December 15, 2019

MASTER THESIS

A formalisation of EMF by expressing Ecore as GROOVE graphs

Remco de Man

Faculty of Electrical Engineering, Mathematics and Computer Science (EEMCS)
Formal Methods and Tools

Exam committee:
prof. dr. ir. A. Rensink
dr. ir. S.J.C. Joosten
dr. ir. M.J. van Sinderen

Documentnumber
—

UNIVERSITY OF TWENTE.

Abstract

Within the field of software verification, software is verified to be correct using models. However, the modelling landscape is very diverse, and multiple modelling techniques exist to model software. Model transformations can help to bridge the gap between these techniques, but often do not have a formal foundation, which is problematic for software verification. Within this work, the model transformations between models based on EMF's Ecore and GROOVE grammars are formalised. A transformation framework is introduced to create model transformations between Ecore models and GROOVE grammars while maintaining a formal foundation. This framework allows for creating significant model transformations out of smaller transformations that are more easy to proof. An application is used to show how model transformations can be built using this framework.

Contents

Chapter 1

Introduction

Software engineering is becoming an increasingly challenging task nowadays. Developing software with complex architectures and nontrivial implementations is a prevalent task for the modern software engineer, having implications on how software is developed. At the same time, software that can be proven to be error-free has become increasingly important. The reason for this is apparent. Sophisticated systems automate more and more crucial tasks. Failure of these systems might have enormous consequences, especially for safety-critical and healthcare systems. Therefore, multiple strategies have been developed over the years to ensure that crucial parts of these systems are error-free.

An increasingly popular method for dealing with the development of complex systems is by using domain-specific models. Model-Driven Engineering (MDE) is a field within software engineering that focuses on using and creating domain models that describe complex software systems on a domain level. These models can then be used for different tasks, depending on the type of model. These tasks include code generation, but also different forms of verification of the software. By using these models, it becomes easier to reason about the developed software, while also allowing for systematic code generation and verification.

Although domain-specific models provide a strategy to deal with the development of complex systems, they do not automatically ensure that the software is error-free. Software verification is an essential strategy in ensuring that software systems are error-free. Modern methods of software verification use automated tools that can use software models to verify the correctness of a system. These tools use some model of the system to verify a set of requirements provided by the software engineer. By using structural checks on the model, the tool can tell if these requirements are met.

A possible problem that might arise when using domain-specific models for software development is the interoperability of different models. Within the area of MDE, a lot of different frameworks and tools exist. Each of these frameworks and tools focuses on a specific set of functionality. As a result, models created in one framework well suited for code generation, might not be useful in the context of software verification. In an ideal world, the format of the produced models would be standardised across all frameworks for smooth interoperability. In reality, different frameworks use different formats which are optimised for their specific set of functions. These different formats make it difficult to share models across different frameworks and applications.

Model transformation is a concept in the field of MDE that focuses on solving this problem. Model transformation is an automated way of modifying and creating models by transforming existing models. By using model transformations, it is possible to transform a model that is tailored towards code generation into a model that is suited for software verification, without the need to create a new model for this purpose.

Model transformations have already led to various tools and services that can export and import models in different tools and frameworks. These tools and services allow a software engineer to transform a model suited for code generation into a model suited for verification, and therefore use one model and its transformations to achieve both tasks. Sadly, these model transformations rarely have a formal foundation. Having a formal foundation for the model transformations is useful in the context of software verification since it allows for proving the correctness of the transformation itself. When a transformed model is used to verify a software system, the results of the verification can only be considered correct if the transformation is correct. Without proof of correctness of the transformation, it might be that the verification results are incorrect because the original model might have a different meaning than the transformed model.

This thesis will contribute to fields of MDE and Software Verification by specifying a formal foundation for model transformations between EMF/Ecore (Section 2.1), a framework for software modelling in

which various models can be created, and GROOVE (Section 2.2), a tool for software verification based on graph grammars. Furthermore, a framework is presented in which these model transformations can be proven correct, allowing the user to build correct model transformations iteratively.

1.1 Formalisation of model transformations

As explained earlier, model transformations are an automated way of modifying and creating models by transforming existing models. Model transformations can be used in a variety of scenarios, from simple modifications within the same domain and language (an endogenous transformation) to conversions between different domains and languages (an exogenous transformation). Furthermore, model transformations can be unidirectional, meaning that a model can only be transformed one way, or bidirectional, meaning that the model can be transformed in both directions. Unidirectional transformations are particularly useful in situations where the output model is meant to be used as a final result, such as code generation. Bidirectional transformations are necessary for situations where the models must be kept consistent. In that case, a change to one model might necessitate a change to the other model, which then can be automated using model transformations.

Since this thesis focuses on model transformations between EMF/Ecore and GROOVE, this thesis focuses on bidirectional exogenous transformations. The transformations between EMF/Ecore and GROOVE are exogenous by definition, since the languages of EMF/Ecore and GROOVE are different, as will be shown later. The bidirectionality of the transformations is beneficial to ensure consistency, which is a useful property to have in software verification.

In order to prove any property on these model transformations, the transformations need to be formalised. The formalisation of a model transformation consists of mathematical definitions and functions that describe the behaviour of the transformation, allowing to mathematically translate an input model to an output model as described by the model transformation. These definitions and functions directly depend on the formalisations of the input and output models themselves, as these are needed to describe the input and output models of the transformations. Because of this dependency, the formalisations of EMF/Ecore and GROOVE must be established as well.

The main disadvantage of the formalisation of model transformations is the direct relationship between the transformation and its input and output language. As a consequence, the formalisation of a model transformation directly depends on the formalisations of its input and output languages. Therefore, it is not possible to give an abstract formalisation for model transformations between different languages. Creating such a formalisation would mean making the formalisations of the input and output languages more abstract. Making these more abstract might result in loss of information, which is undesirable, or an increase in complexity. Within this thesis, this disadvantage was dealt with by only focusing on the model transformations between EMF/Ecore and GROOVE.

1.2 Correctness of model transformations

As explained in Section 1.1, this thesis will define a formalisation for model transformations from EMF/Ecore to GROOVE and vice versa. However, a formalisation of the transformation itself does not prove anything about its properties and correctness. In order for the formalisation of the model transformation to be useful in the context of software verification, it is essential to prove its correctness. Therefore, it is crucial to establish what it means for a model transformation to be correct.

As explained earlier, the model transformations between EMF/Ecore and GROOVE are exogenous and bidirectional. This bidirectionality means that for every transformation from EMF/Ecore to GROOVE, there exists a transformation back, from GROOVE to EMF/Ecore. Since GROOVE and EMF/Ecore are very different, there are elements in EMF/Ecore that cannot be expressed in GROOVE and vice versa. Because of the difference, it might not be possible to use one mapping in both directions. Therefore, it might be the case that for a transformation from EMF/Ecore to GROOVE, a different transformation function is used to convert the model back from GROOVE to EMF/Ecore. In this case, two unidirectional transformations are used to achieve bidirectionality.

Throughout this thesis, the correctness of a model transformation is defined as the syntactical correctness. The semantics are not further discussed as the semantics might differ from model to model, depending on what the creator intended to model. The following properties must hold for the formalisation for it to be correct. Please note that since GROOVE is based on graph grammars, one does not speak of a GROOVE model, but rather a GROOVE graph:

- For each valid EMF/Ecore model that is transformed to GROOVE, the resulting GROOVE graph must be syntactically valid.

- For each valid GROOVE graph that is transformed to EMF/Ecore, the resulting EMF/Ecore model must be syntactically valid.
- For each valid EMF/Ecore model that is transformed to GROOVE, there exists a known transformation from the resulting GROOVE graph back to the original EMF/Ecore model.
- For each valid GROOVE graph that is transformed to EMF/Ecore, there exists a known transformation from the resulting EMF/Ecore model back to the original GROOVE graph.

These properties assume that it is clear what it means for EMF/Ecore models and GROOVE graphs to be syntactically valid. Therefore, the formalisations of EMF/Ecore and GROOVE will specify the syntactical correctness of their models and graphs.

The properties discussed above are useful in the context of software verification since they show that the transformed models and graphs are indeed a valid transformation of their original counterparts. Therefore, this thesis will not only define the formalisation for the model transformations but also show that the properties discussed above hold for these transformations.

1.3 Approach and composability

As explained in the previous sections, this thesis will provide a formalisation for the model transformations between Ecore and GROOVE and also prove the correctness of the transformations. Although this a noble goal, it comes with many complexities.

First of all, Ecore and GROOVE both have a very different nature. Ecore is mostly based on a subset of UML, as discussed in Section 2.1. On the other hand, GROOVE is based around graph grammars and therefore mathematical graph theory. As a consequence, the set of features is very different. Ecore has elements that are not directly expressable in GROOVE and vice versa. When providing the formalisation for the transformations, the different features within both languages should be taken into account.

Furthermore, Ecore and GROOVE have a lot of different elements within their models and grammars. When transforming these models and grammars, all these elements need to be transformed. Transforming all these elements at once is a very complex problem, as these different elements can be used in infinitely many combinations, each requiring a different transformation. Not only must the formalisation be able to express all these different combinations, but each of these combinations must also be proven correct.

In order to overcome the problems that are raised by these complexities, the divide and conquer-principle will be applied. This thesis will provide a framework in which model transformations and their proofs can be composed out of smaller transformations and their proofs. This composability allows for proving only small parts of the problem, which then can be composed to express the countless combinations of model transformations.

1.4 Research question

This thesis will focus on defining a formalisation for model transformations from Ecore to GROOVE and vice versa, and also proving the correctness of these transformations. It will try to achieve this goal by providing a way to compose more substantial model transformations out of smaller ones. In short, the thesis will answer the following research question:

"What is a suitable formalisation for composable model transformations between Ecore and GROOVE that gives rise to correct model transformations between Ecore and GROOVE?"

It is immediately clear that this research question consists of multiple facets. In order to make answering the research question easier, the research question will be split into smaller questions based on the different facets of the main question. The following subquestions will be answered:

1. "What is a suitable formalisation of Ecore models and what Ecore models are valid within this formalisation?"

   In order to transform between Ecore and GROOVE, a formalisation of Ecore is needed. As explained earlier, this formalisation needs to give rise to a definition of valid Ecore models, which are needed to prove the correctness of the transformations later.

2. "What is a suitable formalisation of GROOVE grammars and what GROOVE grammars are valid within this formalisation?"

   Just like the previous question, a formalisation that captures GROOVE grammars is needed. Like the previous question, this formalisation should also give rise to a definition of valid GROOVE grammars for use in proving the correctness of the transformations.

3. "What is a suitable formalisation for the model transformations between Ecore and GROOVE?"

   A suitable formalisation for the model transformations between Ecore and GROOVE is needed to describe the model transformations between Ecore and GROOVE formally. Such a formalisation must be able to express the infinite combinations of possible model transformations. This formalisation forms the basis of the correctness of model transformations and their composability. Therefore, this question is the foundation of the main result of this thesis.

4. "What model transformations are correct within the formalisation?"

   This question will answer the question which model transformations within the formalisation are correct model transformations between Ecore and GROOVE. These transformations are of interest, as only these transformations can be used with confidence within formal applications.

5. "How can correct model transformations between Ecore and GROOVE be composed?"

   A fundamental part of this thesis is to compose small model transformations into larger ones. This composability allows for only proving the correctness of small model transformations and then combining them without loss of correctness. This question answers how to compose correct model transformations into a new model transformation while preserving correctness.

When these subquestions are answered, it is possible to formulate an answer to the main research question. A suitable formalisation for model transformations between Ecore and GROOVE will follow from subquestions 1, 2 and 3. Subquestions 1 and 2 provide the formalisations of Ecore and GROOVE themselves, which will be used to formalise their model transformations. Subquestion 3 defines the formalisation of the model transformations. The correctness of model transformations within this formalisation will follow from subquestions 1, 2 and 4. Subquestions 1 and 2 will provide the definitions needed to prove correctness, while subquestion 4 will give a proof for the correct model transformations. Finally, the composability of these model transformations follows from subquestion 5, which answers how to combine correct model transformations while preserving correctness.

1.5 Validation

This section describes how the research questions of this thesis will be validated. The main research question of this thesis will be validated by validating the subquestions. For each subquestion, the validation process is different:

- "What is a suitable formalisation of Ecore models and what Ecore models are valid within this formalisation?" and "What is a suitable formalisation of GROOVE grammars and what GROOVE grammars are valid within this formalisation?"

  The answer to these questions will be validated through existing theory about these modelling languages. Existing theories describe the different elements in these languages and the constraints between them. These give rise to domains for both languages, which can be used to formalise the language. The correctness of the grammars and models in these languages follow from literature in the same way, as the literature defines which grammars and models are valid within these languages.

- "What is a suitable formalisation for the model transformations between Ecore and GROOVE?"

  A suitable formalisation must be able to express a reasonable set of model transformations. If the formalisation is not able to express such a set, the formalisation is useless. Therefore, the thesis will show examples of model transformations within this formalisation and give an intuition of which transformations are possible. The existence of these examples validates the suitability of the formalisation.

- "What model transformations are correct within the formalisation?"

  The correctness of the model transformations follows from a correctness proof. This proof is validated using a theorem prover, which ensures that the proof is sound and complete. Therefore, the theorem prover validates the proof, while the proof validates the answer to the question. Furthermore, examples of correct model transformations will be provided, which validates that correct model transformations exist within the formalisation.

- "How can correct model transformations between Ecore and GROOVE be composed?"

  This subquestions answers how correct model transformations can be composed such that the result is also correct. Validating this question consists of two parts. In the first part, a correctness proof is given, which shows that the composed model transformations are indeed a correct model transformation itself. This correctness proof is validated using a theorem prover. In the second part, an application of the composability of model transformations is shown, which validates that composing model transformations is possible in practice.

Since the answer to the main research question follows directly from the answers to the subquestions, the answer to the main question is validated using the validation of the subquestions.

1.6 Related work

In this section, the work related to this thesis will be discussed. The related work is divided into multiple sections that each describe a different facet related to this thesis.

1.6.1 Formalisations of modelling languages

This section discusses some related work in the field of formalisations of modelling languages. The work presented here is relevant to this thesis as the formalisations of Ecore and GROOVE have an essential role throughout this thesis.

In [14], Kleppe and Rensink present a straightforward formalisation of UML models using graph theory and graph constraints. Since Ecore is many facets similar to UML, this formalisation provides a reasonable basis for formalising Ecore as well. Such formalisation has an advantage that it is already built upon graph theory, which allows for an easy formalisation of the transformation to other graph languages. Although the work presented does include formalisations for most relevant elements of UML models, it does not have enough expressive power to formalise concepts unique to Ecore. Within this thesis, a formalisation of Ecore is used that is much closer to the Ecore implementation, with enough expressive power to formalise all the relevant concepts.

Within UML, it is possible to describe a model and its constraints using the Object Constraint Language (OCL) [18]. Most queries and invariants written in OCL can also be applied to Ecore models. Moreover, EMF has its declarative language EMF-IncQuery [9], which can handle complex constraints that cannot be expressed using OCL.

In [17], Semeráth et al. present a way to formalise EMF/Ecore by expressing a subset of OCL and EMF-IncQuery in first-order logic. Within this work, each Ecore model is expressed as multiple sets of named elements. These elements are constrained by OCL and EMF-IncQuery invariants, expressed in first-order logic. The goal is to use automated reasoners to analyse the models automatically. Because OCL and EMF-IncQuery are more expressive languages than first-order logic, approximations are used where necessary.

The work presented by Semeráth et al. has a particular relation to this thesis since they try to formalise Ecore to be able to perform formal verification on the Ecore models. In a way, this goal is similar to the goal of this thesis, but the approach is different. Instead of formalising Ecore with the goal of verification, formalising Ecore is in this thesis merely a tool for providing a formalisation of model transformations to GROOVE. Verification is achieved through GROOVE, which is developed solely for this purpose.

1.6.2 Formalisations of model transformations

This section discusses related work in the field of formalisations of model transformations. Existing work in this field that is relevant is mostly related to the concept of a Triple Graph Grammar (TGG). Whereas a Graph Grammar can be used to describe the evolution of a single graph model, TGGs allow for describing the relation between two graph models and also allow for transforming one kind of model to the other [13]. The formal description of model transformations using TGGs is especially relevant to this thesis, as this thesis will also formalise a specific set of model transformations.

In [10], Hermann, Ehrig, Golas, and Orejas approach the problem of formal analysis of model transformations using triple graph grammars. They explain how triple graph grammars can be used to describe model transformations and which problems arise when performing this task. Properties related to the syntactical correctness, functional behaviour and information preservation are discussed.

The work of Hermann, Ehrig, Golas, and Orejas discusses model transformations on a more abstract level than this thesis, by providing mathematical properties and mathematical structures to approach the problem. These structures and properties are not applied to specific modelling languages. In this thesis uses a more practical approach where Ecore models are transformed to GROOVE graph grammars and vice versa. This approach allows for a mathematical specification that is tailored for these modelling languages and can, therefore, discuss specific properties of these languages in detail.

An application of TGGs on the model transformation of Ecore models is shown by [3]. In this work, Biermann, Ermel, and Taentzer use TGGs to formalise the behaviour of model transformations between EMF models. This formalisation is done by formalising EMF models as graph grammars first and then using these graph grammars as part of the TGGs for formalising model transformations within EMF.

Ermel, Hermann, Gall, and Binanzer later use this work in [6] to create an Eclipse plugin that can describe model transformations between Ecore diagrams visually, including the possibility to edit them.

The work presented by Biermann, Ermel, and Taentzer uses a formalisation of EMF to describe model transformations formally. This formalisation is similar to the work presented by this thesis but focuses on endogenous transformations (transformations between EMF models) instead of exogenous transformations (transformations from Ecore to GROOVE, in case of this thesis).

In [4], Bruintjes has worked on mapping multiple languages to GROOVE and back using an intermediate conceptual model. This intermediate conceptual model can express Ecore diagrams as well, and therefore Bruintjes provides an implementation of model transformations between Ecore and GROOVE. Because the approach of this work focuses on the implementation, the model transformations are not formalised in this work. It is still worth mentioning because it is the only work that has a focus on transformations between Ecore and GROOVE specifically. Moreover, the conceptual model used within this work does not use graph grammars as a basis, which provides more freedom in expressing specific properties of Ecore.

The work presented by Bruintjes uses a similar approach for formalising Ecore models itself. This thesis will a formalisation inspired by this work, which is like the work of Bruintjes not based on graph grammars. It differs from the work of Bruintjes by focusing on the formal foundation rather than the implementation. Moreover, this thesis only focuses on the model transformations between Ecore and GROOVE, rather than multiple languages and GROOVE.

1.7 Contribution

This section discusses the intended contribution of this thesis to the active field of research. This thesis will propose a transformation framework for bidirectional transformations between EMF/Ecore and GROOVE. This transformation framework makes it possible to compose transformations while maintaining a formal proof of its syntactical correctness. As discussed in Section 1.6, most active research uses Triple Graph Grammars to deal with the problem of the formalisation of model transformations. This thesis will take a different approach by not modelling EMF/Ecore as a graph language, but rather using a more specific formalisation. Therefore, the formalisation of the transformations will not be based on Triple Graph Grammars, but it will borrow some similar concepts.

Within this work, there will be a focus on the transformations between EMF/Ecore and GROOVE. No earlier work exists that focuses on the formalisation of the transformations between these languages specifically. Because of the focus on these two languages, a practical approach can be used that results in a framework that can be used to create transformations between these two languages directly. Within existing work, either a more abstract method is used, or the formalised transformations are endogenous (e.g., in the work of Biermann, Ermel, and Taentzer [3]).

The result of this work can be a valuable foundation for verifying Ecore software models within GROOVE. Furthermore, it could be a valuable contribution to the field of formalised model transformations in general, since it uses an approach different than using TGGs for achieving a formalisation of exogenous transformations.

1.8 Outline

Within this thesis, a framework for formalising model transformations will be provided, including examples and applications. In Chapter 2, more information on EMF/Ecore and GROOVE is provided. Furthermore, the theorem prover that is part of validating the proofs is introduced. In Chapter 3, the formalisations of Ecore and GROOVE are introduced. In Chapter 4, a framework is introduced for formally expressing composable model transformations. As a part of this chapter, the formalisation of model transformations between Ecore and GROOVE is introduced. The chapter also introduces the definitions needed to compose these model transformations. Chapter 5 introduces a non-exhaustive library of model transformations within this framework with corresponding proofs, which provides examples of the model transformations, which can be expressed within this framework. Furthermore, Chapter 6 shows the composability of these model transformations by providing an example of composing smaller model transformations in a practical example. Finally, Chapter 7 concludes the thesis by answering the research questions and discussing possible future work.

1.8.1 Mathematical notation

Throughout this thesis, a lot of mathematical definitions and proofs are introduced. In order to accommodate for these definitions and proofs, prior knowledge of commonly used mathematical notations is assumed. For completeness, the meaning of the different braces and parentheses is as follows:

- Braces, "{}", are used to denote mathematical sets;
- Angle brackets, "⟨⟩", are used to denote mathematical sequences and named tuples;
- Parentheses, "()", are used to denote unnamed tuples or grouping within expressions.

Besides commonly used notations, new notations are introduced as part of some definitions throughout this thesis.

1.8.2 References to validated proofs

As explained Section 1.5, the formal proofs within this thesis will be validated using a theorem prover. In order to easily find the validated proofs corresponding to definitions and theorems, all relevant definitions and theorems will include a reference to the validated proof. Such a reference can be recognised by the  symbol and includes the corresponding name of the definition or theorem. For example, a reference to the theorem *mult_zero_unbounded_valid* from Appendix A would be written as  `mult_zero_unbounded_valid` in `Ecore.Multiplicity`. The proofs referenced by this thesis can be found on https://github.com/RemcodM/thesis-ecore-groove-formalisation. For more information on the theorem prover used for validating the proofs within this thesis, please refer to Section 2.3.

Chapter 2

Background

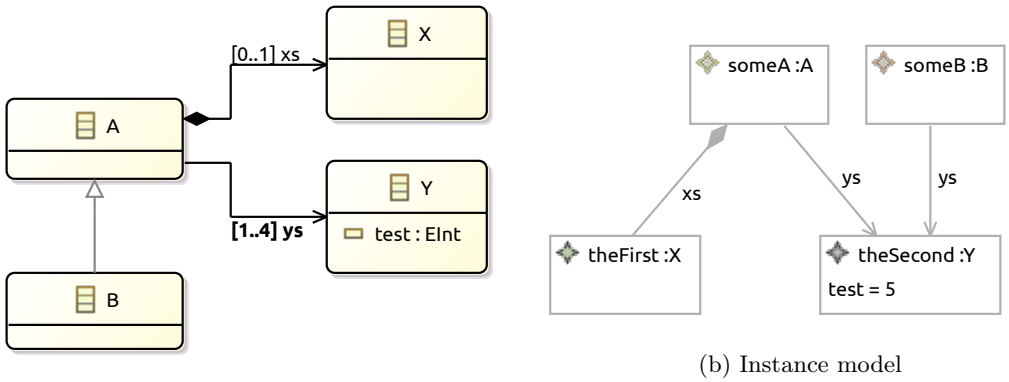
This chapter discusses the background required to understand the different formalisations and the transformations framework introduced within this thesis. Within this chapter, EMF/Ecore is explained in more detail, as well as GROOVE. Furthermore, this chapter introduces the Isabelle proof assistant, a theorem prover which will be used to validate the proofs throughout this thesis.

2.1 Eclipse Modeling Framework

The Eclipse Modeling Framework (EMF) [7] is a modelling framework and code generation facility for building applications based on a structured model. It is quite popular in the field of Model-Driven Engineering because of its open-source nature. EMF offers support for creating, editing and translating models based on its metamodel Ecore [1]. Models based on the Ecore metamodel are very comparable to UML class diagrams, but with properties specifically focused on software development. This focus makes models based on Ecore very suitable for object-oriented code generation as the structure of the model is already very similar to the class diagram of the corresponding application.

Because of the open-source nature of the Ecore metamodel and EMF, it has become increasingly popular for expressing domain models, creating editors for domain logic and code generation from domain models. However, EMF does not provide functionality for automated verification of its models out of the box. Different tools should be used to accomplish this task.

This thesis will focus on two levels of models based on the Ecore metamodel. The first level of models are models directly based on the Ecore metamodel, which will be called type models throughout this thesis. The second level of models are models based on a type model, and thus indirectly on the Ecore metamodel, and will be called instance models throughout this thesis. A simplified version of the Ecore metamodel [5] with elements relevant to the formalisation is given in Figure 2.1.



(a) Type model

(b) Instance model

Figure 2.2: Examples of different models in Ecore

2.1.1 Type models

A type model represents the first level of models based on the Ecore metamodel that will be used within this thesis. Since a type model is directly based on the Ecore metamodel, the metamodel of a type model is the Ecore metamodel. Since models based on the Ecore metamodel can best be understood as UML class diagrams, a type model can best be compared to a UML class diagram. Figure 2.2a shows the visual

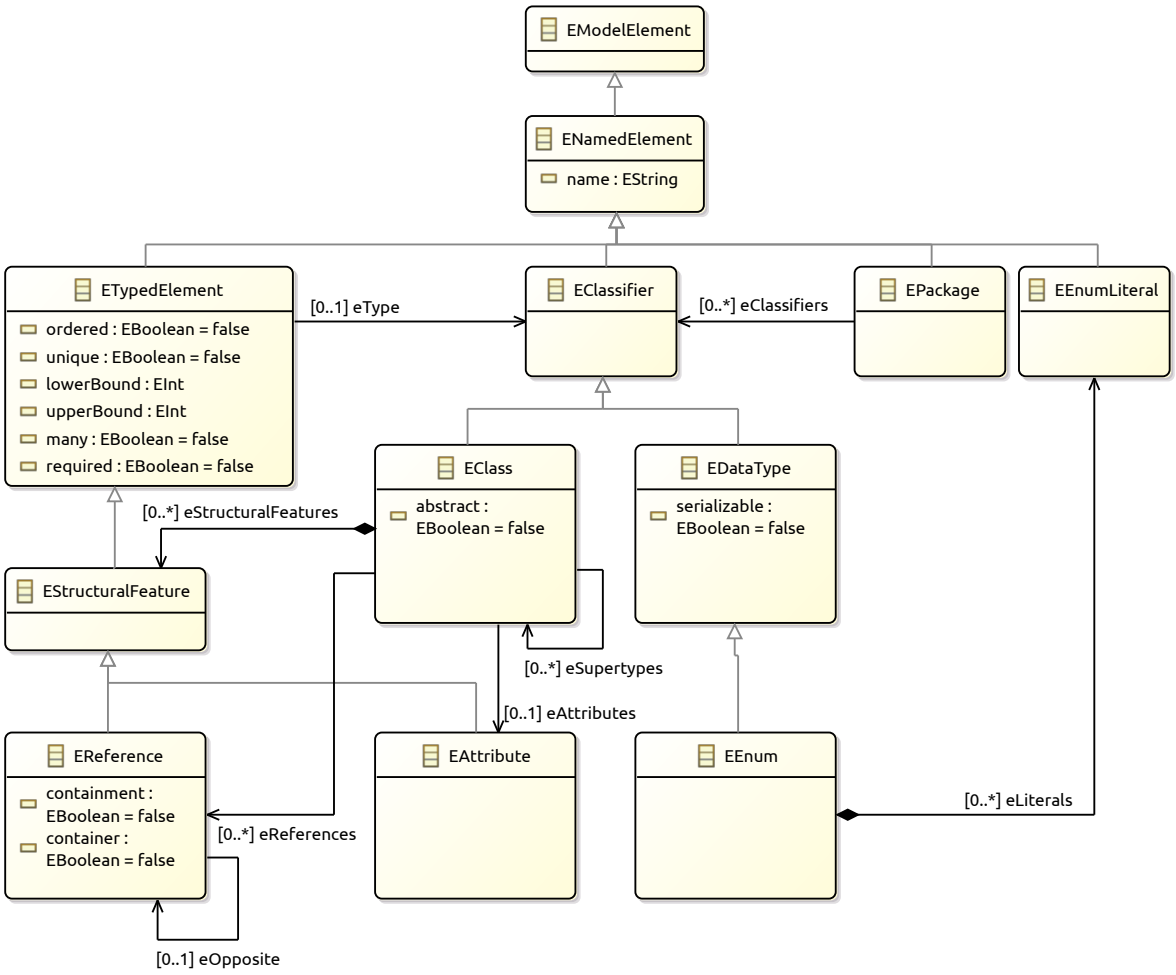


Figure 2.1: Simplified version of the Ecore metamodel

notation of a type model in EMF's own visual notation. Familiar concepts from class diagrams can be found in this visualisation. First of all, the figure shows four class types, A, B, X and Y. An example of inheritance of class types is shown, as class B extends class A, so class B is a subtype of class A. A has two relations, named xs and ys. Relation xs is a relation to class X. Furthermore, the figure shows that xs is a containment relation with a multiplicity of 0..1. There is a second relation ys, which has a multiplicity of 1..4. Finally, class Y has an attribute named test, which represents an integer.

2.1.2 Instance models

An instance model is the second level of models based on the Ecore metamodel that will be used in this thesis. An instance model is directly based on a type model. Therefore, the metamodel of an instance model is its corresponding type model. As a consequence, the metametamodel of an instance model is the Ecore metamodel. Figure 2.2b shows the visual notation of an instance model based on EMF's own notation, typed by the type model of Figure 2.2a. The figure shows one instance of every class type. The instance of class A has values for both the relations xs and ys. The xs relation references the instance of class X and the ys relation the instance of class Y. The instance of class B only has a value for the relation ys, which references the instance of class Y. The instance of class Y has a value set for the test attribute, which is equal to integer 5. Finally, all instances have a corresponding identifier, which is *someA* for the instance of class A, *someB* for the instance of class B, *theFirst* for the instance of class X and *theSecond* for the instance of class Y.

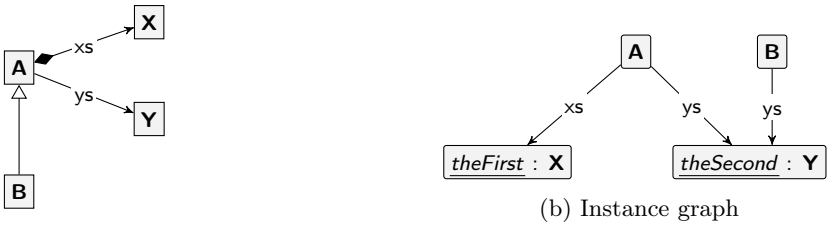
2.2 GROOVE

GROOVE [8] is an open source tool which uses graphs for modelling object-oriented software and for performing verification on these graphs. GROOVE is based on graph theory and makes uses the concept of graph grammars to relate the different kind of graphs. The graphs created within a graph grammar can be further analysed using LTL and CTL properties to verify if specific properties hold on the specified graphs. When the graphs represent the design-time, compile-time, or run-time structure of a software system, the results of this analysis can be used to verify which properties hold for the software system.

GROOVE defines multiple graph types, including (but not limited to) *type graphs*, *instance graphs* and *rule graphs*. These different graph types are used to achieve the grammar structure. Type graphs define the structure of instance graphs and rule graphs, while rule graphs describe a translation rule of an instance graph to another instance graph while maintaining the structure enforced by the type graph.

GROOVE is specially created for verification of software and uses proven techniques from logic and graph theory to verify properties on the graphs created within the tool. Although GROOVE provides excellent tools for performing verification on its graphs, there are no tools to achieve other goals, such as code generation.

This thesis will focus solely on type graphs and instance graphs. Although rule graphs might be useful in the context of model transformations and their formalisations, they are out of the scope of this thesis.



(a) Type graph

(b) Instance graph

Figure 2.3: Examples of different graphs in GROOVE

2.2.1 Type graphs

As explained before, a type graph defines the structure of instance graphs and rule graphs. It is a graph type which supports concepts as inheritance, abstractness of nodes and multiplicities of edges. Figure 2.3a shows the visual notation of a type graph in GROOVE its own notation. It consists of 4 nodes types, A, B, X and Y, with 2 relations. The first relation is the xs relation between A and X and the second relation is the ys relation between A and Y. Finally, we also see the concept of inheritance, with a subtype relation between node type B and A.

2.2.2 Instance graphs

An instance graph is a graph that describes actual instances of the types defined by a type graph. The description of these instances consists of the instance itself, optional identifiers and the relation to other instances. Figure 2.3b shows the visual notation of an instance graph in GROOVE its own notation. This instance graph is based on the type graph of Figure 2.3a and shows one instance of every type. The A-typed instance has a relation of type xs to the X-typed instance. Furthermore, it has a relation of type ys to the Y-typed instance. Finally, the B-typed instance also has a relation of type ys to the same Y-typed instance.

It should also be noted that the X-typed and Y-typed instances have identifiers. For the X-typed instance, this identifier is *theFirst*, while for the Y-typed instance, the identifier is *theSecond*.

2.3 Theorem proving using Isabelle

As mentioned before in Section 1.5 all formal proofs within this paper are verified within a theorem prover, sometimes also called a proof assistant. A theorem prover is a software solution to assist with the task of proving mathematical theorems. It achieves this goal by using automated reasoning and mathematical logic to provide the user with information on the correctness of the written proof. Furthermore, a theorem prover might have tools to prove simple theorems automatically.

In this thesis, the Isabelle proof assistant is used to prove the relevant theorems. Isabelle was chosen as theorem prover for several reasons:

- Isabelle stays close to mathematical definitions while still maintaining much automation in the process, which is different from other theorem provers. A comparison of theorem provers has shown that most other theorem provers which stay close to the mathematical definition do not have much automation, and vice versa [21].
- Isabelle provides a plug-in to jEdit, a text editor, which deeply integrated Isabelle into the jEdit editor. This integration allows for interactively creating theories and checking proofs without the need to use the command-line application for this purpose.
- Isabelle has its own proof language Isar, which makes proofs more readable to the human reader, without giving up on automation and functionality for delivering proofs.
- One of the supervisors of this thesis has experience with Isabelle, meaning that there is local expertise available in case of problems.

The remaining part of this section will discuss different parts of Isabelle and its proof language Isar, to provide some background on theorem proofing in Isabelle.

2.3.1 About Isabelle

Isabelle [12] is a generic theorem prover written in ML. It was originally developed at the University of Cambridge and Technische Universität München, but now includes numerous contributions from institutions and individuals worldwide. It has been designed to be able to support reasoning in several object-logics, which include but are not limited to:

- first-order logic, constructive and classical versions (Isabelle/FOL)
- higher-order logic (Isabelle/HOL)
- Zermelo-Fraenkel set theory (Isabelle/ZF)

Isabelle is distributed for free under a mix of open-source licenses, but the main code-base is subject to BSD-style regulations. More specifically, the binary distributions of Isabelle come with the 3-Clause BSD License [19].

Isabelle has quite a large user base and a well-maintained community. Besides a mailing list for users, there is also a wiki [11] and active support on StackOverflow, when questions are tagged under 'isabelle' [15].

The first release of Isabelle is published in 1986. Nowadays, it receives yearly releases with new updates. At the time of writing, Isabelle 2019 is the newest release, which is also the release used to prove the theorems in this thesis.

2.3.2 Basics

This section will discuss some basics on Isabelle that are relevant for this thesis. The constructs mentioned here will not be discussed in much detail, as that would be a thesis on its own. The documentation provided with Isabelle does a decent job explaining all constructs in as much detail as possible.

Theories

Each document in Isabelle is a *theory*, which can define a set of definitions, theorems and proofs. A theory can import other theories in order to reuse its definitions, theorems and proofs. An example of a theory is given in Appendix A. This theory is the actual formalisation of Definition 3.1.1 in Isabelle used for this thesis. It will be used as an example throughout the remaining parts of this section.

Datatypes

Within an Isabelle theory, it is possible to define inductive datatypes. Inductive datatypes are the most used way to define new types in Isabelle. Famous data structures, such as lists, can be defined using datatypes.

The example provided in Appendix A defines a new datatype for the set $\mathbb{N} \cup *$. The definition is specified within the theory as:

datatype $\mathcal{M} = Star \mid Nr\ nat$

This example defines a datatype called $\mathcal{M}$, which can have two values: $Star$ and Nr. $Star$ and Nr are called datatype constructors and can get arguments of different types. For example, the Nr value gets an additional type, nat, which is Isabelle's type for representing natural numbers. On the other hand, the $Star$ constructor gets no additional arguments and is just a value for $\mathcal{M}$ on itself.

The formalisation achieved here should be straightforward. $Star$ is used to denote $*$, the unbounded value, while $Nr\ nat$ is used to denote a bounded value for a multiplicity.

Record types

Another way of defining new types within Isabelle is by using record types. A record can be defined using the **record** keyword. The concept of a record is borrowed from programming languages, but it provides a way to define a named n-tuple. Effectively, a record type is a type consisting of multiple named fields that can each have a different type. Each field of a record can be accessed using its name.

Within this thesis, records are actively used to introduce types for type models, type graphs, instance models and instance graphs. These are all named-tuples which are easiest defined using records. Sadly, the example provided in Appendix A does not define such a record, therefore, we provide the record of an instance model (Definition 3.2.12) here:

record $('o, 'nt)$ *instance-model* =
  Tm :: $('nt)$ *type-model*
  Object :: $'o$ *set*
  ObjectClass :: $'o \Rightarrow 'nt\ Id$
  ObjectId :: $'o \Rightarrow 'nt$
  FieldValue :: $('o \times ('nt\ Id \times 'nt)) \Rightarrow ('o, 'nt)\ ValueDef$
  DefaultValue :: $'nt\ Id \Rightarrow ('o, 'nt)\ ValueDef$

The record of an instance model directly shows the structure. It has 6 named fields, the corresponding type model and the 5 elements defined in Definition 3.2.12. Each of these fields has a corresponding type, corresponding to the type described within the definition of an instance model. This way, we have a direct formalisation of an instance graph in Isabelle.

Type synonyms

Isabelle can form new types out of existing types by using generic types. For example, *list*s in Isabelle use this functionality. A list was created using a generic type that can be replaced with any concrete type on usage. For example '$nat\ list$' would represent a list of natural numbers, and '$\mathcal{M}\ list$' a list of elements of datatype $\mathcal{M}$.

