instance models. For example, when loading an instance model that has an instance of *EClass ClassB* from *EPackage packageB* into the *ResourceSet*, the *ResourceSet* needs to know which *Resource* contains the specification of *packageB* and *ClassB*. Finally, we create sets containing all *EClasses*, *EEnums*, *EEnumLiterals*, *EReferences*, *EAttributes* and *EDataTypes*. We will be using these later.

Listing 5.1: Pseudo code for loading an Ecore model.

```
1  create new ResourceSet;
2  create new Resource in the ResourceSet;
3  load an Ecore model file into the Resource;
4
5  for all EPackages from the contents of the Resource
6     register the EPackage in the ResourceSet;
7     if EPackage is the root EPackage, mark it;
8  endfor;
9
10 for all elements from the root EPackage
11    add element to a set of EClass, EEnum, EEnumLiteral, EReference,
          EAttribute or EDataType;
12 endfor;
```

Now that we have loaded an Ecore model, we can transform it to a type graph. Listing 5.2 shows in pseudo code how this is done. This part continues from the part in Listing 5.1, so we have sets of Ecore elements from the Ecore model available to us.

First we create an empty type graph and an Ecore type graph with an *EClass*, *EReference* and *EAttribute* node type that we can add nodes and edges to. Then we iterate over the sets in a specific order to add their graph representations to the type graph. Elements have their own representation, but can also be connected to other elements. First we add graph representations to the type graph of elements that can be represented without required connections, then we add graph representations of elements that are connected to previously added elements.

We start by iterating over the set of *EClasses*. For each *EClass*, we add a node type to the type graph to represent it. We then put the pair of the *EClass* and the node type in a map, so we can later retrieve the node that represents this *EClass* to connect it to other nodes. We also add a node type representing the *EClass* to the Ecore type graph as a subtype of the *EClass* node type.

Next, we iterate over the set of *EClasses* again. This time, for each *EClass* we get the set of its *super EClasses*, and for each *super EClass* we add a *sub:* edge from the node that represents the *EClass* to the node that represents the *super EClass*. Because we already iterated over all *EClasses*, we know that all *EClasses* already have a node representation. We use the *EClass* to node map we mentioned earlier to get the node representations.

EReferences connect *EClasses*, and now that all *EClasses* have a node representation, we can add representations of *EReferences* to the type graph. We iterate over all *EReferences*, and add a node type to the type graph to represent each *EReference*. As with *EClasses*, we also put a pair of *EReference* and node type in a map so we can later retrieve the node representing an *EReference*. Then, if an *EReference* is both ordered and has an upper bound of at least 2, we add a *next* selfedge to its node type. If *EReference* has an *opposite EReference*, we check if this *opposite EReference* is already represented in the type graph by checking the map of *EReferences* to node types, and if so add *opposite* edges from and to this *opposite EReference* node type. Finally, we add a node type to the Ecore type graph as a subtype of the *EReference* node type.

Then, representations for *EEnums* and *EEnumLiterals* are added to the type graph. First, node types are added to the type graph to represent *EEnums*, and again put into a map. Next, flags to represent *EEnumLiterals* are added to the node types that represent the *EEnums*.

Now we iterate over all *EDataTypes* in the Ecore model. We need to check if the *EDataType* is supported, but also if an *EDataType* that is represented by the same GROOVE attribute was already added. For example, the *EShort* and *EInt EDataTypes* are both represented by an *int:* node type, but node types must be unique in a type graph so we cannot add another node of the same type. Instead, we must use the node type that is already in the type graph. So if an *EDataType* is supported, we add a node type to the type graph only if it does not already exist. Then, we add a pair of the *EDataType* and either the newly created node type or an already existing node type to a map.

Finally, we add representations of *EAttributes* to the type graph. For each *EAttribute* we add a node type to both the type graph and the Ecore type graph to represent it. Then, an edge from the *containing EClass* node type, retrieved from the *EClass* to node type map, to the *EReference* node type is added. As with *EReferences*, if the *EAttribute* is ordered and the upper bound is at least 2, a *next* selfedge is also added. Finally we need to add an edge to the *EDataType* node type. If the *EDataType* of an *EAttribute* is supported, an edge is added from the *EAttribute* node type to either the *EEnum* node type or the *EDataType* node type. The respective map is used to retrieve this node type.

Listing 5.2: Pseudo code for transforming an Ecore model to a type graph.

```
1  create new TypeGraph;
2  create new EcoreTypeGraph;
3
4  for all EClasses from the set of EClasses
5     add node type to TypeGraph and EcoreTypeGraph to represent EClass
          ;
6     add EClass and node type to map;
7  endfor;
8
9  for all EClasses from the set of EClasses
10    add sub: edges from EClass node to superEClasses nodes;
11 endfor;
12
13 for all EReferences from the set of EReferences
14    add node type to TypeGraph and EcoreTypeGraph represent
          EReference;
15    add EReference and node type to map;
16    add edges from source and to target nodes of EReference;
17    if EReference is ordered and many mult., add next self edge;
18    if EReference has an opposite
19       if opposite EReference already in typegraph, add opposite
             edges;
20    endif;
21 endfor;
22
23 for all EEnums from set of EEnums
24    add node type to TypeGraph to represent EEnum;
25    add EEnum and node type to map;
26 endfor;
27
28 for all EEnumLiterals from set of EEnumLiterals
29    add literal flag to EEnum node type;
30 endfor;
31
32 for all EDataTypes from set of EDataTypes
33    if EDataType is supported and not already in TypeGraph
34       add node type to TypeGraph to represent EDataType;
35    endif;
36    add EDataType and node type to map;
37 endfor;
38
39 for all EAttributes from set of EAttributes
40    add node type to TypeGraph and EcoreTypeGraph to represent
          EAttribute
41    add edge from container EClass node
42    if EReference is ordered and many mult., add next edge;
43    if EDataType is supported, add edge to EDataType or EEnum node;
44 endfor;
```

Adding constraint rules is straightforward, as seen in Listing 5.3. We iterate over all *EClasses*, *EEnums*, *EReferences* and *EAttributes*, and when required, create a constraint rule for GROOVE. Each of these rules is then added to the GROOVE grammar. We do not explain the creation of each constraint rule in detail, since they have already been described in detail in section 4.4 and creating an actual constraint rule is done by just adding nodes and edges to a new graph. This listing is given for completeness, to show that all constraint rules are generated during transformation, and which conditions must be satisfied for a constraint rule to be created.

Listing 5.3: Pseudo code for creating constraint rules from an Ecore model.

```
1  for all EClasses from set of EClasses
2     if the EClass is abstract, add abstract constraint rule;
3     add cyclicity constraint rule;
4     add one container constraint rule;
5     add root constraint rule;
6  endfor;
7
8  for all EEnums from set of EEnums
9     add no literals constraint rule;
10    add many literals constraint rule;
11    add no incoming val constraint rule;
12    add many incoming val constraint rule;
13 endfor;
14
15 for all EReferences from set of EReferences
16    add no val constraint rule;
17    add many val constraint rule;
18    add no container constraint rule;
19    add many container constraint rule;
20    if EReference is unique and upper bound at least 2, add unique
          constraint rule;
21    if EReference has lower bound at least 1, add lower bound
          constraint rule;
22    if EReference has upper bound is not unlimited, add upper bound
          constraint rule;
23    if EReference has opposite, add opposite constraint rule;
24    if EReference has eKeys, add eKeys constraint rule;
25    if EReference is ordered and upper bound at least 2, add ordered
          constraint rules;
26    if EReference is containment, add containment constraint rule;
27    if EReference is not containment, add not containment constraint
           rule;
28 endfor;
29
30 for all EAttributes from set of EAttributes
31    add many val constraint rule;
32    if values of EAttribute must be bound more than default int, add
          values constraint rule;
33    if EAttribute is unique and upper bound at least 2, add unique
          constraint rule;
34    if EAttribute has lower bound at least 1, add lower bound
          constraint rule;
35    if EAttribute has upper bound is not unlimited, add upper bound
          constraint rule;
36    if EAttribute is unchangeable and has a default value, add
          unchangeable constraint rule;
37    if EAttribute is id, add id constraint rule;
38    if EAttribute is ordered and upper bound at least 2, add ordered
          constraint rules;
39 endfor;
```

5.2.2 Ecore instance models to instance graphs

After loading an Ecore model and transforming it to a type graph, we can now load instance models and transform them to instance graphs. *Ecore2groove* can transform several instance models at a time for the same Ecore model. For each

instance model, an instance graph is created in the GROOVE grammar that already contains the type graph.

Listing 5.4 contains in pseudo code how an Ecore instance model is transformed into a GROOVE instance graph. First, we load the instance model into the ResourceSet that already contains the Ecore model. Then a new graph is created that will be stored in the GROOVE grammar.

Next, we iterate over all *EObjects* that are instances of *EClasses* in the instance model. For each instance, we add a node to the instance graph. Each time we mention that a node or edge is added to the instance graph to represent an instance of some element, it is typed by the node type or edge in the type graph that represents this element. So in this case, the node is typed by the node type that represents the *EClass* that the *EObject* is an instance of. After adding the node, we also add the *EObject* and the node to a map.

Now that all instances of *EClasses* are represented in the instance graph, we can add representations of instances of *EReferences* and *EAttributes*. These are not *EObjects* like instances of *EClasses* are, but values for the instances of *EClasses* that contain them. For example, an instance of *ClassA* contains through the *containment EReference has* an instance of *ClassB*. The target of this *EReference* is retrieved by getting the values of *has* from *ClassA*. This will be an *EList* of *EObjects* in the case of a many multiplicity, or just an *EObject* otherwise. The *EObjects* are *ClassB* instances.