To make it more convenient to use these types, it is possible to define these composed types as a type synonym. An example of such a type synonym is given in Appendix A:

type-synonym $multiplicity = \mathcal{M} \times \mathcal{M}$

This type synonym defines the multiplicity type, effectively the formalisation of $\mathbb{M}$ from Definition 3.1.1. It is a tuple of two elements of datatype $\mathcal{M}$, thus a tuple $\mathbb{N} \cup \{*\} \times \mathbb{N} \cup \{*\}$. Using this type synonym, it is now possible to refer to the multiplicity type as 'multiplicity'.

Definitions and functions

In an Isabelle theory, multiple definitions can be provided. The most basic definition can be created using the **definition** keyword. Effectively, a **definition** is simply an abbreviation, i.e. a new name for an existing construction.

An example of such a definition is the *upper* definition in Appendix A:

definition *upper* :: *multiplicity* $\Rightarrow$ $\mathcal{M}$ **where**
upper $m \equiv$ *snd* m

As can be seen, the upper definition receives one argument m of type 'multiplicity', a multiplicity-tuple. It returns the upper bound of the multiplicity. In other words, it returns the second element of the tuple. This behaviour matches what the definition tells us, as *snd* is the Isabelle function to return the second element of a tuple.

Besides the **definition** keyword, it is possible to give recursive function definitions using the **fun** and **function** keywords. An example of this is given as part of the linear order of $\mathcal{M}$ in Appendix A:

fun *less-eq-*$\mathcal{M}$ :: $\mathcal{M} \Rightarrow \mathcal{M} \Rightarrow$ *bool* **where**
less-eq-$\mathcal{M}$ - $\star$ = *True* |
less-eq-$\mathcal{M}$ (**a**) (**b**) = ($a \leq b$) |
less-eq-$\mathcal{M}$ - - = *False*

The function defined here, 'less-eq-$\mathcal{M}$', defines the less than or equal to ($\leq$) relation for $\mathcal{M}$. Effectively, the function describes that any value is always smaller or equal to $*$ and that two numbers (two instances of Nr) are only less or equal when the first number is less or equal to the second number.

The same function could also have been defined using the **function** keyword, with the only difference that for the **function** keyword, a proof for termination of the function must be provided manually. Using the **fun** keyword, Isabelle will try to automatically proof termination of the function by using the specification. This automation is very powerful and works in a variety of functions (in fact, for all functions defined in this thesis, termination is proven automatically using **fun**).

Abbreviations and notation

As can be seen from the 'less-eq-$\mathcal{M}$' function in Appendix A, the Nr and $Star$ constructors for $\mathcal{M}$ are not used directly. Instead, numbers and a star symbol ($\star$) are used to respectively represent a value of Nr and the $Star$ constructor. The use of these symbols has been achieved using the **notation** keyword:

notation
Star (($\star$) *1000*) **and**
Nr ((**-**) [*1000*] *1000*)

The notation keyword allows us to introduce a new notation for many different constructs, in this case, the $Star$ and Nr constructors of $\mathcal{M}$. Custom notations are very powerful, as Isabelle automatically rewrites $Star$ and Nr constructors back to this notation, so introducing this notation works in two ways. It can help to make theorems and proofs more readable, as can also be seen from the example in Appendix A.

Besides the introduction of an alternative notation for existing constructs, it is also possible to introduce a new notation with the corresponding definition of what the notation means. Such a notation is achieved using the **abbreviation** keyword. It has the same properties as the **notation** keyword, but then for a newly defined definition. An example of this is given in Appendix A:

abbreviation *multiplicity-notation* :: $\mathcal{M} \Rightarrow \mathcal{M} \Rightarrow$ *multiplicity* ((-/..-) [*52*, *52*] *51*) **where**
$l..u \equiv (l,u)$

This example introduces a new notation for writing down a multiplicity tuple. It does so by writing the newly introduced notation on the left-hand side and writing the corresponding definition on the right-hand side. Although an **abbreviation** looks the same as a **definition**, they are different in the sense that **abbreviation** only introduces a notation. To Isabelle, it is syntactic sugar, as internally, the notation does not exist. It is only used when representing constructs to the user. This behaviour is different from **definition**, as definitions exist internally and are used by the proof reasoners.

It should be noted that there are shortcuts possible to introduce notations while defining a definition. An example of this is the 'within-multiplicity' definition in Appendix A:

definition *within-multiplicity* :: *nat* ⇒ *multiplicity* ⇒ *bool* (**infixl** *in 50*) **where**
n in m ≡ *lower m* ≤ **n** ∧ **n** ≤ *upper m*

This function uses the infix-left (**infixl**) construction to define an infix notation for the definition. Just like the **abbreviation** command, it is possible to use this definition on the left-hand side of the definition, for readability.

Locales

When defining new types, there is no way to constrain the values for any of its elements. For example, for the 'multiplicity' type, there is no way to prevent the second value of the tuple to be 0, since the natural numbers include 0. In Isabelle, functions and types are always total, and there is no way to exclude specific values of a type.

A way to work around this is by using **locale**s. Locales are Isabelle's approach for dealing with parametric theories. With locales, it is possible to define a context in which specific assumptions hold. An example of a locale is given in Appendix A:

locale *multiplicity* = **fixes** *mult* :: *multiplicity*
assumes *lower-bound-valid*[*simp*]: *lower mult* ≠ ⋆
assumes *upper-bound-valid*: *upper mult* ≠ **0**
assumes *properly-bounded*[*simp*]: *lower mult* ≤ *upper mult*

This example introduces the multiplicity locale. Within this locale, we introduce a named-construct 'mult', which is a multiplicity. Then we make some assumptions which hold in the context of a multiplicity. In this case, there are three assumptions. First of all, there are assumptions on the lower and upper bound, excluding specific (but invalid) values. The final assumption captures that the lower bound is always smaller or equal to the upper bound.

With this locale in place, it is possible to prove theorems and lemmas within the context of a multiplicity. That means that when proving theorems and lemmas within the multiplicity context, all introduced assumptions for 'mult' hold:

context *multiplicity*
begin

lemma *upper-bound-valid-alt*[*simp*]: *upper mult* ≥ **1**
using *less-*$\mathcal{M}$*.elims not-less upper-bound-valid* **by** *fastforce*

end

In above example, it is possible to prove that '*upper mult* ≥ 1 because the assumptions ensure that '*upper mult* ≠ 0'. Since natural numbers cannot be negative, we have that '*upper mult* ≥ 1. This theorem can only be proven within the multiplicity context, as otherwise, the assumptions do not hold, and '*upper mult*' might be 0.

Within this thesis, **locale**s are mostly used to denote valid constructs, such as a valid type graph, type model, instance graph or instance model. These locales limit the respective **record** types for these constructs by assuming the validity constraints presented in their respective sections.

Theorems and proofs

Theorems (also called lemmas) are statements that can be proven correct. For this thesis, all theorems are either defined using the **theorem** or **lemma** keywords in Isabelle. A theorem can be defined to be only valid under certain assumptions or can be defined to be true without any assumptions.

An example of a simple **theorem** can be found in Appendix A:

theorem *mult-zero-unbounded-valid*[*simp*]: *n in* **0**..⋆
unfolding *within-multiplicity-def*
by *simp*

This theorem states that for a multiplicity 0..∗, any natural number is within bounds (any n is in 0..∗). It can easily be proven using the definition of a natural number within a multiplicity.

A proof for a theorem is written directly after the statement. It can either be a short proof using apply-scripts, or a proof within Isabelle's proof language Isar. In the example above, it is a short proof using

apply-scripts. In this case, the proof is done within one step: by simplification of the definition.

Once lemmas or theorems are proven, they can be used in the proof of other lemmas and statements. Reusing them is done by referring to them manually, or by adding them to a set of lemmas and theorems that Isabelle will try by default. Adding theorems to a set of default rules is done by adding a specific keyword. For example, add [*simp*] to add the theorem to the set of simplification rules, or add [*intro*] or [*elim*], to specify the theorem to be an introduction rule or elimination rule. Introduction and elimination rules will not be further specified here; more information on these can be found in the Isabelle documentation.

Isar

Isar stands for *Intelligible semi-automated reasoning*, and is an interpreted language environment for structured formal proof documents. It allows to write down mostly humanly readable proofs in Isabelle while still getting the advantage of semi-automated reasoning. Isar is built on the principle of writing down multiple steps of the proof, doing less automation, in favour of readability. As a consequence, writing Isar proofs is more work for the writer of the proof but eventually results in better humanly readable proofs that also have value without the automated reasoning of a theorem prover.

This section will not discuss the full Isar environment in detail. The Isabelle/Isar Reference Manual [20], included with each copy of Isabelle, already contains a very detailed explanation of all features that Isar has to offer. Instead, we consider a small example of a proof written in Isar, picked directly from Appendix A:

```
proof
  fix x y z :: 𝓜
  show (x < y) = (x ≤ y ∧ ¬ y ≤ x)
  proof (induction x arbitrary: y)
    case Star
    then show ?case by simp-all
  next
    case (Nr x)
    then show ?case by (cases y) auto
  qed

  show x ≤ x by (induction x) simp-all
  then show x ≤ y ⟹ y ≤ x ⟹ x = y
  proof (induction x arbitrary: y)
    case Star
    then show ?case by (cases y) simp-all
  next
    case (Nr x)
    then show ?case by (cases y) simp-all
  qed

  show x ≤ y ⟹ y ≤ z ⟹ x ≤ z
  proof (induction x arbitrary: y z)
    case Star
    then show ?case by (cases y) simp-all
  next
    case (Nr x)
    then show ?case
    proof (induction y arbitrary: z)
      case Star
      then show ?case by (cases z) simp-all
    next
      case (Nr x)
      then show ?case by (cases z) simp-all
    qed
  qed

  show x ≤ y ∨ y ≤ x
  proof (induction x arbitrary: y)
    case Star
    then show ?case by simp
  next
    case (Nr x)
    then show ?case by (cases y) auto
  qed
```

qed

In this example, we see the proof that proves that type $\mathcal{M}$ is an instantiation of a linear order. In order to show that type $\mathcal{M}$ gives rise to a linear order, we have to proof multiple subgoals, which are:

- Correctness of $<$: $(x < y) = (x \leq y \wedge \neg y \leq x)$
- Reflexivity of $\leq$: $(x \leq x)$
- Transitivity of $\leq$: $x \leq y \wedge y \leq z \implies x \leq z$
- Correctness of the linear order: $x \leq y \vee y \leq x$

Each of these subgoals is proven separately within the Isar proof. The proof of each subgoal is defined by the **show** keyword. Important to see from the example above is that each subgoal is proven using a nested subproof. Such subproofs can be written as apply-scripts, or using a nested Isar proof, as we see in the example above.

Proof tactics

In order to deliver proofs, we make use of automated reasoning. Essential aspects of automated reasoning are the different proof tactics in Isabelle. Proof tactics can be applied to a proof goal to either solve the proof goal entirely or to somehow make the goal simpler to solve.

A vital proof tactic shown in the example above is 'induction'. This tactic applies mathematical induction to the proof goal, splitting the goal into new subgoals that follow the structure of mathematical induction.

The following proof tactics are extensively used within this thesis:

- 'induction' (also called 'induct'): Applies mathematical induction to the proof goal. Splits the subgoal into two or more subgoals that follow the structure of mathematical induction.
- 'cases': Applies a case distinction to the proof goal. It will split the proof goal into multiple subgoals, one for each applicable case. This proof tactic works especially well for inductive definitions and datatypes with a finite set of possible values.
- 'intro': Splits a proof goal and introduces new subgoals based on an introduction rule. An important example of an introduction rule is $conjI$, which splits a proof goal of $A \wedge B$ into the two subgoals A and B.
- 'elim': Splits a proof goal by eliminating operations and relations and providing smaller subgoals instead of those. For example, the elimination rule $disjE$ splits a proof goal of the form $A \vee B \implies C$ into two subgoals, $A \implies C$ and $B \implies C$.
- 'simp' (and 'simp_all'): Apply simplification to a proof goal in order to solve the problem completely. It uses simplification rules to rewrite the statement until it arrives at 'True', finishing the proof.
- 'fastforce': Solves the proof goal by using a tactic similar to brute force. It tries all possible outcomes but tries to be smart by excluding similar cases.
- 'fast': A classical solver which solves the proof goal by structurally checking cases based on a depth-first search algorithm. Not frequently used within this thesis.
- 'auto': A combination of 'simp' and 'fastforce', which can also use introduction rules and elimination rules when rewriting. In general, this proof tactic more powerful than 'simp' and 'fastforce'.
- 'blast': Solves the proof goal by using a semantic tableau. Frequently used for solving logic problems.
- 'metis': Solves the proof goal by using resolution. Frequently used for more complex logic problems that cannot be solved by 'blast'.

Isabelle is not limited to the above-discussed proof tactics, but these tactics are the most important ones for this thesis. Other tactics are not used either because they apply to a different kind of problem (number arithmetic instead of logic, for example) or because they are not transparent in solving their problem. For example, the 'smt' proof tactic solves a problem by using an external SMT solver. Although these proof tactics can solve many problems, it is not transparent to the reader what steps the SMT solver has taken to solve the problem, as opposed to the proof tactics described above. Therefore these tactics have been excluded.

2.3.3 Archive of Formal Proofs

Isabelle has an Archive of Formal Proofs (AFP), which is a collection of proof libraries, examples, and larger scientific developments, mechanically checked in the theorem prover Isabelle. All theories within this archive are organised in the way of a scientific journal such that they can be referred to by new theories.

Graph Theory

The Isabelle AFP submission Graph Theory [16] is used as part of this thesis. This submission to the Isabelle AFP is a formalization of directed graphs, supporting labelled multi-edges and infinite graphs. Theorems proven for these graphs include, but are not limited to, walks, cyclicity, connectedness and some properties of isomorphisms. All the theorems proven as part of this submission are discussed in [2].

Within this thesis, the submission is used as part of the GROOVE formalisation within Isabelle. Within the GROOVE formalisation, GROOVE type graphs and instance graphs are extensions of the directed graph introduced by the Graph Theory submission. This allows Isabelle to apply theorems proven for graphs within this submission to GROOVE graphs presented in the theories of this thesis.

Within this thesis, only a small selected set of theorems from the submission is used. This set mostly includes theorems related to walks and cyclicity of graphs. These theorems are used to show the acyclicity of the containment relation for instance graphs.

Chapter 3

Formalisations

As explained in Section 1.1, the formalisation of the model transformations depends on the formalisation of the model languages. Therefore, the formalisations of Ecore and GROOVE need to be established. In this chapter, the formalisations for Ecore and GROOVE used throughout this thesis are introduced.

3.1 Global definitions

This section defines a multiplicity, which is a two tuple consisting of a lower and upper bound. In Ecore, the notion of a multiplicity is used within a field signature (Definition 3.2.6) in order to specify a limit on the allowed amount of values for a field. In GROOVE, multiplicities are used to bound the number of incoming and outgoing edges for each node type via multiplicity pairs (Definition 3.3.4).

Definition 3.1.1 (Multiplicity)
A multiplicity is a two tuple consisting of a lower bound (which is any natural number) and an upper bound (which is possibly unbounded).

$$\mathbb{M} \subseteq (\mathbb{N} \times \mathbb{N}^+ \cup *) \cap \leq$$

The first value represents the lower bound, the second value of the tuple represents the upper bound. The set of multiplicities $\mathbb{M}$ *is formally defined as*

$$\mathbb{M} = \{(l, u) \mid l \in \mathbb{N} \wedge u \in (\mathbb{N}^+ \cup *) \wedge l \leq u\}$$

It holds that $*$ *is larger than each natural number, so* $\forall n \in \mathbb{N} : n < *$*. Furthermore, the notation* $l..u$ *is used to denote* $(l, u) \in \mathbb{M}$*.*

Finally, any natural number n *is said to be part of a multiplicity if it is within bounds, meaning:*

$$\forall m = l..u \in \mathbb{M}, n \in \mathbb{N} : n \in m \Leftrightarrow l \leq n \leq u$$

 *Also see* `multiplicity` *in* `Ecore.Multiplicity`

3.2 Ecore formalisation

This section discusses a partial formalisation of Ecore based on the conceptual model discussed in [4]. The formalisation discusses both type models and instance models, as discussed in Section 2.1. This formalisation has enough expressive power to capture all the elements of Ecore that are relevant for this thesis.

3.2.1 Definitions

This section discusses some definitions specific to the Ecore formalisation. The definitions need to be in place before the formalisation of type models and instance models are given.

In Ecore, all elements should be identifiable by a name. For this, we define a globally unique set of names *Name*, which type and instance models share. We write down elements of *Name* in a sans-serif font, such as aName.

Definition 3.2.1 (Name)
Name is a globally fixed set of names (shared between instance models and type models). This set contains at least the names boolean*,* integer*,* real *and* string*, as well as* true*,* false *and* nil*.*

A name is not enough to uniquely identify an element in Ecore. All elements in Ecore belong to a namespace. Within a namespace, all names have to be unique, but names can be shared between different namespaces. The combination of a namespace and a name is referred to as an identifier. An identifier can be used to identify an element within a model uniquely.

Namespaces can be nested, which means that a namespace can contain other namespaces. Therefore namespaces are recursively defined up to the root namespace, which we define as $\bot$. This means all namespaces are part of the root namespace, either directly or via one or more parent namespaces.

Definition 3.2.2 (Identifier/Namespace)
Identifiers and namespaces are defined as the smallest sets satisfying

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

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

 *Also see* `Namespace` *in* `Ecore.Model_Namespace`

For notation, we will separate namespace from the name using a dot, omitting the root namespace. For example, $\langle\langle\bot, \mathsf{namespace}\rangle, \mathsf{name}\rangle$ becomes .namespace.name.

To distinguish between the different types of data that may be present in an instance model, we define a set of data types that can be used within the type model. The set of data types gives rise to a set of data values, which a corresponding instance model may use.

Definition 3.2.3 (Data types)
The set of data types is defined by

$$DataType = \{\mathsf{boolean}, \mathsf{integer}, \mathsf{real}, \mathsf{string}\}$$

 *Also see* `DataType` *in* `Ecore.Type_Model`

Definition 3.2.4 (Data type values)
For each of the various data types a single set defines the possible values. The following sets of values are defined:

- $\mathbb{B} = \{\mathsf{true}, \mathsf{false}\}$*, the set of* boolean *values.*
- $\mathbb{C}$*, the set of all printable characters.*
- $\mathbb{R}$*, the set of all rational (*real*) numbers.*
- $\mathbb{S}$*, defined as the set of all finite subsequences of elements in $\mathbb{C}$, the set of all possible* strings*.*
- $\mathbb{Z}$*, the set of all* integer *numbers.*

 *Also see* `LiteralValue` *in* `Ecore.Instance_Model`

Ecore also supports the notion of assigning a nil value, indicating there is no actual reference to any element in an instance model.

Definition 3.2.5 (Nil value)
nil *defines the unassigned value for nullable types in the type model.*

 *Also see* `ClassValue` *in* `Ecore.Instance_Model`

Definition 3.2.7 and Definition 3.2.15 specify in what context this value can be used.

3.2.2 Type models

This section provides the formal definition of type models, which are models that are based on the Ecore metamodel, of which a simplified version is given in Figure 2.1. A type model provides a set of definitions and constraints that describe a set of instance models, which may or may not be valid according to the type model. On the top level, it defines a set of *classes*, `EClass`es, of which instances (*objects*) may be used within the instance model. The classes contain a set of *fields*, `EStructuralFeature`s, which are identified by a name which is unique within the class. Each of these `EStructuralFeature`s is typed and has a multiplicity. Class instances in the instance model may assign values to these fields (specific for that instance) which must adhere to both the type and multiplicity of the field. The type model also defines an *inheritance* relation between these classes, modelled by `eSupertypes` in the Ecore metamodel, which allows classes to inherit from other classes, providing a specialisation of that class.

A type model also defines a set of *enumerations*, `EEnum`s, and their values, `EEnumLiteral`s. Each enumeration defines a unique type with a fixed set of values. Furthermore, a set of *constants* and their types define a symbolic typed value, which relates to a specific value in an instance model. A set of *custom data types*, `EDataType`s, is also provided by the type model, which allows the representation of user-defined data types.

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

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

The suffix Tm is used when the definition of something depends on any type model Tm, for example, $Class_{Tm}$.

Definition 3.2.6 (Type model)
A single type model Tm is a 10-tuple, consisting of 8 sets and 2 functions, which is defined as:

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

with

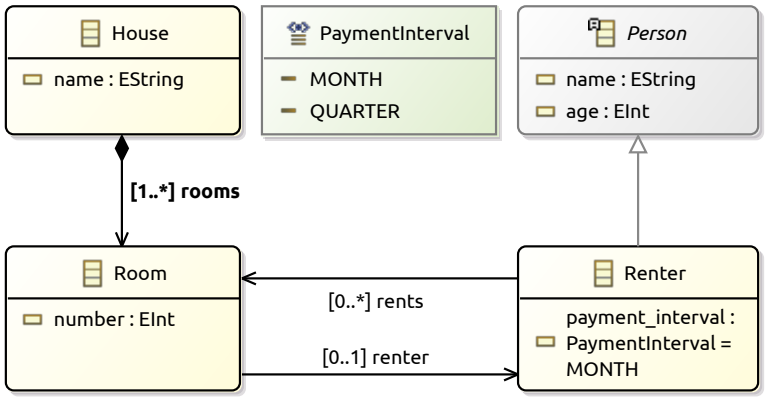
- $Class \subseteq Id$ *is the set of classes (`EClass` objects) in* Tm.
- $Enum \subseteq Id$ *is the set of enumerations (`EEnum` objects) in* Tm.
- $UserDataType \subseteq Id$ *is the set of custom data types (`EDataType` objects) in* Tm.
- $Field \subseteq (Class \times Name)$ *is the set that maps a class to a set of field names (`EStructuralFeature` objects) in* Tm.
- $\text{FieldSig} : Field \Rightarrow (Type_{Tm} \times \mathbb{M})$ *is the function that maps fields to their type (as defined in Definition 3.2.7) and multiplicity (as defined in Definition 3.1.1).*
- $EnumValue \subseteq Enum \times Name$ *is the set of possible values (the `EEnumLiteral`s) for the enumerations in* Tm.
- $Inh \subseteq Class \times Class$ *is the inheritance relation between the classes in* Tm.
- $Prop \subseteq Property_{Tm}$ *is the set of properties that apply to* Tm *(see definition Definition 3.2.10).*
- $Constant \subseteq Id$ *is the set that contains all possible constants that may be used as a (symbolic) default value.*
- $\text{ConstType} : Constant \Rightarrow Type_{Tm}$ *is the function that maps constants to their respective types.*

where

- *Class, DataType (Definition 3.2.3), Enum and* $UserDataType$ *are pairwise disjoint.*
- *None of the elements in* $Class \cup DataType \cup Enum \cup UserDataType$ *may be in the namespace of another element in that set.*
- Inh *is an asymmetric relation, of which the transitive closure is irreflexive.*


Also see `type_model` *in* `Ecore.Type_Model`

An example type model is given in Figure 3.1. It shows 4 classes (House, Person, Renter and Room) and a single enumeration (PaymentInterval). The Person class has 2 fields, age and name. The Renter class



(a) Type model in Ecore notation

$$
\begin{aligned}
Class_{Tm} &= \{.\text{House}, .\text{Person}, .\text{Renter}, .\text{Room}\} \\
Enum_{Tm} &= \{.\text{PaymentInterval}\} \\
UserDataType_{Tm} &= \emptyset \\
Field_{Tm} &= \{(.\text{House}, \text{name}), (.\text{House}, \text{rooms}), \\
&\quad (.\text{Person}, \text{age}), (.\text{Person}, \text{name}), \\
&\quad (.\text{Renter}, \text{payment\_interval}), (.\text{Renter}, \text{rents}), \\
&\quad (.\text{Room}, \text{number}), (.\text{Room}, \text{renter})\} \\
\text{FieldSig}_{Tm} &= \Big\{\Big((.\text{House}, \text{name}), (\text{string}, 1..1)\Big), \Big((.\text{House}, \text{rooms}), ([\text{setof}, !.\text{Room}], 1..*)\Big), \\
&\quad \Big((.\text{Person}, \text{age}), (\text{integer}, 1..1)\Big), \Big((.\text{Person}, \text{name}), (\text{string}, 1..1)\Big), \\
&\quad \Big((.\text{Renter}, \text{payment\_interval}), (.\text{PaymentInterval}, 1..1)\Big), \\
&\quad \Big((.\text{Renter}, \text{rents}), ([\text{setof}, !.\text{Room}], 0..*)\Big), \\
&\quad \Big((.\text{Room}, \text{number}), (\text{integer}, 1..1)\Big), \Big((.\text{Room}, \text{renter}), (!.\text{Renter}, 0..1)\Big)\Big\} \\
EnumValue_{Tm} &= \{(.\text{PaymentInterval}, \text{MONTH}), (.\text{PaymentInterval}, \text{QUARTER})\} \\
Inh_{Tm} &= \{(.\text{Renter}, .\text{Person})\} \\
Prop_{Tm} &= \{[\text{abstract}, .\text{Person}], \\
&\quad [\text{identity}, \{(.\text{Person}, \text{age}), (.\text{Person}, \text{name})\}], [\text{containment}, (.\text{House}, \text{rooms})], \\
&\quad [\text{opposite}, (.\text{Room}, \text{renter}), (.\text{Renter}, \text{rents})], [\text{opposite}, (.\text{Renter}, \text{rents}), (.\text{Room}, \text{renter})], \\
&\quad [\text{defaultValue}, (.\text{Renter}, \text{payment\_interval}), .\text{Constant.PaymentInterval.Month}]\} \\
Constant_{Tm} &= \{.\text{Constant.PaymentInterval.Month}\} \\
\text{ConstType}_{Tm} &= \{(.\text{Constant.PaymentInterval.Month}, .\text{PaymentInterval})\}
\end{aligned}
$$

(b) Formal definition of the type model

Figure 3.1: Example of a type model corresponding with Definition 3.2.6

has a field payment_interval which makes use of the PaymentInterval enumeration. It also has a field rents which defines a relation between Room and Renter. The House class has also 2 fields, a field name and a field rooms which is a containment relation between House and Room. Moreover, we have the Room class itself, which has 2 fields. One is called number and the other one is called renter, which is the opposite relation between Room and Renter. Finally, we see that Renter inherits from Person.

The fields in a type model are always associated with a specific type, which defines the set of possible values that may be assigned in an instance model. The possible types are defined by the set of data types, classes, enumerations and user-defined data types. The set of types also consists of various aggregations of these types, namely containers. Containers provide types for multiple values of the same type, but of which the values may differ in number and order.

Definition 3.2.7 (Types)
Given any type model Tm, the set of types is defined as

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

The $ClassType_{Tm}$ set defines both a set of nullable and proper classes. Nullable classes are classes for which the nil *(see Definition 3.2.5) value is valid, and proper classes are those classes for which the* nil *value is not valid (hence both sets of classes are disjoint).*

The $ClassType_{Tm}$ set is defined as

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

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

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

For the interpretation of the values in $\{\mathsf{bagof}, \mathsf{setof}, \mathsf{seqof}, \mathsf{ordof}\}$*, see Definition 3.2.13.*

The set of types is recursively defined as the smallest solution of the equations for $Type_{Tm}$ and $Container_{Tm}$.

Tuples within $Type_{Tm}$ are written using square brackets, e.g. $[\mathsf{bagof}, \mathsf{int}]$ *or* $[\mathsf{nullable}, C]$*. Furthermore, given a $C \in Class_{Tm}$, $?C$ is a short notation for the* nullable *variant of C,* $[\mathsf{nullable}, C]$*. In the same fashion, $!C$ is the short notation for the* proper *variant of C,* $[\mathsf{proper}, C]$*.*

Finally, we define the function uncontainer$: Container_{Tm} \Rightarrow Type_{Tm}$ *which returns the type contained by a container.*

Also see `Type` *in* `Ecore.Type_Model`

In the example in Figure 3.1, the various fields make use of different types. The fields depicted as a relation between two classes are actually of a type based on the $Class_{Tm}$ set. For example, the rooms field of class House is typed by the Room class. In this case, we assume that each Room in a House is unique. Thus the relationship is best typed by a setof container. For a setof container, the order does not matter, and each value should be unique. We also see the example of the usage of an enumeration, the payment_interval field on Renter uses the PaymentInterval enumeration. Finally, the Person class shows two fields that are typed by some of the data types available, integer and string for the age and name fields respectively.

Definition 3.2.8 (Field)
Given any type model Tm, the $Field_{Tm}$ relation defines a binary relation between classes and fields. In order to retrieve the set of fields for a given class (and the fields inherited from superclasses), the following function is defined:

$$\text{fields}: Class_{Tm} \Rightarrow \mathcal{P}(Field_{Tm})$$

such that

$$\text{fields}_{Tm}(c) = \{f \in Field_{Tm} \mid f = (c', n) \wedge c \sqsubseteq_{Tm} c'\}$$

Given any type model Tm, the FieldSig_{Tm} *function defines a mapping between fields and their signatures. The following functions are defined to retrieve various components of a field signature:*

- class$: Field_{Tm} \Rightarrow Class_{Tm}$
- type$: Field_{Tm} \Rightarrow Type_{Tm}$
- lower$: Field_{Tm} \Rightarrow \mathbb{N}$

- upper: $Field_{Tm} \Rightarrow \mathbb{N}^+ \cup *$

These functions are defined as follows:

- $\text{class}_{Tm}(f) = class \quad \text{iff} \quad f = (class, name)$
- $\text{type}_{Tm}(f) = type \quad \text{iff} \quad \text{FieldSig}_{Tm}(f) = (type, (lower, upper))$
- $\text{lower}_{Tm}(f) = lower \quad \text{iff} \quad \text{FieldSig}_{Tm}(f) = (type, (lower, upper))$
- $\text{upper}_{Tm}(f) = upper \quad \text{iff} \quad \text{FieldSig}_{Tm}(f) = (type, (lower, upper))$

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

$$\begin{aligned} Attr_{Tm} = &\{f \in Field_{Tm} \mid type(f) \in (DataType \cup Enum_{Tm} \cup UserDataType_{Tm}) \vee \\ &type(f) \in \{\mathsf{setof}, \mathsf{bagof}, \mathsf{ordof}, \mathsf{seqof}\} \times Attr_{Tm}\} \\ Rel_{Tm} = &Field_{Tm} \setminus Attr_{Tm} \end{aligned}$$

The set of attributes is recursively defined as the smallest solution of the given set of equations for $DataType$, $Enum_{Tm}$, $UserDataType_{Tm}$ and $Attr_{Tm}$

Also see `fields` *in* `Ecore.Type_Model`

Taking for example the field rents from the Renter class in the type model example in Figure 3.1, the following properties can be identified: The type refers to Room, which is an element of the $Class_{Tm}$ set. The lower and upper values are 0 and $*$ respectively (which means a Renter can rent an arbitrary number of rooms). Furthermore, the rents field is an element of the Rel_{Tm} set (it is a class container type) and part of the $fields_{Tm}(\mathsf{Renter})$ set, which is $\{\mathsf{payment\_interval}, \mathsf{rents}\}$.

The various types have an underlying subtype relation, which generalises inheritance. A subtype defines a specialisation of a supertype. Because of that, all values valid for the subtype are also valid for the supertype (see Definition 3.2.15 for details).

Definition 3.2.9 (Subtype relation)
Given any type model Tm, $\sqsubseteq_{Tm} \subseteq Type_{Tm} \times Type_{Tm}$ defines the subtype relation. It is a reflexive partial order relation, for which the following rules can be defined (with $t_1, t_2, t_3 \in Type_{Tm}$ and $c_1, c_2 \in Class_{Tm}$):

Transitivity:

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

Reflexivity:

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

Generalization of inheritance:

$$\frac{(c_1, c_2) \in Inh_{Tm}}{?c_1 \sqsubseteq_{Tm} ?c_2} \qquad \frac{(c_1, c_2) \in Inh_{Tm}}{!c_1 \sqsubseteq_{Tm} !c_2}$$

Nullable/Proper classes:

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

Also see `subtype` *in* `Ecore.Type_Model`

Thus, in the example, [nullable, Renter] $\sqsubseteq_{Tm}$ [nullable, Person] (since (Student, Person) $\in Inh_{Tm}$). Furthermore, it also holds that [proper, Renter] $\sqsubseteq_{Tm}$ [nullable, Renter], as a proper class is a subtype of a nullable class.

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

- The abstract property. This property, specified for a specific class in the type model, forbids the instantiation of that class in any instance model. As such, it is satisfied when no object exists in an instance model which is an instance of that class. Please note that this only holds for instances of that exact class. Subtyping may be allowed (under the condition that those classes are not abstract).
- The containment property. This property, specified for a relation, states that a single source object contains all objects that are the target of this relation. Objects are contained by at most one other object, and containment cycles are not allowed. When an object with contained objects is removed from an instance model, all contained objects are removed as well. The constraint is satisfied when an object is the target of no more than one containment relation, and there exists no cycle between containment relations.
- The defaultValue property. This property specifies a default value for a field which has not been assigned a value in an instance model. It specifies a constant for a field, which represents a value in an instance model. It is always satisfied and influences the behaviour of possible model transformations.
- The identity property. This property is specified for a class and a set of attributes. It is used to specify that the values for the set of attributes uniquely identify an object on instance model level. As such, it is satisfied when no two objects are both an instance of the class and have pairwise the same value for all the attributes.
- The keyset property. This property is specified for a set of attributes of a class and a relation towards that class. It ensures that each instance of that class within the given relation is uniquely identified by the values of the set of attributes. It is satisfied that two objects must be the same object if they are the target of the given relation, and they have pairwise-identical values for their attributes.
- The opposite property. This property specifies that two relations are the opposite of each other, which means that for each instance of the first relation, an instance of the second relation exists with a switched target and source. The property is satisfied when for each pair of objects, for each relation that exists between these objects, a reverse relation exists if both these relations are opposite.
- The readonly property. This property, specified for a field in the type model, forbids the assignment of a new value to that field in any instance model. This property only affects the possible transformations of an instance model and is always satisfied for any specific instance model.

Definition 3.2.10 (Type model properties)
For a type model Tm a set of properties $Property_{Tm}$ is defined which contains all the possible properties. This set is defined as

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

where

- $[\mathsf{defaultValue}, f, v]$ *is defined such that* $\text{ConstType}_{Tm}(v) \sqsubseteq_{Tm} \text{type}_{Tm}(f)$.
- $[\mathsf{keyset}, r, A]$ *is defined such that* $\text{type}_{Tm}(r) \in (\{\mathsf{setof}, \mathsf{ordof}\} \times ClassType_{Tm})$ *with* $\text{type}_{Tm}(r) = [\mathsf{setof}, c] \vee \text{type}_{Tm}(r) = [\mathsf{ordof}, c]$. *Then also have that* $\forall (ac, an) \in A : !c \sqsubseteq_{Tm} !ac$.
- $[\mathsf{opposite}, r, r']$ *is defined such that when* $r = (c1, n1), r' = (c2, n2)$, *then it should hold that* $!c1 \sqsubseteq_{Tm} \text{uncontainer}(\text{type}_{Tm}(r'))$, $!c2 \sqsubseteq_{Tm} \text{uncontainer}(\text{type}_{Tm}(r))$, $\text{type}_{Tm}(r) \notin \{\mathsf{bagof}, \mathsf{seqof}\} \times Type_{Tm}$ *and finally* $type_{Tm}(r') \notin \{\mathsf{bagof}, \mathsf{seqof}\} \times Type_{Tm}$ *(containers must have unique values).*

Also see `Property` *in* `Ecore.Type_Model`

The example in Figure 3.1 also shows a few properties:

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

Table 3.1: The allowed multiplicities for each type

- The Person class is declared abstract (indicated by the grey looking class layout and italic name)
- The age and name fields of a Person are its identity (as indicated by the formal definition). Although it is highly unlikely that these details would be unique in the real world, they must be in our instance models.
- The rooms relation of House is a containment relation (shown by the filled diamond at the House end of the containment relation).
- The rents and renter relations are opposites (as indicated by the formal definition).
- The payment_interval field has a default value MONTH (as indicated with the field definition in the model).

With the given definitions, it is possible to define an inconsistent type model. Such a type model is correct according to the given definitions but does not specify any valid instance model using the definitions in Section 3.2.3. Therefore, a definition is given for a consistent type model. A consistent type model enforces some constraints on multiplicities and properties within the type model to ensure the satisfiability of the definitions in Section 3.2.3. Without these constraints, one or more of these definitions can never be satisfied at all. Please note that these constraints merely support the satisfiability of these definitions, it does not guarantee the existence of any meaningful instance model, as it might still be possible to define a 'consistent' type model with conflicting multiplicities or properties.

Definition 3.2.11 (Type model consistency)
The multiplicities of the fields in $Field_{Tm}$ are consistent if it holds that:

$$\begin{aligned} \text{type}_{Tm}(f) \in DataType \cup Enum_{Tm} \cup UserDataType_{Tm} \cup (\mathsf{proper} \times Class_{Tm}) &\implies \text{lower}_{Tm}(f) = 1 \\ \text{type}_{Tm}(f) \in (\mathsf{nullable} \times Class_{Tm}) &\implies \text{lower}_{Tm}(f) = 0 \end{aligned}$$

and

$$\text{type}_{Tm}(f) \notin Container_{Tm} \implies \text{upper}_{Tm}(f) = 1$$

as indicated by Table 3.1.

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

- $[\mathsf{containment}, r] \in Prop_{Tm} \wedge [\mathsf{opposite}, r, r'] \in Prop_{Tm} \implies upper_{Tm}(r') = 1$ *(opposite of containment relation must have upper bound of 1).*
- $[\mathsf{defaultValue}, f, v] \in Prop_{Tm} \wedge [\mathsf{defaultValue}, f, v'] \in Prop_{Tm} \implies v = v'$ *(unique for f).*
- $[\mathsf{identity}, c_1, A_1] \in Prop_{Tm} \wedge [\mathsf{identity}, c_2, A_2] \in Prop_{Tm} \wedge\, !c_1 \sqsubseteq_{Tm} !c_2 \implies A_1 \subseteq A_2$ *(classes may only have identities with attributes that are a subset of attributes part of the superclass' identity).*
- $[\mathsf{keyset}, r, A] \in Prop_{Tm} \wedge [\mathsf{keyset}, r, A'] \in Prop_{Tm} \implies A = A'$ *(unique for r).*
- $[\mathsf{opposite}, r, r'] \in Prop_{Tm} \wedge [\mathsf{opposite}, r, r''] \in Prop_{Tm} \implies r' = r''$ *(unique for r).*
- $[\mathsf{opposite}, r, r'] \in Prop_{Tm} \iff [\mathsf{opposite}, r', r] \in Prop_{Tm}$ *(symmetry).*

Then a type model Tm is consistent if and only if

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

 *Also see* `type_model` *in* `Ecore.Type_Model`

3.2.3 Instance models

An instance model represents an instance of a type model. In other words, the metamodel of an instance model is its type model. Because of our definitions of type models, this means that the metametamodel of an instance model is the Ecore metamodel.

An instance model consists of a set of objects, which have a corresponding class they instantiate and an optional identifier. All objects are an instance of a specific class and are therefore typed by that class and its superclasses. Furthermore, an instance model also specifies the values for each field of an object. Its type determines the fields present for each object. Finally, the instance model specifies a set of default values, which assigns a value to each of the named constants from the type model ($Constant_{Tm}$), allowing to assign default values to fields.

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

The suffix Im is used when the definition of something depends on any instance model Im, which itself depends on the definition of any type model Tm.

Definition 3.2.12 (Instance model)
For a type model Tm,

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

a single instance model Im is defined as

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

with

- *$Object$ is the set of objects (class instances) in Im.*
- ObjectClass : *$Object \Rightarrow Class_{Tm}$ is the function that maps each object in Im to a class.*
- ObjectId : *$Object \Rightarrow Name$ is the injective partial function that maps each object in Im to an identifier.*
- FieldValue : *$(Object \times Field_{Tm}) \Rightarrow Value_{Im}$ is the partial function between each $Field_{Tm}$ of an $Object_{Im}$ and a $Value_{Im}$ (see Definition 3.2.13).*
- DefaultValue : *$Constant_{Tm} \Rightarrow Value_{Im}$ is the function that assigns a value to each constant in the corresponding type model Tm.*

where

- $\forall (o, n), (o', n') \in ObjectId : n = n' \implies o = o'$.
- $\forall o \in Object, f \in Field_{Tm} : (o, f) \in \text{dom } FieldValue \iff ObjectClass(o) \sqsubseteq_{Tm} \text{class}(f)$.

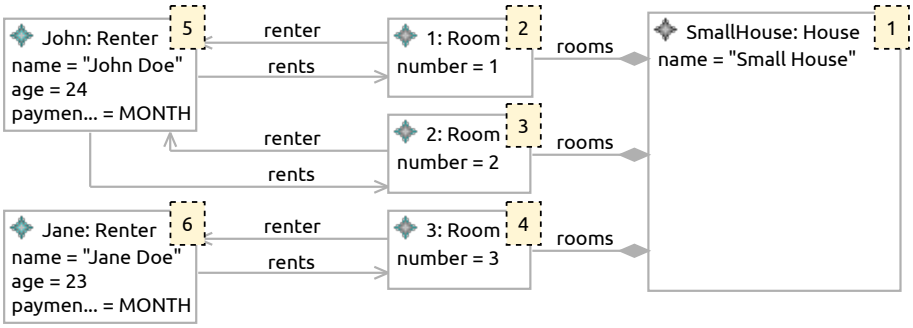
 Also see `instance_model` *in* `Ecore.Instance_Model`

Please note that ObjectId is injective because each object must have a unique identifier. It is partial because an object does not necessarily need an identifier: The internal object identifiers (the elements of the set *Object*) are already unique. The ObjectId function is for adding an explicit identifier that is not generated internally.

The FieldValue function maps a combination of an object and field to a value. Please note that the function is partial because not every combination of object and field is valid. The domain of this function is therefore made explicit by the constraints of the definition. Please note that this function is not injective: Values can be shared across objects and do not have to be unique.

An important function is the DefaultValue function, which is defined on an instance model rather than a type model. This definition has been chosen to accommodate for default values that reference another object. In order to reference another object, the possible object references need to be known. These object references are only known on the instance level, as the type level does not define any objects.

An example model is represented by Figure 3.2. It is based on the type model from the example in Figure 3.1. It shows 2 instantiations of the Renter class: the John and Jane objects. Furthermore, there are three instantiations of the Room class (1, 2 and 3) and one instantiation of the House class (SmallHouse). The text after the colon in the header of each object represents the $ObjectClass_{Im}$ of each object. Additionally, the text preceding the colon represents the $ObjectId_{Im}$. The Renter objects have values assigned for all fields, including the fields of their superclasses. This also holds for the Room



(a) Instance model based on Ecore notation

$$
\begin{aligned}
Object_{Im} &= \{1,2,3,4,5,6\} \\
\text{ObjectClass}_{Im} &= \{(1,.\mathsf{House}),(2,.\mathsf{Room}),(3,.\mathsf{Room}),(4,.\mathsf{Room}),(5,.\mathsf{Renter}),(6,.\mathsf{Renter}),\} \\
\text{ObjectId}_{Im} &= \{(1,.\mathsf{SmallHouse}),(2,.1),(3,.2),(4,.3),(5,.\mathsf{John}),(6,.\mathsf{Jane})\} \\
\text{FieldValue}_{Im} &= \Big\{\Big((1,(.\mathsf{House},\mathsf{name})),[\mathsf{string},\text{“Small House”}]\Big), \\
&\quad \Big((1,(.\mathsf{House},\mathsf{rooms})),[\mathsf{setof},\langle[\mathsf{obj},2],[\mathsf{obj},3],[\mathsf{obj},4]\rangle]\Big), \\
&\quad \Big((2,(.\mathsf{Room},\mathsf{number})),[\mathsf{int},1]\Big),\ \Big((2,(.\mathsf{Room},\mathsf{renter})),[\mathsf{obj},5]\Big), \\
&\quad \Big((3,(.\mathsf{Room},\mathsf{number})),[\mathsf{int},2]\Big),\Big((3,(.\mathsf{Room},\mathsf{renter})),[\mathsf{obj},5]\Big), \\
&\quad \Big((4,(.\mathsf{Room},\mathsf{number})),[\mathsf{int},3]\Big),\Big((4,(.\mathsf{Room},\mathsf{renter})),[\mathsf{obj},6]\Big), \\
&\quad \Big((5,(.\mathsf{Person},\mathsf{name})),[\mathsf{string},\text{“John Doe”}]\Big), \\
&\quad \Big((5,(.\mathsf{Person},\mathsf{age})),[\mathsf{int},24]\Big), \\
&\quad \Big((5,(.\mathsf{Renter},\mathsf{payment\_interval})),[\mathsf{enum},(.\mathsf{PaymentInterval},\mathsf{MONTH})]\Big), \\
&\quad \Big((5,(.\mathsf{Renter},\mathsf{rents})),[\mathsf{setof},\langle[\mathsf{obj},2],[\mathsf{obj},3]\rangle]\Big), \\
&\quad \Big((6,(.\mathsf{Person},\mathsf{name})),[\mathsf{string},\text{“Jane Doe”}]\Big), \\
&\quad \Big((6,(.\mathsf{Person},\mathsf{age})),[\mathsf{int},23]\Big), \\
&\quad \Big((6,(.\mathsf{Renter},\mathsf{payment\_interval})),[\mathsf{enum},(.\mathsf{PaymentInterval},\mathsf{MONTH})]\Big), \\
&\quad \Big((6,(.\mathsf{Renter},\mathsf{rents})),[\mathsf{setof},\langle[\mathsf{obj},4]\rangle]\Big)\Big\} \\
\text{DefaultValue}_{Im} &= \Big\{\Big(.\mathsf{Constant.PaymentInterval.Month},[\mathsf{enum},(.\mathsf{PaymentInterval},\mathsf{MONTH})]\Big)\Big\}
\end{aligned}
$$

(b) Formal definition of the instance model

Figure 3.2: Example of an instance model corresponding with Definition 3.2.12

and House objects. For attributes, the assignment to a field name represents the value of a field. For relations, a named arrow between two objects represents the value of the field. The name of the arrow represents the field name, and multiple arrows with the same name represent multiple values for the same field.

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

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

Definition 3.2.13 (Values)
Given any instance model Im, the set of values is $Value_{Im}$.

The set of values is then defined as

$$Value_{Im} = AtomValue_{Im} \cup ContainerValue_{Im}$$

with

- $AtomValue_{Im} = ClassValue_{Im} \cup LiteralValue \cup (\{\mathsf{enum}\} \times EnumValue_{Tm}) \cup (\{\mathsf{data} \times \mathbb{S}\})$
- $LiteralValue = (\{\mathsf{bool}\} \times \mathbb{B}) \cup (\{\mathsf{int}\} \times \mathbb{Z}) \cup (\{\mathsf{real}\} \times \mathbb{R}) \cup (\{\mathsf{string}\} \times \mathbb{S})$
- $ClassValue_{Im} = \{\mathsf{obj}\} \times (Object_{Im} \cup \mathsf{nil})$
- $ContainerValue_{Im} = \{\mathsf{setof}, \mathsf{bagof}, \mathsf{seqof}, \mathsf{ordof}\} \times Value^*_{Im}$ *(where $Value^*_{Im}$ allows containers to recursively contain other containers.)*

The set of values is recursively defined as the smallest solution of the given set of equations for $Value_{Im}$ and $ContainerValue_{Im}$. Furthermore, elements of the set $Value_{Im}$ are written using square brackets, e.g. $[\mathsf{string}, \text{“}Example\text{”}]$ *or* $[\mathsf{setof}, \langle[\mathsf{int}, 4], [\mathsf{int}, 8]\rangle]$.

 Also see `Value` *in* `Ecore.Instance_Model`

For custom data types, the value is an element from the set $\mathbb{S}$. In Ecore, custom data types can be made serializable, which means a value from $\mathbb{S}$ can be stored for the custom data type. Thus, the value for a custom data type can be stored in the model, but it cannot be further interpreted.

Containers attributed as setof or ordof are considered to have unique values, whereas containers attributed as bagof or seqof are not. This means for example that a tuple with two or more identical values is not a valid value for a container attributed as setof or ordof, see also Definition 3.2.15.

Additionally, the values of a container attributed as bagof or setof are considered unordered, and seqof or ordof ordered. This affects the equivalency of containers, as defined in Definition 3.2.14.

In the example, the set of atomic values that are assigned consists of

$$\begin{aligned}\{&[\mathsf{string}, \text{“Small House”}], [\mathsf{string}, \text{“John Doe”}], [\mathsf{string}, \text{“Jane Doe”}],\\ &[\mathsf{int}, 1], [\mathsf{int}, 2], [\mathsf{int}, 3], [\mathsf{int}, 24], [\mathsf{int}, 23],\\ &[\mathsf{enum}, (.\mathsf{PaymentInterval}, \mathsf{MONTH})]\\ &[\mathsf{obj}, 5], [\mathsf{obj}, 6]\}\end{aligned}$$

Note that only the Renter objects are in an atomic assigned value for the field $(.\mathsf{Room}, \mathsf{renter})$, as it is the only field that references a single object. All other relations in the type model are container types, and as such all the objects are contained in a container value as well. For example, the container value for the rooms field of the House object is $[\mathsf{setof}, \langle[\mathsf{obj}, 2], [\mathsf{obj}, 3], [\mathsf{obj}, 4]\rangle]$ (in no particular order, as the relation is of a set container type).

Each instance model also defines an equivalence relation for values. This relation allows the comparison of aggregate values and explicitly defines equivalency for unordered container values.

Definition 3.2.14 (Value equivalency)
Two values are equivalent ($\equiv_{Im} \subseteq Value_{Im} \times Value_{Im}$) if both the type is identical and the actual value

content is equivalent. It is defined as the smallest reflexive relation between values and the relations defined by the rules given next.

For atomic values equivalence is defined as

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

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

Sequence container equivalency

$$\frac{c_1 = \left[\mathsf{seqof}, \langle v_1, \ldots, v_n \rangle\right] \qquad c_2 = \left[\mathsf{seqof}, \langle u_1, \ldots, u_n \rangle\right] \qquad v_1 \equiv_{Im} u_1, \ldots, v_n \equiv_{Im} u_n}{c_1 \equiv_{Im} c_2}$$

Ordered set container equivalency

$$\frac{c_1 = \left[\mathsf{ordof}, \langle v_1, \ldots, v_n \rangle\right] \qquad c_2 = \left[\mathsf{ordof}, \langle u_1, \ldots, u_n \rangle\right] \qquad v_1 \equiv_{Im} u_1, \ldots, v_n \equiv_{Im} u_n}{c_1 \equiv_{Im} c_2}$$

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

Set container equivalency

$$\frac{c_1 = \left[\mathsf{setof}, \langle v_1, \ldots, v_n \rangle\right] \qquad c_2 = \left[\mathsf{setof}, \langle u_1, \ldots, u_n \rangle\right] \qquad \exists f : \{1, \ldots, n\} \rightarrowtail\!\!\!\!\!\rightarrow \{1, \ldots, n\} : v_i \equiv_{Im} u_{f(i)}}{c_1 \equiv_{Im} c_2}$$

Bag container equivalency

$$\frac{c_1 = \left[\mathsf{bagof}, \langle v_1, \ldots, v_n \rangle\right] \qquad c_2 = \left[\mathsf{bagof}, \langle u_1, \ldots, u_n \rangle\right] \qquad \exists f : \{1, \ldots, n\} \rightarrowtail\!\!\!\!\!\rightarrow \{1, \ldots, n\} : v_i \equiv_{Im} u_{f(i)}}{c_1 \equiv_{Im} c_2}$$

Also see `value_equiv` *in* `Ecore.Instance_Model`

In the example, the value $\left[\mathsf{setof}, \left\langle [\mathsf{obj}, 2], [\mathsf{obj}, 3] \right\rangle\right]$ would thus be equivalent to $\left[\mathsf{setof}, \left\langle [\mathsf{obj}, 3], [\mathsf{obj}, 2] \right\rangle\right]$, as the ordering does not matter for 'setof' container types.

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

Definition 3.2.15 (Valid type values)
The $Valid_{Im}$ set contains tuples which indicate what values are valid for a given type, which is defined by

$$Valid_{Im} \subseteq (Type_{Tm} \times Value_{Im})$$

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

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

The contents of the $Valid_{Im}$ set is then defined as follows:

Data type values:

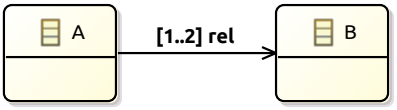
$$\frac{v \in \mathbb{B}}{[\mathsf{bool}, v] :_{Im} \mathsf{boolean}} \qquad \frac{v \in \mathbb{Z}}{[\mathsf{int}, v] :_{Im} \mathsf{integer}} \qquad \frac{v \in \mathbb{R}}{[\mathsf{real}, v] :_{Im} \mathsf{real}} \qquad \frac{v \in \mathbb{S}}{[\mathsf{string}, v] :_{Im} \mathsf{string}}$$

Class values:

$$\frac{ObjectClass_{Im}(o) = c \qquad !c \sqsubseteq_{Tm} t \qquad t \in ClassType_{Tm}}{[\mathsf{obj}, o] :_{Im} t} \qquad \frac{t \in \{\mathsf{nullable}\} \times Class_{Tm}}{[\mathsf{obj}, \mathsf{nil}] :_{Im} t}$$

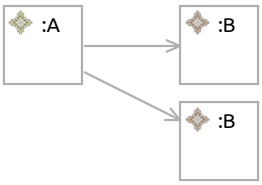
Enumeration values:

$$\frac{(ename, eval) \in EnumValue_{Tm} \qquad ename \in Enum_{Tm}}{[\mathsf{enum}, (ename, eval)] :_{Im} ename}$$

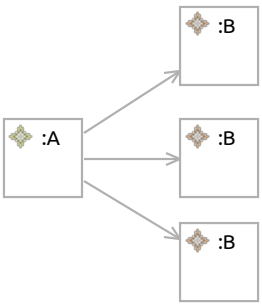


(a) Example type model

(b) Invalid instance model: cardinality of rel too low



(c) Valid instance model: cardinality of rel within bounds



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

Figure 3.3: Examples of valid and invalid multiplicities

User-defined data type values:

$$\frac{v \in \mathbb{S} \qquad t \in UserDataType_{Tm}}{[\mathsf{data}, v] :_{Im} t}$$

Container values:

$$\frac{v_1 :_{Im} T, \ldots, v_n :_{Im} T \qquad \langle v_1, \ldots, v_n \rangle \text{ distinct} \qquad [\mathsf{setof}, T] \in Container_{Tm}}{[\mathsf{setof}, \langle v_1, \ldots, v_n \rangle] :_{Im} [\mathsf{setof}, T]}$$

$$\frac{v_1 :_{Im} T, \ldots, v_n :_{Im} T \qquad [\mathsf{bagof}, T] \in Container_{Tm}}{[\mathsf{bagof}, \langle v_1, \ldots, v_n \rangle] :_{Im} [\mathsf{bagof}, T]}$$

$$\frac{v_1 :_{Im} T, \ldots, v_n :_{Im} T \qquad \langle v_1, \ldots, v_n \rangle \text{ distinct} \qquad [\mathsf{ordof}, T] \in Container_{Tm}}{[\mathsf{ordof}, \langle v_1, \ldots, v_n \rangle] :_{Im} [\mathsf{ordof}, T]}$$

$$\frac{v_1 :_{Im} T, \ldots, v_n :_{Im} T \qquad [\mathsf{seqof}, T] \in Container_{Tm}}{[\mathsf{seqof}, \langle v_1, \ldots, v_n \rangle] :_{Im} [\mathsf{seqof}, T]}$$

Also see `Valid` *in* `Ecore.Instance_Model`

The validity of an instance model depends on the multiplicity of field values. The valid multiplicities depend on the types and field signatures in the corresponding type model. As a consequence, a valid multiplicity also requires the type of the value to be valid. The multiplicity is of most influence for container values, as they can contain an arbitrary amount of values.

Definition 3.2.16 (Multiplicity validity)
A field value $((object, field), value) \in FieldValue_{Im}$ *has a valid multiplicity if the following property holds:*

$$value :_{Im} \text{type}_{Tm}(field) \land value = [t, \langle v_1, \ldots, v_n \rangle] \in ContainerValue_{Im} \implies \text{lower}_{Im}(field) \leq n \leq \text{upper}_{Im}(field)$$

This may be written as $\text{validMul}_{Im}\big(((object, field), value)\big)$.

Also see `validMul` *in* `Ecore.Instance_Model`

The examples shown in Figure 3.3 show different multiplicities in instance models. More specifically, Figure 3.3a shows a type model that specifies a multiplicity of 1..2 for the rel relation. Figure 3.3b and Figure 3.3d show two instance models that have an invalid multiplicity (too low and too high respectively),

whereas Figure 3.3c shows an instance model with correct multiplicity (an alternative correct instance model could have only a single instance of class B).

In order to simplify reasoning over assignments of values, the edgeCount and edge operators are defined. These operators specify the number of relations (and the existence thereof) between any two objects.

Definition 3.2.17 (Value edges)
Let $a, b \in Object_{Im}$ *and* $r \in Field_{Tm}$ *where* $r \in fields_{Tm}(\text{ObjectClass}_{Im}(a))$. *Furthermore, we define* $\text{containerCount}_{Im}(a, r, b)$ *as*

$$\text{containerCount}_{Im}(a, r, b) = \left|\left\{i \in \mathbb{N} \mid \left((a, r), \left[t, \langle v_1, \ldots, v_n\rangle\right]\right) \in \text{FieldValue}_{Im} \wedge v_i = [\mathsf{obj}, b]\right\}\right|$$

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

$$\text{edgeCount}_{Im}(a, r, b) = \begin{cases} 0, & \text{if } \text{type}_{Tm}(r) \notin Container_{Tm} \\ & \quad \wedge \left((a, r), [\mathsf{obj}, b]\right) \notin \text{FieldValue}_{Im} \\ 1, & \text{if } \text{type}_{Tm}(r) \notin Container_{Tm} \\ & \quad \wedge \left((a, r), [\mathsf{obj}, b]\right) \in \text{FieldValue}_{Im} \\ \text{containerCount}_{Im}(a, r, b), & \text{otherwise} \end{cases}$$

Also see `edgeCount` *in* `Ecore.Instance_Model`

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

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

Also see `edge` *in* `Ecore.Instance_Model`

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

Definition 3.2.18 (Property satisfaction)
Given an instance model Im *and a type model* Tm, *a property* $p \in Prop_{Tm}$ *can be satisfied, written as* $Im \models p$, *if the satisfaction formula holds for* p.

- *The abstract property* $[\mathsf{abstract}, c]$ *is satisfied by some instance model* Im *if none of the objects in* Im *is typed by class* c.

  Formally, the satisfaction formula for $Im \models [\mathsf{abstract}, c]$ *is defined as:*

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

- *The containment property* $[\mathsf{containment}, r]$ *is satisfied for an instance model* Im *when any object in* Im *that is the target for a containment relation is contained by no more than one object, and there are no cycles in the instance model given the containment values.*

  Let $CR_{Tm} = \{r \mid r \in Rel_{Tm} \wedge [\mathsf{containment}, r] \in Prop_{Tm}\}$ *be the set of all containment relations in a type model* Tm. *The satisfaction formula for* $Im \models [\mathsf{containment}, r]$ *is then defined as:*

  $$\begin{aligned} &\forall o \in Object_{Im} : \left|\left\{\left((fo, ff), fv\right) \mid \left((fo, ff), fv\right) \in \text{FieldValue}_{Im} \wedge [\mathsf{obj}, o] = fv \wedge ff \in CR_{Tm}\right\}\right| \leq 1 \\ &\quad \wedge \left\{(fo, fv) \mid \left((fo, ff), fv\right) \in \text{FieldValue}_{Im} \wedge ff \in CR_{Tm}\right\} \text{ is acyclic} \end{aligned}$$

- *The identity property* $[\mathsf{identity}, c, A]$. *is satisfied for an instance model* Im, *when for each pair of objects of class* c, *the values for at least one of the attributes in* A *is different.*

  Formally, the satisfaction formula for $Im \models [\mathsf{identity}, c, A]$ *is defined as:*

  $$\begin{aligned} \forall o, o' \in Object_{Im} : &\text{ObjectClass}_{Im}(o) = c \wedge \text{ObjectClass}_{Im}(o') = c \\ &\wedge \forall a \in A : \text{FieldValue}_{Im}((o, a)) \equiv_{Im} \text{FieldValue}_{Im}((o', a)) \\ &\implies o = o' \end{aligned}$$

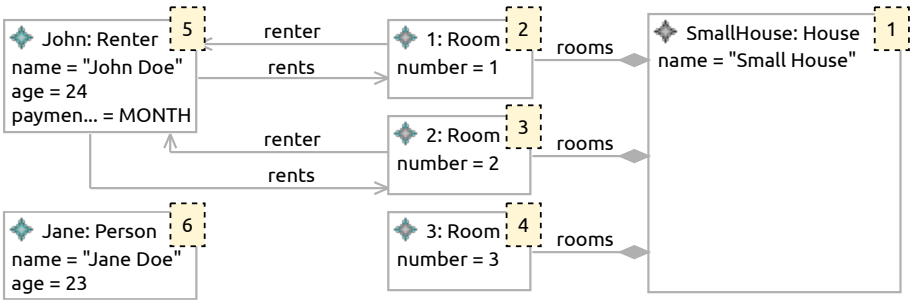


Figure 3.4: Model not satisfying the abstract property.

- *The keyset property* $[\mathsf{keyset}, r, A]$ *is satisfied for an instance model* Im *when for each object containing relation* r*, each pair of objects referenced by* r *has a different set of values for the attributes in* A*. In other words, for each such pair, there is at least one value for the attributes in* A *that is different for both objects.*

  The satisfaction formula for $Im \models [\mathsf{keyset}, r, A]$ *is defined as:*

$$\begin{aligned}\forall o, o', p \in Object_{Im} : r \in \text{fields}_{Tm}(\text{ObjectClass}_{Im}(p)) \\ \wedge\, \text{edge}_{Im}(p, r, o) \wedge \text{edge}_{Im}(p, r, o') \\ \wedge\, \forall a \in A : \text{FieldValue}_{Im}((o, a)) \equiv_{Im} \text{FieldValue}_{Im}((o', a)) \\ \implies o = o'\end{aligned}$$

- *The opposite property* $[\mathsf{opposite}, r, r']$ *is satisfied for an instance model* Im *when for each object* o *with a value for* r*, the referenced objects by* r *have a value for* r'*, which references object* o*. In other words: each object referenced by* r *must also have a reference* r' *that references the source object that defined* r*.*

  Formally, the satisfaction formula for $Im \models [\mathsf{opposite}, r, r']$ *given an instance is defined as:*

$$\forall o, o' \in Object_{Im} : \text{edgeCount}_{Im}(o, r, o') = \text{edgeCount}_{Im}(o', r', o)$$

  Also see `property_satisfaction` *in* `Ecore.Instance_Model`

Figure 3.2 shows an example of an instance model that satisfies the $[\mathsf{abstract}, \mathsf{Person}]$ property, as no direct instantiations of the Person class exist. On the other hand, Figure 3.4 shows an instance model that does not satisfy the property, as the Person class has been instantiated (by the object Jane).

Figure 3.5a shows a type model that defines two containment relations, in opposite direction. The instance model given in Figure 3.5b does not satisfy the satisfaction formula for the containment property, as there exists a cycle of containment relations. This is corrected in the instance model in Figure 3.5c, where such a cycle does not exist (and each object is containment by at most one other object).

To illustrate the satisfaction of a identity property, assume the type model in Figure 3.1 specifies an identity property for the name and age attributes of the Renter object. Formally, we define the following property:

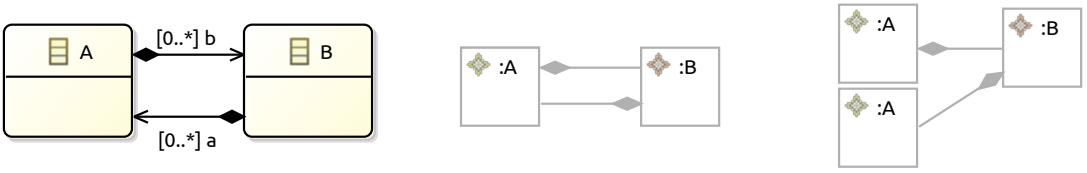
$$\big[\mathsf{identity}, .\mathsf{Renter}, \big\{(.\mathsf{Person}, \mathsf{name}), (.\mathsf{Person}, \mathsf{age})\big\}\big]$$

The example in Figure 3.6 shows an instance model that does not satisfy the property, as the Renter objects share the same values for the name and age attributes, but are still identified as different objects. In Figure 3.2 the values of the name and age attributes are not the same, and thus the property would be satisfied.

An example of the keyset property is shown in Figure 3.7. In Figure 3.7a, we see the type model of this example. We assume there exists a class A which can reference objects of class B through relation rel. Furthermore, we assume that the key field on class B is used as key for the relation rel. In that case, Figure 3.7b shows a violation of the keyset property, because the 2 objects of type B have the same value for key. In Figure 3.7c, the property is satisfied as both objects of type B have a different value for key.

In Figure 3.8, an example model is shown which does not satisfy the opposite property for the rents and renter relations. Although the number of relations is equal, they do not have the same source and target objects (in opposite direction). The example model in Figure 3.2 does in fact satisfy the property.

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



(a) Example type model with two containment relations

(b) Model not satisfying the containment property

(c) Model satisfying the containment property

Figure 3.5: Examples of the containment property.

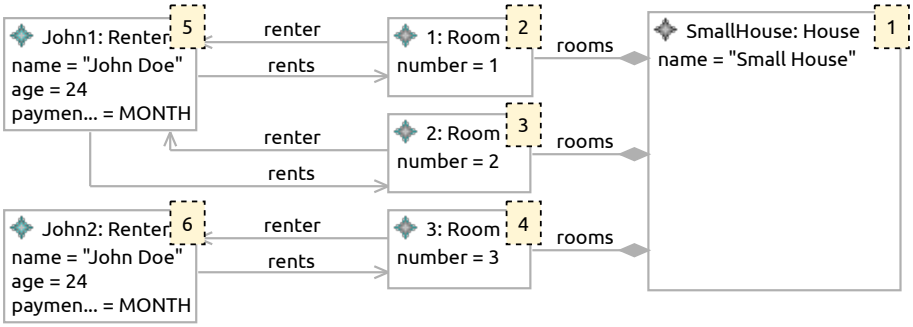
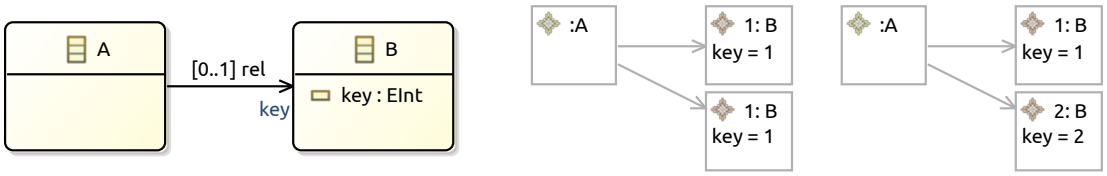


Figure 3.6: Model not satisfying the identity property.



(a) Example type model with relation rel that has the keyset property defined on field key

(b) Model not satisfying the keyset property

(c) Model satisfying the keyset property

Figure 3.7: Examples of the keyset property.

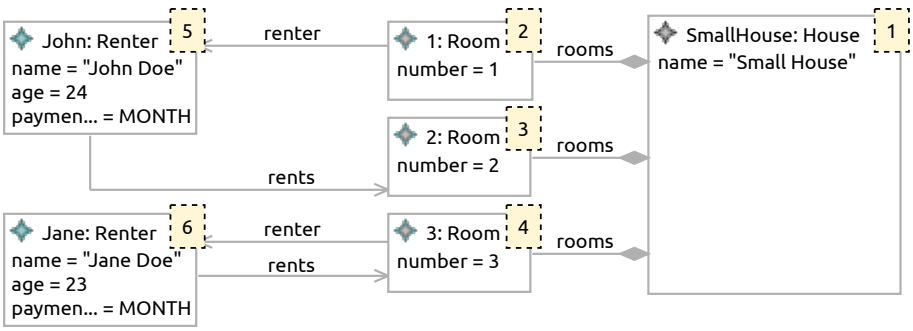


Figure 3.8: Model not satisfying the opposite property.

Definition 3.2.19 (Model validity)
An instance model Im is said to be valid with respect to type model Tm if and only if

- *All values are correctly typed:* $\forall((obj, field), val) \in \text{FieldValue}_{Im} : val :_{Im} \text{type}_{Tm}(field)$.
- *All container multiplicities are valid:* $\forall fv \in \text{FieldValue}_{Im} : \text{validMul}_{Im}(fv)$.
- *All properties are satisfied:* $\forall p \in Prop_{Tm} : Im \models p$
- *All default values have the correct type:* $\forall c \in Constant_{Tm} : \text{DefaultValue}_{Im}(c) :_{Im} \text{ConstType}_{Tm}(c)$.
- *Tm is consistent, as defined in Definition 3.2.11.*

The validity of Im with respect to Tm is written as $Tm \vdash Im$.

Also see `instance_model` *in* `Ecore.Instance_Model`

3.3 GROOVE formalisation

This section discusses a (partial) formalisation of GROOVE. This formalisation is limited to type graphs and instance graphs, as discussed in Section 2.2. These are the only GROOVE graph types that are relevant to this thesis.

3.3.1 Definitions

This section discusses some definitions specific to the GROOVE formalisation. The definitions need to be in place before the formalisations of the different GROOVE graphs are given.

GROOVE internally uses a set of labels *Lab* for each defined grammar. These labels are used by multiple graph types of GROOVE, including (but not limited to) type graphs and instance graphs.

Definition 3.3.1 (Labels)
Lab is the set of labels used by GROOVE graphs. It can be subdivided into three sets:

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

The intersection of each of the sets Lab_t, Lab_f and Lab_e has to be empty:

$$Lab_t \cap Lab_f = \emptyset \wedge Lab_t \cap Lab_e = \emptyset \wedge Lab_f \cap Lab_e = \emptyset$$

Also see `Lab` *in* `GROOVE.Type_Graph`

The set Lab_t will be used to denote the types of nodes in the graph, while the Lab_e set will be used to distinguish between different edges. The Lab_f set are labels for particular kind of edges which always have an identical source and target node. These are used as flags on nodes to indicate that a specific property holds for a node.

Although the intersection of each of the sets Lab_t, Lab_f and Lab_e has to be empty, the sets do not necessarily form a partition of *Lab*, as one or more of the subsets can be empty. Furthermore, *Lab* itself can be empty if the GROOVE grammar does not define any types, flags or edge labels.

Besides *Lab* that belongs to a grammar, GROOVE uses a set of reserved primitive type labels Lab_{prim} that can never be part of the label set of a grammar.

Definition 3.3.2 (Primitive type labels)
GROOVE has a set of reserved primitive type labels Lab_{prim}:

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

It should hold that $Lab \cap Lab_{prim} = \emptyset$ since the primitive type labels are reserved.

The primitive type labels allow the use of primitive types as attributes and values. The label bool represents the type for boolean values $\mathbb{B}$, int represents the type for integer values $\mathbb{Z}$, real represents the type for real numbers $\mathbb{R}$ and finally string represents the type for string values $\mathbb{S}$.

3.3.2 Type graphs

In GROOVE, type graphs are used to constrain the valid instance graphs within the grammar. From a type graph follows a set of valid instance graphs that can be used for verification.

Definition 3.3.3 (Type graph)
A type graph is modeled as tuple TG:

$$TG = \langle NT, ET, \sqsubseteq, abs, \text{mult}, contains \rangle \tag{3.1}$$

with

- $NT \subseteq Lab_t \cup Lab_{prim}$ *is the set of nodes in the type graph. The nodes can consist of type labels (see Definition 3.3.1) or primitive type labels (see Definition 3.3.2).*
- $ET \subseteq NT \times (Lab_e \cup Lab_f) \times NT$ *is the set of (directed) edges in the type graph, which is a set of triples containing the source and target node, as well as the edge label or flag label (see Definition 3.3.1) used to identify the edge.*
- $\sqsubseteq \subseteq NT \times NT$ *is the inheritance relation, the set of tuples of nodes between which an inheritance relation exists.*
- $abs \subseteq NT$ *is the (possibly empty) subset of nodes in the type graph which are considered abstract. An instance graph cannot instantiate abstract nodes.*
- $\text{mult} : ET \Rightarrow \mathbb{M} \times \mathbb{M}$ *is the function which maps edges to their multiplicity pair. See Definition 3.3.4 for the definition.*
- $contains \subseteq ET$ *is the set of edges which identify an containment relation.*

Also see `type_graph` *in* `GROOVE.Type_Graph`

An example of a type graph is given in Figure 3.9. This example is similar to the type model example discussed in Section 3.2.2. There is a node House which contains Rooms. A House also has an edge to a primitive type label string under edge label name which represents the name of the house. Please note that in the visual representation, syntactic sugar is used to represent this edge. Instead of an extra node and edge, it is represented as part of the House node. This syntactic sugar can be used for edges to primitive types and are in reality still treated as an edge to a separate node type. A Room has an edge number, targeting the primitive type label int, which represents the number of the room within the house. A Room can be rented by a Renter. The Renters have edges to the Rooms they rented under the edge label rents, while a Room can access its Renter through the edge with edge label renter. A Renter extends the abstract Person node type, which has 2 edges age and name, targeting the primitive type labels int and string respectively. These edges represent the age and the name of the Person. Finally, a Renter has an edge under the edge label payment_interval, which points to a PaymentInterval node type. This node type is abstract and the edge should therefore point to one of its subtypes, PaymentInterval$MONTH or PaymentInterval$QUARTER. This represents the interval in which the Renter pays the rent. Notable from the definition is that the nodes set N can contain primitive type labels. As a consequence, primitive type labels need to be added explicitly to a type graph in order to use primitive type values in an instance graph.

Furthermore, each edge has a multiplicity pair tied to it, which is defined as the mult function in the type graph definition. The multiplicity pair consists of an incoming multiplicity and an outgoing multiplicity. The incoming multiplicity determines the allowed amount of nodes that share the same target node with this edge type. On the other hand, the outgoing multiplicity determines the number of edges a single source node may have to its target nodes.

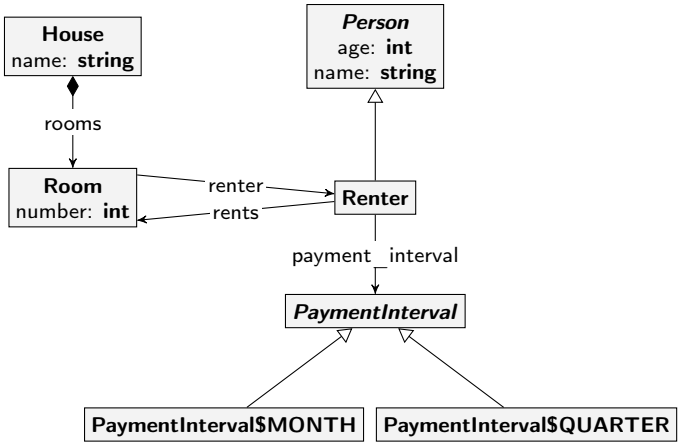
Definition 3.3.4 (Multiplicity pair)
A multiplicity pair is defined as a tuple of two multiplicities, $\mathbb{M} \times \mathbb{M}$, in which the first value denotes the incoming multiplicity and the second value the outgoing multiplicity.

For any multiplicity pair, we define two functions:

$$\text{in} : \mathbb{M} \times \mathbb{M} \Rightarrow \mathbb{M}$$
$$\text{out} : \mathbb{M} \times \mathbb{M} \Rightarrow \mathbb{M}$$

The in *function being the function which from a multiplicity pair returns the incoming multiplicity and the* out *function being the function that returns the outgoing multiplicity, so:*

$$\forall m = (m_{in}, m_{out}) \in \text{mult}_{TG} : \text{in}(m) = m_{in} \wedge \text{out}(m) = m_{out}$$

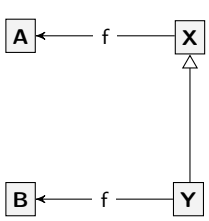


(a) Type graph in GROOVEs visual notation. Multiplicities are omitted for clarity.