We iterate again over all instances of *EClasses*. For each instance, we iterate over all *EReferences* of its *EClass* in the Ecore model, and then over all values (or targets) of this *EReference* in the instance model. This can either be an *EList* of target instances, or just one target instance. Each of these targets is already represented in the instance graph, so we just need to add a node to represent the *EReference*, and edges from the containing instance *EClass* node and to the target instance *EClass* node. Next, we need to add this representation to a map. Since there is no Java Object for an instance of an *EReference*, we do this by adding a pair of a *Triple(containing instance, EReference, target instance)* and the instance *EReference* node to a map. Then, we check if the *EReference* has an opposite, and if so whether or not it is already in the instance graph by consulting the map. If it is present, we add *opposite* edges from and to it. Finally, if the *EReference* is ordered, a *next* edge is added from the previous instance of this *EReference*, unless this was the first instance of the list. The list of values is always ordered the same as how the values occur in the XML file, so this ensures that the instances are properly ordered in the instance graph.

Adding instances of *EAttributes* is done in a similar way, except that we also need to add nodes to represent the values of the *EDataType* in the instance graph when supported, and that *EAttributes* have no opposites. Because they have no opposites, it is also not necessary to add representations of *EAttribute* instances to a map.

Listing 5.4: Pseudo code for transforming an Ecore instance model to an instance graph.

```
load instance model file into ResourceSet;
create new instance graph;

for all instances of EClasses in the instance model
   add node type to instance graph to represent instance;
   add pair of instance and node type to map;
endfor;

for all instances of EClasses in the instance model
   for all EReferences from the EClass of the instance
      for all targets of the EReference from the instance EClass
         add node type to represent EReference instance;
         add edges from and to source an target EClass instances;
         add pair of instance of EReference and node type to map;
         if EReference has opposite
            if opposite is in instance graph, add opposite edges;
         endif;
         if EReference is ordered
            add next edge from previous node type to node type;
         endif;
      endfor;
   endfor;
   for all EAttributes from the EClass of the instance
      for all values of the EAttribute of the instance EClass
         add node type to represent EAttribute instance
         add edge from containing EClass instance node type;
         if EDataType is supported
            add node type to represent EDataType value
            add edge from EAttribute to EDataType/EEnum node type;
         endif;
         if EAttribute is ordered
            add next edge from previous node type to node type;
         endif;
      endfor;
   endfor;
endfor;
```

5.2.3 Instance graphs to Ecore instance models

The pseudo code in Listing 5.5 shows how an instance graph is transformed to an Ecore instance model. First we determine the root *EClass* element in the instance graph. Constraints enforce that there should be exactly one node with a *root* flag, so we locate it and set this node to be the *rootNodeElement*. Next, we create a new Resource in the ResourceSet, and then an instance of the *EClass* that the *rootNodeElement* node represents. Instances of *EClasses* are created by getting the *EPackage* of the *EClass* from the Ecore model, and then the *Factory* of the *EPackage*. This *Factory* can create instances of *EClasses* and *EDataTypes* that belong to the *EPackage*. The created instance of the *EClass* is then added to the contents of the Resource. Further elements are not explicitly added to the contents, since they are contained by the root *EClass* instance and already added implicitly. Next, we add the *rootNodeElement* and the *EClass* instance to the *nodeToObject* map, and finally call the recursive

function *addContainedClasses* to add all *EClass* instances contained by the root element.

This recursive function is given in lines 9-27. For the *eclassNode* passed to it, we iterate over all connected nodes that represent instances of *containment EReferences*. If this *EReference* is not ordered, the node that represents that target *EClass* instance is located, and an instance of this *EClass* is created. The target *EClass* node and the created *EClass* instance are put into the *nodeToObject* map. Then, the *EClass* instance is added as an *EReference* value of the *EClass* instance represented by the *eclassNode*, and *addContainedClasses* is called for the target *EClass* node. If the *EReference* is ordered, we check if the found instance of the *EReference* is the first in the list, by checking that it has no incoming *next* edge. If so, we do the same procedure as in the case of an instance of a non ordered *EReference*, and then continue with the next *EReference* instance in the list by following the outgoing *next* edge until there are no more outgoing *next* edges. The order of values of an *EReference* or *EAttribute* is the same as the order in which they are added, so this way the instance model will have the values in the correct order.

Lines 29-67 show how values of non containment *EReferences* and *EAttributes* are added to the instance model. First, in lines 29-49 values of *EReferences* are added. We iterate over all nodes *eclassNode* that represent *EClass* instances. Then we iterate over all connected nodes *erefNode* that represent *non containment EReferences*. If this *EReference* is not ordered, we look up the target of this *EReference* instance in the instance graph, and then get the already created instance *EClass* from the *nodeToObject* map. Next, we check if this value is already in the *EList* of *EReference* values of the *EClass* instance represented by *eclassNode*, and if not it is added. It is possible that it is already added, because values of *opposite EReferences* are added at the same time when the value of an *EReference* is added. If the *EReference* is ordered, we check if the found *EReference* is the first in the list. If so, we do the same again as for instances of *non ordered EReferences* but this time if the value is already in the *EList* it is not omitted, but instead moved to the last position in the *EList*. It is possible that the value of the *opposite EReference* was already added to the instance model, which could have put this instance in the wrong positing in the *EList*. Since all values of an *ordered EReference* in the instance graph are always iterated over in the correct order, just moving any already present values to the end of the *EList* ensures that the *EList* of values is always in the correct order in the final Ecore instance model. This is repeated for the next *EReference* instance in the instance graph until there is no more outgoing *next* edge. Finally, in lines 51-71 we add values of *EAttributes* to the instance model. The idea for this is the same as for values of *EReferences* with two differences. Firstly, we need to create an instance of the *EDataType* of the *EAttribute* that contains the value. This is again done by getting the *Factory* of the *EPackage* of the *EClass* that the *eclassNode* represents. Secondly, we do not have to consider opposites, so we can always create an *EDataType* value and then add it to the list of values in the instance model.

Listing 5.5: Pseudo code for transforming an instance graph to an Ecore instance model.

```
1  rootNodeElement = node type with root flag;
2
3  create new Resource in ResourceSet;
4  create instance of EClass represented by rootNodeElement;
5  add this instance to contents of Resource;
6  add a pair of node and EClass instance to nodeToObject map;
7  addContainedClasses(rootNodeElement);
8
9  function addContainedClasses(Node eclassNode)
10    for all nodes erefNode that represent containment EReferences
          from eclassNode
11       if EReference is not ordered
12          create instance of EClass represented by target of
                erefNode;
13          add node type and new EClass instance to nodeToObject map;
14          add this instance as EReference value to EClass instance
                of eclassNode;
15          addContainedClasses(target of erefNode);
16       else
17          if erefNode is first in list
18             while erefNode is not null
19                create instance of EClass represented by target of
                      erefNode;
20                add node type and new EClass instance to nodeToObject
                       map;
21                add this instance as EReference value to EClass
                      instance of eclassNode;
22                addContainedClasses(target of erefNode);
23                erefNode = next erefNode in list;
24             endwhile;
25          endif;
26       endif;
27 endfunction;
28
29 for all node types eclassNode that represent EClasses
30    for all nodes erefNode that represent non-containment
          EReferences from eclassNode
31       if erefNode is not ordered
32          get instance of EClass represented by target of erefNode
                from nodeToObject map;
33          if instance not present as EReference value for EClass
                instance of eclassNode
34             add this instance as EReference value to EClass
                   instance of eclassNode;
35          endif;
36       else
37          if erefNode is first in list
38             while erefNode is not null
39                get instance of EClass represented by target of
                      erefNode from nodeToObject map;
40                if instance not present as EReference value for
                      EClass instance of eclassNode
41                   add this instance as EReference value to EClass
                         instance of eclassNode;
42                else
43                   move this instance to last position in list of
                         values;
44                endif;
45                erefNode = next erefNode in list;
46             endwhile;
```

```
47          endif;
48       endif;
49    endfor;
50
51    for all nodes attrNode that represent EAttributes from
          eclassNode
52       if datatype of attrNode is supported
53          if attrNode is not ordered
54             create instance of EDataType represented by target of
                   attrNode;
55             add instance as EAttribute value to EClass instance of
                   eclassNode;
56          else
57             if attrNode is first in list
58                while attrNode is not null
59                   create instance of EDataType represented by
                         target of attrNode;
60                   add instance as EAttribute value to EClass
                         instance of eclassNode;
61                   attrNode = next attrNode in list;
62                endwhile;
63             endif;
64          endif;
65       endif;
66    endfor;
67 endfor;
```

5.3 Tool demonstration

To demonstrate the *ecore2groove* tool and some capabilities of GROOVE as a model transforming tool, we use an Ecore model of a buffer and an instance model with two buffer slots. We then use our transformation tool to transform these models to a GROOVE grammar with the graph representations and constraint rules. The models and graph representations will be shown and explained in section 5.3.1.

The next step is to transform the instance graph in GROOVE using graph transformation rules. We will show several rules to add and remove slots for the buffer, and put and get objects from the slots. Additionally we will show how a new instance graph can be created to represent an instance model. Any resulting instance graphs that do not violate any constraints can then be transformed to an instance model using the *ecore2groove* tool. This is demonstrated in section 5.3.2.

Finally, other features of using GROOVE as a model transforming tool, like creating a new graph or repairing invalid instance graphs, adding semantics to custom constraints or deleting large parts of an instance graph, are explained in section 5.3.3.