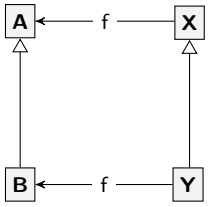
$$
\begin{aligned}
NT_{TG} &= \{\mathsf{House}, \mathsf{PaymentInterval}, \mathsf{PaymentInterval\$MONTH}, \\
&\quad \mathsf{PaymentInterval\$QUARTER}, \mathsf{Person}, \mathsf{Renter}, \mathsf{Room}, \mathsf{int}, \mathsf{string}\} \\
ET_{TG} &= \{(\mathsf{House}, \mathsf{name}, \mathsf{string}), (\mathsf{House}, \mathsf{rooms}, \mathsf{Room}), \\
&\quad (\mathsf{Person}, \mathsf{age}, \mathsf{int}), (\mathsf{Person}, \mathsf{name}, \mathsf{string}), \\
&\quad (\mathsf{Renter}, \mathsf{payment\_interval}, \mathsf{PaymentInterval}), (\mathsf{Renter}, \mathsf{rents}, \mathsf{Room}), \\
&\quad (\mathsf{Room}, \mathsf{number}, \mathsf{int}), (\mathsf{Room}, \mathsf{renter}, \mathsf{Renter})\} \\
\sqsubseteq_{TG} &= \{(\mathsf{House}, \mathsf{House}), (\mathsf{PaymentInterval}, \mathsf{PaymentInterval}), \\
&\quad (\mathsf{PaymentInterval\$MONTH}, \mathsf{PaymentInterval}), \\
&\quad (\mathsf{PaymentInterval\$MONTH}, \mathsf{PaymentInterval\$MONTH}), \\
&\quad (\mathsf{PaymentInterval\$QUARTER}, \mathsf{PaymentInterval}), \\
&\quad (\mathsf{PaymentInterval\$QUARTER}, \mathsf{PaymentInterval\$QUARTER}), \\
&\quad (\mathsf{Person}, \mathsf{Person}), (\mathsf{Renter}, \mathsf{Person}), (\mathsf{Renter}, \mathsf{Renter}), (\mathsf{Room}, \mathsf{Room}), (\mathsf{int}, \mathsf{int}), (\mathsf{string}, \mathsf{string})\} \\
abs_{TG} &= \{\mathsf{PaymentInterval}, \mathsf{Person}\} \\
\mathrm{mult}_{TG} &= \big\{\big((\mathsf{House}, \mathsf{name}, \mathsf{string}), (0..*, 1..1)\big), \big((\mathsf{House}, \mathsf{rooms}, \mathsf{Room}), (1..1, 1..*)\big), \\
&\quad \big((\mathsf{Person}, \mathsf{age}, \mathsf{int}), (0..*, 1..1)\big), \big((\mathsf{Person}, \mathsf{name}, \mathsf{string}), (0..*, 1..1)\big), \\
&\quad \big((\mathsf{Renter}, \mathsf{payment\_interval}, \mathsf{PaymentInterval}), (0..*, 1..1)\big), \\
&\quad \big((\mathsf{Renter}, \mathsf{rents}, \mathsf{Room}), (0..*, 0..*)\big), \\
&\quad \big((\mathsf{Room}, \mathsf{number}, \mathsf{int}), (0..*, 1..1)\big), \big((\mathsf{Room}, \mathsf{renter}, \mathsf{Renter}), (0..*, 0..1)\big)\big\} \\
contains_{TG} &= \{(\mathsf{House}, \mathsf{rooms}, \mathsf{Room})\}
\end{aligned}
$$

(b) Formal definition of the type graph

Figure 3.9: Example of a type graph corresponding with Definition 3.3.3



(a) Valid type graph without ambiguity



(b) Invalid type graph with an ambiguous edge type labelled f

Figure 3.10: Example of ambiguity within edge types

Also see `multiplicity_pair` *in* `GROOVE.Multiplicity_Pair`

With all definitions in place, it is possible to define a valid type graph. The definition of a valid type graph introduces some new constraint that should hold for a type graph to be valid.

Definition 3.3.5 (Type graph validity)
For a type graph to be valid, the following properties must hold:

1. *There may not be any ambiguity in the use of edges:* $\forall (s_1, l, t_1) \in ET_{TG} \land (s_2, l, t_2) \in ET_{TG} : \big((s_1, s_2) \in \sqsubseteq_{TG} \lor (s_2, s_1) \in \sqsubseteq_{TG}\big) \land \big((t_1, t_2) \in \sqsubseteq_{TG} \lor (t_2, t_1) \in \sqsubseteq_{TG}\big) \implies s_1 = s_2 \land t_1 = t_2$.
2. *Flags should have the same source and target node:* $\forall (s, l, t) \in ET_{TG} : l \in Lab_f \implies s = t$.
3. $\sqsubseteq_{TG}$ *is a partial order (*$\sqsubseteq_{TG}$ *is reflexive, transitive and anti-symmetric on N).*
4. *The incoming multiplicities of edges that identify a containment relation are valid:* $\forall e \in contains_{TG} : \text{in}(\text{mult}_{TG}(e)) = (0, 1) \lor \text{in}(\text{mult}_{TG}(e)) = (1, 1)$.

Also see `type_graph` *in* `GROOVE.Type_Graph`

The last 3 properties presented here are mostly self-explanatory. The first property might be unclear at first. This property prevents type graphs from having ambiguous edge types. Figure 3.10 shows an example of such an ambiguity. In essence, when creating edges within an instance graph, there should be an unique solution for typing the edge. In Figure 3.10a, this is always the case, even though both edges are labelled f. If an edge labelled f references a node of type B, then the edge type should be $(\mathsf{Y}, \mathsf{f}, \mathsf{B})$. When an edge labelled f references a node of type A, the edge type should be $(\mathsf{X}, \mathsf{f}, \mathsf{A})$. There is no ambiguity possible.

Figure 3.10b shows an example of a type graph where ambiguity is possible. When a node of type Y references a node of B using an edge labelled f, it is unclear which edge type was meant. Both $(\mathsf{Y}, \mathsf{f}, \mathsf{B})$ and $(\mathsf{X}, \mathsf{f}, \mathsf{A})$ would be valid edge types here. This means there is ambiguity in how edges are typed. The first property of Definition 3.3.5 excludes this case, since $(\mathsf{Y}, \mathsf{f}, \mathsf{B})$ and $(\mathsf{X}, \mathsf{f}, \mathsf{A})$ are both edge types, while $(\mathsf{Y}, \mathsf{X}) \in \sqsubseteq_{TG}$ and $(\mathsf{B}, \mathsf{A}) \in \sqsubseteq_{TG}$. Then according to the first property, Y should be equal to X and B should be equal to A, which is not the case, so Figure 3.10b violates the first property, hence the example is invalid.

3.3.3 Instance graphs

Before giving a formal definition of instance graphs, the set *Node* for a grammar needs to be defined. *Node* is the set of possible nodes that can be part of an instance graph.

Definition 3.3.6 (Node)
We define a set Node containing all possible instance nodes. Node can be subdivided into two disjoint and covering sets

- $Node_t$ *is the set of typed nodes for every label in* Lab_t
- $Node_v$ *is the set of typed nodes for every label in* Lab_{prim}

By definition, there exists a mapping for each element in $Node_t$ *to an element in* Lab_t *and a mapping for each element in* $Node_v$ *to* Lab_{prim}. *We call the mapping* nodeType*:*

$$\text{nodeType} : (Node_t \Rightarrow Lab_t) \cup (Node_v \Rightarrow Lab_{prim})$$

Furthermore, the set $Node_v$ *can also be mapped to actual data values in GROOVE by the function value.*

$$\text{value} : Node_v \rightarrowtail \mathbb{B} \cup \mathbb{Z} \cup \mathbb{R} \cup \mathbb{S}$$

Also see `Node` *in* `GROOVE.Instance_Graph`

Besides the set of nodes, GROOVE also has a set Id, which is a set of identifiers that can be used to assign each node a unique identifier.

Definition 3.3.7 (Identifiers)
Define a global set Id which consists of unique identifiers.

With these definitions, it is possible to define an instance graph in GROOVE.

Definition 3.3.8 (Instance graph)
An instance graph corresponding to a type graph TG is modeled as tuple IG:

$$IG = \langle N, E, ident \rangle \tag{3.2}$$

where

- $N \subseteq Node_t \cup Node_v$ *is the set of nodes in the instance graph.*
- $E \subseteq N \times ET_{TG} \times N$ *is the set of edges in the instance graph. They consist of a source and target node from N and are typed by an edge from the corresponding type graph.*
- $ident : Id \Rightarrow (N \cap Node_t)$ *is a partial injective function which maps selected identifiers from the set Id to a node from N typed by a label in Lab_t.*

Also see `instance_graph` *in* `GROOVE.Instance_Graph`

An example of an instance graph is given in Figure 3.11. This instance graph is typed by the type graph in the example in Figure 3.9. As we can see, there is one node typed House, three nodes typed Room and two nodes typed Renter. The node typed House is identified by identifier 'TwoRem' and is named "Small House" and contains all the three room nodes. The first two nodes that are typed room Room have identifiers 'Longhorn' and 'Shorthorn' and are rented by the first Renter identified as 'Renter1', as is visible from the rents and renter edges. The last node typed Room has identifier 'onghornLay' and is rented by the second Renter identified as 'Renter2'. Please note how the instance graph only has one instance of PaymentInterval$MONTH which is reused by both Renter nodes. Also, notice how there is no instance of PaymentInterval$QUARTER since there is no Renter referencing it. Also notice how the Renter typed objects now have an age and name edge, which are inherited from the Person node type.

As we have seen in the example, all nodes and edges in the instance graph are typed based on the types defined in its type graph. We define two functions for mapping nodes and edges to their type in the type graph. The nodes in the instance graph can be mapped to their type in the type graph using the function type_n, while edges in the instance graph can be mapped to their type in the type graph using the function type_e.

Definition 3.3.9 (Types)
The type of a node in the instance graph can be determined using type_n

$$\text{type}_n : N_{IG} \Rightarrow Lab_t \cup Lab_{prim}$$

for which holds that $\forall n \in N_{IG} : \text{type}_n(n) = nodeType(n)$.

The type of an edge in the instance graph can be determined using type_e

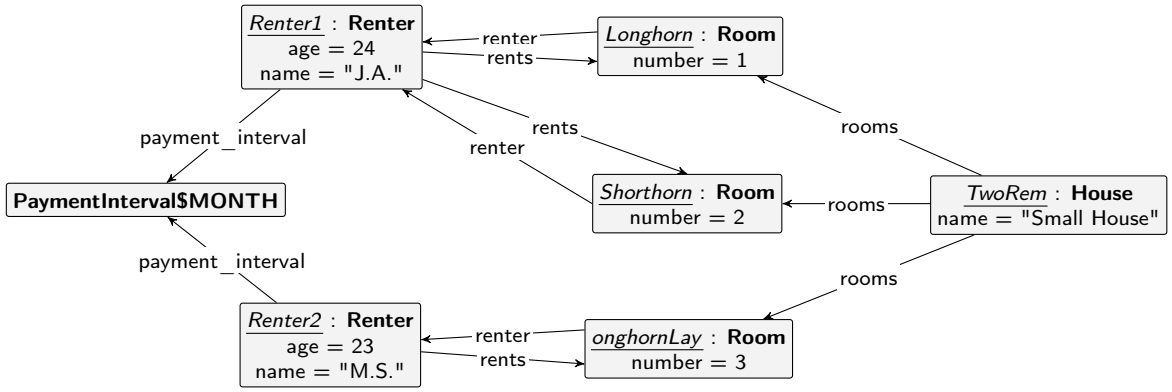
$$\text{type}_e : E_{IG} \Rightarrow ET_{TG}$$

for which holds that $\forall e = (src, etype, tgt) \in E_{IG} : \text{type}_e(e) = etype$

Since instance graphs are typed by a corresponding type graph, the type graph imposes some constraints on the instance graph to ensure that the instance graph is valid. If any of these constraints are violated, the instance graph is considered invalid.

Definition 3.3.10 (Instance graph validity)
An instance graph is valid if the following constraints hold:

1. *The nodes must be properly typed:* $\forall n \in N_{IG} : \text{type}_n(n) \in NT_{TG}$.
2. *The source of each edge must be properly typed:* $\forall e \in E_{IG} : \text{type}_n\big(\text{src}(e)\big) \sqsubseteq_{TG} \text{src}\big(\text{type}_e(e)\big)$.



(a) Instance graph in GROOVEs visual notation

$$
\begin{aligned}
N_{IG} = \{&house, payintervalmonth, renter1, renter2, room1, room2, room3,\\
&1, 2, 3, 23, 24, \text{“J.A.”}, \text{“M.S.”}, \text{“Small House”}\}\\
type_n = \big\{&(house, \mathsf{House}), (payintervalmonth, \mathsf{PaymentInterval\$MONTH}),\\
&(renter1, \mathsf{Renter}), (renter2, \mathsf{Renter}), (room1, \mathsf{Room}), (room2, \mathsf{Room}), (room3, \mathsf{Room}),\\
&(1, \mathsf{int}), (2, \mathsf{int}), (3, \mathsf{int}), (23, \mathsf{int}), (24, \mathsf{int}), (\text{“J.A.”}, \mathsf{string}), (\text{“M.S.”}, \mathsf{string}), (\text{“Small House”}, \mathsf{string})\big\}\\
E_{IG} = \big\{&\big(house, (\mathsf{House}, \mathsf{name}, \mathsf{string}), \text{“Small House”}\big), \big(house, (\mathsf{House}, \mathsf{rooms}, \mathsf{Room}), room1\big),\\
&\big(house, (\mathsf{House}, \mathsf{rooms}, \mathsf{Room}), room2\big), \big(house, (\mathsf{House}, \mathsf{rooms}, \mathsf{Room}), room3\big),\\
&\big(renter1, (\mathsf{Person}, \mathsf{age}, \mathsf{int}), 24\big), \big(renter2, (\mathsf{Person}, \mathsf{age}, \mathsf{int}), 23\big),\\
&\big(renter1, (\mathsf{Person}, \mathsf{name}, \mathsf{string}), \text{“J.A.”}\big), \big(renter2, (\mathsf{Person}, \mathsf{name}, \mathsf{string}), \text{“M.S.”}\big),\\
&\big(renter1, (\mathsf{Renter}, \mathsf{payment\_interval}, \mathsf{PaymentInterval}), payintervalmonth\big),\\
&\big(renter2, (\mathsf{Renter}, \mathsf{payment\_interval}, \mathsf{PaymentInterval}), payintervalmonth\big),\\
&\big(renter1, (\mathsf{Renter}, \mathsf{rents}, \mathsf{Room}), room1\big), \big(renter1, (\mathsf{Renter}, \mathsf{rents}, \mathsf{Room}), room2\big),\\
&\big(renter2, (\mathsf{Renter}, \mathsf{rents}, \mathsf{Room}), room3\big), \big(room1, (\mathsf{Room}, \mathsf{number}, \mathsf{int}), 1\big),\\
&\big(room2, (\mathsf{Room}, \mathsf{number}, \mathsf{int}), 2\big), \big(room3, (\mathsf{Room}, \mathsf{number}, \mathsf{int}), 3\big),\\
&\big(room1, (\mathsf{Room}, \mathsf{renter}, \mathsf{Renter}), renter1\big), \big(room2, (\mathsf{Room}, \mathsf{renter}, \mathsf{Renter}), renter1\big),\\
&\big(room3, (\mathsf{Room}, \mathsf{renter}, \mathsf{Renter}), renter2\big)\big\}\\
ident_{IG} = \{&(\text{TwoRem}, house), (\text{Renter1}, renter1), (\text{Renter2}, renter2),\\
&(\text{Longhorn}, room1), (\text{Shorthorn}, room2), (\text{onghornLay}, room3)\}
\end{aligned}
$$

(b) Formal definition of the instance graph

Figure 3.11: Example of an instance graph corresponding with Definition 3.3.8

3. *The target of each edge must be properly typed:* $\forall e \in E_{IG} \colon \mathrm{type}_n\big(\mathrm{tgt}(e)\big) \sqsubseteq_{TG} \mathrm{tgt}\big(\mathrm{type}_e(e)\big)$.
4. *Abstract types cannot have instances:* $\forall n \in N_{IG} \colon \mathrm{type}_n(n) \notin abs_{TG}$.
5. *The outgoing multiplicity of each edge type must be adhered to:* $\forall et \in ET_{TG} : \forall n \in N_{IG} : \mathrm{type}_n(n) \sqsubseteq_{TG} \mathrm{src}(et) \implies |\{e \in E_{IG} \mid \mathrm{src}(e) = n \wedge \mathrm{type}_e(e) = et\}| \in \mathrm{out}(\mathrm{mult}_{TG}(et))$.
6. *The incoming multiplicity of each edge type must be adhered to:* $\forall et \in ET_{TG} : \forall n \in N_{IG} : \mathrm{type}_n(n) \sqsubseteq_{TG} \mathrm{tgt}(et) \implies |\{e \in E_{IG} \mid \mathrm{tgt}(e) = n \wedge \mathrm{type}_e(e) = et\}| \in \mathrm{in}(\mathrm{mult}_{TG}(et))$.
7. *Nodes must be contained by at most one other node:* $\forall n \in N_{IG} \colon |\{e \in E_{IG} \mid \mathrm{tgt}(e) = n \wedge \mathrm{type}_e(e) \in contains_{TG}\}| \leq 1$.
8. *There may be no cycle between the containment edges in* E_{IG}.

 Also see `instance_graph` *in* `GROOVE.Instance_Graph`

The first property just ensures that all nodes in an instance graph are typed by the corresponding type graph. Figure 3.12 shows an example of an invalid instance graph typed by the type graph of Figure 3.9. The instance graph is invalid, since the type Kitchen is not defined within the type graph. Changing the type of this node to Room would make the instance graph valid.

The second and third property ensure that the source and target nodes of edges are correctly typed. Figure 3.13 shows an example of an invalid instance graph typed by the type graph of Figure 3.9. In this example, the Room-typed node has a number edge connected to a boolean value. Since the target type of the (Room, number, int) is an integer, the third property is violated. Therefore, the instance graph is invalid. Changing the boolean node to any integer node would make the example valid.

The fourth property ensures that an instance graph cannot instantiate abstract node types. Figure 3.14 shows an example of an invalid instance graph typed by the type graph of Figure 3.9. In the example, the only node in the graph is typed by the Person type. However, the Person type is abstract within the type graph. Therefore, the fourth property is violated, and the instance graph is invalid. Changing the type of the node to Renter would make the instance graph valid.

The fifth and sixth property ensure the correctness of multiplicities within an instance graph. The fifth property ensures that the outgoing multiplicity is not violated. The sixth property ensures that the incoming multiplicity is not violated. Figure 3.15 shows two instance graphs that are once more typed by type graph Figure 3.9 but are both invalid. Figure 3.15a violates the fifth property, since the node typed by type House needs to have at least one outgoing edge labelled rooms to a node of type Room. Figure 3.15b violates the sixth property, since the node typed by type Room needs to have at least one incoming edge labelled rooms from a node of type House. Merging these two instance graphs and adding an edge labelled rooms from the House-typed node to the Room-typed node would result in a valid instance graph.

The seventh property ensures that nodes can only be contained by one other node. Figure 3.16 shows an example of an instance graph that is typed by type graph Figure 3.9, but for this example we assume that Hotel is a defined type similar to House, with the same outgoing edge types and properties. In this case, the incoming multiplicity property (the sixth property) will not be violated, as (House, rooms, Room) and (Hotel, rooms, Room) are different edge types with their own incoming multiplicity. However, the example is still invalid. (House, rooms, Room) and (Hotel, rooms, Room) are both containment edges, which means that the Room-typed node now has 2 incoming containment edges. The seventh property prohibits this, and therefore the example is invalid.

The eighth and final property ensures that there are no cycles in the containment edges of an instance graph. Consider the instance graph shown in Figure 3.17b with is typed by type graph Figure 3.17a. This instance graph is valid, as there is no cycle in the containment edges. However, the instance graph shown in Figure 3.17c is invalid, because it has a cycle in the containment edges. This violates the eighth property and is therefore invalid.

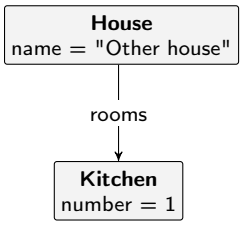


Figure 3.12: Invalid instance graph with improper node types

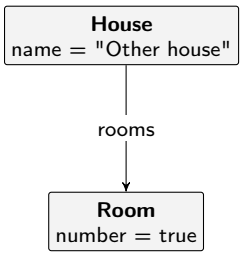


Figure 3.13: Invalid instance graph with improper type for the target of edge *number*

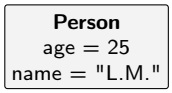
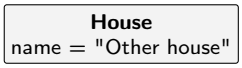
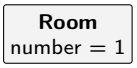


Figure 3.14: Invalid instance graph with node of type Person



(a) Outgoing multiplicity violated



(b) Incoming multiplicity violated

Figure 3.15: Invalid instance graphs with their multiplicity constraints violated

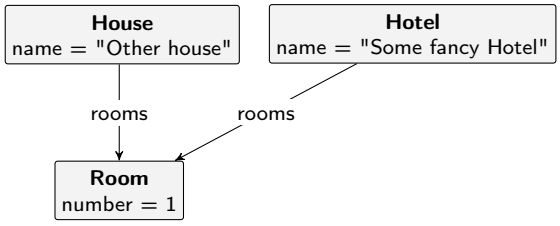
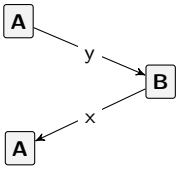
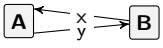


Figure 3.16: Invalid instance graph with multiple containments for the Room-typed node

(a) Type graph



(b) Correct use: No cycles



(c) Incorrect use: Cycle present

Figure 3.17: Example of correct and incorrect use of containment edges

Chapter 4

Transformation framework

The previous chapter introduced formalisations for GROOVE graphs and Ecore models. These formalisations allow us to reason about the GROOVE graphs and Ecore models in a formal way. In this chapter, these formalisations will be the foundation of a framework that allows for the formalisation of model transformations between Ecore and GROOVE.

Creating a formal transformation between Ecore models and GROOVE graphs is non-trivial, first and foremost because Ecore has more instrumentation to express individual elements that GROOVE cannot express directly. For example, Ecore models can directly express enumeration types and values, whereas GROOVE cannot. The same holds for properties related to relations and attributes, as well as user-defined data types and constants. This difference in instrumentation can be solved using encodings, which the transformation framework should support.

Another complexity with a formal transformation between Ecore models and GROOVE graphs is the infinite number of possible transformation functions. Because of the existence of infinitely many models and graphs, there is also an infinite number of model transformations possible. Since it is impractical to prove the correctness of each transformation function, a more systematic solution is needed. As discussed in Section 1.3, the transformation framework will be structured such that individual transformation functions can be composed while preserving the correctness. This composability allows the user to combine simple transformations into more substantial transformations, without the need of proving these transformations separately.

Section 4.1 explains how encodings are used to deal with the elements that GROOVE cannot express directly. Section 4.2 explains the structure of the transformation framework, which is set up to allow the composability of transformation functions. The remaining sections in this chapter further explain how to apply this structure.

4.1 Encodings

As explained earlier, GROOVE graphs cannot express all elements of Ecore directly. A way to deal with the limitations of GROOVE graphs is by using encodings for these Ecore elements. By encoding, we mean expressing these Ecore model elements as one or more GROOVE graph elements, which preserve all information. The set of GROOVE graph elements then represents a code for the corresponding Ecore element.

For example, consider an enumeration type in Ecore, `EnumExample`, given in Figure 4.1. GROOVE has no direct way to express enumerations. However, we can use different encodings to express the enumeration type indirectly. One of such encodings would be to create an abstract node type for the enumeration type and then create a separate node type for each of the values corresponding to the enumeration type. Each of these node types then inherits from the abstract enumeration node type created earlier. An example

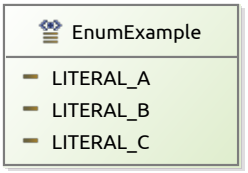
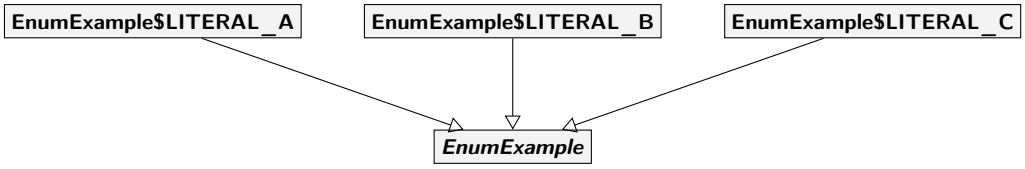


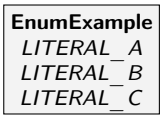
Figure 4.1: Example of an enumeration type in Ecore notation



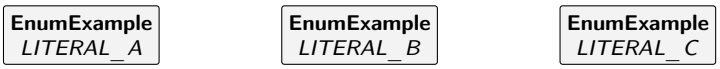
(a) GROOVE type graph

(b) GROOVE instance graph

Figure 4.2: Encoding of the `EnumExample` enumeration type as different nodes types



(a) GROOVE type graph



(b) GROOVE instance graph

Figure 4.3: Encoding of the `EnumExample` enumeration type as one type with multiple flags

of this specific encoding is given in Figure 4.2. Within an instance graph, each value of an enumeration type is expressed by creating a single node. Each of these nodes is typed by the value node types created earlier. When referencing a value, an edge to the corresponding node can be created.

Another possible encoding uses flags in GROOVE. In that case, a single node type is created for the enumeration type. This node type gets multiple flags, one for each possible enumeration value. An example of this is shown in Figure 4.3a. Within an instance graph, a node is created for each value of the enumeration type. Each of these nodes is typed by the enumeration node type and has a single flag corresponding to the value. An example of these instances is given in Figure 4.3b. This way, each node expresses one value of the enumeration type. When referencing an enumeration value, an edge to the corresponding node can be created.

As the previous examples have shown, creating encodings allows for expressing elements that are unique to Ecore in GROOVE. Moreover, each of these elements might correspond to multiple encodings. The examples for enumeration types presented earlier are non-exhaustive. There are more encodings for enumeration types possible. In other words, there is no single encoding for each element. The choice between different encodings depends on the user, as each encoding might have its advantages and disadvantages.

4.2 Structure

Creating a transformation function between Ecore models and GROOVE graphs is a difficult task because there are infinitely many possible transformations that vary in complexity. Complexity in these transformation functions is created by the validity constraints of Ecore models and GROOVE graphs. Another factor that adds complexity is the possibility to mix encodings. For example, a transformation function might encode enumeration type `A` different from enumeration type `B`.

The transformation framework presented in this chapter deals with the complexity by providing a structure for building transformation functions out of smaller building blocks. The framework starts with a trivial transformation (the transformation between an empty model and an empty graph). It then adds elements to this transformation, such that we build the model and graph iteratively. When applying each building block, the transformation function is extended while preserving its correctness. Furthermore, the framework ensures that when applying each building block, the correctness of both the model and graph is preserved.

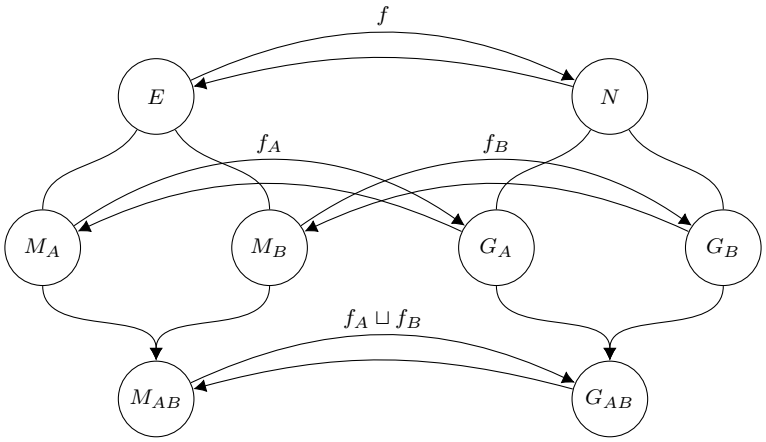


Figure 4.4: General structure for building a transformation function

In order for the framework to guarantee these properties, the smaller building blocks are applied in a general structure. A visualisation of this structure is given in Figure 4.4. The structure assumes that we have a partially build model M_A, which is valid and corresponds to a valid graph G_A under a certain bijective transformation function f_A. In other words, f_A transforms model M_A to graph G_A and the inverse of f_A transforms graph G_A to model M_A.

The next step is to add a building block. This building block is represented by valid model M_B, which corresponds to a valid graph G_B under a specific bijective transformation function f_B. Furthermore, we assume that M_A and M_B are entirely distinct except for some set of elements E, which are the only shared elements among the models M_A and M_B. In the same way, we assume that graphs G_A and G_B are entirely distinct except for a set of nodes N, which are the only shared nodes between graphs G_A and G_B. Finally, there also exists a specific bijective transformation function f, which transforms the set of elements E to the set of nodes N and vice-versa.

Within the framework, we will present a way to merge models M_A and M_B into a model M_{AB}, while preserving validity. We also present a way to merge graphs G_A and G_B intro graph G_{AB}, again while preserving validity. Then we present a way to merge two transformation functions f_A and f_B in order to create a transformation function $f_A \sqcup f_B$, which transforms the combined model M_{AB} to the combined graph G_{AB}. By performing these operations, a larger transformation function is created which can transform the larger model M_{AB} to the larger graph G_{AB}. Since M_{AB} and G_{AB} are valid and $f_A \sqcup f_B$ is a bijective transformation function between model M_{AB} and graph G_{AB}, it is possible to reuse them as the next M_A, G_A and f_A in the model and add another building block. This way, complex model structures and transformation functions are created iteratively while each step in the process maintains a formal proof.

In the following sections, the explained structure is applied to type models and type graphs, and then also to instance models and instance graphs. These sections also discuss the necessary definitions and proofs needed to apply the framework.

4.3 Type models and type graphs

In this section, the proposed framework structure is applied to type models and type graphs. First, the general structure and its requirements are discussed. Then the required definitions and theorems are given.

Figure 4.5 shows an alternation of the structure proposed in Section 4.2 applied to type models and type graphs. As before, type model Tm_A represents the partially build model which corresponds to type graph TG_A under the transformation function f_A. Type model Tm_B represents the next building block to add to this model. It corresponds to type graph TG_B under the bijective transformation function f_B.

Type models Tm_A and Tm_B are entirely distinct except for a set types T, which means $T \subseteq Type_{Tm_A} \wedge T \subseteq Type_{Tm_B}$. In a similar way, type graphs TG_A and TG_B are entirely distinct except for a set of node types N, so $N \subseteq NT_{TG_A} \wedge N \subseteq NT_{TG_B}$.

Type models Tm_A and Tm_B are combined into type model Tm_{AB} using Definition 4.3.1. In a similar way type graphs TG_A and TG_B are combined into type graph TG_{AB} using Definition 4.3.14. Lemma 4.3.12 and Lemma 4.3.23 respectively show that Tm_{AB} and TG_{AB} are valid. Then Definition 4.3.26 and Definition 4.3.31 can be used to merge the transformation functions f_A and f_B into $f_A \sqcup f_B$, where

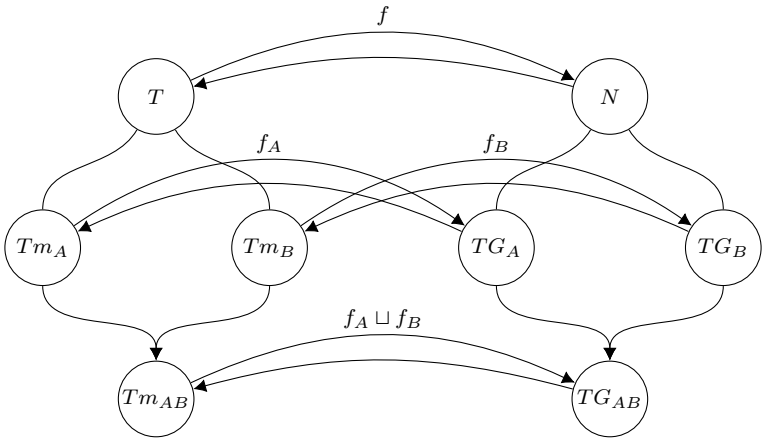


Figure 4.5: Structure for transforming between type models and type graphs

Theorem 4.3.28 and Theorem 4.3.29 show that $f_A \sqcup f_B$ is again a valid transformation function transforming Tm_{AB} to TG_{AB}. Similarly, Theorem 4.3.35 and Theorem 4.3.36 show that the inverse function of $f_A \sqcup f_B$ is again a valid transformation function transforming TG_{AB} to Tm_{AB}.

4.3.1 Combining type models

The structure of Figure 4.5 shows that the type models Tm_A and Tm_B are combined into one type model Tm_{AB}. This section provides the definition of this combination and its corresponding theorems. Please note that the definitions presented here are as generic as possible, and do not actively take into account that Tm_A and Tm_B are mostly distinct. This bit of information is added later as part of a theorem and proof.

Definition 4.3.1 (Combination function on type models)
combine *is a binary function on two type models which combines two type models into one type model. It is defined as follows:*

$$\begin{aligned}\text{combine}(Tm_A, Tm_B) = \langle & Class = Class_{Tm_A} \cup Class_{Tm_B} \\ & Enum = Enum_{Tm_A} \cup Enum_{Tm_B} \\ & UserDataType = UserDataType_{Tm_A} \cup UserDataType_{Tm_B} \\ & Field = Field_{Tm_A} \cup Field_{Tm_B} \\ & \text{FieldSig} = \text{fieldsig\_combine}(Tm_A, Tm_B) \\ & EnumValue = EnumValue_{Tm_A} \cup EnumValue_{Tm_B} \\ & Inh = Inh_{Tm_A} \cup Inh_{Tm_B} \\ & Prop = \text{prop\_combine}(Tm_A, Tm_B) \\ & Constant = Constant_{Tm_A} \cup Constant_{Tm_B} \\ & \text{ConstType} = \text{consttype\_combine}(Tm_A, Tm_B)\rangle\end{aligned}$$

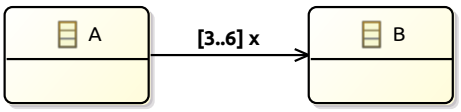
In which fieldsig_combine *is given as part of Definition 4.3.2,* prop_combine *as part of Definition 4.3.4 and* consttype_combine *as part of Definition 4.3.3.*

Also see `tmod_combine` *in* `Ecore.Type_Model_Combination`

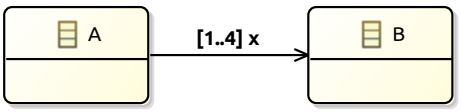
The combination of two type models is rather simple in its definition, at least for all the sets defined as part of a type model. Intuitively, the definition makes sense. To combine two type models, we need the types from both type models, so we merge the classes, enumerations types, enumeration values and user-defined data types. The constants should also be preserved, so these are merged too. To preserve all attributes and relations, we merge the set of fields and the inheritance relation as well.

Merging the different functions is done by using a new function. First, the combination of field signatures will be discussed.

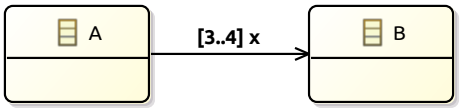
Definition 4.3.2 (Combination function for field signatures)
fieldsig_combine *is a partial function on two type models which returns a new function* $Field_{Tm_{AB}} \Rightarrow$



(a) First type model Tm_A



(b) Second type model Tm_A



(c) Combined type model Tm_{AB}

Figure 4.6: Combination of field signatures when field is present in both type models

$(Type_{Tm_{AB}} \times \mathbb{M})$. *It is defined as follows:*

$$\text{fieldsig\_combine}(Tm_A, Tm_B, f) = \begin{cases} s & \text{if } f \in Field_{Tm_A} \cap Field_{Tm_B} \wedge \text{type}_{Tm_A}(f) = \text{type}_{Tm_B}(f) \\ \text{FieldSig}_{Tm_A}(f) & \text{if } f \in Field_{Tm_A} \setminus Field_{Tm_B} \\ \text{FieldSig}_{Tm_B}(f) & \text{if } f \in Field_{Tm_B} \setminus Field_{Tm_A} \end{cases}$$

where

$$s = \Big(\text{type}_{Tm_A}(f), \big(\max\big(\text{lower}(\text{FieldSig}_{Tm_A}(f)), \text{lower}(\text{FieldSig}_{Tm_B}(f))\big) .. \min\big(\text{upper}(\text{FieldSig}_{Tm_A}(f)), \text{upper}(\text{FieldSig}_{Tm_B}(f))\big)\big)\Big)$$

Also see `tmod_combine_fieldsig` *in* `Ecore.Type_Model_Combination`

Although the above definition looks quite complex, the intuition behind it is straightforward. For a field that only occurs in type model Tm_A, the field signature over from Tm_A is copied. For a field that only occurs in Tm_B, the field signature from Tm_B is copied. In the case that a field occurs in both Tm_A and Tm_B, it should be the case that the type of the fields is the same. If this is indeed the case, the field type is copied, and a new multiplicity is created. This multiplicity takes the maximum of the lower bounds of the field in Tm_A and Tm_B as new lower bound, and the minimum of the upper bounds of the field in Tm_A and Tm_B as new upper bound.

An example of the combination of two field signatures in the case of a field being present in both Tm_A and Tm_B is given in Figure 4.6. It is possible to combine the field x, since in both Tm_A and Tm_B field x references class type B. The multiplicity for both field signatures is different and is combined as defined. The maximum of the lower bounds is taken, which results in $\max(3, 1) = 3$. Furthermore, the minimum of the upper bounds is taken, which results in $\min(6, 4) = 4$. Therefore the multiplicity of x in Tm_{AB} will become 3..4.

Besides a function for field signatures, a type model also defines a function for constant types. The combination of constant types is discussed in the next definition.