5.3.1 Ecore model and graph representation

Figure 5.3 shows an Ecore model of a buffer. The *Buffer* itself is abstract, and instances must either be a *FiLoBuffer* for a *first-in-last-out* buffer, or a

FiFoBuffer for a *first-in-first-out* buffer.

The *Buffer* can have zero to four *BufferSlots* that can hold the objects. This *containment EReference* is ordered as indicated in the figure. Each *BufferSlot* has an *EAttribute status* to keep track whether or not a slot is filled with an object. This *EAttribute* is of an *EEnum* type, *BufferSlotStatus*, which has two literal values, *empty* and *filled*. So a slot of the buffer can either be empty or filled with an object. Next, a *BufferSlot* can have a *value*, which is a *JObject* with an identifier *id*. We want a *Buffer* to only hold unique instances of *JObject*, and we use the *EAttribute id* to check this. The *EReference values* refers to all *JObjects* and *eKeys* has been set to the *EAttribute id* so that this identifier must uniquely identify a referred instance of *values*.

An instance model of the buffer is given in **Figure 5.4**. It represents a *first-in-last-out* buffer with two empty slots. It has an instance of a *FiLoBuffer* with two instances of *BufferSlot* with *status* set to *empty*.

When running *ecore2groove* with the Ecore model and instance model as input, a GROOVE grammar is created with a type graph representation of the Ecore model, instance graph representation of the instance model, and 39 constraints rules. In this example, the tool spends most time loading the Ecore model and the GROOVE grammar without doing any transformation. Creating graphs and constraint rules take far less time. As an indication, on our test setup with a 3.0Ghz dual core processor running Ubuntu Linux, loading the Ecore model took 600ms, creating a type graph representation 20 ms, creating an empty GROOVE grammar 300ms, creating 39 constraint rules 70ms, loading the instance model 10 ms and creating an instance graph to represent it 2 ms. When using a graph grammar that already existed, loading it took 700 ms.

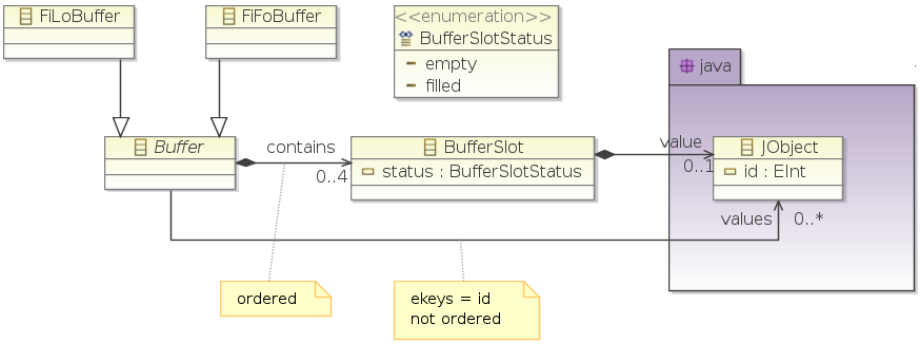


Figure 5.3: Diagram of an Ecore model of a buffer that can both be a first-in-first-out or first-in-last-out buffer.

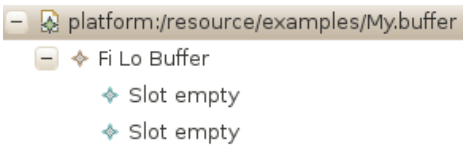


Figure 5.4: Instance model of the Ecore buffer model. This model represents a first-in-last-out buffer with two empty slots.

With a larger Ecore model that contained 211 *EClasses*, 178 *EReferences* and 36 *EAttributes*, loading the Ecore model took 800 ms, creating the type graphs 75 ms, creating a graph grammar 300 ms and 1290 constraint rules 650 ms. However, loading an existing graph grammar took almost 2 s. This is probably because the large amount of constraint rules that need to be loaded by GROOVE. We did not transform an instance of this larger model, but it is expected that loading the model will take most time and not creating the instance graph representation. In general, creating constraint rules and loading graph grammars with constraint rules are the largest bottleneck of our implementation. This can only be solved by having fewer constraint rules, but this is not possible if we want to support all elements and properties of Ecore.

Figure 5.5 shows the type graph representation of the Ecore model of the buffer. *EClasses* and *EReferences* in the Ecore models are represented as node types, which in turn are subtypes of the node types *EClass* and *EReference* in the Ecore type graph. The *ordered EReference contains* has a *next* selfedge so instances can be ordered in instance graphs. *EAttributes* are represented as node types as well, and the *EEnum* is represented as a node type with its *EEnumLiterals* as flags of this node. Finally, *EDataTypes* are GROOVE attributes in the type graph. The type graph representation of the Ecore model is explained in detail in chapter 4.

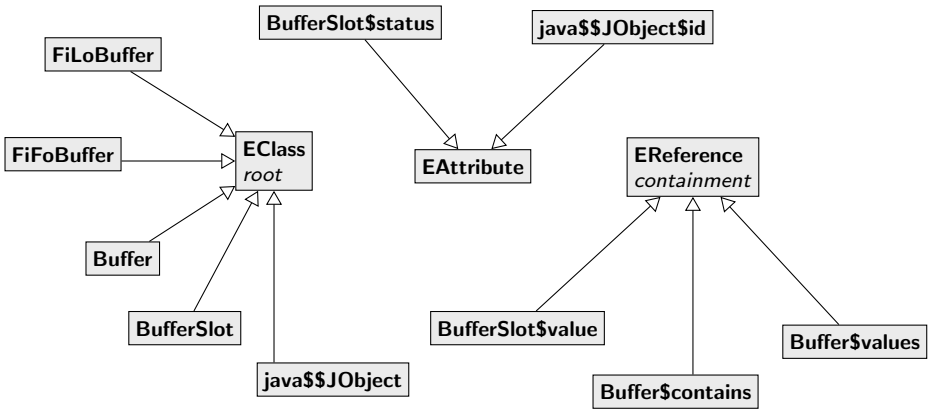
Figure 5.6 shows the instance graph representation of the instance model of the *first-in-last-out* buffer with two slots. The flags to indicate the root element of the model and the instances of the *containment EReference* have been set, and the buffer slots are marked to be empty. None of the constraint rules can find a match in this graph, so it does not violate any constraints.

5.3.2 Transformations in instance graphs

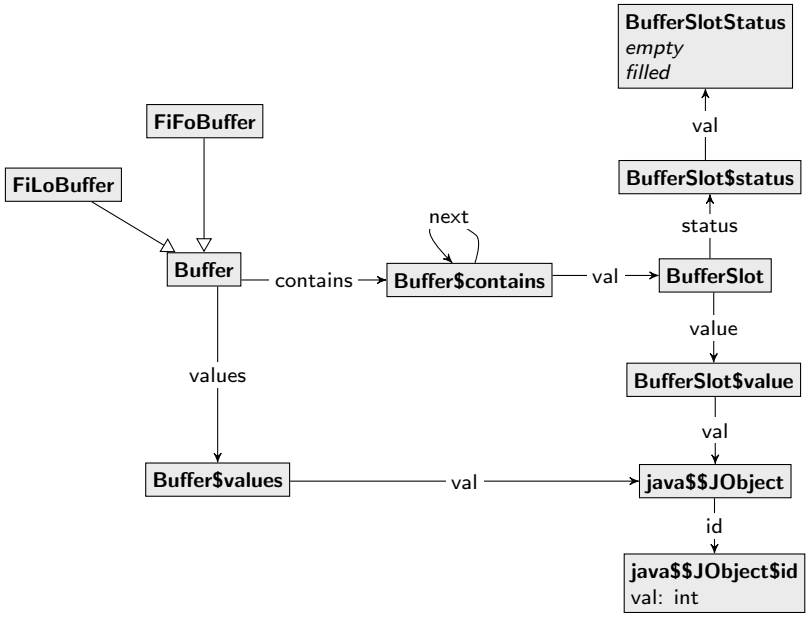
Since the Ecore model represents a buffer, we want to put and get objects from slots of this buffer. First we demonstrate transformation rules to change the size of the buffer by adding and removing slots. Then we show how to put and get objects into and from the slots. The constraint rules that detect invalid instance graphs have a priority of 50. Any transformation rules that are created manually have a priority of 0 by default. This means that whenever a constraint violation is found by a constraint rule, the transformation rules to transform instance graphs are disabled.

The buffer supports zero to four slots as defined in the Ecore model, and the instance model has two. In the instance graph there are two node types for buffer slots, and we want to be able to change this number. We use two transformation rules for this, one to add a new slot and one to remove a slot. The *containment EReference* containing the *BufferSlots* is ordered, so the instances of the slots in the instance graph are ordered with *next* edges. When adding or removing a *BufferSlot* in the instance graph we want to do this at the end of the list, and we only want to remove a slot if it does not contain an object.

Figure 5.7 shows the graph transformation rule to add a new slot to the buffer. It matches a *Buffer* node, which can be either a *FiFoBuffer* or a



(a) Ecore type graph



(b) Type graph

Figure 5.5: Type graph representation of the buffer Ecore model. Both type graphs are merged internally in GROOVE to form one type graph that instance graphs must be typed by.