Definition 4.3.3 (Combination function for constant types)
consttype_combine *is a partial function on two type models which returns a new function* $Constant_{Tm_{AB}} \Rightarrow Type_{Tm_{AB}}$. *It is defined as follows:*

$$\text{consttype\_combine}(Tm_A, Tm_B, c) = \begin{cases} \text{ConstType}_{Tm_A}(c) & \text{if } c \in Constant_{Tm_A} \cap Constant_{Tm_B} \wedge \text{ConstType}_{Tm_A}(c) = \text{ConstType}_{Tm_B}(c) \\ \text{ConstType}_{Tm_A}(c) & \text{if } c \in Constant_{Tm_A} \setminus Constant_{Tm_B} \\ \text{ConstType}_{Tm_B}(c) & \text{if } c \in Constant_{Tm_B} \setminus Constant_{Tm_A} \end{cases}$$

Also see `tmod_combine_const_type` *in* `Ecore.Type_Model_Combination`

The definition of the combination of constant types is similar to the combination of field signatures. The combination of constant types is less complicated because there is no notion of multiplicities involved.

By definition, if a constant only occurs in Tm_A, the constant type of Tm_A is copied. For constants that only occur in Tm_B, the constant type of Tm_B is copied. In case that a constant occurs in both Tm_A and Tm_B, the constant is copied if the constant types for that constant are the same in both Tm_A and Tm_B.

The last definition that remains to be given is the definition of combining the set of properties. The set of properties cannot be united by merely taking the union of $Prop_{Tm_A}$ and $Prop_{Tm_B}$ since this might invalidate the satisfaction of these properties on the level of an instance graph. Instead, the inductive set prop_combine is defined to specify under which circumstances a property can be combined.

Definition 4.3.4 (Combination of model properties)
prop_combine(Tm_A, Tm_B) *is defined as a subset of* $Prop_{Tm_A} \cup Prop_{Tm_B}$. *The contents of the set are then defined as follows:*

For abstract *properties:*

$$\frac{[\mathsf{abstract}, c] \in Prop_{Tm_A} \qquad c \notin Class_{Tm_B}}{[\mathsf{abstract}, c] \in \text{prop\_combine}(Tm_A, Tm_B)} \qquad \frac{[\mathsf{abstract}, c] \in Prop_{Tm_B} \qquad c \notin Class_{Tm_A}}{[\mathsf{abstract}, c] \in \text{prop\_combine}(Tm_A, Tm_B)}$$

$$\frac{[\mathsf{abstract}, c] \in Prop_{Tm_A} \qquad [\mathsf{abstract}, c] \in Prop_{Tm_B}}{[\mathsf{abstract}, c] \in \text{prop\_combine}(Tm_A, Tm_B)}$$

For containment *properties:*

$$\frac{[\mathsf{containment}, r] \in Prop_{Tm_A}}{[\mathsf{containment}, r] \in \text{prop\_combine}(Tm_A, Tm_B)} \qquad \frac{[\mathsf{containment}, r] \in Prop_{Tm_B}}{[\mathsf{containment}, r] \in \text{prop\_combine}(Tm_A, Tm_B)}$$

For defaultValue *properties:*

$$\frac{[\mathsf{defaultValue}, f, v] \in Prop_{Tm_A} \qquad f \notin Field_{Tm_B}}{[\mathsf{defaultValue}, f, v] \in \text{prop\_combine}(Tm_A, Tm_B)} \qquad \frac{[\mathsf{defaultValue}, f, v] \in Prop_{Tm_B} \qquad f \notin Field_{Tm_A}}{[\mathsf{defaultValue}, f, v] \in \text{prop\_combine}(Tm_A, Tm_B)}$$

$$\frac{[\mathsf{defaultValue}, f, v] \in Prop_{Tm_A} \qquad [\mathsf{defaultValue}, f, v] \in Prop_{Tm_B}}{[\mathsf{defaultValue}, f, v] \in \text{prop\_combine}(Tm_A, Tm_B)}$$

For identity *properties:*

$$\frac{[\mathsf{identity}, c, A] \in Prop_{Tm_A} \qquad c \notin Class_{Tm_B}}{[\mathsf{identity}, c, A] \in \text{prop\_combine}(Tm_A, Tm_B)} \qquad \frac{[\mathsf{identity}, c, A] \in Prop_{Tm_B} \qquad c \notin Class_{Tm_A}}{[\mathsf{identity}, c, A] \in \text{prop\_combine}(Tm_A, Tm_B)}$$

$$\frac{[\mathsf{identity}, c, A] \in Prop_{Tm_A} \qquad [\mathsf{identity}, c, A] \in Prop_{Tm_B}}{[\mathsf{identity}, c, A] \in \text{prop\_combine}(Tm_A, Tm_B)}$$

For keyset *properties:*

$$\frac{[\mathsf{keyset}, r, A] \in Prop_{Tm_A} \qquad r \notin Field_{Tm_B}}{[\mathsf{keyset}, r, A] \in \text{prop\_combine}(Tm_A, Tm_B)} \qquad \frac{[\mathsf{keyset}, r, A] \in Prop_{Tm_B} \qquad r \notin Field_{Tm_A}}{[\mathsf{keyset}, r, A] \in \text{prop\_combine}(Tm_A, Tm_B)}$$

$$\frac{[\mathsf{keyset}, r, A] \in Prop_{Tm_A} \qquad [\mathsf{keyset}, r, A] \in Prop_{Tm_B}}{[\mathsf{keyset}, r, A] \in \text{prop\_combine}(Tm_A, Tm_B)}$$

For opposite *properties:*

$$\frac{[\mathsf{opposite}, r1, r2] \in Prop_{Tm_A} \qquad r1 \notin Field_{Tm_B} \qquad r2 \notin Field_{Tm_B}}{[\mathsf{opposite}, r1, r2] \in \text{prop\_combine}(Tm_A, Tm_B)}$$

$$\frac{[\mathsf{opposite}, r1, r2] \in Prop_{Tm_B} \qquad r1 \notin Field_{Tm_A} \qquad r2 \notin Field_{Tm_A}}{[\mathsf{opposite}, r1, r2] \in \text{prop\_combine}(Tm_A, Tm_B)}$$

$$\frac{[\mathsf{opposite}, r1, r2] \in Prop_{Tm_A} \qquad [\mathsf{opposite}, r1, r2] \in Prop_{Tm_B}}{[\mathsf{opposite}, r1, r2] \in \text{prop\_combine}(Tm_A, Tm_B)}$$

For readonly *properties:*

$$\frac{[\text{readonly}, f] \in Prop_{Tm_A} \qquad f \notin Field_{Tm_B}}{[\text{readonly}, f] \in \text{prop\_combine}(Tm_A, Tm_B)} \qquad \frac{[\text{readonly}, f] \in Prop_{Tm_B} \qquad f \notin Field_{Tm_A}}{[\text{readonly}, f] \in \text{prop\_combine}(Tm_A, Tm_B)}$$

$$\frac{[\text{readonly}, f] \in Prop_{Tm_A} \qquad [\text{readonly}, f] \in Prop_{Tm_B}}{[\text{readonly}, f] \in \text{prop\_combine}(Tm_A, Tm_B)}$$

Also see `tmod_combine_prop` *in* `Ecore.Type_Model_Combination`

As can be seen from the definition of prop_combine(Tm_A, Tm_B), properties are only copied under specific circumstances. For abstract properties, it holds that a class is only abstract in the combination of Tm_A and Tm_B if the class is abstract in both Tm_A and Tm_B or if the class is abstract in one of them, and the class does not occur in the other. Intuitively, this makes sense for correctness. If a class is abstract in both type models, there will not be instances of that class in any of the combined instance models type by those type models. The same holds if the class only occurs in one of the type models, as an instance model cannot contain an instance of a class that is not present in its type model.

For the containment property, it holds that the containment property is always copied over. There are no other conditions here. If there is a containment property in $Prop_{Tm_A} \cup Prop_{Tm_B}$ it will also be in the combination of Tm_A and Tm_B.

A default value property is copied over from one of the type models if the other type model does not have the corresponding field defined. Furthermore, a default value may be in the combination of properties of Tm_A and Tm_B if the field occurs in both, and both have the same constant set as the default value for the field. Intuitively, this last requirement makes sense. If we set a default value for a field within a type model, it should not change after combining the type model with another type model, as instance models might depend on the default value set for that field.

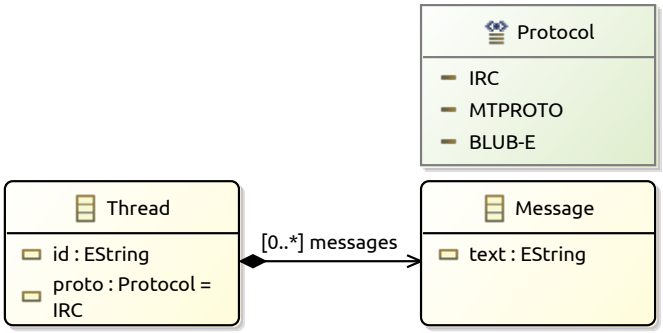
Identity properties follow a similar pattern to default value properties. An identity is copied over from one of the type models if the corresponding class is not defined in the other type model. Furthermore, an identity can be preserved if it is set for the same class and attributes in both type models. Again, intuitively, this is the desired solution. If a type model has a class of which a set of attributes can uniquely define the instances, then this should also be the case after the combination with another type model. Merging two sets of attributes might have preserved the identity property as well, but this would be a questionable decision from a practical standpoint, as this means that the identity of instances changes, which makes no sense in real-world scenarios.

The argumentation for identity properties also holds for keyset properties. Therefore these follow a similar pattern, in which the keyset properties of a type model are only copied if the corresponding field does not occur in the other type model, or if the keyset property for a field occurs with the same attributes in both type models.

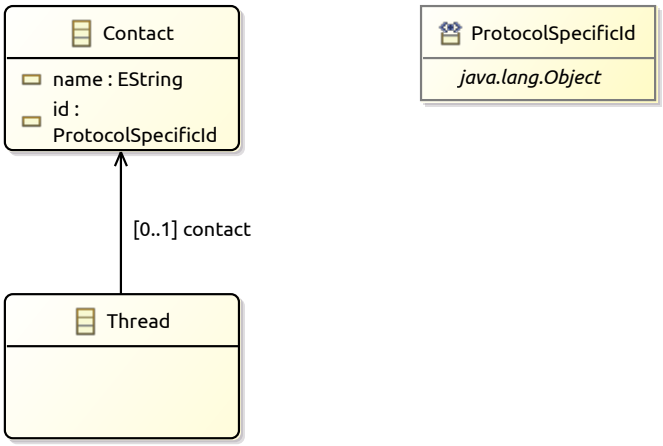
The opposite property is preserved if it occurs in a type model, but the other type model does not define both of the corresponding fields. Alternatively, the property is preserved if it occurs in both type models with the same fields. All different ways to combine an opposite property would result in an invalid type model according to Definition 3.2.11, which is undesired.

The read-only properties follow a similar pattern to the abstract properties. If a field is read-only in both type models, then it is read-only in the combination. Furthermore, if a field is read-only in one of the type models and the field is not defined in the other type model, then the read-only property can be copied too.

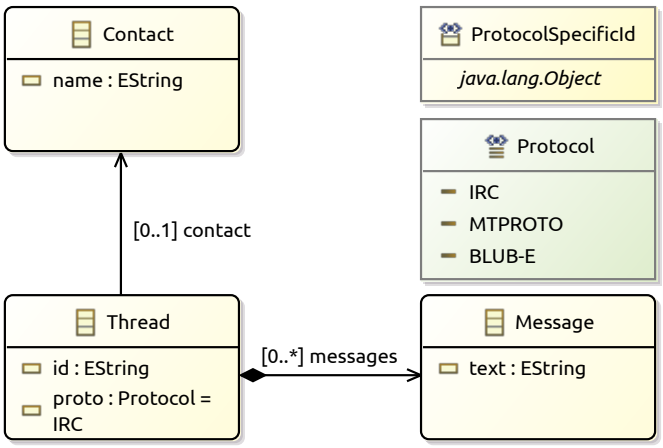
With all definitions in place, it is possible to provide a larger example. Suppose the model of a multi-protocol chat application. It consists of Threads of Messages. Since the application is multi-protocol, each Thread can use one of the supported Protocols. The formal definition of the model of such an application could be as follows:



(a) The chat application model Tm_{Chat}



(b) The contact extension model $Tm_{Extension}$



(c) The extended chat application model $Tm_{ChatExt}$

Figure 4.7: Example of the combination of type models

$$
\begin{aligned}
Tm_{Chat} = \langle \quad & Class = \{.\mathsf{Message}, .\mathsf{Thread}\} \\
& Enum = \{.\mathsf{Protocol}\} \\
& UserDataType = \{\} \\
& Field = \{(.\mathsf{Message}, \mathsf{text}), (.\mathsf{Thread}, \mathsf{id}), (.\mathsf{Thread}, \mathsf{messages}), (.\mathsf{Thread}, \mathsf{proto})\} \\
& \text{FieldSig} = \big\{\big((.\mathsf{Message}, \mathsf{text}), (\mathsf{string}, 1..1)\big), \big((.\mathsf{Thread}, \mathsf{id}), (\mathsf{string}, 1..1)\big), \\
& \qquad \big((.\mathsf{Thread}, \mathsf{messages}), ([\mathsf{seqof}, !.\mathsf{Message}], 0..*)\big), \\
& \qquad \big((.\mathsf{Thread}, \mathsf{proto}), (.\mathsf{Protocol}, 1..1)\big)\big)\big\} \\
& EnumValue = \{(.\mathsf{Protocol}, \mathsf{IRC}), (.\mathsf{Protocol}, \mathsf{MTPROTO}), (.\mathsf{Protocol}, \mathsf{BLUB-E})\} \\
& Inh = \{\} \\
& Prop = \big\{\big(\mathsf{identity}, .\mathsf{Message}, \{(.\mathsf{Thread}, \mathsf{id})\}\big)\big\} \\
& Constant = \{\} \\
& \text{ConstType} = \{\} \\
\rangle &
\end{aligned}
$$

An visual representation of Tm_{Chat} is included as Figure 4.7a. Now, assume a model that represents an extension to this application, adding support for Contacts. Each thread can belong to a Contact. A Contact has a name and some identifier that is protocol specific. For that identifier, the user-defined data type ProtocolSpecificId is introduced. This extension could formally be defined as:

$$
\begin{aligned}
Tm_{Extension} = \langle \quad & Class = \{.\mathsf{Contact}, .\mathsf{Thread}\} \\
& Enum = \{\} \\
& UserDataType = \{.\mathsf{ProtocolSpecificId}\} \\
& Field = \{(.\mathsf{Contact}, \mathsf{name}), (.\mathsf{Contact}, \mathsf{id}), (.\mathsf{Thread}, \mathsf{contact})\} \\
& \text{FieldSig} = \big\{\big((.\mathsf{Contact}, \mathsf{name}), (\mathsf{string}, 1..1)\big), \\
& \qquad \big((.\mathsf{Contact}, \mathsf{id}), (.\mathsf{ProtocolSpecificId}, 1..1)\big), \\
& \qquad \big((.\mathsf{Thread}, \mathsf{contact}), (?.\mathsf{Contact}, 0..1)\big)\big)\big\} \\
& EnumValue = \{\} \\
& Inh = \{\} \\
& Prop = \big\{\big(\mathsf{identity}, .\mathsf{Contact}, \{(.\mathsf{Contact}, \mathsf{id})\}\big)\big\} \\
& Constant = \{\} \\
& \text{ConstType} = \{\} \\
\rangle &
\end{aligned}
$$

The visual representation of the extension is included as Figure 4.7b. Now, using Definition 4.3.1, it is possible to combine these models into one model. This will yield the following model:

$$
\begin{aligned}
Tm_{ChatExt} = \langle \quad Class &= \{.\mathsf{Contact}, .\mathsf{Message}, .\mathsf{Thread}\} \\
Enum &= \{.\mathsf{Protocol}\} \\
UserDataType &= \{.\mathsf{ProtocolSpecificId}\} \\
Field &= \{(.\mathsf{Contact}, \mathsf{name}), (.\mathsf{Contact}, \mathsf{id}), (.\mathsf{Message}, \mathsf{text}), \\
&\quad (.\mathsf{Thread}, \mathsf{contact}), (.\mathsf{Thread}, \mathsf{id}), (.\mathsf{Thread}, \mathsf{messages}), \\
&\quad (.\mathsf{Thread}, \mathsf{proto})\} \\
\mathrm{FieldSig} &= \{\big((.\mathsf{Contact}, \mathsf{name}), (\mathsf{string}, 1..1)\big), \\
&\quad \big((.\mathsf{Contact}, \mathsf{id}), (.\mathsf{ProtocolSpecificId}, 1..1)\big), \\
&\quad \big((.\mathsf{Message}, \mathsf{text}), (\mathsf{string}, 1..1)\big), \\
&\quad \big((.\mathsf{Thread}, \mathsf{contact}), (?.\mathsf{Contact}, 0..1)\big)\big), \\
&\quad \big((.\mathsf{Thread}, \mathsf{id}), (\mathsf{string}, 1..1)\big), \\
&\quad \big((.\mathsf{Thread}, \mathsf{messages}), ([\mathsf{seqof}, !.\mathsf{Message}], 0..*)\big), \\
&\quad \big((.\mathsf{Thread}, \mathsf{proto}), (.\mathsf{Protocol}, 1..1)\big)\big)\} \\
EnumValue &= \{(.\mathsf{Protocol}, \mathsf{IRC}), (.\mathsf{Protocol}, \mathsf{MTPROTO}), (.\mathsf{Protocol}, \mathsf{BLUB-E})\} \\
Inh &= \{\} \\
Prop &= \{\big(\mathsf{identity}, .\mathsf{Contact}, \{(.\mathsf{Contact}, \mathsf{id})\}\big), \\
&\quad \big(\mathsf{identity}, .\mathsf{Message}, \{(.\mathsf{Thread}, \mathsf{id})\}\big)\} \\
Constant &= \{\} \\
\mathrm{ConstType} &= \{\} \\
\rangle &
\end{aligned}
$$

A visual representation of this combined model is included as Figure 4.7c. The example perfectly shows why the combination of type models is useful: It allows for building larger models out of smaller building blocks. This is the exact goal of this definition within the transformation framework.

Although the definitions of the combination of type models are given, no mathematical properties or theorems are defined yet. Some mathematical properties hold for the combination of type models, that will be presented in the following theorems.

Theorem 4.3.5 (Commutativity of the combination of type models)
Assume that Tm_A and Tm_B are type models, then the combine *function is commutative:*

$$\mathrm{combine}(Tm_A, Tm_B) = \mathrm{combine}(Tm_B, Tm_A)$$

Also see `tmod_combine_commute` *in* `Ecore.Type_Model_Combination`

Theorem 4.3.6 (Associativity of the combination of type models)
Assume that Tm_A, Tm_B and Tm_C are type models, then the combine *function is associative:*

$$\mathrm{combine}(\mathrm{combine}(Tm_A, Tm_B), Tm_C) = \mathrm{combine}(Tm_A, \mathrm{combine}(Tm_B, Tm_C))$$

Also see `tmod_combine_assoc` *in* `Ecore.Type_Model_Combination`

Theorem 4.3.7 (Idempotence of the combination of type models)
Assume that Tm_A is a type model and that it is consistent in the sense of Definition 3.2.11. Then the following property holds:

$$\mathrm{combine}(Tm_A, Tm_A) = Tm_A$$

Also see `tmod_combine_idemp_alt` *in* `Ecore.Type_Model_Combination`

These properties follow directly from Definition 4.3.1, but the corresponding proofs will not be included here. It should be noted that these properties are indeed proven correct as part of this thesis, and the corresponding proofs are validated within Isabelle.

Besides these properties, the combination of type models also has an identity element. The empty type model represents this identity element, but it needs to be defined first:

Definition 4.3.8 (Empty type model)
Let Tm_ϵ be the empty type model. Tm_ϵ is defined as:

$$\begin{aligned} Tm_\epsilon = \langle & Class = \{\} \\ & Enum = \{\} \\ & UserDataType = \{\} \\ & Field = \{\} \\ & \text{FieldSig} = undefined \\ & EnumValue = \{\} \\ & Inh = \{\} \\ & Prop = \{\} \\ & Constant = \{\} \\ & \text{ConstType} = undefined\rangle \end{aligned}$$

Theorem 4.3.9 (Correctness of the empty type model)
The empty type model, Tm_ϵ, is consistent with respect to Definition 3.2.11.

Also see `tmod_empty_correct` *in* `Ecore.Type_Model`

The proof for the correctness of the empty type model is trivial. Still, a validated version of this proof can be found within the Isabelle theories of this thesis.

As mentioned earlier, the empty type model acts as an identity element when combining two type models. The following theorem specifies this behaviour.

Theorem 4.3.10 (Identity of the combination of type models)
Assume that Tm_A is a type model and that it is consistent in the sense of Definition 3.2.11. Then Tm_ϵ acts as an identity element in the combination function:

$$\text{combine}(Tm_\epsilon, Tm_A) = Tm_A$$

Also see `tmod_combine_identity_alt` *in* `Ecore.Type_Model_Combination`

Once more, the proof of this theorem follows directly from the definition. Therefore, the corresponding proof will not be included here, but a validated version can be found within the Isabelle theories of this thesis.

A final desired property for the combination of type models is a correctness property. Theorem 4.3.11 defines the theorem under which the combination of type models is a consistent type model. Please note that this theorem is a generic theorem, which does not take into account that the type models are mostly distinct.

Theorem 4.3.11 (Consistency of the combination of type models)
Assume that Tm_A and Tm_B are consistent type models in the sense of Definition 3.2.11. Furthermore, assume the following properties:

- *For all shared fields, the type is the same in both type models:* $\forall f \in Field_{Tm_A} \cap Field_{Tm_B} : \text{type}_{Tm_A}(f) = \text{type}_{Tm_B}(f)$
- *For all shared fields, the combination of the multiplicities is a valid multiplicity:* $\forall f \in Field_{Tm_A} \cap Field_{Tm_B} : \max\big(\text{lower}(\text{FieldSig}_{Tm_A}(f)), \text{lower}(\text{FieldSig}_{Tm_B}(f))\big) .. \min\big(\text{upper}(\text{FieldSig}_{Tm_A}(f)), \text{upper}(\text{FieldSig}_{Tm_B}(f))\big) \in \mathbb{M}$
- *For all shared constants, the constant type is the same in both models:* $\forall f \in Constant_{Tm_A} \cap Constant_{Tm_B} : \text{ConstType}_{Tm_A}(f) = \text{ConstType}_{Tm_B}(f)$
- *Identifiers used for a class in Tm_A cannot be used for an enumeration type or user-defined data type in Tm_B:* $\forall c \in Class_{Tm_A} : c \notin Enum_{Tm_B} \wedge c \notin UserDataType_{Tm_B}$.
- *Identifiers used for a class in Tm_B cannot be used for an enumeration type or user-defined data type in Tm_A:* $\forall c \in Class_{Tm_B} : c \notin Enum_{Tm_A} \wedge c \notin UserDataType_{Tm_A}$.
- *Identifiers used for an enumeration type in Tm_A cannot be used for a class or user-defined data type in Tm_B:* $\forall c \in Enum_{Tm_A} : c \notin Class_{Tm_B} \wedge c \notin UserDataType_{Tm_B}$.
- *Identifiers used for an enumeration type in Tm_B cannot be used for a class or user-defined data type in Tm_A:* $\forall c \in Enum_{Tm_B} : c \notin Class_{Tm_A} \wedge c \notin UserDataType_{Tm_A}$.

- *Identifiers from Tm_A may not be in the namespace of an identifier in Tm_B:* $\forall x \in Class_{Tm_A} \cup Enum_{Tm_A} \cup UserDataType_{Tm_A}; y \in Class_{Tm_B} \cup Enum_{Tm_B} \cup UserDataType_{Tm_B}$: *x not in the namespace of y*
- *Identifiers from Tm_B may not be in the namespace of an identifier in Tm_A:* $\forall x \in Class_{Tm_B} \cup Enum_{Tm_B} \cup UserDataType_{Tm_B}; y \in Class_{Tm_A} \cup Enum_{Tm_A} \cup UserDataType_{Tm_A}$: *x not in the namespace of y*
- *The transitive closure of the inheritance relation is irreflexive:* $(Inh_{Tm_A} \cup Inh_{Tm_B})^+$ *is irreflexive*
- *For any superclass with an identity, the identity of the subclasses must be a superset of the identity of the superclass:* $\forall c_1\ c_2\ A_1\ A_2 : [\mathsf{identity}, c_1, A_1] \in \text{prop\_combine}(Tm_A, Tm_B) \wedge [\mathsf{identity}, c_2, A_2] \in \text{prop\_combine}(Tm_A, Tm_B) \wedge c_1 \neq c_2 \wedge\ !c_1 \not\sqsubseteq_{Tm_A}\ !c_2 \wedge\ !c_1 \not\sqsubseteq_{Tm_B}\ !c_2 \wedge\ !c_1 \sqsubseteq_{\text{combine}(Tm_A, Tm_B)}\ !c_2 \implies A \subseteq B$
- *For all shared fields, if Tm_A defines a default value, Tm_B should define the same default value, and vice versa:* $\forall f \in Field_{Tm_A} \cap Field_{Tm_B} : [\mathsf{defaultValue}, f, v] \in Prop_{Tm_A} \iff [\mathsf{defaultValue}, f, v] \in Prop_{Tm_B}$.
- *For all shared classes, if Tm_A defines a identity, Tm_B should define the same identity, and vice versa:* $\forall c \in Class_{Tm_A} \cap Class_{Tm_B} : [\mathsf{identity}, c, A] \in Prop_{Tm_A} \iff [\mathsf{identity}, c, A] \in Prop_{Tm_B}$.
- *For all shared fields, if Tm_A defines a keyset, Tm_B should define the same keyset, and vice versa:* $\forall r \in Field_{Tm_A} \cap Field_{Tm_B} : [\mathsf{keyset}, r, A] \in Prop_{Tm_A} \iff [\mathsf{keyset}, r, A] \in Prop_{Tm_B}$.
- *For all shared fields, if Tm_A defines an opposite property, Tm_B should define the same opposite property, and vice versa:* $\forall r \in Field_{Tm_A} \cap Field_{Tm_B} : [\mathsf{opposite}, r, r'] \in Prop_{Tm_A} \iff [\mathsf{opposite}, r, r'] \in Prop_{Tm_B}$

Then $\text{combine}(Tm_A, Tm_B)$ *is a consistent type model in the sense of Definition 3.2.11*

Also see `tmod_combine_correct` *in* `Ecore.Type_Model_Combination`

Proof. To proof that $\text{combine}(Tm_A, Tm_B)$ is a consistent type model, it needs to be shown that $\text{combine}(Tm_A, Tm_B)$ gives rise to a valid structure for a type model and that Definition 3.2.11 holds. For readability, define Tm_{AB} to be $\text{combine}(Tm_A, Tm_B)$.

Structural properties

- All elements of $Class_{Tm_{AB}}$ are elements of Id.

  Follows from $Class_{Tm_A} \subseteq Id$ and $Class_{Tm_B} \subseteq Id$.
- All elements of $Enum_{Tm_{AB}}$ are elements of Id.

  Follows from $Enum_{Tm_A} \subseteq Id$ and $Enum_{Tm_B} \subseteq Id$.
- All elements of $UserDataType_{Tm_{AB}}$ are elements of Id.

  Follows from $UserDataType_{Tm_A} \subseteq Id$ and $UserDataType_{Tm_B} \subseteq Id$.
- All elements of $Field_{Tm_{AB}}$ are elements of $(Class_{Tm_{AB}} \times Name)$.

  Follows from $Field_{Tm_A} \subseteq (Class_{Tm_A} \times Name)$ and $Field_{Tm_B} \subseteq (Class_{Tm_B} \times Name)$. To complete the proof, use $Class_{Tm_{AB}} = Class_{Tm_A} \cup Class_{Tm_B}$.
- For each field f, $\text{FieldSig}_{Tm_{AB}}(f)$ must be an element of $(Type_{Tm_{AB}} \times \mathbb{M})$.

  First, note that $Type_{Tm_{AB}} = Type_{Tm_A} \cup Type_{Tm_B}$
  (see `tmod_combine_type` in `Ecore.Type_Model_Combination`).

  If $f \in Field_{Tm_A} \setminus Field_{Tm_B}$, then $\text{FieldSig}_{Tm_{AB}}(f) \in (Type_{Tm_{AB}} \times \mathbb{M})$.

  Similarly, if $f \in Field_{Tm_B} \setminus Field_{Tm_A}$, then $\text{FieldSig}_{Tm_{AB}}(f) \in (Type_{Tm_{AB}} \times \mathbb{M})$.

  If $f \in Field_{Tm_A} \cap Field_{Tm_B}$, then $\text{type}_{Tm_A}(f) = \text{type}_{Tm_B}(f)$ by assumption. Also, the combined multiplicity is correct by assumption. Therefore $\text{FieldSig}_{Tm_{AB}}(f) \in (Type_{Tm_{AB}} \times \mathbb{M})$.
- All elements of $EnumValue_{Tm_{AB}}$ are elements of $(Enum_{Tm_{AB}} \times Name)$.

  Follows from $EnumValue_{Tm_A} \subseteq (Enum_{Tm_A} \times Name)$ and $EnumValue_{Tm_B} \subseteq (Enum_{Tm_B} \times Name)$. To complete the proof, use $Enum_{Tm_{AB}} = Enum_{Tm_A} \cup Enum_{Tm_B}$.
- All elements of $Inh_{Tm_{AB}}$ are elements of $(Class_{Tm_{AB}} \times Class_{Tm_{AB}})$.

  Follows from $Inh_{Tm_A} \subseteq (Class_{Tm_A} \times Class_{Tm_A})$ and $Inh_{Tm_B} \subseteq (Class_{Tm_B} \times Class_{Tm_B})$. Furthermore, $Class_{Tm_{AB}} = Class_{Tm_A} \cup Class_{Tm_B}$.

- All elements of $Prop_{Tm_{AB}}$ are elements of $Property_{Tm_{AB}}$.

  Make a case distinction for the different possible properties.

  - For $[\mathsf{abstract}, c] \in Prop_{Tm_{AB}}$, use the fact that $c \in Class_{Tm_A} \cup Class_{Tm_B}$. Therefore, $[\mathsf{abstract}, c] \in Property_{Tm_{AB}}$.
  - For $[\mathsf{containment}, r] \in Prop_{Tm_{AB}}$, use the fact that $Rel_{Tm_{AB}} = Rel_{Tm_A} \cup Rel_{Tm_B}$ (see `tmod_combine_rel` in `Ecore.Type_Model_Combination`). Then have $[\mathsf{containment}, r] \in Property_{Tm_{AB}}$.
  - For $[\mathsf{defaultValue}, f, v] \in Prop_{Tm_{AB}}$, use the fact that $f \in Field_{Tm_A} \cup Field_{Tm_B}$ and $v \in Constant_{Tm_A} \cup Constant_{Tm_B}$. Using a case disinction on the combination of properties, it is possible to show that $\text{ConstType}_{Tm_{AB}}(v) \sqsubseteq_{Tm_{AB}} \text{type}_{Tm_{AB}}(f)$. Therefore, $[\mathsf{defaultValue}, f, v] \in Property_{Tm_{AB}}$.
  - For $[\mathsf{identity}, c, A] \in Prop_{Tm_{AB}}$, use the fact that $c \in Class_{Tm_A} \cup Class_{Tm_B}$ and $A \subseteq fields_{Tm_A} \vee A \subseteq fields_{Tm_B}$. Then have that $A \subseteq fields_{Tm_{AB}}$ Therefore, $[\mathsf{identity}, c, A] \in Property_{Tm_{AB}}$.
  - For $[\mathsf{keyset}, r, A] \in Prop_{Tm_{AB}}$, use the fact that $Rel_{Tm_{AB}} = Rel_{Tm_A} \cup Rel_{Tm_B}$ (see `tmod_combine_rel` in `Ecore.Type_Model_Combination`) to show $r \in Rel_{Tm_{AB}}$. Also use the fact that $Attr_{Tm_{AB}} = Attr_{Tm_A} \cup Attr_{Tm_B}$ to show $A \subseteq Attr_{Tm_{AB}}$ (see `tmod_combine_attr` in `Ecore.Type_Model_Combination`). Because types are preserved after combining, it is possible to show that $\forall f \in A : \text{uncontainer}(\text{type}_{Tm_{AB}}(r)) \sqsubseteq_{Tm_{AB}} \text{class}_{Tm_{AB}}(f)$. Furthermore, $\text{type}_{Tm_{AB}}(r) \in (\{\mathsf{setof}, \mathsf{ordof}\} \times ClassType_{Tm_{AB}})$. Therefore, $[\mathsf{keyset}, r, A] \in Property_{Tm_{AB}}$.
  - For $[\mathsf{opposite}, r, r'] \in Prop_{Tm_{AB}}$, use the fact that $Rel_{Tm_{AB}} = Rel_{Tm_A} \cup Rel_{Tm_B}$ (see `tmod_combine_rel` in `Ecore.Type_Model_Combination`) to show $r \in Rel_{Tm_{AB}}$ and $r' \in Rel_{Tm_{AB}}$. Because types are preserved after combining, it is possible to show that $!c1 \sqsubseteq_{Tm_{AB}} \text{uncontainer}(\text{type}_{Tm_{AB}}(r'))$, $!c2 \sqsubseteq_{Tm_{AB}} \text{uncontainer}(\text{type}_{Tm_{AB}}(r))$, $\text{type}_{Tm_{AB}}(r) \notin \{\mathsf{bagof}, \mathsf{seqof}\} \times Type_{Tm_{AB}}$ and finally $type_{Tm_{AB}}(r') \notin \{\mathsf{bagof}, \mathsf{seqof}\} \times Type_{Tm_{AB}}$. Therefore, $[\mathsf{opposite}, r, r'] \in Property_{Tm_{AB}}$.
  - For $[\mathsf{readonly}, f] \in Prop_{Tm_{AB}}$, use the fact that $f \in Field_{Tm_A} \cup Field_{Tm_B}$. Therefore, $[\mathsf{readonly}, f] \in Property_{Tm_{AB}}$.
- All elements of $Constant_{Tm_{AB}}$ are elements of Id.

  Follows from $Constant_{Tm_A} \subseteq Id$ and $Constant_{Tm_B} \subseteq Id$.
- For each constant c, $\text{ConstType}_{Tm_{AB}}(c)$ must be an element of $Type_{Tm_{AB}}$.

  First, note that $Type_{Tm_{AB}} = Type_{Tm_A} \cup Type_{Tm_B}$
  (see `tmod_combine_type` in `Ecore.Type_Model_Combination`).

  If $c \in Constant_{Tm_A} \setminus Constant_{Tm_B}$, then $\text{ConstType}_{Tm_{AB}}(f) \in Type_{Tm_{AB}}$.

  Similarly, if $c \in Constant_{Tm_B} \setminus Constant_{Tm_A}$, then $\text{ConstType}_{Tm_{AB}}(c) \in Type_{Tm_{AB}}$.

  If $c \in Constant_{Tm_A} \cap Constant_{Tm_B}$, then $\text{ConstType}_{Tm_A}(c) = \text{ConstType}_{Tm_B}(c)$ by assumption. Therefore $\text{ConstType}_{Tm_{AB}}(c) \in Type_{Tm_{AB}}$.
- $Class_{Tm_{AB}}$, $DataType$, $Enum_{Tm_{AB}}$ and $UserDataType_{Tm_{AB}}$ are pairwise disjoint.

  Notice that $Class_{Tm_A}$, $DataType$, $Enum_{Tm_A}$, $UserDataType_{Tm_A}$ are pairwise disjoint. Also, $Class_{Tm_B}$, $DataType$, $Enum_{Tm_B}$, $UserDataType_{Tm_B}$ are pairwise disjoint.

  Use that $Class_{Tm_{AB}} = Class_{Tm_A} \cup Class_{Tm_B}$, $Enum_{Tm_{AB}} = Enum_{Tm_A} \cup Enum_{Tm_B}$ and $UserDataType_{Tm_{AB}} = UserDataType_{Tm_A} \cup UserDataType_{Tm_B}$. Use this to split the possible cases.

  Only the case where one element is from $Class_{Tm_A} \cup Enum_{Tm_A} \cup UserDataType_{Tm_A}$ and one element is from $Class_{Tm_B} \cup Enum_{Tm_B} \cup UserDataType_{Tm_B}$ cannot be proven directly. For this case, the proof follows from the assumptions.
- None of the elements in $Class_{Tm_{AB}}$, $DataType$, $Enum_{Tm_{AB}}$ and $UserDataType_{Tm_{AB}}$ may be in the namespace of another element in that set.

  Use that $Class_{Tm_{AB}} = Class_{Tm_A} \cup Class_{Tm_B}$, $Enum_{Tm_{AB}} = Enum_{Tm_A} \cup Enum_{Tm_B}$ and $UserDataType_{Tm_{AB}} = UserDataType_{Tm_A} \cup UserDataType_{Tm_B}$. Use this to split the possible cases.

  It is not possible to directly proof the cases where the identifier comes from $Class_{Tm_A} \cup Enum_{Tm_A} \cup UserDataType_{Tm_A}$ and the namespace comes from $Class_{Tm_B} \cup Enum_{Tm_B} \cup UserDataType_{Tm_B}$.

Furthermore, it is also not possible for the cases where the identifier comes from $Class_{Tm_B} \cup Enum_{Tm_B} \cup UserDataType_{Tm_B}$ and the namespace comes from $Class_{Tm_A} \cup Enum_{Tm_A} \cup UserDataType_{Tm_A}$. For these cases, the proof follows from the assumptions.

- $Inh_{Tm_{AB}}$ is an asymmetric relation, of which the transitive closure is irreflexive.

  The transitive closure of $Inh_{Tm_{AB}}$ is irreflexive by assumption. Then show that $Inh_{Tm_{AB}}$ is an asymmetric relation using the assumption that the transitive closure of $Inh_{Tm_{AB}}$ is irreflexive.

Consistency properties

- For all $\text{type}_{Tm_{AB}}(f) \in DataType \cup Enum_{Tm_{AB}} \cup UserDataType_{Tm_{AB}} \cup (\mathsf{proper} \times Class_{Tm_{AB}})$, it holds that $\text{lower}_{Tm_{AB}}(f) = 1$.

  Use the fact that $\text{type}_{Tm_{AB}}(f) = \text{type}_{Tm_A}(f)$ or $\text{type}_{Tm_{AB}}(f) = \text{type}_{Tm_B}(f)$.

  If $\text{type}_{Tm_{AB}}(f) = \text{type}_{Tm_A}(f)$, then $\text{type}_{Tm_{AB}}(f) \in DataType \cup Enum_{Tm_{AB}} \cup UserDataType_{Tm_{AB}} \cup (\mathsf{proper} \times Class_{Tm_{AB}})$ only when $\text{type}_{Tm_A}(f) \in DataType \cup Enum_{Tm_A} \cup UserDataType_{Tm_A} \cup (\mathsf{proper} \times Class_{Tm_A})$.

  Then if $\text{type}_{Tm_A}(f) \in DataType \cup Enum_{Tm_A} \cup UserDataType_{Tm_A} \cup (\mathsf{proper} \times Class_{Tm_A})$, then $\text{lower}_{Tm_A}(f) = 1$. As a consequence, it must be that $\text{lower}_{Tm_{AB}}(f) = 1$.

  If $\text{type}_{Tm_{AB}}(f) = \text{type}_{Tm_B}(f)$, then $\text{type}_{Tm_{AB}}(f) \in DataType \cup Enum_{Tm_{AB}} \cup UserDataType_{Tm_{AB}} \cup (\mathsf{proper} \times Class_{Tm_{AB}})$ only when $\text{type}_{Tm_B}(f) \in DataType \cup Enum_{Tm_B} \cup UserDataType_{Tm_B} \cup (\mathsf{proper} \times Class_{Tm_B})$.

  Then if $\text{type}_{Tm_B}(f) \in DataType \cup Enum_{Tm_B} \cup UserDataType_{Tm_B} \cup (\mathsf{proper} \times Class_{Tm_B})$, then $\text{lower}_{Tm_B}(f) = 1$. As a consequence, it must be that $\text{lower}_{Tm_{AB}}(f) = 1$.

- For all $\text{type}_{Tm_{AB}}(f) \in (\mathsf{nullable} \times Class_{Tm_{AB}})$, it holds that $\text{lower}_{Tm_{AB}}(f) = 0$.

  Use the fact that $\text{type}_{Tm_{AB}}(f) = \text{type}_{Tm_A}(f)$ or $\text{type}_{Tm_{AB}}(f) = \text{type}_{Tm_B}(f)$.

  If $\text{type}_{Tm_{AB}}(f) = \text{type}_{Tm_A}(f)$, then $\text{type}_{Tm_{AB}}(f) \in (\mathsf{nullable} \times Class_{Tm_{AB}})$ only when $\text{type}_{Tm_A}(f) \in (\mathsf{nullable} \times Class_{Tm_A})$.

  Then if $\text{type}_{Tm_A}(f) \in (\mathsf{nullable} \times Class_{Tm_A})$, then $\text{lower}_{Tm_A}(f) = 0$. As a consequence, it must be that $\text{lower}_{Tm_{AB}}(f) = 0$.

  If $\text{type}_{Tm_{AB}}(f) = \text{type}_{Tm_B}(f)$, then $\text{type}_{Tm_{AB}}(f) \in (\mathsf{nullable} \times Class_{Tm_{AB}})$ only when $\text{type}_{Tm_B}(f) \in (\mathsf{nullable} \times Class_{Tm_B})$.

  Then if $\text{type}_{Tm_B}(f) \in (\mathsf{nullable} \times Class_{Tm_{AB}})$, then $\text{lower}_{Tm_B}(f) = 0$. As a consequence, it must be that $\text{lower}_{Tm_{AB}}(f) = 0$.

- For all $\text{type}_{Tm_{AB}}(f) \notin Container_{Tm_{AB}}$, it holds that $\text{upper}_{Tm_{AB}}(f) = 1$.

  Use the fact that $\text{type}_{Tm_{AB}}(f) = \text{type}_{Tm_A}(f)$ or $\text{type}_{Tm_{AB}}(f) = \text{type}_{Tm_B}(f)$.

  If $\text{type}_{Tm_{AB}}(f) = \text{type}_{Tm_A}(f)$, then $\text{type}_{Tm_{AB}}(f) \notin Container_{Tm_{AB}}$ only when $\text{type}_{Tm_A}(f) \notin Container_{Tm_A}$.

  Then if $\text{type}_{Tm_A}(f) \notin Container_{Tm_A}$, then $\text{upper}_{Tm_A}(f) = 1$. As a consequence, must also be $\text{upper}_{Tm_{AB}}(f) = 1$.

  If $\text{type}_{Tm_{AB}}(f) = \text{type}_{Tm_B}(f)$, then $\text{type}_{Tm_{AB}}(f) \notin Container_{Tm_{AB}}$ only when $\text{type}_{Tm_B}(f) \notin Container_{Tm_B}$.

  Then if $\text{type}_{Tm_B}(f) \notin Container_{Tm_B}$, then $\text{upper}_{Tm_B}(f) = 1$. As a consequence, must also be $\text{upper}_{Tm_{AB}}(f) = 1$.

- $[\mathsf{containment}, r] \in Prop_{Tm_{AB}} \wedge [\mathsf{opposite}, r, r'] \in Prop_{Tm_{AB}} \implies upper_{Tm_{AB}}(r') = 1$.

  Use the fact that $[\mathsf{containment}, r] \in Prop_{Tm_{AB}}$ means that $[\mathsf{containment}, r] \in Prop_{Tm_A}$ or $[\mathsf{containment}, r] \in Prop_{Tm_B}$

  Also use the fact that $[\mathsf{opposite}, r, r'] \in Prop_{Tm_{AB}}$ means that $[\mathsf{opposite}, r, r'] \in Prop_{Tm_A} \setminus Prop_{Tm_B}$, $[\mathsf{opposite}, r, r'] \in Prop_{Tm_B} \setminus Prop_{Tm_A}$ or $\mathsf{opposite}, r, r'] \in Prop_{Tm_A} \cap Prop_{Tm_B}$.

  Based on these two facts, make a case distinction of all 6 possible cases. The case where $[\mathsf{opposite}, r, r'] \in Prop_{Tm_A} \setminus Prop_{Tm_B}$ and $[\mathsf{containment}, r] \in Prop_{Tm_B}$ is invalid, since r cannot be part of $Field_{Tm_B}$ by definition of the combination of properties. Similarly, the case $[\mathsf{opposite}, r, r'] \in Prop_{Tm_B} \setminus Prop_{Tm_A}$ and $[\mathsf{containment}, r] \in Prop_{Tm_A}$ is also invalid.

For the other cases, at least $\text{upper}_{Tm_A}(r') = 1$ or $\text{upper}_{Tm_B}(r') = 1$. The upper bound of r in the other type model may be larger than 1. By the definition of the combination of field signatures, $\text{upper}_{Tm_{AB}}(r') = 1$.

- $[\mathsf{defaultValue}, f, v] \in Prop_{Tm_{AB}} \wedge [\mathsf{defaultValue}, f, v'] \in Prop_{Tm_{AB}} \Longrightarrow v = v'$.

  Use the fact that $[\mathsf{defaultValue}, f, v] \in Prop_{Tm_{AB}}$ means that $[\mathsf{defaultValue}, f, v] \in Prop_{Tm_A} \setminus Prop_{Tm_B}$, $[\mathsf{defaultValue}, f, v] \in Prop_{Tm_B} \setminus Prop_{Tm_A}$ or $[\mathsf{defaultValue}, f, v] \in Prop_{Tm_A} \cap Prop_{Tm_B}$.

  Also use the similar fact for $[\mathsf{defaultValue}, f, v'] \in Prop_{Tm_{AB}}$.

  Now make a case distinction based on these facts. The case where $[\mathsf{defaultValue}, f, v] \in Prop_{Tm_A} \setminus Prop_{Tm_B}$ and $[\mathsf{defaultValue}, f, v'] \in Prop_{Tm_B} \setminus Prop_{Tm_A}$ is invalid, since f cannot be part of $Field_{Tm_B}$ by definition of the combination of properties. Similarly, the case $[\mathsf{defaultValue}, f, v] \in Prop_{Tm_B} \setminus Prop_{Tm_A}$ and $[\mathsf{defaultValue}, f, v'] \in Prop_{Tm_A} \setminus Prop_{Tm_B}$ is also invalid.

  In all other cases, use the fact that $v = v'$ in both Tm_A and Tm_B to show that $v = v'$ in Tm_{AB}.

- $[\mathsf{identity}, c_1, A_1] \in Prop_{Tm_{AB}} \wedge [\mathsf{identity}, c_2, A_2] \in Prop_{Tm_{AB}} \wedge\, !c_1 \sqsubseteq_{Tm_{AB}} !c_2 \Longrightarrow A_1 \subseteq A_2$.

  First establish that $!c_1 \sqsubseteq_{Tm_A} !c_2$, $!c_1 \sqsubseteq_{Tm_B} !c_2$ or $!c_1 \not\sqsubseteq_{Tm_A} !c_2 \wedge\, !c_1 \not\sqsubseteq_{Tm_B} !c_2$.

  Then use the fact that $[\mathsf{identity}, c_1, A_1] \in Prop_{Tm_{AB}}$ means that $[\mathsf{identity}, c_1, A_1] \in Prop_{Tm_A} \setminus Prop_{Tm_B}$, $[\mathsf{identity}, c_1, A_1] \in Prop_{Tm_B} \setminus Prop_{Tm_A}$ or $[\mathsf{identity}, c_1, A_1] \in Prop_{Tm_A} \cap Prop_{Tm_B}$.

  Also use the similar fact for $[\mathsf{identity}, c_2, A_2] \in Prop_{Tm_{AB}}$.

  Make a case distinction using the facts above. The case where $!c_1 \sqsubseteq_{Tm_A} !c_2$, $[\mathsf{identity}, c_1, A_1] \in Prop_{Tm_A} \setminus Prop_{Tm_B}$ and $[\mathsf{identity}, c_2, A_2] \in Prop_{Tm_B} \setminus Prop_{Tm_A}$ is invalid, since c_2 must be part of $Class_{Tm_A}$ to have $!c_1 \sqsubseteq_{Tm_A} !c_2$. Similarly, the case $!c_1 \sqsubseteq_{Tm_B} !c_2$, $[\mathsf{identity}, c_1, A_1] \in Prop_{Tm_B} \setminus Prop_{Tm_A}$ and $[\mathsf{identity}, c_2, A_2] \in Prop_{Tm_A} \setminus Prop_{Tm_B}$ is also invalid.

  Furthermore, the case where $!c_1 \sqsubseteq_{Tm_A} !c_2$ and $[\mathsf{identity}, c_1, A_1] \in Prop_{Tm_B} \setminus Prop_{Tm_A}$ is invalid, as well as $!c_1 \sqsubseteq_{Tm_B} !c_2$ and $[\mathsf{identity}, c_1, A_1] \in Prop_{Tm_A} \setminus Prop_{Tm_B}$

  Then solve the proof for all cases where $!c_1 \sqsubseteq_{Tm_A} !c_2$ or $!c_1 \sqsubseteq_{Tm_B} !c_2$. In the cases where $!c_1 \sqsubseteq_{Tm_B} !c_2$ or $!c_1 \not\sqsubseteq_{Tm_A} !c_2 \wedge\, !c_1 \not\sqsubseteq_{Tm_B} !c_2$, distinguish once more two cases: $c1 = c2$ and $c1 \neq c2$.

  In case that $c1 = c2$, show that when $!c_1 \sqsubseteq_{Tm_A} !c_2$ or $!c_1 \sqsubseteq_{Tm_B} !c_2$, it cannot be the case that $!c_1 \sqsubseteq_{Tm_{AB}} !c_2$ because of the reflexitivity of the subtype relation.

  Finally, if $c1 \neq c2$, the proof is given by assumption.

- $[\mathsf{keyset}, r, A] \in Prop_{Tm_{AB}} \wedge [\mathsf{keyset}, r, A'] \in Prop_{Tm_{AB}} \Longrightarrow A = A'$.

  Use the fact that $[\mathsf{keyset}, r, A] \in Prop_{Tm_{AB}}$ means that $[\mathsf{keyset}, r, A] \in Prop_{Tm_A} \setminus Prop_{Tm_B}$, $[\mathsf{keyset}, r, A] \in Prop_{Tm_B} \setminus Prop_{Tm_A}$ or $[\mathsf{keyset}, r, A] \in Prop_{Tm_A} \cap Prop_{Tm_B}$.

  Also use the similar fact for $[\mathsf{keyset}, r, A'] \in Prop_{Tm_{AB}}$.

  Now make a case distinction based on these facts. The case where $[\mathsf{keyset}, r, A] \in Prop_{Tm_A} \setminus Prop_{Tm_B}$ and $[\mathsf{keyset}, r, A'] \in Prop_{Tm_B} \setminus Prop_{Tm_A}$ is invalid, since r cannot be part of $Field_{Tm_B}$ by definition of the combination of properties. Similarly, the case $[\mathsf{keyset}, r, A] \in Prop_{Tm_B} \setminus Prop_{Tm_A}$ and $[\mathsf{keyset}, r, A'] \in Prop_{Tm_A} \setminus Prop_{Tm_B}$ is also invalid.

  In all other cases, use the fact that $A = A'$ in both Tm_A and Tm_B to show that $A = A'$ in Tm_{AB}.

- $[\mathsf{opposite}, r, r'] \in Prop_{Tm_{AB}} \wedge [\mathsf{opposite}, r, r''] \in Prop_{Tm_{AB}} \Longrightarrow r' = r''$.

  Use the fact that $[\mathsf{opposite}, r, r'] \in Prop_{Tm_{AB}}$ means that $[\mathsf{opposite}, r, r'] \in Prop_{Tm_A} \setminus Prop_{Tm_B}$, $[\mathsf{opposite}, r, r'] \in Prop_{Tm_B} \setminus Prop_{Tm_A}$ or $[\mathsf{opposite}, r, r'] \in Prop_{Tm_A} \cap Prop_{Tm_B}$.

  Also use the similar fact for $[\mathsf{opposite}, r, r''] \in Prop_{Tm_{AB}}$.

  Now make a case distinction based on these facts. The case where $[\mathsf{opposite}, r, r'] \in Prop_{Tm_A} \setminus Prop_{Tm_B}$ and $[\mathsf{opposite}, r, r''] \in Prop_{Tm_B} \setminus Prop_{Tm_A}$ is invalid, since r cannot be part of $Field_{Tm_B}$ by definition of the combination of properties. Similarly, the case $[\mathsf{opposite}, r, r'] \in Prop_{Tm_B} \setminus Prop_{Tm_A}$ and $[\mathsf{opposite}, r, r''] \in Prop_{Tm_A} \setminus Prop_{Tm_B}$ is also invalid.

  In all other cases, use the fact that $r' = r''$ in both Tm_A and Tm_B to show that $r' = r''$ in Tm_{AB}.

- $[\mathsf{opposite}, r, r'] \in Prop_{Tm_{AB}} \Longleftrightarrow [\mathsf{opposite}, r', r] \in Prop_{Tm_{AB}}$.

  Use the fact that $[\mathsf{opposite}, r, r'] \in Prop_{Tm_{AB}}$ means that $[\mathsf{opposite}, r, r'] \in Prop_{Tm_A} \setminus Prop_{Tm_B}$, $[\mathsf{opposite}, r, r'] \in Prop_{Tm_B} \setminus Prop_{Tm_A}$ or $[\mathsf{opposite}, r, r'] \in Prop_{Tm_A} \cap Prop_{Tm_B}$.

Show that if $[\mathsf{opposite}, r, r'] \in Prop_{Tm_A} \setminus Prop_{Tm_B}$, then $[\mathsf{opposite}, r', r] \in Prop_{Tm_A} \setminus Prop_{Tm_B}$. And therefore, $[\mathsf{opposite}, r', r] \in Prop_{Tm_{AB}}$.

Also show that if $[\mathsf{opposite}, r, r'] \in Prop_{Tm_B} \setminus Prop_{Tm_A}$, then $[\mathsf{opposite}, r', r] \in Prop_{Tm_B} \setminus Prop_{Tm_A}$. And therefore, $[\mathsf{opposite}, r', r] \in Prop_{Tm_{AB}}$.

Finally, show that if $[\mathsf{opposite}, r, r'] \in Prop_{Tm_A} \cap Prop_{Tm_B}$, then $[\mathsf{opposite}, r', r] \in Prop_{Tm_A} \cap Prop_{Tm_B}$. And therefore, $[\mathsf{opposite}, r', r] \in Prop_{Tm_{AB}}$.s

The proofs of all these individual properties complete the entire proof. □

As explained before, Theorem 4.3.11 does not take into account that the type models are supposed to be distinct except for a set of types. The following lemma is an alternation of the previous theorem, which takes this into account.

Lemma 4.3.12 (Consistency of the combination (mostly) distinct of type models)
Assume that Tm_A and Tm_B are consistent type models in the sense of Definition 3.2.11. Also, ensure that the type models are fully distinct except for a set of types T. Furthermore, assume the following properties:

- *Identifiers used for a class in Tm_A cannot be used for an enumeration type or user-defined data type in Tm_B: $\forall c \in Class_{Tm_A} : c \notin Enum_{Tm_B} \wedge c \notin UserDataType_{Tm_B}$.*
- *Identifiers used for a class in Tm_B cannot be used for an enumeration type or user-defined data type in Tm_A: $\forall c \in Class_{Tm_B} : c \notin Enum_{Tm_A} \wedge c \notin UserDataType_{Tm_A}$.*
- *Identifiers used for an enumeration type in Tm_A cannot be used for a class or user-defined data type in Tm_B: $\forall c \in Enum_{Tm_A} : c \notin Class_{Tm_B} \wedge c \notin UserDataType_{Tm_B}$.*
- *Identifiers used for an enumeration type in Tm_B cannot be used for a class or user-defined data type in Tm_A: $\forall c \in Enum_{Tm_B} : c \notin Class_{Tm_A} \wedge c \notin UserDataType_{Tm_A}$.*
- *Identifiers from Tm_A may not be in the namespace of an identifier in Tm_B: $\forall x \in Class_{Tm_A} \cup Enum_{Tm_A} \cup UserDataType_{Tm_A}; y \in Class_{Tm_B} \cup Enum_{Tm_B} \cup UserDataType_{Tm_B}$: x not in the namespace of y*
- *Identifiers from Tm_B may not be in the namespace of an identifier in Tm_A: $\forall x \in Class_{Tm_B} \cup Enum_{Tm_B} \cup UserDataType_{Tm_B}; y \in Class_{Tm_A} \cup Enum_{Tm_A} \cup UserDataType_{Tm_A}$: x not in the namespace of y*
- *The transitive closure of the inheritance relation is irreflexive: $(Inh_{Tm_A} \cup Inh_{Tm_B})^+$ is irreflexive*
- *For any superclass with an identity, the identity of the subclasses must be a superset of the identity of the superclass: $\forall c_1\ c_2\ A_1\ A_2 : [\mathsf{identity}, c_1, A_1] \in \text{prop\_combine}(Tm_A, Tm_B) \wedge [\mathsf{identity}, c_2, A_2] \in \text{prop\_combine}(Tm_A, Tm_B) \wedge c_1 \neq c_2 \wedge\ !c_1 \not\sqsubseteq_{Tm_A} !c_2 \wedge\ !c_1 \not\sqsubseteq_{Tm_B} !c_2 \wedge\ !c_1 \sqsubseteq_{\text{combine}(Tm_A, Tm_B)} !c_2 \implies A \subseteq B$*
- *For all shared classes, if Tm_A defines a identity, Tm_B should define the same identity, and vice versa: $\forall c \in Class_{Tm_A} \cap Class_{Tm_B} : [\mathsf{identity}, c, A] \in Prop_{Tm_A} \iff [\mathsf{identity}, c, A] \in Prop_{Tm_B}$.*

Then combine(Tm_A, Tm_B) *is a consistent type model in the sense of Definition 3.2.11.*

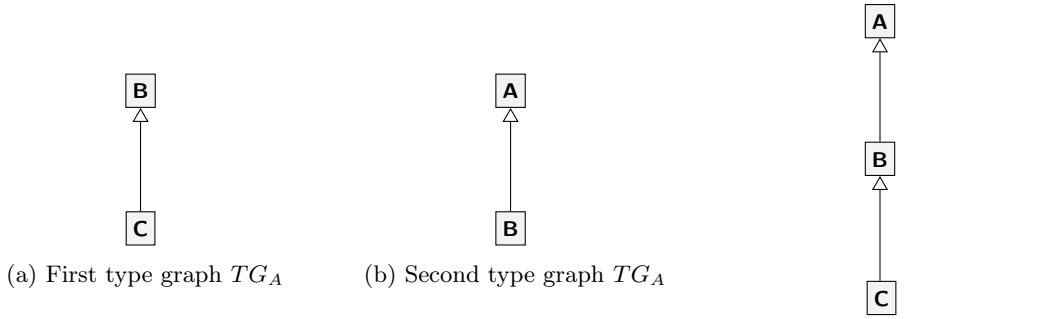
Also see `tmod_combine_merge_correct` *in* `Ecore.Type_Model_Combination`

Proof. Use Theorem 4.3.11 to show that combine(Tm_A, Tm_B) is a consistent type model. Use the assumptions given. Some assumptions of Theorem 4.3.11 become irrelevant because Tm_A and Tm_B are mostly distinct. □

Finally, the concept of compatibility between two type models is defined.

Definition 4.3.13 (Compatibility of type models)
Assume type models Tm_A and Tm_B. We say that Tm_A is compatible with Tm_B if combine(Tm_A, Tm_B) *is a consistent type model in the sense of Definition 3.2.11.*

The notion of compatibility will be used later as a way to denote type models that can be combined with other type models without loss of consistency.



(a) First type graph TG_A (b) Second type graph TG_A (c) Combined type graph TG_{AB}

Figure 4.8: Combination of the inheritance relation

4.3.2 Combining type graphs

The structure of Figure 4.5 shows that the type graphs TG_A and TG_B are combined into one type graph TG_{AB}. This section provides the definition of this combination and its corresponding theorems. Please note that the definitions presented here, just like the previous section, are as generic as possible, and do not actively take into account that TG_A and TG_B are mostly distinct. This bit of information is added later as part of a theorem and proof.

Definition 4.3.14 (Combination function on type graphs)
combine *is a binary function on two type graphs which combines two type graphs into one type graph. It is defined as follows:*

$$\begin{aligned}\text{combine}(TG_A, TG_B) = \langle & NT = NT_{TG_A} \cup NT_{TG_B} \\ & ET = ET_{TG_A} \cup ET_{TG_B} \\ & \sqsubseteq = (\sqsubseteq_{TG_A} \cup \sqsubseteq_{TG_B})^+ \\ & abs = (abs_{TG_A} \setminus NT_{TG_B}) \cup (abs_{TG_B} \setminus NT_{TG_A}) \cup (abs_{TG_A} \cap abs_{TG_B}) \\ & \text{mult} = \text{mult\_combine}(TG_A, TG_B) \\ & contains = contains_{TG_A} \cup contains_{TG_B} \rangle\end{aligned}$$

In which mult_combine *is given as part of Definition 4.3.15.*

Also see `tg_combine` *in* `GROOVE.Type_Graph_Combination`

Intuitively, the presented definition makes sense. In order to combine two (type) graphs, the nodes and edges of the graph need to be merged. The definition accurately describes this behaviour.

To preserve the correctness of the inheritance relation, the inheritance relation is merged and the transitive closure it taken. This is to ensure that the inheritance relation is correct and contains all subtypes. An example of this is given in Figure 4.8. The inheritance relation of the first type graph, TG_A is $((\mathsf{B},\mathsf{B}),(\mathsf{C},\mathsf{C}),(\mathsf{C},\mathsf{B}))$. The inheritance relation of the second type graph, TG_B is $((\mathsf{A},\mathsf{A}),(\mathsf{B},\mathsf{B}),(\mathsf{B},\mathsf{A}))$. Taking the union of these relations is not enough to get the correct inheritance relation for the combination of TG_A and TG_B, as the inheritance relation is transitive. If the union is taken, the new relation would become $((\mathsf{B},\mathsf{B}),(\mathsf{C},\mathsf{C}),(\mathsf{C},\mathsf{B}),(\mathsf{A},\mathsf{A}),(\mathsf{B},\mathsf{A}))$. This is not enough, as C is also a subtype of A in the combination (see Figure 4.8c). Taking the transitive closure of the union solves this, which results in $((\mathsf{B},\mathsf{B}),(\mathsf{C},\mathsf{C}),(\mathsf{C},\mathsf{B}),(\mathsf{A},\mathsf{A}),(\mathsf{B},\mathsf{A}),(\mathsf{C},\mathsf{A}))$.

Abstract node types are merged in a similar way to the abstract property in type models (Definition 4.3.4). For node types that are only present in one of the type graphs, their abstract property is preserved. For node types that are present in both type graphs, the abstract property is only preserved if the node type is abstract in both graphs. Intuitively, this makes sense, as instances of a class can only appear if the node type is not abstract. When the node type was not abstract in one of the type graphs, making it abstract within the combination would make all instances of the class invalid.

Containment edges are just merged, meaning that if an edge is a containment edge in one of the graphs, it is a containment edge in the combination. This behaviour makes sense from a practical standpoint. When applying models, it is undesired that ownership over a node might get lost after combining two graphs.

The multiplicity function is merged using a new function, which is discussed in the next definition.

Definition 4.3.15 (Combination function for multiplicity pairs)
mult_combine(TG_A, TG_B) *is a partial function on two type graphs which returns a new function* $ET_{TG_{AB}} \Rightarrow (\mathbb{M} \times \mathbb{M})$*. It is defined as follows:*

$$\text{mult\_combine}(TG_A, TG_B, e) = \begin{cases} (\max(l^A_{in}, l^B_{in}) .. \min(u^A_{in}, u^B_{in}), \max(l^A_{out}, l^B_{out}) .. \min(u^A_{out}, u^B_{out})) & \text{if } e \in ET_{TG_A} \cap ET_{TG_B} \\ \text{mult}_{TG_A}(e) & \text{if } e \in ET_{TG_A} \setminus ET_{TG_B} \\ \text{mult}_{TG_B}(e) & \text{if } e \in ET_{TG_B} \setminus ET_{TG_A} \end{cases}$$

where

$$l^A_{in} .. u^A_{in} = \text{in}(\text{mult}_{TG_A}(e)) \qquad l^A_{out} .. u^A_{out} = \text{out}(\text{mult}_{TG_A}(e))$$
$$l^B_{in} .. u^B_{in} = \text{in}(\text{mult}_{TG_B}(e)) \qquad l^B_{out} .. u^B_{out} = \text{out}(\text{mult}_{TG_B}(e))$$

Also see `tg_combine_mult` *in* `GROOVE.Type_Graph_Combination`

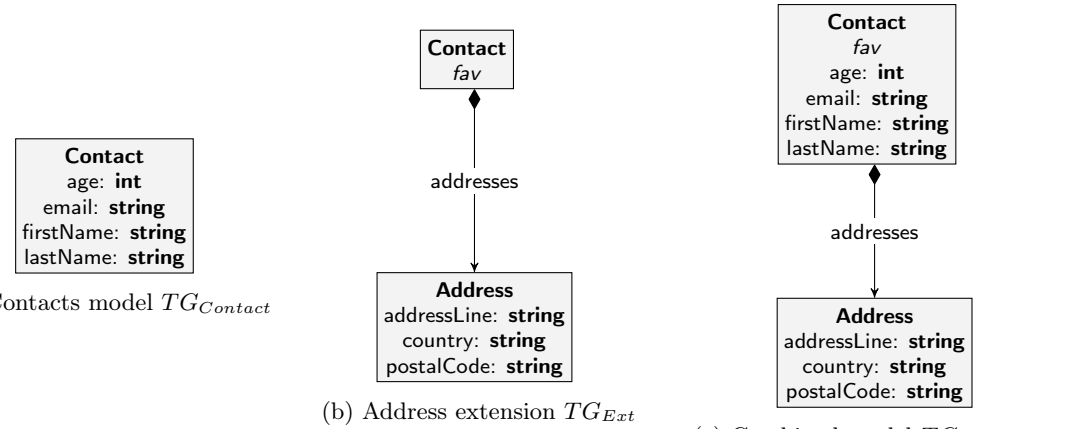
Although the presented function looks quite complicated, it is similar to the way multiplicities are handled in type models (see Definition 4.3.2), the only difference being that there are now two multiplicities, an incoming and outgoing multiplicity. In the case that an edge e is shared across TG_A and TG_B, the incoming and outgoing multiplicities are merged. For each of these multiplicities, the maximum of the corresponding lower bounds is taken, and the minimum of the corresponding upper bounds. For an edge e that only occurs in once of the type graphs, the multiplicity pair is copied.

With all definitions in place, it is possible to provide a more significant example. Suppose the model of a straightforward contacts list. It consists of Contacts of which the name, age and email address can be stored. The formal definition of the model of such a contacts list could be as follows:

$$\begin{aligned} TG_{Contact} = \langle \quad & NT = \{\mathsf{Contact}, \mathsf{int}, \mathsf{string}\} \\ & ET = \{(\mathsf{Contact}, \mathsf{age}, \mathsf{int}), (\mathsf{Contact}, \mathsf{email}, \mathsf{string}), \\ & \qquad (\mathsf{Contact}, \mathsf{firstName}, \mathsf{string}), (\mathsf{Contact}, \mathsf{lastName}, \mathsf{string})\} \\ & \sqsubseteq = \{(\mathsf{Contact}, \mathsf{Contact}), (\mathsf{int}, \mathsf{int}), (\mathsf{string}, \mathsf{string})\} \\ & abs = \{\} \\ & \text{mult} = \big\{\big((\mathsf{Contact}, \mathsf{age}, \mathsf{int}), (0..*, 0..1)\big), \\ & \qquad \big((\mathsf{Contact}, \mathsf{email}, \mathsf{string}), (0..*, 0..1)\big), \\ & \qquad \big(\mathsf{Contact}, \mathsf{firstName}, \mathsf{string}), (0..*, 0..1)\big), \\ & \qquad \big(\mathsf{Contact}, \mathsf{lastName}, \mathsf{string}), (0..*, 0..1)\big)\big\} \\ & contains = \{\} \\ \rangle & \end{aligned}$$

An visual representation of $TG_{Contact}$ is included as Figure 4.9a. Now, assume a model that represents an extension to this application, adding support for adding Addresses. Furthermore, it adds the possibility to select favourite Contacts. This extension could formally be defined as:

$$\begin{aligned} TG_{Ext} = \langle \quad & NT = \{\mathsf{Address}, \mathsf{Contact}, \mathsf{int}, \mathsf{string}\} \\ & ET = \{(\mathsf{Contact}, \mathsf{fav}, \mathsf{Contact}), (\mathsf{Contact}, \mathsf{addresses}, \mathsf{Address}), \\ & \qquad (\mathsf{Address}, \mathsf{addressLine}, \mathsf{string}), (\mathsf{Address}, \mathsf{country}, \mathsf{string}), \\ & \qquad (\mathsf{Address}, \mathsf{postalCode}, \mathsf{string})\} \\ & \sqsubseteq = \{(\mathsf{Address}, \mathsf{Address}), (\mathsf{Contact}, \mathsf{Contact}), (\mathsf{int}, \mathsf{int}), (\mathsf{string}, \mathsf{string})\} \\ & abs = \{\} \\ & \text{mult} = \big\{\big((\mathsf{Contact}, \mathsf{fav}, \mathsf{Contact}), (0..1, 0..1)\big), \\ & \qquad \big((\mathsf{Contact}, \mathsf{addresses}, \mathsf{Address}), (1..1, 1..4)\big), \\ & \qquad \big((\mathsf{Address}, \mathsf{addressLine}, \mathsf{string}), (0..*, 0..1)\big), \\ & \qquad \big(\mathsf{Address}, \mathsf{country}, \mathsf{string}), (0..*, 0..1)\big), \\ & \qquad \big(\mathsf{Address}, \mathsf{postalCode}, \mathsf{string}), (0..*, 0..1)\big)\big\} \\ & contains = \{(\mathsf{Contact}, \mathsf{addresses}, \mathsf{Address})\} \\ \rangle & \end{aligned}$$



(a) Contacts model $TG_{Contact}$

(b) Address extension TG_{Ext}

(c) Combined model $TG_{ContactExt}$

Figure 4.9: Example of the combination of type graphs

The visual representation of the extension is included as Figure 4.9b. Please note that fav is modelled as a flag in this model. In the visual notation, syntactic sugar is added to flags by writing them inside the node, in an italic font. Now, using Definition 4.3.14, it is possible to combine these graphs into one model. This will yield the following graph:

$$
\begin{aligned}
TG_{ContactExt} = \langle \quad & NT = \{\mathsf{Address}, \mathsf{Contact}, \mathsf{int}, \mathsf{string}\} \\
& ET = \{(\mathsf{Contact}, \mathsf{age}, \mathsf{int}), (\mathsf{Contact}, \mathsf{email}, \mathsf{string}), \\
& \quad (\mathsf{Contact}, \mathsf{firstName}, \mathsf{string}), (\mathsf{Contact}, \mathsf{lastName}, \mathsf{string}), \\
& \quad (\mathsf{Contact}, \mathsf{fav}, \mathsf{Contact}), (\mathsf{Contact}, \mathsf{addresses}, \mathsf{Address}), \\
& \quad (\mathsf{Address}, \mathsf{addressLine}, \mathsf{string}), (\mathsf{Address}, \mathsf{country}, \mathsf{string}), \\
& \quad (\mathsf{Address}, \mathsf{postalCode}, \mathsf{string})\} \\
& \sqsubseteq = \{(\mathsf{Address}, \mathsf{Address}), (\mathsf{Contact}, \mathsf{Contact}), (\mathsf{int}, \mathsf{int}), (\mathsf{string}, \mathsf{string})\} \\
& abs = \{\} \\
& \mathrm{mult} = \big\{\big((\mathsf{Contact}, \mathsf{age}, \mathsf{int}), (0..*, 0..1)\big), \\
& \quad \big((\mathsf{Contact}, \mathsf{email}, \mathsf{string}), (0..*, 0..1)\big), \\
& \quad \big(\mathsf{Contact}, \mathsf{firstName}, \mathsf{string}), (0..*, 0..1)\big), \\
& \quad \big(\mathsf{Contact}, \mathsf{lastName}, \mathsf{string}), (0..*, 0..1)\big), \\
& \quad \big((\mathsf{Contact}, \mathsf{fav}, \mathsf{Contact}), (0..1, 0..1)\big), \\
& \quad \big((\mathsf{Contact}, \mathsf{addresses}, \mathsf{Address}), (1..1, 1..4)\big), \\
& \quad \big((\mathsf{Address}, \mathsf{addressLine}, \mathsf{string}), (0..*, 0..1)\big), \\
& \quad \big(\mathsf{Address}, \mathsf{country}, \mathsf{string}), (0..*, 0..1)\big), \\
& \quad \big(\mathsf{Address}, \mathsf{postalCode}, \mathsf{string}), (0..*, 0..1)\big)\big\} \\
& contains = \{(\mathsf{Contact}, \mathsf{addresses}, \mathsf{Address})\} \\
\rangle &
\end{aligned}
$$

A visual representation of this combined graph is included as Figure 4.9c. The example perfectly shows why the combination of type graphs is useful: It allows for building larger graphs out of smaller building blocks. This behaviour is the exact goal of this definition within the transformation framework.

Although the definitions of the combination of type graphs are given, no mathematical properties or theorems are defined yet. Some mathematical properties hold for the combination of type graphs, that will be presented in the following theorems.

Theorem 4.3.16 (Commutativity of the combination of type graphs)
Assume that TG_A and TG_B are type graphs, then the combine *function is commutative:*

$$\mathrm{combine}(TG_A, TG_B) = \mathrm{combine}(TG_B, TG_A)$$

Also see `tg_combine_commute` *in* `GROOVE.Type_Graph_Combination`

Theorem 4.3.17 (Associativity of the combination of type graphs)
Assume that TG_A, TG_B and TG_C are type graphs, then the combine *function is associative:*

$$\text{combine}(\text{combine}(TG_A, TG_B), TG_C) = \text{combine}(TG_A, \text{combine}(TG_B, TG_C))$$

Also see `tg_combine_assoc` *in* `GROOVE.Type_Graph_Combination`

Theorem 4.3.18 (Idempotence of the combination of type graphs)
Assume that TG_A is a type graph and that it is valid in the sense of Definition 3.3.5. Then the following property holds:

$$\text{combine}(TG_A, TG_A) = TG_A$$

Also see `tg_combine_idemp_alt` *in* `GROOVE.Type_Graph_Combination`

These properties follow directly from Definition 4.3.14, but the corresponding proofs will not be included here. It should be noted that these properties are indeed proven correct as part of this thesis, and the corresponding proofs are validated within Isabelle.

Besides these properties, the combination of type graphs also has an identity element. The empty type graph represents this identity element, but it needs to be defined first:

Definition 4.3.19 (Empty type graph)
Let TG_ϵ be the empty type graph. TG_ϵ is defined as:

$$\begin{aligned} TG_\epsilon = \langle & NT = \{\} \\ & ET = \{\} \\ & \sqsubseteq = \{\} \\ & abs = \{\} \\ & \text{mult} = undefined \\ & contains = \{\}\rangle \end{aligned}$$

Theorem 4.3.20 (Correctness of the empty type graph)
The empty type graph, TG_ϵ, is valid with respect to Definition 3.3.5.

Also see `tg_empty_correct` *in* `GROOVE.Type_Graph`

The proof for the correctness of the empty type graph is trivial. Still, a validated version of this proof can be found within the Isabelle theories of this thesis.

As mentioned earlier, the empty type graph acts as an identity element when combining two type graphs. The following theorem specifies this behaviour.

Theorem 4.3.21 (Identity of the combination of type graphs)
Assume that TG_A is a type graph and that it is valid in the sense of Definition 3.3.5. Then TG_ϵ acts as an identity element in the combination function:

$$\text{combine}(TG_\epsilon, TG_A) = TG_A$$

Also see `tg_combine_identity_alt` *in* `GROOVE.Type_Graph_Combination`

Once more, the proof of this theorem follows directly from the definition. Therefore, the corresponding proof will not be included here, but a validated version can be found within the Isabelle theories of this thesis.