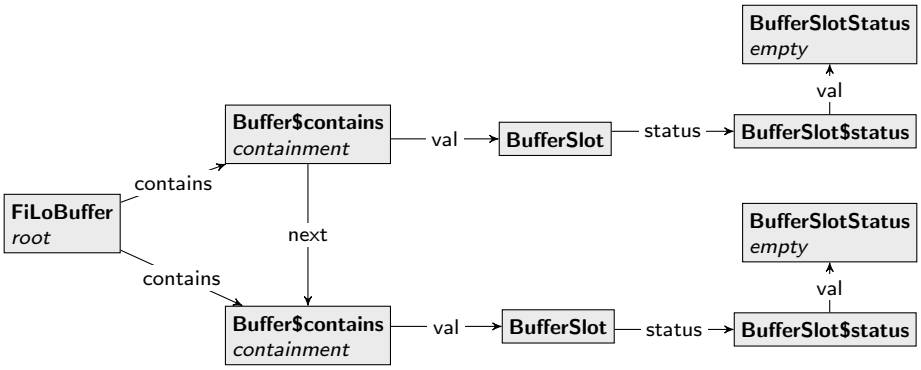


Figure 5.6: Instance graph representation of the buffer instance model. This *first-in-last-out* buffer has two slots that can contain an object each.

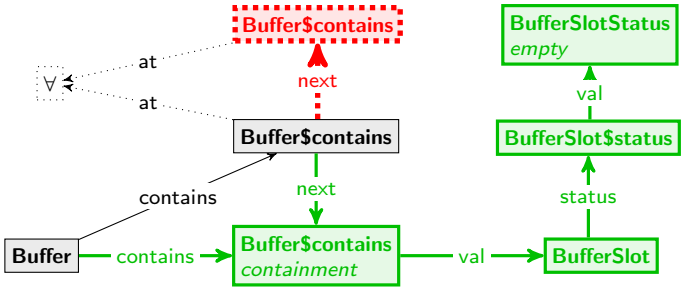


Figure 5.7: Graph transformation rule to add a new slot to the buffer. This slot will be placed at the end of the list of slots.

FiLoBuffer node because of subtyping. New nodes are created when applying this rule; a *Buffer$contains* node with *containment* flag to represent the *containment EReference*, *BufferSlot* node for the slot itself, and a *BufferSlot$status* and *BufferSlotStatus* with an *empty* flag for the *EAttribute* and *EEnum* to indicate that this slot is empty. Now this slot must be placed at the end of the list. The last *BufferSlot* in the list is the one where the *EReference* representation containing it does not have an outgoing *next* edge. The rule matches all *Buffer$contains* nodes without outgoing *next* edges, which can be either one or zero. From each of these *Buffer$contains* nodes, a *next* edge is added that refers to the new *Buffer$contains* node. This way the rule is applicable when there are already slots in the list, but also for the first slot of a buffer.

The graph transformation rule in **Figure 5.8** deletes a slot from a buffer. It matches a *Buffer* node which can again either be a *FiFoBuffer* or a *FiLoBuffer* node. It then looks for a *Buffer$contains* node that that does not have an outgoing next edge, a *BufferSlot* node that represents a buffer slot, a *BufferSlot$status* node and a *BufferSlotStatus* node with an *empty* flag, and deletes

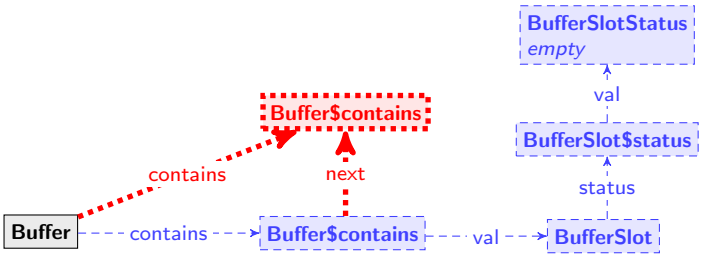


Figure 5.8: Graph transformation rule to delete a slot from the buffer. The slot to be removed is at the end of the list and must be empty.

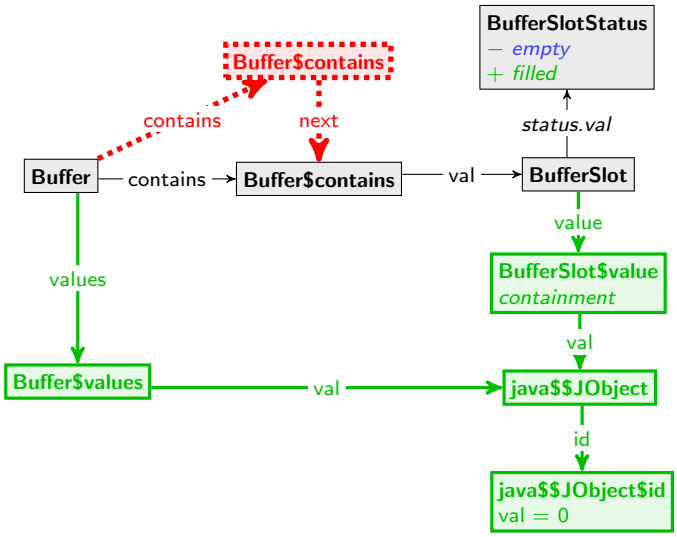


Figure 5.9: Graph transformation rule to put an object into the first slot of the buffer. The *id* attribute of the first object is set to 0

these nodes and the edges connecting them from the graph when applying the rule. There can be at most one *Buffer$contains* node without an outgoing *next* edge in an instance graph, namely the instance at the end of the list, so when this rules finds a match it is for this last buffer slot in the list. The rule also only matches when the buffer slot to be removed is flagged to be empty, which indicates in our model that it does not contain an object.

Putting an object into a slot of the buffer is split into two transformation rules. The first one in **Figure 5.9** adds an object to the first slot of the buffer, and the other one in **Figure 5.10** adds an object to any subsequent slot of the buffer. The first rule matches the first *BufferSlot* in the list, but only when it is flagged to be empty. When the rule is applied, the *empty* flag is deleted from the *BufferSlotStatus* and a *filled* flag is added. We use a path edge labelled *status.val* from the *Buffer* node to match the *BufferSlotStatus* node connected to it. The rule also creates nodes that represent an instance of a *JObject*, the *containment EReference* that contains it, the *EAttribute id* with value 0 and finally an instance of the *EReference values* from the *Buffer* that refers to this new *JObject*.

The second rule to put an object into the buffer can only be applied when there is a slot in the buffer that is filled with an object that has an *id* integer value, and also has a next slot that is still empty. The new object then has an *id* value of 1 plus the *id* integer value of the object in the previous slot. The flag of the *BufferSlotStatus* node that is matched is again changed from *empty* to *filled*, and nodes to represent the new object are created. The integer attribute

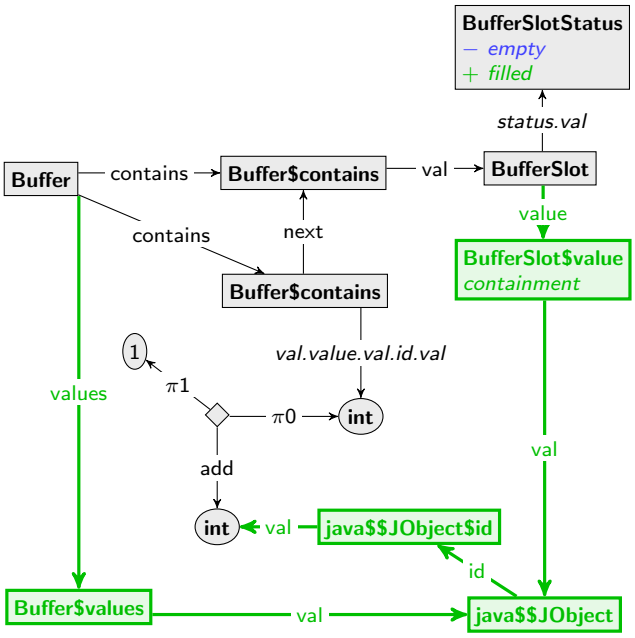


Figure 5.10: Graph transformation rule to put an object into a subsequent slot of the buffer. The *id* attribute of this object is set to the *id* of the previous object in the list, plus one.

that represents the *id* value of the previous object is matched by the path edge labelled *val.value.val.id.val* from the node that represents the *EReference* instance containing the previous slot of the buffer. An integer operation is then used to add 1 and this value to each other, and the result is the value of the *id* attribute of the new object.

The final transformation rule is used to get an object from a *first-in-last-out* buffer slot, effectively deleting it from the instance graph. The rule in **Figure 5.11** matches the last slot in the buffer that is filled, deletes its object and references to it, and flags it to be empty again. The last filled buffer slot is matched by looking for a *Buffer$contains* node that does not have a *next* edge to a *Buffer$contains* node with a *val.status.val* path to a *BufferSlotStatus* with a *filled* flag. When the rule is applied, the nodes and connecting edges representing the *JObject*, the *EReferences* to it and its *EAttribute* are deleted from the instance graph. The flag of the *BufferSlotStatus* connected to the *BufferSlot* that is made empty is also changed from *filled* to *empty* to mark the buffer slot empty.