Just like type models, the final desired property for the combination of type graphs is a correctness property. Theorem 4.3.22 defines the theorem under which the combination of type graphs is a valid type graph. Please note that this theorem is a generic theorem, which does not take into account that the type graphs are mostly distinct.

Theorem 4.3.22 (Validity of the combination of type graphs)
Assume that TG_A and TG_B are valid type graphs in the sense of Definition 3.3.5. Furthermore, assume the following properties:

- *For all edges in TG_A that share the same label, ensure that there is no possible confusion of edge types because of a new subtype introduced at the source of the edge:* $\forall (s_1, l, t_1) \in ET_{TG_A} \wedge (s_2, l, t_2) \in ET_{TG_A}$: $\big((s_1, s_2) \in (\sqsubseteq_{TG_A} \cup \sqsubseteq_{TG_B})^+ \setminus \sqsubseteq_{TG_A} \vee\ (s_2, s_1) \in (\sqsubseteq_{TG_A} \cup \sqsubseteq_{TG_B})^+ \setminus \sqsubseteq_{TG_A}\big) \wedge$
$\big((t_1, t_2) \in (\sqsubseteq_{TG_A} \cup \sqsubseteq_{TG_B})^+ \vee (t_2, t_1) \in (\sqsubseteq_{TG_A} \cup \sqsubseteq_{TG_B})^+\big) \implies s_1 = s_2 \ \wedge\ t_1 = t_2$.
- *For all edges in TG_A that share the same label, ensure that there is no possible confusion of edge types because of a new subtype introduced at the target of the edge:* $\forall (s_1, l, t_1) \in ET_{TG_A} \wedge (s_2, l, t_2) \in ET_{TG_A}$: $\big((s_1, s_2) \in (\sqsubseteq_{TG_A} \cup \sqsubseteq_{TG_B})^+ \vee (s_2, s_1) \in (\sqsubseteq_{TG_A} \cup \sqsubseteq_{TG_B})^+\big) \wedge$
$\big((t_1, t_2) \in (\sqsubseteq_{TG_A} \cup \sqsubseteq_{TG_B})^+ \setminus \sqsubseteq_{TG_A} \vee\ (t_2, t_1) \in (\sqsubseteq_{TG_A} \cup \sqsubseteq_{TG_B})^+ \setminus \sqsubseteq_{TG_A}\big)$
$\implies s_1 = s_2 \ \wedge\ t_1 = t_2$.
- *For all edges in TG_B that share the same label, ensure that there is no possible confusion of edge types because of a new subtype introduced at the source of the edge:* $\forall (s_1, l, t_1) \in ET_{TG_B} \wedge (s_2, l, t_2) \in ET_{TG_B}$: $\big((s_1, s_2) \in (\sqsubseteq_{TG_A} \cup \sqsubseteq_{TG_B})^+ \setminus \sqsubseteq_{TG_B} \vee\ (s_2, s_1) \in (\sqsubseteq_{TG_A} \cup \sqsubseteq_{TG_B})^+ \setminus \sqsubseteq_{TG_B}\big) \wedge$
$\big((t_1, t_2) \in (\sqsubseteq_{TG_A} \cup \sqsubseteq_{TG_B})^+ \vee (t_2, t_1) \in (\sqsubseteq_{TG_A} \cup \sqsubseteq_{TG_B})^+\big) \implies s_1 = s_2 \ \wedge\ t_1 = t_2$.
- *For all edges in TG_B that share the same label, ensure that there is no possible confusion of edge types because of a new subtype introduced at the target of the edge:* $\forall (s_1, l, t_1) \in ET_{TG_B} \wedge (s_2, l, t_2) \in ET_{TG_B}$: $\big((s_1, s_2) \in (\sqsubseteq_{TG_A} \cup \sqsubseteq_{TG_B})^+ \vee (s_2, s_1) \in (\sqsubseteq_{TG_A} \cup \sqsubseteq_{TG_B})^+\big) \wedge$
$\big((t_1, t_2) \in (\sqsubseteq_{TG_A} \cup \sqsubseteq_{TG_B})^+ \setminus \sqsubseteq_{TG_B} \vee\ (t_2, t_1) \in (\sqsubseteq_{TG_A} \cup \sqsubseteq_{TG_B})^+ \setminus \sqsubseteq_{TG_B}\big)$
$\implies s_1 = s_2 \ \wedge\ t_1 = t_2$.
- *Ensure that there is no possible confusion of edge types between an edge from Tm_A and an edge from Tm_B:* $\forall (s_1, l, t_1) \in ET_{TG_A} \wedge (s_2, l, t_2) \in ET_{TG_B}$:
$\big((s_1, s_2) \in (\sqsubseteq_{TG_A} \cup \sqsubseteq_{TG_B})^+ \vee (s_2, s_1) \in (\sqsubseteq_{TG_A} \cup \sqsubseteq_{TG_B})^+\big) \wedge$
$\big((t_1, t_2) \in (\sqsubseteq_{TG_A} \cup \sqsubseteq_{TG_B})^+ \vee (t_2, t_1) \in (\sqsubseteq_{TG_A} \cup \sqsubseteq_{TG_B})^+\big)$
$\implies s_1 = s_2 \ \wedge\ t_1 = t_2$.
- *For all shared edges, ensure the combination of multiplicity pairs is a valid multiplicity pair:* $\forall e \in ET_{TG_A} \cap ET_{TG_B}$: mult_combine$(TG_A, TG_B, e) \in (\mathbb{M} \times \mathbb{M})$.
- *The transitive closure of the combine inheritance relation is antisymmetric:* $(\sqsubseteq_{TG_A} \cup \sqsubseteq_{TG_B})^+$ *is antisymmetric.*

Then combine(TG_A, TG_B) *is a valid type graph in the sense of Definition 3.3.5*

Also see `tg_combine_correct` *in* `GROOVE.Type_Graph_Combination`

Proof. To proof that combine(TG_A, TG_B) is a valid type model, it needs to be shown that combine(TG_A, TG_B) gives rise to a valid structure for a type graph and that Definition 3.3.5 holds. For readability, define TG_{AB} to be combine(TG_A, TG_B).

Structural properties

- All elements of $NT_{TG_{AB}}$ are elements of $Lab_t \cup Lab_{prim}$.

  Follows from $NT_{TG_A} \subseteq Lab_t \cup Lab_{prim}$ and $NT_{TG_B} \subseteq Lab_t \cup Lab_{prim}$.
- All elements of $ET_{TG_{AB}}$ are elements of $NT_{TG_{AB}} \times (Lab_e \cup Lab_f) \times NT_{TG_{AB}}$.

  Follows from $ET_{TG_A} \subseteq (NT_{TG_A} \times (Lab_e \cup Lab_f) \times NT_{TG_A})$ and $ET_{TG_B} \subseteq (NT_{TG_B} \times (Lab_e \cup Lab_f) \times NT_{TG_B})$. To complete the proof, use $NT_{TG_{AB}} = NT_{TG_A} \cup NT_{TG_B}$.
- All elements of $\sqsubseteq$ are elements of $NT_{TG_{AB}} \times NT_{TG_{AB}}$.

  Follows from $\sqsubseteq_{TG_A} \subseteq (NT_{TG_A} \times NT_{TG_A})$ and $\sqsubseteq_{TG_B} \subseteq (NT_{TG_B} \times NT_{TG_B})$. To complete the proof, use $NT_{TG_{AB}} = NT_{TG_A} \cup NT_{TG_B}$.
- All elements of $abs_{TG_{AB}}$ are elements of $NT_{TG_{AB}}$.

  Follows from $abs_{TG_A} \subseteq NT_{TG_A}$ and $abs_{TG_B} \subseteq NT_{TG_B}$. To complete the proof, use $NT_{TG_{AB}} = NT_{TG_A} \cup NT_{TG_B}$.
- For each edge e, $\text{mult}_{TG_{AB}}(e)$ must be an element of $\mathbb{M} \times \mathbb{M}$.

  It holds that $e \in ET_{TG_A} \setminus ET_{TG_B}$, $e \in ET_{TG_B} \setminus ET_{TG_A}$ or $e \in ET_{TG_A} \cap ET_{TG_B}$.

  Perform the proof by a case distinction based on this fact. In case of $e \in ET_{TG_A} \cap ET_{TG_B}$, use the assumption to complete the proof.
- All elements of $contains_{TG_{AB}}$ are elements of $ET_{TG_{AB}}$.

  Follows from $contains_{TG_A} \subseteq ET_{TG_A}$ and $contains_{TG_B} \subseteq ET_{TG_B}$. To complete the proof, use $ET_{TG_{AB}} = ET_{TG_A} \cup ET_{TG_B}$.

Properties for validity

- $\forall (s_1, l, t_1) \in ET_{TG_{AB}} \wedge (s_2, l, t_2) \in ET_{TG_{AB}} : \big((s_1, s_2) \in \sqsubseteq_{TG_{AB}} \vee (s_2, s_1) \in \sqsubseteq_{TG_{AB}}\big) \wedge \big((t_1, t_2) \in \sqsubseteq_{TG_{AB}} \vee (t_2, t_1) \in \sqsubseteq_{TG_{AB}}\big) \implies s_1 = s_2 \wedge t_1 = t_2$.

  Establish that $(s_1, l, t_1) \in ET_{TG_A}$ or $(s_1, l, t_1) \in ET_{TG_B}$. Also establish that $(s_2, l, t_2) \in ET_{TG_A}$ or $(s_2, l, t_2) \in ET_{TG_B}$.

  Perform a case distinction on these facts. If $(s_1, l, t_1) \in ET_{TG_A}$ and $(s_2, l, t_2) \in ET_{TG_B}$, then proof by the corresponding assumption. The case $(s_1, l, t_1) \in ET_{TG_B}$ and $(s_2, l, t_2) \in ET_{TG_A}$ is proven by the same assumption.

  If $(s_1, l, t_1) \in ET_{TG_A}$ and $(s_2, l, t_2) \in ET_{TG_A}$, then check if $\big((s_1, s_2) \in \sqsubseteq_{TG_A} \vee (s_2, s_1) \in \sqsubseteq_{TG_A}\big) \wedge \big((t_1, t_2) \in \sqsubseteq_{TG_A} \vee (t_2, t_1) \in \sqsubseteq_{TG_A}\big)$. If this is the case, solve using the properties of TG_A, when this is not the case, prove the statement using the corresponding assumptions.

  If $(s_1, l, t_1) \in ET_{TG_B}$ and $(s_2, l, t_2) \in ET_{TG_B}$, then check if $\big((s_1, s_2) \in \sqsubseteq_{TG_B} \vee (s_2, s_1) \in \sqsubseteq_{TG_B}\big) \wedge \big((t_1, t_2) \in \sqsubseteq_{TG_B} \vee (t_2, t_1) \in \sqsubseteq_{TG_B}\big)$. If this is the case, solve using the properties of TG_B, when this is not the case, prove the statement using the corresponding assumptions.

- $\forall (s, l, t) \in ET_{TG_{AB}} : l \in Lab_f \implies s = t$.

  Establish that $(s, l, t) \in ET_{TG_A}$ or $(s, l, t) \in ET_{TG_B}$. Then if $l \in Lab_f$, show that $s = t$.

- $\sqsubseteq_{TG_{AB}}$ is a partial order.

  First show that $(\sqsubseteq_{TG_A} \cup \sqsubseteq_{TG_B})^+$ is reflexive. Since $\sqsubseteq_{TG_A}$ is reflexive and $\sqsubseteq_{TG_B}$ is reflexive, $(\sqsubseteq_{TG_A} \cup \sqsubseteq_{TG_B})^+$ is reflexive as well.

  Then show that $(\sqsubseteq_{TG_A} \cup \sqsubseteq_{TG_B})^+$ is transitive. This is easily proven using the definition of the transitive closure.

  Finally, have that $(\sqsubseteq_{TG_A} \cup \sqsubseteq_{TG_B})^+$ is antisymmetric by assumption.

- $\forall e \in contains_{TG_{AB}} : \text{in}(\text{mult}_{TG_{AB}}(e)) = (0, 1) \vee \text{in}(\text{mult}_{TG_{AB}}(e)) = (1, 1)$.

  Use the fact that $e \in contains_{TG_A}$ or $e \in contains_{TG_B}$. Then have that incoming multiplicity is valid in Tm_A or Tm_B. Use the assumption that the combined multiplicities are valid and the definition of mult_combine(TG_A, TG_B, e) to show that the statement holds.

The proofs of all these individual properties completes the proof. □

As explained before, Theorem 4.3.22 does not take into account that the type graphs are supposed to be distinct except for a set of node types. The following lemma is an alternation of the previous theorem, which takes this into account.

Lemma 4.3.23 (Validity of the combination (mostly) distinct of type graphs)
Assume that TG_A and TG_B are valid type graphs in the sense of Definition 3.3.5. Also, assume that TG_A and TG_B are fully distinct except for a shared set of node types N. Furthermore, assume the following properties:

- *For all edges in TG_A that share the same label, ensure that there is no possible confusion of edge types because of a new subtype introduced at the source of the edge:* $\forall (s_1, l, t_1) \in ET_{TG_A} \wedge (s_2, l, t_2) \in ET_{TG_A} : \big((s_1, s_2) \in (\sqsubseteq_{TG_A} \cup \sqsubseteq_{TG_B})^+ \setminus \sqsubseteq_{TG_A} \vee (s_2, s_1) \in (\sqsubseteq_{TG_A} \cup \sqsubseteq_{TG_B})^+ \setminus \sqsubseteq_{TG_A}\big) \wedge \big((t_1, t_2) \in (\sqsubseteq_{TG_A} \cup \sqsubseteq_{TG_B})^+ \vee (t_2, t_1) \in (\sqsubseteq_{TG_A} \cup \sqsubseteq_{TG_B})^+\big) \implies s_1 = s_2 \wedge t_1 = t_2$.
- *For all edges in TG_A that share the same label, ensure that there is no possible confusion of edge types because of a new subtype introduced at the target of the edge:* $\forall (s_1, l, t_1) \in ET_{TG_A} \wedge (s_2, l, t_2) \in ET_{TG_A} : \big((s_1, s_2) \in (\sqsubseteq_{TG_A} \cup \sqsubseteq_{TG_B})^+ \vee (s_2, s_1) \in (\sqsubseteq_{TG_A} \cup \sqsubseteq_{TG_B})^+\big) \wedge \big((t_1, t_2) \in (\sqsubseteq_{TG_A} \cup \sqsubseteq_{TG_B})^+ \setminus \sqsubseteq_{TG_A} \vee (t_2, t_1) \in (\sqsubseteq_{TG_A} \cup \sqsubseteq_{TG_B})^+ \setminus \sqsubseteq_{TG_A}\big) \implies s_1 = s_2 \wedge t_1 = t_2$.
- *For all edges in TG_B that share the same label, ensure that there is no possible confusion of edge types because of a new subtype introduced at the source of the edge:* $\forall (s_1, l, t_1) \in ET_{TG_B} \wedge (s_2, l, t_2) \in ET_{TG_B} : \big((s_1, s_2) \in (\sqsubseteq_{TG_A} \cup \sqsubseteq_{TG_B})^+ \setminus \sqsubseteq_{TG_B} \vee (s_2, s_1) \in (\sqsubseteq_{TG_A} \cup \sqsubseteq_{TG_B})^+ \setminus \sqsubseteq_{TG_B}\big) \wedge \big((t_1, t_2) \in (\sqsubseteq_{TG_A} \cup \sqsubseteq_{TG_B})^+ \vee (t_2, t_1) \in (\sqsubseteq_{TG_A} \cup \sqsubseteq_{TG_B})^+\big) \implies s_1 = s_2 \wedge t_1 = t_2$.
- *For all edges in TG_B that share the same label, ensure that there is no possible confusion of edge types because of a new subtype introduced at the target of the edge:* $\forall (s_1, l, t_1) \in ET_{TG_B} \wedge (s_2, l, t_2) \in ET_{TG_B} : \big((s_1, s_2) \in (\sqsubseteq_{TG_A} \cup \sqsubseteq_{TG_B})^+ \vee (s_2, s_1) \in (\sqsubseteq_{TG_A} \cup \sqsubseteq_{TG_B})^+\big) \wedge \big((t_1, t_2) \in (\sqsubseteq_{TG_A} \cup \sqsubseteq_{TG_B})^+ \setminus \sqsubseteq_{TG_B} \vee (t_2, t_1) \in (\sqsubseteq_{TG_A} \cup \sqsubseteq_{TG_B})^+ \setminus \sqsubseteq_{TG_B}\big) \implies s_1 = s_2 \wedge t_1 = t_2$.

- *Ensure that there is no possible confusion of edge types between an edge from Tm_A and an edge from Tm_B:* $\forall (s_1, l, t_1) \in ET_{TG_A} \wedge (s_2, l, t_2) \in ET_{TG_B}:$
$\big((s_1, s_2) \in (\sqsubseteq_{TG_A} \cup \sqsubseteq_{TG_B})^+ \vee (s_2, s_1) \in (\sqsubseteq_{TG_A} \cup \sqsubseteq_{TG_B})^+\big) \wedge$
$\big((t_1, t_2) \in (\sqsubseteq_{TG_A} \cup \sqsubseteq_{TG_B})^+ \vee (t_2, t_1) \in (\sqsubseteq_{TG_A} \cup \sqsubseteq_{TG_B})^+\big)$
$\implies s_1 = s_2 \ \wedge \ t_1 = t_2.$
- *The transitive closure of the combine inheritance relation is antisymmetric:* $(\sqsubseteq_{TG_A} \cup \sqsubseteq_{TG_B})^+$ *is antisymmetric.*

Then combine(TG_A, TG_B) *is a valid type graph in the sense of Definition 3.3.5.*

Also see `tg_combine_merge_correct` *in* `GROOVE.Type_Graph_Combination`

Proof. Use Theorem 4.3.22 to show that combine(TG_A, TG_B) is a valid type graph. Use the assumptions given. The assumption for the correctness of the multiplicity pair for shared edges has become irrelevant because there are no shared edges. □

Finally, the concept of compatibility between two type graphs is defined.

Definition 4.3.24 (Compatibility of type graphs)
Assume type graphs TG_A and TG_B. We say that TG_A is compatible with TG_B if combine(TG_A, TG_B) *is a valid type graph in the sense of Definition 3.3.5.*

The notion of compatibility will be used later as a way to denote type graphs that can be combined with other type graphs without loss of validity.

4.3.3 Combining transformation functions

The previous sections discussed the combination of type models and type graphs. In this section, the combination of transformation functions between type models and type graphs is discussed. This combination is the last key element shown in Figure 4.5. If it is possible to combine f_A and f_B into $f_A \sqcup f_B$, then it is possible to build transformation functions between type models and type graphs iteratively.

Before it is possible to define a definition for the combination of two transformation functions, it is essential to define what functions are considered to be transformation functions.

Definition 4.3.25 (Transformation function from a type model to a type graph)
Let f be a function from type models to type graphs, Tm be a type model and TG the corresponding type graph. f is a transformation function iff:

- *f projects Tm onto TG:* $f(Tm) = TG$;
- *After combination with another type model, f preserves the node types:*
$\forall Tm_x : NT_{f(Tm)} \subseteq NT_{f(\text{combine}(Tm, Tm_x))}$;
- *After combination with another type model, f preserves the edge types:*
$\forall Tm_x : ET_{f(Tm)} \subseteq ET_{f(\text{combine}(Tm, Tm_x))}$;
- *After combination with another type model, f preserves the inheritance relation:*
$\forall Tm_x : \sqsubseteq_{f(Tm)} \subseteq \sqsubseteq_{f(\text{combine}(Tm, Tm_x))}$;
- *After combination with another type model, f preserves the abstract node types:*
$\forall Tm_x : abs_{f(Tm)} \subseteq abs_{f(\text{combine}(Tm, Tm_x))}$;
- *For all edges in the projected type graph, f preserves the multiplicity if the type model is combined with another type model:*
$\forall Tm_x : \forall e \in ET_{f(Tm)} : \text{mult}_{f(Tm)}(e) = \text{mult}_{f(\text{combine}(Tm, Tm_x))}(e)$;
- *After combination with another type model, f preserves the containment edges:*
$\forall Tm_x : contains_{f(Tm)} \subseteq contains_{f(\text{combine}(Tm, Tm_x))}$.

Also see `tg_combine_mapping_function` *in* `Ecore-GROOVE-Mapping.Type_Model_Graph_Mapping`

As expected, a transformation must project some type model Tm to its corresponding type graph TG. Furthermore, it has to preserve properties of the projection, even after Tm is combined with some other type model. The rationale behind these properties is that after combining Tm with some other type model, there must still be a way to transform the elements that originated from Tm. If that is possible, it is possible to use the transformation function as the basis for the combined transformation function, which can transform the combined type model to a combined type graph.

The following definition will describe how two transformation functions from type models to type graphs can be combined into a new transformation function, which projects the combination of two type models onto the combination of the two corresponding type graphs.

Definition 4.3.26 (Combination of transformation functions from a type model to a type graph)
Let f_A and f_B be a transformation functions in the sense of Definition 4.3.25. f_A projects a type model Tm_A onto type graph TG_A. f_B projects a type model Tm_B onto type graph TG_B. Then the combination of f_A and f_B is defined as:

$$
\begin{aligned}
f_A \sqcup f_B(Tm) = \langle \quad & NT = \{n \mid n \in NT_{f_A(Tm)} \wedge n \in NT_{TG_A}\} \cup \{n \mid n \in NT_{f_B(Tm)} \wedge n \in NT_{TG_B}\} \\
& ET = \{e \mid e \in ET_{f_A(Tm)} \wedge e \in ET_{TG_A}\} \cup \{e \mid e \in ET_{f_B(Tm)} \wedge e \in ET_{TG_B}\} \\
& \sqsubseteq = (\{i \mid i \in \sqsubseteq_{f_A(Tm)} \wedge i \in \sqsubseteq_{TG_A}\} \cup \{i \mid i \in \sqsubseteq_{f_B(Tm)} \wedge i \in \sqsubseteq_{TG_B}\})^+ \\
& abs = (\{n \mid n \in abs_{f_A(Tm)} \wedge n \in abs_{TG_A}\} \setminus \{n \mid n \in NT_{f_B(Tm)} \wedge n \in NT_{TG_B}\}) \cup \\
& \qquad (\{n \mid n \in abs_{f_B(Tm)} \wedge n \in abs_{TG_B}\} \setminus \{n \mid n \in NT_{f_A(Tm)} \wedge n \in NT_{TG_A}\}) \cup \\
& \qquad (\{n \mid n \in abs_{f_A(Tm)} \wedge n \in abs_{TG_A}\} \cap \{n \mid n \in abs_{f_B(Tm)} \wedge n \in abs_{TG_B}\}) \\
& \text{mult} = \text{mult\_mapping}(f_A, TG_A, f_B, TG_B, Tm) \\
& contains = \{e \mid e \in contains_{f_A(Tm)} \wedge e \in contains_{TG_A}\} \cup \\
& \qquad \{e \mid e \in contains_{f_B(Tm)} \wedge e \in contains_{TG_B}\} \\
\rangle &
\end{aligned}
$$

In which mult_mapping *is given as part of Definition 4.3.27.*

Also see `tg_combine_mapping` *in* `Ecore-GROOVE-Mapping.Type_Model_Graph_Mapping`

The definitions for the combination of transformation functions from a type model to a type graph looks quite complex, but a careful reader will find that this definition is an alternation of Definition 4.3.14. Intuitively, this is what is expected from this definition, as the combination of the transformation functions should be able to transform the combination of two type models to the combination of the two corresponding type graphs, as visually represented in Figure 4.5.

Unsurprisingly, the definition of the multiplicity function of $f_A \sqcup f_B$ is very similar to Definition 4.3.15.

Definition 4.3.27 (Combination of the multiplicity function for two transformation functions)
$\text{mult\_mapping}(f_A, TG_A, f_B, TG_B, Tm)$ *is a partial function on two transformation functions f_A and f_B, their corresponding projections TG_A and TG_B and a type model Tm which returns a new function $ET_{f(Tm)} \Rightarrow (\mathbb{M} \times \mathbb{M})$. It is defined as follows:*

$$
\text{mult\_mapping}(f_A, TG_A, f_B, TG_B, Tm, e) = \begin{cases}
m & \text{if } e \in \{e \mid e \in ET_{f_A(Tm)} \wedge e \in ET_{TG_A}\} \cap \{e \mid e \in ET_{f_B(Tm)} \wedge e \in ET_{TG_B}\} \\
\text{mult}_{f_A(Tm)}(e) & \text{if } e \in \{e \mid e \in ET_{f_A(Tm)} \wedge e \in ET_{TG_A}\} \setminus \{e \mid e \in ET_{f_B(Tm)} \wedge e \in ET_{TG_B}\} \\
\text{mult}_{f_B(Tm)}(e) & \text{if } e \in \{e \mid e \in ET_{f_B(Tm)} \wedge e \in ET_{TG_B}\} \setminus \{e \mid e \in ET_{f_A(Tm)} \wedge e \in ET_{TG_A}\}
\end{cases}
$$

where

$$
m = (\max(l^A_{in}, l^B_{in}) .. \min(u^A_{in}, u^B_{in}), \max(l^A_{out}, l^B_{out}) .. \min(u^A_{out}, u^B_{out}))
$$

and

$$
\begin{aligned}
l^A_{in} .. u^A_{in} &= \text{in}(\text{mult}_{f_A(Tm)}(e)) & \qquad l^A_{out} .. u^A_{out} &= \text{out}(\text{mult}_{f_A(Tm)}(e)) \\
l^B_{in} .. u^B_{in} &= \text{in}(\text{mult}_{f_B(Tm)}(e)) & \qquad l^B_{out} .. u^B_{out} &= \text{out}(\text{mult}_{f_B(Tm)}(e))
\end{aligned}
$$

With these definitions in place, it is possible to provide the necessary theorems for the correctness of the combined function $f_A \sqcup f_B$.

Theorem 4.3.28 (The projection of a combined transformation function from a type model to a type graph)
Let f_A and f_B be a transformation functions in the sense of Definition 4.3.25. f_A projects a type model Tm_A onto type graph TG_A. f_B projects a type model Tm_B onto type graph TG_B. Then the combination of f_A and f_B, $f_A \sqcup f_B$ projects $\text{combine}(Tm_A, Tm_B)$ *onto* $\text{combine}(TG_A, TG_B)$*, so:*

$$
f_A \sqcup f_B(\text{combine}(Tm_A, Tm_B)) = \text{combine}(TG_A, TG_B)
$$

Also see `tg_combine_mapping_correct` *in* `Ecore-GROOVE-Mapping.Type_Model_Graph_Mapping`

Proof. The corresponding proof follows directly from Definition 4.3.26 as well as Definition 4.3.25. Since the individual transformation functions f_A and f_B preserve the elements of their type graphs when the type model is combined with another one, we can establish that the definition of $f_A \sqcup f_B$ is equal to the definition of combine(TG_A, TG_B). Therefore, $f_A \sqcup f_B(\text{combine}(Tm_A, Tm_B)) = \text{combine}(TG_A, TG_B)$. □

Although the presented theorem is a large step towards being able to build transformation functions from type models to type graphs iteratively, there is still one key element missing. It should be formally argued that $f_A \sqcup f_B$ is once again an transformation function in the sense of Definition 4.3.25. If this is formally argued, it becomes possible to easily combine $f_A \sqcup f_B$ with yet another transformation function. The following theorem states this property.

Theorem 4.3.29 (A combined transformation function from a type model to a type graph is a transformation function)
Let f_A and f_B be a transformation functions in the sense of Definition 4.3.25. f_A projects a type model Tm_A onto type graph TG_A. f_B projects a type model Tm_B onto type graph TG_B. Then the combination of f_A and f_B, $f_A \sqcup f_B$ is again a transformation function in the sense of Definition 4.3.25 which projects combine(Tm_A, Tm_B) *onto* combine(TG_A, TG_B).

Also see `tg_combine_mapping_function_correct` *in* `Ecore-GROOVE-Mapping.Type_Model_Graph_Mapping`

Proof. Use Definition 4.3.25. Since the individual transformation functions f_A and f_B preserve the elements of their type graphs when the type model is combined with another one, we can establish that the definition of $f_A \sqcup f_B$ will also preserve these elements. This can be shown using the commutativity and associativity of the combination of type models, see Theorem 4.3.5 and Theorem 4.3.6 respectively. □

This last theorem completes the recursive behaviour of combining transformation functions and therefore allows for building transformation functions from type models to type graphs iteratively.

The definitions and theorems that are presented so far only work in one direction: for transforming type models into type graphs. As visually shown in Figure 4.5, it must also be possible to transform type graphs back into type models. The definitions and theorems needed for this transformation are similar and will be presented in the remaining part of this section.

Definition 4.3.30 (Transformation function from a type graph to a type model)
Let f be a function from type graphs to type models, TG be a type graph and Tm the corresponding type model. f is a transformation function iff:

- *f projects TG onto Tm: $f(TG) = Tm$;*
- *After combination with another type graph, f preserves the classes:* $\forall TG_x : Class_{f(TG)} \subseteq Class_{f(\text{combine}(TG, TG_x))}$;
- *After combination with another type graph, f preserves the enumerations:* $\forall TG_x : Enum_{f(TG)} \subseteq Enum_{f(\text{combine}(TG, TG_x))}$;
- *After combination with another type graph, f preserves the user-defined data types:* $\forall TG_x : UserDataType_{f(TG)} \subseteq UserDataType_{f(\text{combine}(TG, TG_x))}$;
- *After combination with another type graph, f preserves the fields:* $\forall TG_x : Field_{f(TG)} \subseteq Field_{f(\text{combine}(TG, TG_x))}$;
- *For all fields in the projected type model, f preserves the field signature if the type graph is combined with another type graph:* $\forall TG_x : \forall s \in Field_{f(Tm)} : \text{FieldSig}_{f(TG)}(s) = \text{FieldSig}_{f(\text{combine}(TG, TG_x))}(s)$;
- *After combination with another type graph, f preserves the enumeration values:* $\forall TG_x : EnumValue_{f(TG)} \subseteq EnumValue_{f(\text{combine}(TG, TG_x))}$;
- *After combination with another type graph, f preserves the inheritance relation:* $\forall TG_x : Inh_{f(TG)} \subseteq Inh_{f(\text{combine}(TG, TG_x))}$;
- *After combination with another type graph, f preserves the model properties:* $\forall TG_x : Prop_{f(TG)} \subseteq Prop_{f(\text{combine}(TG, TG_x))}$;
- *After combination with another type graph, f preserves the constants:* $\forall TG_x : Constant_{f(TG)} \subseteq Constant_{f(\text{combine}(TG, TG_x))}$;

- *For all constants in the projected type model, f preserves the constant types if the type graph is combined with another type graph:*
  $\forall TG_x : \forall c \in Constant_{f(Tm)} : \text{ConstType}_{f(TG)}(c) = \text{ConstType}_{f(\text{combine}(TG,TG_x))}(c)$.

Also see `tmod_combine_mapping_function` *in* `Ecore-GROOVE-Mapping.Type_Model_Graph_Mapping`

Just like Definition 4.3.25, the definition of transformation functions from a type graph to a type model preserves all elements if the type graph is combined with another type graph. This will once more be the key to having the property of iterative building of transformation functions.

The following definition will describe how two transformation functions from type graphs to type models can be combined into a new transformation function, which projects the combination of two type graphs onto the combination of the two corresponding type models.

Definition 4.3.31 (Combination of transformation functions from a type graph to a type model)
Let f_A and f_B be a transformation functions in the sense of Definition 4.3.30. f_A projects a type graph TG_A onto type model Tm_A. f_B projects a type graph TG_B onto type model Tm_B. Then the combination of f_A and f_B is defined as:

$$
\begin{aligned}
f_A \sqcup f_B(TG) = \langle \quad Class &= \{c \mid c \in Class_{f_A(TG)} \wedge c \in Class_{Tm_A}\} \cup \\
&\quad \{c \mid c \in Class_{f_B(TG)} \wedge c \in Class_{Tm_B}\} \\
Enum &= \{e \mid e \in Enum_{f_A(TG)} \wedge e \in Enum_{Tm_A}\} \cup \\
&\quad \{e \mid e \in Enum_{f_B(TG)} \wedge e \in Enum_{Tm_B}\} \\
UserDataType &= \{u \mid u \in UserDataType_{f_A(TG)} \wedge u \in UserDataType_{Tm_A}\} \cup \\
&\quad \{u \mid u \in UserDataType_{f_B(TG)} \wedge u \in UserDataType_{Tm_B}\} \\
Field &= \{d \mid d \in Field_{f_A(TG)} \wedge d \in Field_{Tm_A}\} \cup \\
&\quad \{d \mid d \in Field_{f_B(TG)} \wedge d \in Field_{Tm_B}\} \\
\text{FieldSig} &= \text{fieldsig\_mapping}(f_A, Tm_A, f_B, Tm_B, TG) \\
EnumValue &= \{v \mid v \in EnumValue_{f_A(TG)} \wedge v \in EnumValue_{Tm_A}\} \cup \\
&\quad \{v \mid v \in EnumValue_{f_B(TG)} \wedge v \in EnumValue_{Tm_B}\} \\
Inh &= \{i \mid i \in Inh_{f_A(TG)} \wedge i \in Inh_{Tm_A}\} \cup \\
&\quad \{i \mid i \in Inh_{f_B(TG)} \wedge i \in Inh_{Tm_B}\} \\
Prop &= \text{prop\_mapping}(f_A, Tm_A, f_B, Tm_B, TG) \\
Constant &= \{c \mid c \in Constant_{f_A(TG)} \wedge c \in Constant_{Tm_A}\} \cup \\
&\quad \{c \mid c \in Constant_{f_B(TG)} \wedge c \in Constant_{Tm_B}\} \\
\text{ConstType} &= \text{consttype\_mapping}(f_A, Tm_A, f_B, Tm_B, TG) \\
\rangle &
\end{aligned}
$$

In which fieldsig_mapping *is given as part of Definition 4.3.32,* prop_mapping *as part of Definition 4.3.34, and* consttype_mapping *as part of Definition 4.3.33.*

Also see `tmod_combine_mapping` *in* `Ecore-GROOVE-Mapping.Type_Model_Graph_Mapping`

As expected, the definition for the combination of transformation functions from a type graph to a type model is an alternation of Definition 4.3.1. This alternation will once more allow the combined transformation function to project the combination of the type graphs to the combination of the corresponding type models.

The following two definitions will provide the remaining functions, which will closely follow their counterparts from Section 4.3.1.

Definition 4.3.32 (Combination of the field signature function for two transformation functions)
fieldsig_mapping$(f_A, Tm_A, f_B, Tm_B, TG)$ *is a partial function on two transformation functions f_A and f_B, their corresponding projections Tm_A and Tm_B and a type model TG which returns a new function*

$Field_{f_A \sqcup f_B(TG)} \Rightarrow (Type_{f_A \sqcup f_B(TG)} \times \mathbb{M})$. *It is defined as follows:*

$$\text{fieldsig\_combine}(f_A, Tm_A, f_B, Tm_B, TG, d) = \begin{cases} s & \text{if } d \in \{d \mid d \in Field_{f_A(TG)} \wedge d \in Field_{Tm_A}\} \cap \\ & \quad \{d \mid d \in Field_{f_B(TG)} \wedge d \in Field_{Tm_B}\} \wedge \\ & \quad \text{type}_{f_A(TG)}(d) = \text{type}_{f_B(TG)}(d) \\ \text{FieldSig}_{f_A(TG)}(d) & \text{if } d \in \{d \mid d \in Field_{f_A(TG)} \wedge d \in Field_{Tm_A}\} \setminus \\ & \quad \{d \mid d \in Field_{f_B(TG)} \wedge d \in Field_{Tm_B}\} \\ \text{FieldSig}_{f_B(TG)}(d) & \text{if } d \in \{d \mid d \in Field_{f_B(TG)} \wedge d \in Field_{Tm_B}\} \setminus \\ & \quad \{d \mid d \in Field_{f_A(TG)} \wedge d \in Field_{Tm_A}\} \end{cases}$$

where

$$s = \Big(\text{type}_{f_A(TG)}(d), \Big(\max\Big(\text{lower}(\text{FieldSig}_{f_A(TG)}(d)), \text{lower}(\text{FieldSig}_{f_B(TG)}(d))\Big) .. \min\Big(\text{upper}(\text{FieldSig}_{f_A(TG)}(d)), \text{upper}(\text{FieldSig}_{f_B(TG)}(d))\Big)\Big)\Big)$$

Also see `tmod_combine_fieldsig_mapping` *in* `Ecore-GROOVE-Mapping.Type_Model_Graph_Mapping`

Definition 4.3.33 (Combination of the constant type function for two transformation functions) $\text{consttype\_mapping}(f_A, Tm_A, f_B, Tm_B, TG)$ *is a partial function on two transformation functions* f_A *and* f_B*, their corresponding projections* Tm_A *and* Tm_B *and a type model* TG *which returns a new function* $Constant_{f_A \sqcup f_B(TG)} \Rightarrow Type_{f_A \sqcup f_B(TG)}$*. It is defined as follows:*

$$\text{consttype\_mapping}(f_A, Tm_A, f_B, Tm_B, TG, c) = \begin{cases} \text{ConstType}_{f_A(TG)}(c) & \text{if } c \in \{c \mid c \in Constant_{f_A(TG)} \wedge c \in Constant_{Tm_A}\} \cap \\ & \quad \{c \mid c \in Constant_{f_B(TG)} \wedge c \in Constant_{Tm_B}\} \wedge \\ & \quad \text{ConstType}_{f_A(TG)}(c) = \text{ConstType}_{f_B(TG)}(c) \\ \text{ConstType}_{f_A(TG)}(c) & \text{if } c \in \{c \mid c \in Constant_{f_A(TG)} \wedge c \in Constant_{Tm_A}\} \setminus \\ & \quad \{c \mid c \in Constant_{f_B(TG)} \wedge c \in Constant_{Tm_B}\} \\ \text{ConstType}_{f_B(TG)}(c) & \text{if } c \in \{c \mid c \in Constant_{f_B(TG)} \wedge c \in Constant_{Tm_B}\} \setminus \\ & \quad \{c \mid c \in Constant_{f_A(TG)} \wedge c \in Constant_{Tm_A}\} \end{cases}$$

Also see `tmod_combine_const_type_mapping` *in* `Ecore-GROOVE-Mapping.Type_Model_Graph_Mapping`

Now that the definitions for the field signature function and the constant type function are defined, the only remaining definition is that of the combination of properties, which is given in the next definition.