With these graph transformation rules we can transform our example instance graph to a graph that represents a different model, for example a *first-in-last-out* buffer with 3 slots of which the first 2 slots are filled. To get this we must apply the rule the put the first object in the buffer, and then the rule to put a subsequent object into the buffer. Additionally, we need one extra buffer slot so the rule to create a slot must be applied. We now have the instance graph from **Figure 5.12**. We store the state as a new graph after applying the rules, and then run *ecore2groove* to transform the instance graphs back to instance models. Creating each instance model from an instance graph takes about 20ms. The resulting instance models can be loaded into the Eclipse

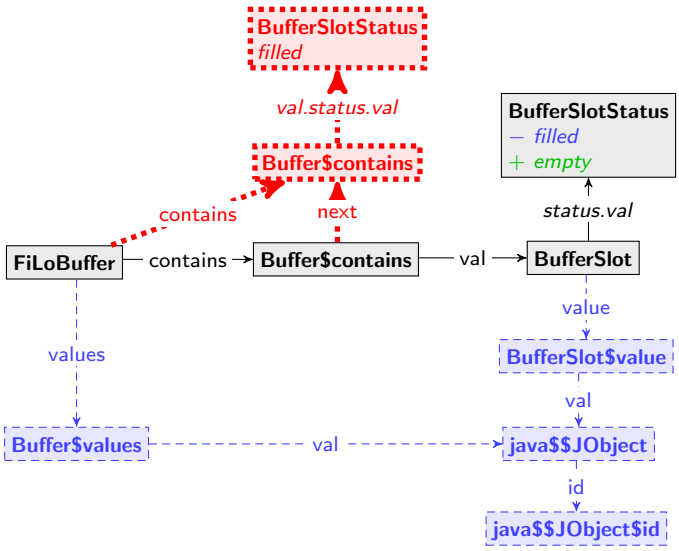


Figure 5.11: Graph transformation rule to get an object from the last slot of a *first-in-last-out* buffer. The slot is flagged to be empty so it can either be filled again or be deleted.

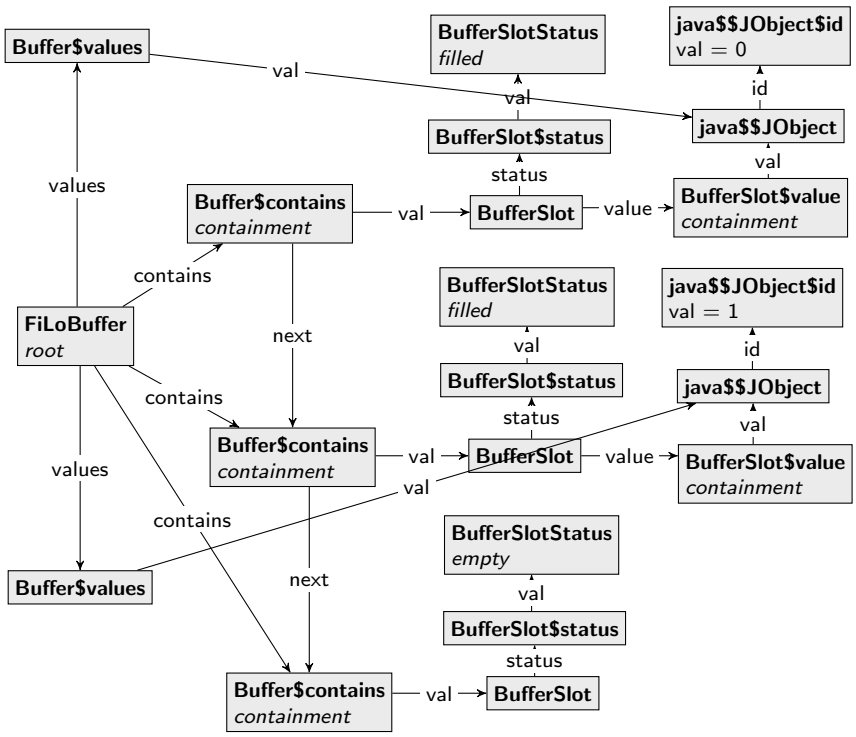


Figure 5.12: Instance graph representation of the transformed buffer instance model. This *first-in-last-out* buffer has three slots of which two are filled with an object that have a unique *id* value.

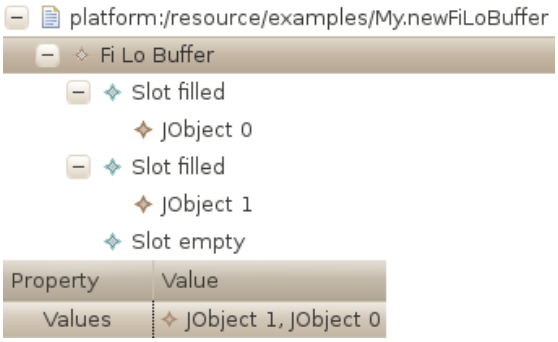


Figure 5.13: Instance model of the transformed buffer instance model.

Figure 5.14: Transformation rule to add a *FiLoBuffer* node to the graph if there is not already a *Buffer* node.

instance model editor, as seen in **Figure 5.13**.

5.3.3 Other features

When starting with an empty graph to create a new instance of a buffer, we find that a constraint is immediately violated, since the empty instance graph has no root element. New graph transformation rules have a priority of 0 by default. Since the constraint rule has a priority of 50, the transformation rules are not applicable. The rule that adds a *Buffer* node to an empty graph like in **Figure 5.14** should have a priority higher than 50 so it can be applied. After applying it, the instance graph does not violate any constraints anymore so the normal transformation rules become applicable.

In general, transformation rules that match a violation and transform an invalid instance graph into a valid one should have a higher priority than the priority of the constraint rules. Whenever either such a rule or a constraint rule is applicable, the instance graph violates a constraint and an instance model cannot be generated from it. These rules can be used when a user knows that an instance graph violates a constraint, but also takes steps to repair the violation. This can be done as explained for empty graphs, but also when a transformation cannot be done in a single step with a single transformation rule, and the intermediate graph between steps violates some constraint. Subsequent steps can be done with transformation rules that detect this, and also perform the next step of the transformation. One has to take care that these rules are not applicable in valid instance graphs, because then the normal transformation rules will be disabled.

Another feature of using GROOVE as a transformation tool is creating custom constraint rules. Let us consider the buffer Ecore model again. We mentioned that all *JObjects* in the slots of a buffer should have unique values for

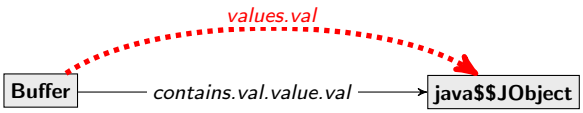


Figure 5.15: Custom constraint rule that detects *JObjects* that are contained by a *Buffer*, but that are not refered to by the *values EReference*.

id. The *eKeys* property of the *values EReference* refers to the *id EAttribute* of *JObject*, indicating they should be unique for each *JObject* referred to by *values*. However, this does not enforce that the *Buffer* actually refers to all *JObject* instances with *values*, and *JObjects* that are not referred to can still have an *id* value which it the same as another. In Ecore models, we can add a constraint as an annotation that all *JObject* instances must be referred to. However, such a constraint has no semantics, and it cannot be checked by the instance model editor.

By using GROOVE, it is possible to create custom constraint rules that detect violations of constraints set by a user, like the one just described. The constraint rule in **Figure 5.15** detects *java$$JObject* nodes that are indirectly contained by a *Buffer* node through the *contains.val.value.val* path, but do not have a *values.val* path to them that represents an instance of the *values EReference* to this *JObject*. These custom constraint rules should have a priority higher than 0. In our case we set the priority to 40 to keep them separate from the generated constraint rules. They should not be named with something starting with *"constraint - "* to prevent them from being deleted when using the GROOVE grammar as output for the *ecore2groove* tool.

In short, in Ecore models it is possible to define constraints on an instance model by using annotations, but semantics cannot be added to the model and violations in instance models cannot be detected. In GROOVE, semantics can be added by defining a custom constraint rule that detects violations. These custom constraint rules cannot be generated, because the semantics are not known in the Ecore model, and must be created by a user.

Finally, special care has to be taken when when deleting nodes from an instance graph. In our buffer example, slots can only be removed when they do not contain an object. If we would not check for this, objects can be left as dangling nodes that are not connected to the root element of the graph. These dangling nodes are constraint violations that will be detected by constraint rules. Dangling nodes can be prevented by specifically checking for the absence of contained elements when deleting a node, like we did when deleting a buffer slot. Another option is to also delete any nodes that represent elements contained by the element which is deleted. Quantifiers can be used to match all nodes that represent elements contained by the element that is deleted.

For example the rule in **Figure 5.16** deletes a *Buffer* node and also all its slots and objects contained in those slots. A *Buffer* node is matched and deleted, and also all *BufferSlots* it contains, as well as all *JObjects* that are contained in those *BufferSlots*. After applying this rule, we end up with an empty graph.

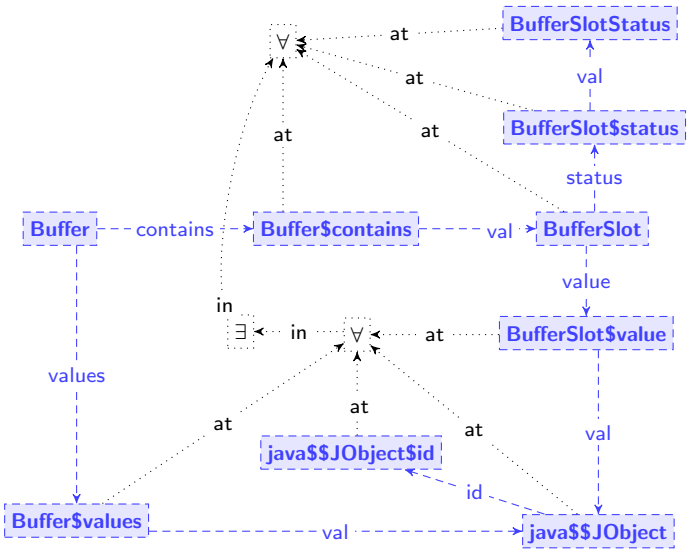


Figure 5.16: Graph transformation rule that deletes a *Buffer* node from the graph, as well as all its *BufferSlots* and *JObjects*.

Chapter 6

Related work

Much work has been done in the area of model transformation within the meta modelling paradigm, and there have been many different approaches. With our approach it is possible to perform model transformation of Ecore instance models by using the graph transformation tool GROOVE. In this chapter we discuss other approaches and compare their interoperability with other tools, their used modelling standards or non-standards, and their notations to describe model transformations to our approach.

Tools exist that perform UML, Ecore or other model transformations within the tool itself, such as EMF Tiger/EMF Henshin [6, 8, 7], Fujaba [47], MOFLON [2, 1], VIATRA2 [46, 45], VMTS [24], MoTMoT [38, 27], AToM^3 [44, 12], mediniQVT [19], Tefkat [23] or ATL [20]. Of these tools, some use a graphical notation for transformation rules with a foundation in graph transformation [8, 7, 47, 24, 44, 12, 2, 1], others use a textual notation, usually an implementation of QVT [32] or something based on it, to define transformation rules [45, 19, 23, 20], or a combination of graphical and textual [38, 27, 44, 12]. EMF Tiger/Henshin can also be used, like ours, to first transform models to a graph formalism of a graph transformation tool, in this case AGG [40, 8], so that graph transformation rules can be applied within an existing graph transformation tool [6, 8, 7]. Some model transformation approaches use a meta modelling standard, usually MOF or EMF, as a basis for the models that can be transformed [8, 19, 23, 2, 1, 20], and again others use a custom internal format for meta models and models [45, 24, 38, 47, 27, 44, 12] but sometimes offer some import/export capabilities.

Some approaches aim to achieve compatibility with other modelling tools by using the XMI standard for exchanging models [46, 45, 38, 27]. To our knowledge, there has been no XMI compatibility research like ours to determine which tools can exchange models to each other using different version of XMI. However, approaches that do use the XMI standard often mention the specific tool it can import models from using XMI [2, 1], issues with importing models using XMI [27], or only plan on supporting it in the future [46, 45].

EMF Henshin started out as the Tiger EMF project [6, 8], and was renamed to EMF Henshin as it became an official subproject of EMF [7] as an Eclipse

plugin. EMF Henshin allows users to specify transformation rules in a graph transformation like graphical notation to transform Ecore models. The LHS (left-hand-side) of a transformation rule matches an instance in an Ecore model, the RHS (right-hand-side) determines the actions of what needs to be deleted or created, and the NAC (negative application constraint) puts a constraint on the applicability of the rule. The LHS, RHS and NAC part of a transformation rule are Ecore models themselves, which makes it possible to define transformation rules on transformation rules. Furthermore, transformations using EMF Henshin can be either endogenous or exogenous, while our approach currently only supports endogenous transformations. EMF Henshin also offers functionality to export models and transformation rules to the graph transformation system AGG [40, 8], which is an approach similar to ours. However, their mapping of Ecore models to a graph representation is very basic, and not complete. Several features of Ecore models are not supported, like the *opposite*, *ordered*, *iD* or *eKeys* properties. They discuss the differences between performing model transformation directly on Ecore models, or on graph representations [8]. In short, the strength of EMF Henshin is to work with existing Ecore models, while using AGG has the benefit that it is strong in performing static analysis of properties of transformation systems, like conflicts, dependencies or the applicability of rules. Unlike their approach with AGG, our tool can also transform instance graphs back to Ecore instance models.

The Fujaba Tool Suite, originally developed at the university of Paderborn, allows a user to create UML class diagrams, and to define model transformations [47]. Fujaba uses a proprietary model transformation technique called Story Driven Modelling (SDM). The behaviour of a model is defined in a so called Story Diagram, which is basically an activity diagram. Each step of the diagram is refined into a Story Pattern, which describes a transformation of the model. Using SDM, it is possible to visually describe the full behaviour and transformations of a UML class diagram. However, without extensions, the tool does not integrate with other tools and transformations can only be described on models developed within the tool itself.

MOFLON [2, 1] is an extension to Fujaba to integrate it with other tools. It extends Fujaba with metamodelling capabilities using the MOF OMG standard, and provides XMI 2.1 import and export capabilities for models. Compatibility with Rational Rose for UML 1.3 diagrams is mentioned explicitly, but no further details about compatibility are given. Based on our own findings we expect that imported models from other tools will have errors, unless specific importers and exporters have been written for individual tools. Imported models can be synchronized to each other using triple graph grammars [39] that are translated to SDM, allowing transformations to one model to be propagated to other models.

MoTMoT [38, 27] is a model transformation tool that is based on Fujaba. It was originally developed because Fujaba provided no exchangeability of models with other tools, although it does now through MOFLON. MoTMoT is based on the SDM technique from Fujaba to define model transformations. It supports model transformations of UML 1.3 models, and models can be imported using XMI 1.x. Performance issues in scanning and building a model from XMI have been mentioned [27], although no further details are given.

VIATRA2 [46, 45], a subproject of Eclipse GMT, is a model transformation tool that works with a non-standard multi-level metamodelling approach, VPM (Visual and Precise Metamodelling), because they claim that MOF and EMF are not expressive enough [45]. A textual syntax called VTML (Viatra Textual Metamodelling Language) is used to specify metamodels and models. Model transformations are also specified textually, using VTCL (Viatra Textual Command Language). This is again a non-standard approach, because they argue that QVT [32] is not suited for unidirectional transformations of models [45]. Rules consist of a precondition and a postcondition, which can be related to LHS and RHS of graph transformation rules. A unique feature of VTCL is the possibility to use negative conditions with an arbitrary depth of negation. VTCL, like EMF Henshin, also supports transformations of transformation rules. A drawback of VIATRA2 is that non-standards are used for specifying metamodels, models, and transformations, and the lack of integration with other tools. Importers and exporters for XMI models are planned, as well as importers for QVT transformations, but have been since the original project plan and do not exist yet [46].

VMTS [24] is a visual modelling and model transformation tool that provides a non-standard multi-level modelling framework in which (meta)models and transformations can be specified. Model transformations can be specified in a way similar to graph transformation rules, with a LHS and a RHS that contain elements specified in the metamodel. Constraints for transformations can be defined using OCL [31]. Models are stored in an XML-like way, but not conform to the standard. VMTS does not offer import or export capabilities, so it cannot be used on existing models.

AToM^3 [44, 12] is a metamodelling and model transformation tool. With it, it is possible to define a metamodel within the tool, after which an editor is generated to create and transform instance models of this metamodel. It is not based on any metamodelling standard and simple ER (Entity-Relation) diagrams are used to define metamodels, but it is possible to define metamodels that simulate the MOF or EMF metamodelling standard. Model transformations are created in the generated instance editor using graph grammars with transformation rules that have a LHS and a RHS. Constraints on the applicability of rules can be specified in OCL [31] or using Python scripts. Their approach, like ours, is to provide a means to transform instance models of a metamodel, although their approach also supports exogenous model transformations. The drawback of AToM^3 is that it provides no integration with existing models and that metamodelling is not conform to any standard, so it can only be used on models created within the tool. Future support for exchangeability of models is mentioned [12], but has not been implemented.

Tefkat [23], developed at the University of Queensland, is a tool that implements the model transformation language QVT [32]. It is an plugin for Eclipse and operates on models specified using EMF. QVT is a textual and declarative transformation language used to define bidirectional transformations of models. Tefkat does not support in-place transformation of models, and instead produces a set of target models that are correct with respect to the transformation specification in a QVT script. As a result of using QVT, all elements that must be present in the target model, even when unchanged, must be mentioned ex-

plicitly in the script [16]. This leads to scripts with a size related to the size of the model and not to the delta of the transformation, so transformations with a small delta can still lead to large scripts. Another limitation is that transformation rules cannot depend on their own negation, ie. it is not possible to create an element in the target model, but only if it does not exist yet [23]. Graph grammars are very suited to define such transformations. Because of QVT, Tefkat is very suited to synchronize different models, but less suited to perform unidirectional model to model transformations.

ATL [20] stands for ATLAS transformation language. It is a textual transformation language with a declarative part similar to QVT, extended with an imperative part, and a toolkit which is an based on the Eclipse framework. It can be applied to QVT transformation scenarios. Models conform to one metamodel can be transformed to models conform to another metamodel. The transformation language of ATL is more expressive than QVT, but the tool still suffers from the same drawback as Tefkat that transformation scripts get very large. However, a so called refining mode exists, aimed to preserve unchanged parts of the model. A comparison of ATL to other model transformation approaches also mentions the large size of transformation scripts and mentions that the refining mode still has limitations [16].

CGT (Concrete syntax-based Graph Transformations) [16] is a new model transformation approach by Grønmo et. al. Transformations on models are specified with a graph transformation-like syntax with a LHS and RHS of a rule. Rules have been extended with a so called *collection operator*, which can be used to match several elements in the LHS of a rule, and combine them in the RHS of the rule using a concrete syntax. With this approach, complex transformations on models are shorter and more intuitive than similar implementations using the graph transformation tool AGG [40, 16] or the model transformation tool ATL [20, 16]. Their approach is still a proof of concept, and the implementation is hard coded to only work with activity diagrams.

GReAT is a tool that generates model transformation tools [11]. Given a source and target metamodel, graph transformation rules and flow control, the tool generates Java code. This code in turn can be executed to perform model transformations from models that are instances of the source metamodel to models that are instances of the target metamodel. Models must be in XMI format, although no specifics can be found of which tools are compatible. Transformation rules have to be defined in a custom textual format.

AutoFOCUS [37] is an approach to transform EMF Ecore models that uses a declarative specification of transformation rules, using a PROLOG rule-based mechanism. The implementation is a proof of concept that does not work with all features of EMF Ecore models, advanced features like multi-valued or ordered attributes are not supported yet.

Other work exists that aim to bring models and metamodels to the world of graph grammars and graph transformations. Recently, Kleppe and Rensink described how to formalize UML class diagrams by representing such diagrams as typed graphs [21]. Their work is rather complete, but some features like *ordered* are missing, and it does not fully apply to EMF Ecore models. Like

us, they concluded that class diagrams cannot be represented by simply typing graphs, and that constraints are required. Kuske et. al. describe a graph based semantics for UML models in general, and how graph transformation principles can be used to simulate and verify UML models [22]. Only core elements of UML models have been discussed however, advanced features have been left for future work. Perez et. al. discusses the suitability to use graph transformation tools for refactoring of models that represent JAVA programs [35]. They map model elements to graphs using examples, but no complete mapping is given, and they provide no real conclusions. Clan morphisms are a formalism for type graphs introduced to bring the concept of class inheritance to graph transformation systems [4]. Finally, some older work already mentioned approaches and ideas to use graph grammars as a visual language for object oriented systems [5]. It particularly mentions the application of graph transformation to those visual languages.

Chapter 7

Conclusion and future work

Contribution

The goal of our work was to add interoperability with modelling tools to GROOVE. We have achieved that GROOVE can import and export Ecore models and instance models using XMI. Ecore models are represented as graphs, for which we created a full Ecore to graph mapping. To make our work applicable in practice, we implemented a transformation tool that can transform Ecore models and instance models to graphs, and instance graphs back to instance models.

We use XMI [30] for model exchange to and from GROOVE. We performed an extensive research with an experiment to determine the compatibility of tools that support XMI for model exchange. The goal of this was to choose the set of tools we want GROOVE to exchange models with, and of which modelling standard these models should be. We conclude that model exchangeability using XMI is very poor. In some cases models from another tool cannot be imported at all, and in many cases models are imported with errors. Exchanging UML 2.1 models using XMI 2.1 led to more, and more serious problems than exchanging UML 1.4 models using XMI 1.2. In our opinion, exchangeability problems when exchanging UML models using XMI are mostly the result of the complex nature of UML and the generic nature of XMI. Ecore models have a native serialization format of XMI 2.0, and the tools that support Ecore models are able to exchange models without major problems. Based on these results, we chose to connect GROOVE to Ecore modelling tools.

We defined a complete mapping of Ecore to GROOVE, and discussed every element and property of the Ecore metamodel. Our mapping supports features often not supported by other approaches, like multi-valued attributes, ordered and opposite references and identifiers. We discussed their impact on instance models and how they should be represented in a graph representation. The graph representations of Ecore models and instance models are specifically targetted to the feature set of GROOVE. Ecore models are represented by a type graphs and constraint rules, instance models are represented by instance graphs.

Some properties of instance graphs cannot be enforced by typing and are instead enforced by constraints. Constraint rules are graph transformations rules that only detect constraint violations and do not alter the graph. When no constraint rule has a match in an instance graph, and if the instance graph is correctly typed, it validly represents an Ecore instance model and can be transformed back to one.

Finally, we implemented a Java transformation tool as a package in the GROOVE project to transform Ecore models and instance models to graph grammars with a type graph, instance graphs and constraint rules. The instance graphs in the graph grammars can, given also the Ecore model and only when no constraints are violated, be transformed back to Ecore instance models. With our mapping from Ecore models to graphs and the transformation tool that works in both directions, GROOVE can be used as a transformation tool for instances of Ecore models.

Evaluation of our work

An XMI compatibility research like ours has to our knowledge not been performed and published before, even though the apparent need. Some model transformation approaches [46, 45, 38, 27] and many modelling tools [18, 42, 26, 25, 43, 34, 3, 15, 9] use XMI to provide compatibility, but fail to report if and to what degree they are truly compatible. Online experiences indicate that model exchange is often not possible without errors. With our research, we were able to confirm that tools can rarely exchange UML models using XMI without errors and that exchanging Ecore model using XMI 2.0 led to the best results.

However, we only confirmed the poor exchangeability, but we did not determine the reasons for this. We believe the reasons are the complex nature and the different versions of UML and the generic nature of XMI. Our beliefs are based on our results and on examples we found, but we did not perform further research to confirm them.

Our mapping of Ecore models and instance models to graphs representations supports all features of Ecore modelling that are of importance for the structure of Ecore instance models. Furthermore, all choices in our graph representation are supported by GROOVE. Other work to define graph-based semantics for UML or Ecore models [21, 22, 35, 8, 5, 4] are often not complete or do not take the capabilities of a target graph transformations tool into consideration. Only with our approach is it possible to transform Ecore models to type graphs and Ecore instance models to instance graphs, and also transform instance graphs back to Ecore instance models.

We were forced to make some counterintuitive choices to represent Ecore models and instance models. References and attributes are represented by a node type with an edge to the target or value. By representing a reference as an edge, like other approaches [21, 22, 8, 1], it would not be possible to support ordered or opposite references. Also, by representing an attribute as an attribute of the node representing the containing class it would not be possible to support multi-valued and ordered attributes. The additional nodes that

represent references in instance graphs are ordered with *next* edges, imposing a local ordering of elements for this reference, and the same for attributes. Additionally, multiplicities of references and attributes are currently enforced with constraint rules instead of by the type graph, because GROOVE does not support multiplicities in type graphs. A more intuitive approach would be to use multiplicities in type graphs to limit the amount of allowed instances of nodes or edges in instance graphs.

The benefit of our choices is that all features of Ecore modelling that are part of the structure of instance models are supported. Furthermore, the graph representations are supported by a graph transformation tool, GROOVE. The drawback is that the nodification introduced by representing references and attributes as node types is counterintuitive for the graph representations, and this complicates graph transformation. We however believe that the benefits outweigh the drawback, since instance models can now be transformed to instance graphs, transformed in GROOVE, and transformed back without losing any information. This is not possible with other approaches.

Two features of Ecore modelling, operations and annotations, are not part of the structure of instance models. Therefore they are not supported in our graph representation of instance models. However, they can indirectly still have some influence on instance models, but it is not possible to support them. Operations defined in an Ecore model are often used to change instance models, for example to change attribute values, instantiate new elements or delete elements. If an operation changes an instance model, it could be possible to devise a graph transformation rule, or a set of rules, to achieve the same transformation in the instance graph representation. However, operations have no semantics specified for them in Ecore models. Semantics will have to be added in some way, using Java code or otherwise, to make a translation from operations to graph transformation rules possible. Annotations of elements are textual comments without semantics. They are sometimes used to denote constraints for valid instance models. For example, the Ecore metamodel uses annotations for constraints on valid Ecore models. Since these constraints have no semantics, it is not possible to create constraint rules to detect violations.

Generic types are also not supported. Generic classes and generic operations are not part of the structure of instance models and therefor do not need a representation. Generic datatypes are not serializable by EMF. Therefore values of them are not supported in Ecore instance models, and hence also not by our graph representation. This is a limitation of EMF.

At the moment, our approach can only be used to perform endogenous model transformations using GROOVE. Ecore instance models imported into GROOVE can be transformed to other models, but must always be instances of the same Ecore model. GROOVE supports multi-typing of graphs, where multiple active type graphs are merged. It could be possible to use GROOVE for exogenous model transformations by using a source and target Ecore model that are represented by a source and target type graph. The source instance graph must then by typed by the source type graph and the target instance graph by the target type graph. Intermediate instance graphs during the transformation must then be typed by the merged source and target type graph.

We implemented a Java transformation tool to transform Ecore models and instance models to graphs and instance graphs back to instance models. Ecore models and instance models are loaded and created using the API provided by the Ecore packages of the EMF project [42] so that we did not have to write our own XMI importer/exporter and could focus on the transformation itself. This means however that our approach does not easily extend to support XMI files containing UML models created by other tools. In order to support other modelling standards in the future, our transformation tool cannot be reused to import or export models of that standard that were serialized using XMI.

With our mapping from Ecore models to graphs and the transformation tool that works in both directions, GROOVE can be used as a transformation tool with verification and simulation features for instances of Ecore models. With the transformation tool we developed, our approach can be used in practice, and does not remain an on-paper idea.

Future work

We confirmed that model exchangeability using XMI is poor, and we believe the reasons for this are the complex nature of UML and the generic nature of XMI. Whether these are indeed the reasons will have to be investigated. Whether tools correctly implement the UML version they claim to support must be determined, and the generated XMI files must be compared to each other and to the XMI specification. Based on such an investigation, our beliefs might be confirmed, or other reasons might be found. Suggestions and work to improve model exchangeability between tools could then be done.

By representing references and attributes as node types instead of edges we introduced nodification into the graph representation. The reason for this representation is to support ordered and opposite references, and ordered and multi-valued attributes. This nodification could be prevented by extending GROOVE with a formalism to support ordered edges, for example by using hyper-edges [13]. Supporting opposite references would still remain an issue this way, so a solution would have to be found for this as well.

Multiplicities are currently not represented in type graphs, but violations of multiplicities are detected with constraint rules. Some other graph transformation tools, like AGG [40] support multiplicities in type graphs, and something like this could be implemented in GROOVE to more intuitively represent Ecore models.

Operations can be used to specify changes to instance models. By adding semantics to operations in Ecore models in some way, for example with Java code, it could be possible to create transformation rules that mimic the behaviour of operations. A suitable method to add semantics to operations will have to be investigated. These semantics will then have to be translated to graph transformation rules for GROOVE. It might not be possible to translate all operations to graph transformation rules, but what is possible and what is not will have to be investigated.

We discussed textual constraints in annotations. They lack semantics and cannot be translated to constraint rules for GROOVE that detect violations. Some model transformation approaches use OCL [31] to define constraints for valid models [12, 2]. If constraints in Ecore models would be added to annotations in the form of OCL constraints, it might be possible to translate them to constraints rules that detect violations. The applicability of this approach will have to be investigated.

In order to extent our approach to handle exogenous model transformations, multi-typing can be used for a source and target type graph for the source and target model. These type graphs can then be merged to type intermediate graphs during the transformation. The suitability of using merged type graphs to type intermediate instance graphs will have to be investigated, as well as how to deal with constraint rules to match constraint violations in intermediate instance graphs.

Currently we import Ecore models and instance models, but do not use diagram information. Ecore models can have diagrams that have layout information of elements of the model. These diagrams can be used to layout type graphs, instead of using the default forest layout of GROOVE. This way, the type graph can more resemble the Ecore model it represents. In future work, type graph layout using an Ecore model diagram could be implemented.

Bibliography

[1] Amelunxen, C., Klar, F., Königs, A., Rötschke, T., and Schürr, A. Metamodel-based Tool Integration with MOFLON. In *30th International Conference on Software Engineering* (New York, 05 2008), ACM Press, ACM Press, pp. 807–810.

[2] Amelunxen, C., Königs, A., Rötschke, T., and Schürr, A. MOFLON: A Standard-Compliant Metamodeling Framework with Graph Transformations. In *Model Driven Architecture - Foundations and Applications: Second European Conference* (Heidelberg, 2006), vol. 4066 of *Lecture Notes in Computer Science (LNCS)*, Springer Verlag, pp. 361–375.

[3] Artisan. Artisan Studio. `http://www.artisansoftwaretools.com/products/artisan-studio/`, July 2010.

[4] Bardohl, R., Ehrig, H., De Lara, J., and Taentzer, G. Integrating meta-modelling aspects with graph transformation for efficient visual language definition and model manipulation. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics) 2984* (2004), 214–228.

[5] Bardohl, R., Minas, M., Taentzer, G., and Schürr, A. Application of graph transformation to visual languages. *Handbook of graph grammars and computing by graph transformation: vol. 2: applications, languages, and tools* (1999), 105–180.

[6] Biermann, E., Ehrig, K., Khler, C., Kuhns, G., Taentzer, G., and Weiss, E. Graphical definition of in-place transformations in the Eclipse Modeling Framework. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics) 4199 LNCS* (2006), 425–439.

[7] Biermann, E., Ermel, C., and Jurack, S. Modeling the "Ecore to GenModel" Transformation with EMF Henshin. In *Transformation Tool Contest* (Malaga, 2010).

[8] Biermann, E., Ermel, C., Lambers, L., Prange, U., Runge, O., and Taentzer, G. Introduction to AGG and EMF Tiger by modeling a Conference Scheduling System. *International Journal on Software Tools for Technology Transfer 12*, 3 (2010), 245–261.

[9] Borland. Borland Together. `http://www.borland.com/us/products/together/index.html`, July 2010.

[10] Buttner, F., and Gogolla, M. Realizing graph transformations by pre- and postconditions and command sequences. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics) 4178 LNCS* (2006), 398–413.

[11] Christoph, A. Describing horizontal model transformations with graph rewriting rules. *Lecture Notes in Computer Science 3599* (2005), 93–107.

[12] De Lara Jaramillo, J., Vangheluwe, H., and Moreno, M. A. Using meta-modelling and graph grammars to create modelling environments. *Electronic Notes in Theoretical Computer Science 72*, 3 (2003), 39–53.

[13] de Mol, M., and Rensink, A. On A Graph Formalism for Ordered Edges. In *Preliminary Proceedings of the Ninth International Workshop on Graph Transformation and Visual Modeling Techniques* (2010), pp. 2–13. (to appear).

[14] Ehrig, H., Ehrig, K., Prange, U., and Taentzer, G. *Fundamentals of Algebraic Graph Transformation*. Springer-Verlag, 2006.

[15] Gentleware AG. Poseidon for UML professional edition. `http://www.gentleware.com/uml-software-pe.html`, July 2010.

[16] Grønmo, R., Møller-Pedersen, B., and Olsen, G. K. Comparison of three model transformation languages. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics) 5562 LNCS* (2009), 2–17.

[17] GROOVE (GRaphs for Object-Oriented VErification). `http://groove.cs.utwente.nl/`, August 2010.

[18] IBM. Rational Software Architect Standard Edition. `http://www-01.ibm.com/software/awdtools/swarchitect/standard/`, July 2010.

[19] ikv++ technologies AG. MediniQVT. `http://projects.ikv.de/qvt/`, August 2010.

[20] Jouault, F., and Kurtev, I. Transforming models with ATL. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics) 3844 LNCS* (2006), 128–138.

[21] Kleppe, A., and Rensink, A. *A Graph-Based Semantics for UML Class and Object Diagrams*. Centre for Telematics and Information Technology, University of Twente, Enschede, January 2008.

[22] Kuske, S., Gogolla, M., Kreowski, H., and Ziemann, P. Towards an integrated graph-based semantics for UML. *Software and Systems Modeling 8*, 3 (2009), 403–422.

[23] Lawley, M., and Steel, J. Practical declarative model transformation with Tefkat. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics) 3844 LNCS* (2006), 139–150.

[24] Levendovszky, T., Lengyel, L., Mezei, G., and Charaf, H. A systematic approach to metamodeling environments and model transformation systems in VMTS. *Electronic Notes in Theoretical Computer Science 127*, 1 (2005), 65–75.

[25] Microsoft. Microsoft Visio. `http://office.microsoft.com/en-us/visio/`, July 2010.

[26] Modeliosoft. Modelio Enterprise Edition. `http://www.modeliosoft.com/en/products/modelio-enterprise-edition.html`, July 2010.

[27] Muliawan, O., and Janssens, D. Model refactoring using MoTMoT. *International Journal on Software Tools for Technology Transfer 12*, 3 (2010), 201–209.

[28] Object Management Group. Meta Object Facility. `http://www.omg.org/mof/`, August 2010.

[29] Object Management Group. Model Driven Architecture. `http://www.omg.org/mda/`, August 2010.

[30] Object Management Group. MOF 2.0 / XMI Mapping Specification, v2.1.1. `http://www.omg.org/technology/documents/formal/xmi.htm`, August 2010.

[31] Object Management Group. Object Constraint Language (OCL). `http://www.omg.org/spec/OCL/2.2/`, August 2010.

[32] Object Management Group. Query/View/Transformation (QVT). `http://www.omg.org/spec/QVT/1.1/Beta2/`, August 2010.

[33] Object Management Group. Unified Modelling Languag. `http://www.uml.org/`, August 2010.

[34] Pagès, B. BoUML. `http://bouml.free.fr/`, July 2010.

[35] Pérez, J., Crespo, Y., Hoffmann, B., and Mens, T. A case study to evaluate the suitability of graph transformation tools for program refactoring. *International Journal on Software Tools for Technology Transfer 12*, 3 (2010), 183–199.

[36] Rensink, A. The GROOVE Simulator: A Tool for State Space Generation. In *Applications of Graph Transformations with Industrial Relevance (AGTIVE)* (Berlin, 2004), vol. 3062 of *Lecture Notes in Computer Science*, Springer Verlag, pp. 479–485.

[37] Schätz, B. Formalization and Rule-Based Transformation of EMF Ecore-Based Models. *Software Language Engineering: First International Conference, SLE 2008, Toulouse, France, September 29-30, 2008. Revised Selected Papers* (2009), 227–244.

[38] Schippers, H., Van Gorp, P., and Janssens, D. Leveraging UML profiles to generate plugins from visual model transformations. *Electronic Notes in Theoretical Computer Science 127*, 3 (2005), 5–16.

[39] Schürr, A. Specification of graph translators with triple graph grammars. In *Graph-Theoretic Concepts in Computer Science*, vol. 903 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, 1995, pp. 151–163.

[40] Taentzer, G. AGG: A graph transformation environment for modeling and validation of software. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics) 3062* (2004), 446–453.

[41] Taentzer, G., and Carughi, G. T. *A graph-based approach to transform XML documents*, vol. 3922 LNCS. Springer, 2006.

[42] The Eclipse Foundation. Eclipse Modeling Framework Project. `http://www.eclipse.org/modeling/emf/`, July 2010.

[43] Tigris.org. ArgoUML. `http://argouml.tigris.org/`, July 2010.

[44] Vangheluwe, H., de Lara, J., and Mosterman, P. An introduction to multi-paradigm modelling and simulation. In *In Proceedings of the 2002 AI, Simulation and Planning in High Autonomy Systems* (2002), pp. 9–20.

[45] Varró, D., and Balogh, A. The model transformation language of the VIATRA2 framework. *Science of Computer Programming 68*, 3 (2007), 187–207.

[46] Varró,D. and Balogh,A. VIsual Automated model TRAnsformations. `http://dev.eclipse.org/viewcvs/indextech.cgi/gmt-home/subprojects/VIATRA2/index.html`, August 2010.

[47] Wagner, R. Developing Model Transformations with Fujaba. In *Proc. of the 4th International Fujaba Days 2006, Bayreuth, Germany* (2006).

[48] World Wide Web Consortium (W3C). XSL Transformations (XSLT) Version 1.0. `http://www.w3.org/TR/xslt`, August 2010.