Definition 4.3.34 (Combination of model properties when combining two transformation functions) $\text{prop\_mapping}(f_A, Tm_A, f_B, Tm_B, TG)$ *is a set depending on two transformation functions* f_A *and* f_B*, their corresponding projections* Tm_A *and* Tm_B *and a type model* TG*. The set is defined as a subset of* $Prop_{Tm_A} \cup Prop_{Tm_B}$*. The contents of the set are then defined as follows:*

For abstract *properties:*

$$\frac{[\textsf{abstract}, c] \in \{p \mid p \in Prop_{f_A(TG)} \wedge p \in Prop_{Tm_A}\} \qquad c \notin \{c \mid c \in Class_{f_B(TG)} \wedge c \in Class_{Tm_B}\}}{[\textsf{abstract}, c] \in \text{prop\_mapping}(f_A, Tm_A, f_B, Tm_B, TG)}$$

$$\frac{[\textsf{abstract}, c] \in \{p \mid p \in Prop_{f_B(TG)} \wedge p \in Prop_{Tm_B}\} \qquad c \notin \{c \mid c \in Class_{f_A(TG)} \wedge c \in Class_{Tm_A}\}}{[\textsf{abstract}, c] \in \text{prop\_mapping}(f_A, Tm_A, f_B, Tm_B, TG)}$$

$$\frac{\begin{array}{c}[\textsf{abstract}, c] \in \{p \mid p \in Prop_{f_A(TG)} \wedge p \in Prop_{Tm_A}\} \\ [\textsf{abstract}, c] \in \{p \mid p \in Prop_{f_B(TG)} \wedge p \in Prop_{Tm_B}\}\end{array}}{[\textsf{abstract}, c] \in \text{prop\_mapping}(f_A, Tm_A, f_B, Tm_B, TG)}$$

For containment *properties:*

$$\frac{[\mathsf{containment}, r] \in \{p \mid p \in Prop_{f_A(TG)} \wedge p \in Prop_{Tm_A}\}}{[\mathsf{containment}, r] \in \text{prop\_mapping}(f_A, Tm_A, f_B, Tm_B, TG)}$$

$$\frac{[\mathsf{containment}, r] \in \{p \mid p \in Prop_{f_B(TG)} \wedge p \in Prop_{Tm_B}\}}{[\mathsf{containment}, r] \in \text{prop\_mapping}(f_A, Tm_A, f_B, Tm_B, TG)}$$

For defaultValue *properties:*

$$\frac{\begin{array}{c}[\mathsf{defaultValue}, f, v] \in \{p \mid p \in Prop_{f_A(TG)} \wedge p \in Prop_{Tm_A}\} \\ f \notin \{d \mid d \in Field_{f_B(TG)} \wedge d \in Field_{Tm_B}\}\end{array}}{[\mathsf{defaultValue}, f, v] \in \text{prop\_mapping}(f_A, Tm_A, f_B, Tm_B, TG)}$$

$$\frac{\begin{array}{c}[\mathsf{defaultValue}, f, v] \in \{p \mid p \in Prop_{f_B(TG)} \wedge p \in Prop_{Tm_B}\} \\ f \notin \{d \mid d \in Field_{f_A(TG)} \wedge d \in Field_{Tm_A}\}\end{array}}{[\mathsf{defaultValue}, f, v] \in \text{prop\_mapping}(f_A, Tm_A, f_B, Tm_B, TG)}$$

$$\frac{\begin{array}{c}[\mathsf{defaultValue}, f, v] \in \{p \mid p \in Prop_{f_A(TG)} \wedge p \in Prop_{Tm_A}\} \\ [\mathsf{defaultValue}, f, v] \in \{p \mid p \in Prop_{f_B(TG)} \wedge p \in Prop_{Tm_B}\}\end{array}}{[\mathsf{defaultValue}, f, v] \in \text{prop\_mapping}(f_A, Tm_A, f_B, Tm_B, TG)}$$

For identity *properties:*

$$\frac{[\mathsf{identity}, c, A] \in \{p \mid p \in Prop_{f_A(TG)} \wedge p \in Prop_{Tm_A}\} \qquad c \notin \{c \mid c \in Class_{f_B(TG)} \wedge c \in Class_{Tm_B}\}}{[\mathsf{identity}, c, A] \in \text{prop\_mapping}(f_A, Tm_A, f_B, Tm_B, TG)}$$

$$\frac{[\mathsf{identity}, c, A] \in \{p \mid p \in Prop_{f_B(TG)} \wedge p \in Prop_{Tm_B}\} \qquad c \notin \{c \mid c \in Class_{f_A(TG)} \wedge c \in Class_{Tm_A}\}}{[\mathsf{identity}, c, A] \in \text{prop\_mapping}(f_A, Tm_A, f_B, Tm_B, TG)}$$

$$\frac{\begin{array}{c}[\mathsf{identity}, c, A] \in \{p \mid p \in Prop_{f_A(TG)} \wedge p \in Prop_{Tm_A}\} \\ [\mathsf{identity}, c, A] \in \{p \mid p \in Prop_{f_B(TG)} \wedge p \in Prop_{Tm_B}\}\end{array}}{[\mathsf{identity}, c, A] \in \text{prop\_mapping}(f_A, Tm_A, f_B, Tm_B, TG)}$$

For keyset *properties:*

$$\frac{[\mathsf{keyset}, r, A] \in \{p \mid p \in Prop_{f_A(TG)} \wedge p \in Prop_{Tm_A}\} \qquad r \notin \{d \mid d \in Field_{f_B(TG)} \wedge d \in Field_{Tm_B}\}}{[\mathsf{keyset}, r, A] \in \text{prop\_mapping}(f_A, Tm_A, f_B, Tm_B, TG)}$$

$$\frac{[\mathsf{keyset}, r, A] \in \{p \mid p \in Prop_{f_B(TG)} \wedge p \in Prop_{Tm_B}\} \qquad r \notin \{d \mid d \in Field_{f_A(TG)} \wedge d \in Field_{Tm_A}\}}{[\mathsf{keyset}, r, A] \in \text{prop\_mapping}(f_A, Tm_A, f_B, Tm_B, TG)}$$

$$\frac{\begin{array}{c}[\mathsf{keyset}, r, A] \in \{p \mid p \in Prop_{f_A(TG)} \wedge p \in Prop_{Tm_A}\} \\ [\mathsf{keyset}, r, A] \in \{p \mid p \in Prop_{f_B(TG)} \wedge p \in Prop_{Tm_B}\}\end{array}}{[\mathsf{keyset}, r, A] \in \text{prop\_mapping}(f_A, Tm_A, f_B, Tm_B, TG)}$$

For opposite *properties:*

$$\frac{\begin{array}{c}[\mathsf{opposite}, r1, r2] \in \{p \mid p \in Prop_{f_A(TG)} \wedge p \in Prop_{Tm_A}\} \\ r1 \notin \{d \mid d \in Field_{f_B(TG)} \wedge d \in Field_{Tm_B}\} \qquad r2 \notin \{d \mid d \in Field_{f_B(TG)} \wedge d \in Field_{Tm_B}\}\end{array}}{[\mathsf{opposite}, r1, r2] \in \text{prop\_mapping}(f_A, Tm_A, f_B, Tm_B, TG)}$$

$$\frac{\begin{array}{c}[\mathsf{opposite}, r1, r2] \in \{p \mid p \in Prop_{f_B(TG)} \wedge p \in Prop_{Tm_B}\} \\ r1 \notin \{d \mid d \in Field_{f_A(TG)} \wedge d \in Field_{Tm_A}\} \qquad r2 \notin \{d \mid d \in Field_{f_A(TG)} \wedge d \in Field_{Tm_A}\}\end{array}}{[\mathsf{opposite}, r1, r2] \in \text{prop\_mapping}(f_A, Tm_A, f_B, Tm_B, TG)}$$

$$\frac{\begin{array}{c}[\mathsf{opposite}, r1, r2] \in \{p \mid p \in Prop_{f_A(TG)} \wedge p \in Prop_{Tm_A}\} \\ [\mathsf{opposite}, r1, r2] \in \{p \mid p \in Prop_{f_B(TG)} \wedge p \in Prop_{Tm_B}\}\end{array}}{[\mathsf{opposite}, r1, r2] \in \text{prop\_mapping}(f_A, Tm_A, f_B, Tm_B, TG)}$$

For readonly *properties:*

$$\frac{[\text{readonly}, f] \in \{p \mid p \in Prop_{f_A(TG)} \wedge p \in Prop_{Tm_A}\} \qquad f \notin \{d \mid d \in Field_{f_B(TG)} \wedge d \in Field_{Tm_B}\}}{[\text{readonly}, f] \in \text{prop\_mapping}(f_A, Tm_A, f_B, Tm_B, TG)}$$

$$\frac{[\text{readonly}, f] \in \{p \mid p \in Prop_{f_B(TG)} \wedge p \in Prop_{Tm_B}\} \qquad f \notin \{d \mid d \in Field_{f_A(TG)} \wedge d \in Field_{Tm_A}\}}{[\text{readonly}, f] \in \text{prop\_mapping}(f_A, Tm_A, f_B, Tm_B, TG)}$$

$$\frac{\begin{array}{c}[\text{readonly}, f] \in \{p \mid p \in Prop_{f_A(TG)} \wedge p \in Prop_{Tm_A}\} \\ [\text{readonly}, f] \in \{p \mid p \in Prop_{f_B(TG)} \wedge p \in Prop_{Tm_B}\}\end{array}}{[\text{readonly}, f] \in \text{prop\_mapping}(f_A, Tm_A, f_B, Tm_B, TG)}$$

Also see `tmod_combine_prop_mapping` *in* `Ecore-GROOVE-Mapping.Type_Model_Graph_Mapping`

As expected, the definition for the properties within the combination is an alternation of Definition 4.3.4.

With these definitions in place, it is possible to provide the necessary theorems for the correctness of the combined function $f_A \sqcup f_B$.

Theorem 4.3.35 (The projection of a combined transformation function from a type graph to a type model)
Let f_A and f_B be a transformation functions in the sense of Definition 4.3.30. f_A projects a type graph TG_A onto type model Tm_A. f_B projects a type graph TG_B onto type model Tm_B. Then the combination of f_A and f_B, $f_A \sqcup f_B$ projects combine(TG_A, TG_B) *onto* combine(Tm_A, Tm_B)*, so:*

$$f_A \sqcup f_B(\text{combine}(TG_A, TG_B)) = \text{combine}(Tm_A, Tm_B)$$

Also see `tmod_combine_mapping_correct` *in* `Ecore-GROOVE-Mapping.Type_Model_Graph_Mapping`

Proof. The corresponding proof follows directly from Definition 4.3.31 as well as Definition 4.3.30. Since the individual transformation functions f_A and f_B preserve the elements of their type models when the type graph is combined with another one, we can establish that the definition of $f_A \sqcup f_B$ is equal to the definition of combine(Tm_A, Tm_B). Therefore, $f_A \sqcup f_B(\text{combine}(TG_A, TG_B)) = \text{combine}(Tm_A, Tm_B)$. □

Like the combined transformation function from type models to type graphs, the combined transformation function from type graphs to type models is also a transformation function, but in the sense of Definition 4.3.30. This is stated in the following theorem.

Theorem 4.3.36 (A combined transformation function from a type graph to a type model is a transformation function)
Let f_A and f_B be a transformation functions in the sense of Definition 4.3.30. f_A projects a type graph TG_A onto type model Tm_A. f_B projects a type graph TG_B onto type model Tm_B. Then the combination of f_A and f_B, $f_A \sqcup f_B$ is again a transformation function in the sense of Definition 4.3.30 which projects combine(TG_A, TG_B) *onto* combine(Tm_A, Tm_B)*.*

Also see `tmod_combine_mapping_function_correct` *in* `Ecore-GROOVE-Mapping.Type_Model_Graph_Mapping`

Proof. Use Definition 4.3.30. Since the individual transformation functions f_A and f_B preserve the elements of their type models when the type graph is combined with another one, we can establish that the definition of $f_A \sqcup f_B$ will also preserve these elements. This can be shown using the commutativity and associativity of the combination of type graphs, see Theorem 4.3.16 and Theorem 4.3.17 respectively. □

4.4 Instance models and instance graphs

In the previous section, the structure of the framework was applied to type models and type graphs. In this section, the structure will be applied to instance models and instance graphs. Since instance models and instance graphs directly depend on type models and type graphs, some definitions will be borrowed from the previous section.

First, the general structure of the framework applied to instance models and instance graphs is discussed. Then the required definitions and theorems are given.

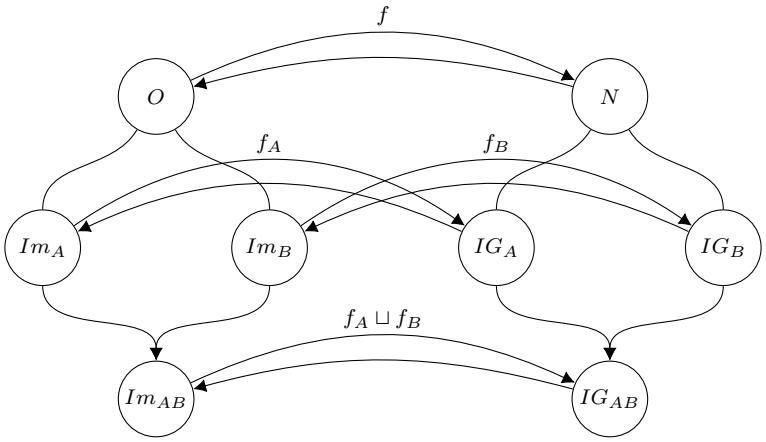


Figure 4.10: Structure for transforming between instance models and instance graphs

Figure 4.10 shows one more alternation of the structure proposed in Section 4.2. This version of the structure is applied to instance models and instance graphs. As before, instance model Im_A represents the partially build model which corresponds to instance graph IG_A under the transformation function f_A. Instance model Im_B represents the next building block to add to this model. It corresponds to instance graph IG_B under the bijective transformation function f_B.

Instance models Im_A and Im_B are entirely distinct except for a set objects O, which means $O \subseteq Object_{Im_A} \wedge O \subseteq Object_{Im_B}$. In a similar way, instance graphs IG_A and IG_B are entirely distinct except for a set of nodes N, so $N \subseteq N_{IG_A} \wedge N \subseteq N_{IG_B}$.

Instance models Im_A and Im_B are combined into instance model Im_{AB} using Definition 4.4.1. In a similar way instance graphs IG_A and IG_B are combined into instance graph IG_{AB} using Definition 4.4.15. Lemma 4.4.13 and Lemma 4.4.24 respectively show that Im_{AB} and IG_{AB} are valid. Then Definition 4.4.27 and Definition 4.4.32 can be used to merge the transformation functions f_A and f_B into $f_A \sqcup f_B$, where Theorem 4.4.29 and Theorem 4.4.30 show that $f_A \sqcup f_B$ is again a valid transformation function transforming Im_{AB} to IG_{AB}. Similarly, Theorem 4.4.37 and Theorem 4.4.38 show that the inverse function of $f_A \sqcup f_B$ is again a valid transformation function transforming IG_{AB} to Im_{AB}.

4.4.1 Combining instance models

The structure of Figure 4.10 shows that the instance models Im_A and Im_B are combined into one instance model Im_{AB}. This section provides the definition of this combination and its corresponding theorems. Please note that the definitions presented here are as generic as possible, and do not actively take into account that Im_A and Im_B are mostly distinct. This bit of information is added later as part of a theorem and proof.

Definition 4.4.1 (Combination function on type models)
combine *is a binary function on two instance models which combines two instance models into one instance model. Assume* Im_A *is an instance model typed by type model* Tm_A *and* Im_B *is an instance model typed by type model* Tm_B*, then* combine(Im_A, Im_B) *is typed by* combine(Tm_A, Tm_B) *and is defined as follows:*

$$\begin{aligned}\text{combine}(Im_A, Im_B) = \langle & Object = Object_{Im_A} \cup Object_{Im_B} \\ & \text{ObjectClass} = \text{objectclass\_combine}(Im_A, Im_B) \\ & \text{ObjectId} = \text{objectid\_combine}(Im_A, Im_B) \\ & \text{FieldValue} = \text{fieldvalue\_combine}(Im_A, Im_B) \\ & \text{DefaultValue} = \text{defaultvalue\_combine}(Im_A, Im_B)\rangle\end{aligned}$$

In which objectclass_combine *is given as part of Definition 4.4.2,* objectid_combine *as part of Definition 4.4.3,* fieldvalue_combine *as part of Definition 4.4.4 and* defaultvalue_combine *as part of Definition 4.4.5.*

Also see `imod_combine` *in* `Ecore.Instance_Model_Combination`

The combination of two instance models knows a surprisingly simple definition. This is mostly caused by the fact that an instance model only contains of a set of objects, which has some properties. The

properties of each object are specified by the different functions, which will be introduced in the following definitions.

First, the function for the combination of object classes is discussed.

Definition 4.4.2 (Combination function for object classes)
objectclass_combine *is a partial function on two instance models which returns a new function* $Object_{Im_{AB}} \Rightarrow Class_{Tm_{AB}}$. *It is defined as follows:*

$$\text{objectclass\_combine}(Im_A, Im_B, o) = \begin{cases} \text{ObjectClass}_{Im_A}(o) & \text{if } o \in Object_{Im_A} \cap Object_{Im_B} \wedge \text{ObjectClass}_{Im_A}(o) = \text{ObjectClass}_{Im_B}(o) \\ \text{ObjectClass}_{Im_A}(o) & \text{if } o \in Object_{Im_A} \setminus Object_{Im_B} \\ \text{ObjectClass}_{Im_B}(o) & \text{if } o \in Object_{Im_B} \setminus Object_{Im_A} \end{cases}$$

Also see `imod_combine_object_class` *in* `Ecore.Instance_Model_Combination`

The combination of two instance models knows a surprisingly simple definition. Because an instance model is essentially a set of objects with properties, no complex definition is needed. The properties of each object are specified by the different functions, which will be introduced in the following definitions.

First, the function of the combination of object classes is discussed.

Definition 4.4.3 (Combination function for object identifiers)
objectid_combine *is a partial function on two instance models which returns a new function* $Object_{Im_{AB}} \Rightarrow Name$. *It is defined as follows:*

$$\text{objectid\_combine}(Im_A, Im_B, o) = \begin{cases} \text{ObjectId}_{Im_A}(o) & \text{if } o \in Object_{Im_A} \cap Object_{Im_B} \wedge \text{ObjectId}_{Im_A}(o) = \text{ObjectId}_{Im_B}(o) \\ \text{ObjectId}_{Im_A}(o) & \text{if } o \in Object_{Im_A} \setminus Object_{Im_B} \\ \text{ObjectId}_{Im_B}(o) & \text{if } o \in Object_{Im_B} \setminus Object_{Im_A} \end{cases}$$

Also see `imod_combine_object_id` *in* `Ecore.Instance_Model_Combination`

As mentioned before, the definition of the combination function of object identifiers is very similar to the definition of the combination function of object classes. If an object only occurs in one of the instance models, its identifier is copied over. If an object appears in both instance models, they must have the same identifier already in order to have an identifier in the final model.

A careful reader might notice that the behaviour of the function is strange. Theoretically, it might give rise to double identities, which is undesired. As will be shown later, the combination function and theorems assume that the identities of the models are already distinct. This assumption is fair, as it is possible to redefine two instance models to have distinct identities, without loss of significance.

The following definition describes the combination of field values.

Definition 4.4.4 (Combination function for field values)
fieldvalue_combine *is a partial function on two instance models which returns a new function* $(Object_{Im_{AB}} \times Field_{Tm_{AB}}) \Rightarrow Value_{Im_{AB}}$. *It is defined as follows:*

$$\text{fieldvalue\_combine}(Im_A, Im_B, (o, f)) = \begin{cases} \text{FieldValue}_{Im_A}((o,f)) & \text{if } o \in Object_{Im_A} \cap Object_{Im_B} \wedge \\ & \quad f \in \text{fields}_{Tm_A}(\text{ObjectClass}_{Im_A}(o)) \wedge \\ & \quad f \in \text{fields}_{Tm_B}(\text{ObjectClass}_{Im_B}(o)) \wedge \\ & \quad \text{FieldValue}_{Im_A}((o,f)) = \text{FieldValue}_{Im_B}((o,f)) \\ \text{FieldValue}_{Im_A}((o,f)) & \text{if } o \in Object_{Im_A} \wedge f \in \text{fields}_{Tm_A}(\text{ObjectClass}_{Im_A}(o)) \wedge \\ & \quad (o \notin Object_{Im_B} \vee f \notin \text{fields}_{Tm_B}(\text{ObjectClass}_{Im_B}(o))) \\ \text{FieldValue}_{Im_B}((o,f)) & \text{if } o \in Object_{Im_B} \wedge f \in \text{fields}_{Tm_B}(\text{ObjectClass}_{Im_B}(o)) \wedge \\ & \quad (o \notin Object_{Im_A} \vee f \notin \text{fields}_{Tm_A}(\text{ObjectClass}_{Im_A}(o))) \end{cases}$$

Also see `imod_combine_field_value` *in* `Ecore.Instance_Model_Combination`

The definition of the combination function of field values is a lot more complicated than the previous ones. The function domain causes this complexity. Not every combination of an object and a field has a value. An object only has values for those fields that are defined for its class or superclasses.

When a value is set on one of the instance models, but not the other, the value is copied. Furthermore, if a combination of object and field is set for both instance models and the value is the same, it is also copied. Please note that equality is used here, instead of equivalence. This property is to support some mathematical properties later on. Since the transformation framework will not allow for shared fields anyhow, this will not impose problems later.

The last function that needs to be defined is the combination function for default values. It is given in the following definition.

Definition 4.4.5 (Combination function for default values)
defaultvalue_combine *is a partial function on two instance models which returns a new function* $Constant_{Tm_{AB}} \Rightarrow Value_{Im_{AB}}$. *It is defined as follows:*

$$\text{defaultvalue\_combine}(Im_A, Im_B, c) = \begin{cases} \text{DefaultValue}_{Im_A}(c) & \text{if } c \in Constant_{Tm_A} \cap Constant_{Tm_B} \wedge \\ & \quad \text{DefaultValue}_{Im_A}(c) = \text{DefaultValue}_{Im_B}(c) \\ \text{DefaultValue}_{Im_A}(c) & \text{if } c \in Constant_{Tm_A} \setminus Constant_{Tm_B} \\ \text{DefaultValue}_{Im_B}(c) & \text{if } c \in Constant_{Tm_B} \setminus Constant_{Tm_A} \end{cases}$$

Also see `imod_combine_default_value` *in* `Ecore.Instance_Model_Combination`

The definition of the combination function of default values is very similar to the combination function of constant types of type models (see Definition 4.3.3). This function gives values to constants defined on the type model level. When a constant only appears in the type model of one of the instance models, the value can be copied from that instance model. This behaviour is logical since the other instance model cannot have a value set for that constant. If a constant is set for both of the corresponding type models, the value set on the instance models must be the same. If this is the case, the value can be copied over. This behaviour is desired, as the value for a constant should not change after the combination of two instance models.

Like the last definition, equality is used here to compare the values, instead of equivalence. Once more, this has been done to support some mathematical properties later on. Since the transformation framework will not allow for shared constants anyhow, this will not impose problems later.

With all definitions in place, it is possible to provide an example. Let us return to the multi-protocol chat application example introduced in Figure 4.7 of Section 4.3.1. An instance model for Tm_{Chat} (Figure 4.7a) could have an instance of a Thread with some Messages. Formally, the instance model could look as follows:

$$\begin{aligned} Im_{Chat} = \langle \quad Object &= \{1,2,3\} \\ \text{ObjectClass} &= \{(1, .\mathsf{Thread}), (2, .\mathsf{Message}), (3, .\mathsf{Message})\} \\ \text{ObjectId} &= \{(1, \text{Thread42}), (2, \text{Message4084}), (3, \text{Message4093})\} \\ \text{FieldValue} &= \Big\{\Big((1, (.\mathsf{Thread}, \mathsf{id})), [\mathsf{string}, \text{“BLUB-E\_Thread\_01”}]\Big), \\ &\quad \Big((1, (.\mathsf{Thread}, \mathsf{proto})), [\mathsf{enum}, (.\mathsf{Protocol}, \mathsf{BLUB-E})]\Big), \\ &\quad \Big((1, (.\mathsf{Thread}, \mathsf{messages})), [\mathsf{seqof}, \langle [\mathsf{obj}, 2], [\mathsf{obj}, 3]\rangle]\Big), \\ &\quad \Big((2, (.\mathsf{Message}, \mathsf{text})), [\mathsf{string}, \text{“This is a test”}]\Big), \\ &\quad \Big((3, (.\mathsf{Message}, \mathsf{text})), [\mathsf{string}, \text{“Did you receive it?”}]\Big)\Big\} \\ \text{DefaultValue} &= \{\} \\ \rangle & \end{aligned}$$

A visual representation of this instance model is included in Figure 4.11a. Now, assume that there also exists some instance model that is typed by the extension represented by $Tm_{Extension}$. This instance model introduces a Contact instance for the Thread instance in Im_{Chat}. Formally, this instance model could be defined as follows:

$$
\begin{aligned}
Im_{Extension} = \langle \quad & Object = \{1, 4\} \\
& \text{ObjectClass} = \{(1, .\mathsf{Thread}), (4, .\mathsf{Contact})\} \\
& \text{ObjectId} = \{(1, \text{Thread42}), (4, \text{Broodkast})\} \\
& \text{FieldValue} = \Big\{ \Big(\big(1, (.\mathsf{Thread}, \mathsf{contact})\big), \big[\mathsf{obj}, 4\big] \Big), \\
& \qquad \Big(\big(4, (.\mathsf{Contact}, \mathsf{id})\big), \big[\mathsf{data}, \text{“BLUB-E\_PubKey\_a8138”}\big] \Big), \\
& \qquad \Big(\big(4, (.\mathsf{Contact}, \mathsf{name})\big), \big[\mathsf{string}, \text{“Lukas”}\big] \Big) \Big\} \\
& \text{DefaultValue} = \{\} \\
\rangle &
\end{aligned}
$$

The visual representation of $Im_{Extension}$ is included in Figure 4.11b. With these instance models formally defined, it is possible to combine them using Definition 4.4.1. This will yield the following model:

$$
\begin{aligned}
Im_{ChatExt} = \langle \quad & Object = \{1, 2, 3, 4\} \\
& \text{ObjectClass} = \{(1, .\mathsf{Thread}), (2, .\mathsf{Message}), (3, .\mathsf{Message}), (4, .\mathsf{Contact})\} \\
& \text{ObjectId} = \{(1, \text{Thread42}), (2, \text{Message4084}), (3, \text{Message4093}), (4, \text{Broodkast})\} \\
& \text{FieldValue} = \Big\{ \Big(\big(1, (.\mathsf{Thread}, \mathsf{id})\big), \big[\mathsf{string}, \text{“BLUB-E\_Thread\_01”}\big] \Big), \\
& \qquad \Big(\big(1, (.\mathsf{Thread}, \mathsf{proto})\big), \big[\mathsf{enum}, (.\mathsf{Protocol}, \mathsf{BLUB-E})\big] \Big), \\
& \qquad \Big(\big(1, (.\mathsf{Thread}, \mathsf{messages})\big), \big[\mathsf{seqof}, \langle [\mathsf{obj}, 2], [\mathsf{obj}, 3] \rangle\big] \Big), \\
& \qquad \Big(\big(1, (.\mathsf{Thread}, \mathsf{contact})\big), \big[\mathsf{obj}, 4\big] \Big), \\
& \qquad \Big(\big(2, (.\mathsf{Message}, \mathsf{text})\big), \big[\mathsf{string}, \text{“This is a test”}\big] \Big), \\
& \qquad \Big(\big(3, (.\mathsf{Message}, \mathsf{text})\big), \big[\mathsf{string}, \text{“Did you receive it?”}\big] \Big), \\
& \qquad \Big(\big(4, (.\mathsf{Contact}, \mathsf{id})\big), \big[\mathsf{data}, \text{“BLUB-E\_PubKey\_a8138”}\big] \Big), \\
& \qquad \Big(\big(4, (.\mathsf{Contact}, \mathsf{name})\big), \big[\mathsf{string}, \text{“Lukas”}\big] \Big) \Big\} \\
& \text{DefaultValue} = \{\} \\
\rangle &
\end{aligned}
$$

A visual representation of this combined model is included as Figure 4.11c. Like the example for the combination of type models, this example shows that the definition of the combination of instance models is useful. It allows to build larger models out of smaller building blocks. Furthermore, the example shows that the combination of the two instance models is typed by the combination of its corresponding type models.

Although the definitions of the combination of instance models are given, no mathematical properties or theorems are defined yet. Some mathematical properties hold for the combination of instance models, that will be presented in the following theorems.

Theorem 4.4.6 (Commutativity of the combination of instance models)
Assume that Im_A and Im_B are instance models, then the combine *function is commutative:*

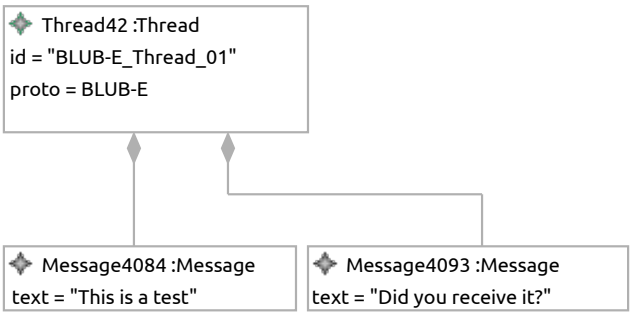
$$\text{combine}(Im_A, Im_B) = \text{combine}(Im_B, Im_A)$$

Also see `imod_combine_commute` *in* `Ecore.Instance_Model_Combination`

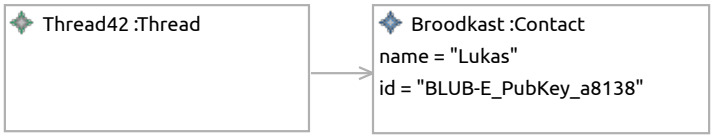
Theorem 4.4.7 (Associativity of the combination of instance models)
Assume that Im_A, Im_B and Im_C are instance models, then the combine *function is associative:*

$$\text{combine}(\text{combine}(Im_A, Im_B), Im_C) = \text{combine}(Im_A, \text{combine}(Im_B, Im_C))$$

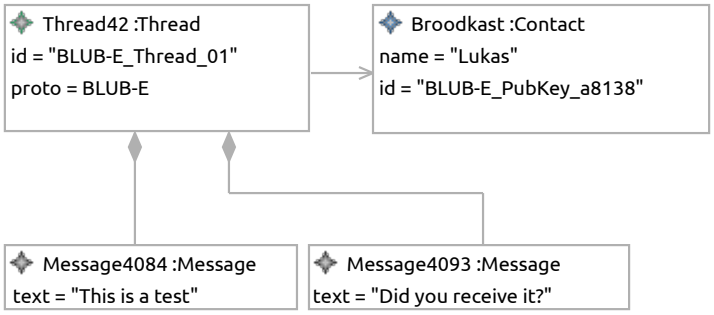
Also see `imod_combine_assoc` *in* `Ecore.Instance_Model_Combination`



(a) The chat application instance model Im_{Chat}



(b) The contact extension instance model $Im_{Extension}$



(c) The extended chat application instance model $Im_{ChatExt}$

Figure 4.11: Example of the combination of type models

Theorem 4.4.8 (Idempotence of the combination of instance models)
Assume that Im_A is an instance model and that it is valid in the sense of Definition 3.2.19. Then the following property holds:

$$\text{combine}(Im_A, Im_A) = Im_A$$

Also see `imod_combine_idemp_alt` *in* `Ecore.Instance_Model_Combination`

These properties follow directly from Definition 4.4.1, but the corresponding proofs will not be included here. It should be noted that these properties are indeed proven correct as part of this thesis, and the corresponding proofs are validated within Isabelle.

Besides these properties, the combination of instance models also has an identity element. The empty instance model represents this identity element, but it needs to be defined first:

Definition 4.4.9 (Empty instance model)
Let Im_ϵ be the empty instance model. It is typed by the empty type model Tm_ϵ. Im_ϵ is defined as:

$$\begin{aligned} Im_\epsilon = \langle & Object = \{\} \\ & \text{ObjectClass} = undefined \\ & \text{ObjectId} = undefined \\ & \text{FieldValue} = undefined \\ & \text{DefaultValue} = undefined \rangle \end{aligned}$$

Theorem 4.4.10 (Correctness of the empty type model)
The empty instance model, Im_ϵ, is valid with respect to Definition 3.2.19.

Also see `imod_empty_correct` *in* `Ecore.Instance_Model`

The proof for the correctness of the empty instance model is trivial. Still, a validated version of this proof can be found within the Isabelle theories of this thesis.

As mentioned earlier, the empty instance model acts as an identity element when combining two instance models. The following theorem specifies this behaviour.

Theorem 4.4.11 (Identity of the combination of instance models)
Assume that Im_A is an instance model and that it is valid in the sense of Definition 3.2.19. Then Im_ϵ acts as an identity element in the combination function:

$$\text{combine}(Im_\epsilon, Im_A) = Im_A$$

Also see `imod_combine_identity_alt` *in* `Ecore.Instance_Model_Combination`

Once more, the proof of this theorem follows directly from the definition. Therefore, the corresponding proof will not be included here, but a validated version can be found within the Isabelle theories of this thesis.

A final desired property for the combination of instance models is a correctness property. Theorem 4.4.12 defines the theorem under which the combination of instance models is a valid instance model. Please note that this theorem is a generic theorem, which does not take into account that the instance models are mostly distinct.

Theorem 4.4.12 (Validity of the combination of instance models)
Assume that Im_A and Im_B are valid instance models in the sense of Definition 3.2.19. Assume that Im_A is typed by type model Tm_A. Furthermore, assume that Im_B is typed by type model Tm_B. Tm_A and Tm_B are consistent by definition. Also assume that $Tm_{AB} = \text{combine}(Tm_A, Tm_B)$ is consistent in the sense of Definition 3.2.11. Finally, assume the following properties:

- *For all shared objects, the object class must be the same in both instance models:* $\forall o \in Object_{Im_A} \cap Object_{Im_B} : \text{ObjectClass}_{Im_A}(o) = \text{ObjectClass}_{Im_B}(o)$.
- *For all shared objects, the object id must be the same in both instance models:* $\forall o \in Object_{Im_A} \cap Object_{Im_B} : \text{ObjectId}_{Im_A}(o) = \text{ObjectId}_{Im_B}(o)$.
- *For all shared constants within the corresponding type graphs, the default value must be the same in both instance models:* $\forall c \in Constant_{Tm_A} \cap Constant_{Tm_B} :$ $\text{DefaultValue}_{Im_A}(c) = \text{DefaultValue}_{Im_B}(c)$.

- *The identifiers must be unique across both instance models:* $\forall o_1 \in Object_{Im_A} \setminus Object_{Im_B} \wedge o_2 \in Object_{Im_B} \setminus Object_{Im_A} : \text{ObjectId}_{Im_A}(o_1) = \text{ObjectId}_{Im_B}(o_2) \implies o_1 = o_2$.
- *If a field value is set for a combination of an object and field in both instance models, that field value must be the same in both instance models:* $\forall o \in Object_{Im_A} \cap Object_{Im_B} \wedge f \in \text{fields}_{Tm_A}(\text{ObjectClass}_{Im_A}(o)) \cap \text{fields}_{Tm_B}(\text{ObjectClass}_{Im_B}(o)) : \text{FieldValue}_{Im_A}((o,f)) = \text{FieldValue}_{Im_B}((o,f))$.
- *If an object needs a field value in the combination of* Im_A *and* Im_B*, but this field value is not set in* Im_A*, then it must be set in* Im_B*:* $\forall o \in Object_{Im_A} \wedge f \notin \text{fields}_{Tm_A}(\text{ObjectClass}_{Im_A}(o)) : f \in \text{fields}_{Tm_{AB}}(\text{ObjectClass}_{\text{combine}(Im_A,Im_B)}(o)) \implies o \in Object_{Im_B} \wedge f \in \text{fields}_{Tm_B}(\text{ObjectClass}_{Im_B}(o))$.
- *If an object needs a field value in the combination of* Im_A *and* Im_B*, but this field value is not set in* Im_B*, then it must be set in* Im_A*:* $\forall o \in Object_{Im_B} \wedge f \notin \text{fields}_{Tm_B}(\text{ObjectClass}_{Im_B}(o)) : f \in \text{fields}_{Tm_{AB}}(\text{ObjectClass}_{\text{combine}(Im_A,Im_B)}(o)) \implies o \in Object_{Im_A} \wedge f \in \text{fields}_{Tm_A}(\text{ObjectClass}_{Im_A}(o))$.
- *For field values copied from* Im_A *that are in* $ContainerValue_{Im_A}$*, the combined multiplicity must be correct:* $\forall o \in Object_{Im_A} \wedge f \in \text{fields}_{Tm_A}(\text{ObjectClass}_{Im_A}(o)) \wedge f \in Field_{Tm_B} : \text{FieldValue}_{Im_A}((o,f)) \in ContainerValue_{Im_A} \implies \text{lower}_{Tm_{AB}}(\text{FieldSig}_{Tm_{AB}}(f)) \leq |\text{FieldValue}_{Im_A}((o,f))| \wedge |\text{FieldValue}_{Im_A}((o,f))| \leq \text{upper}_{Tm_{AB}}(\text{FieldSig}_{Tm_{AB}}(f))$.
- *For field values copied from* Im_B *that are in* $ContainerValue_{Im_B}$*, the combined multiplicity must be correct:* $\forall o \in Object_{Im_B} \wedge f \in \text{fields}_{Tm_B}(\text{ObjectClass}_{Im_B}(o)) \wedge f \in Field_{Tm_A} : \text{FieldValue}_{Im_B}((o,f)) \in ContainerValue_{Im_B} \implies \text{lower}_{Tm_{AB}}(\text{FieldSig}_{Tm_{AB}}(f)) \leq |\text{FieldValue}_{Im_B}((o,f))| \wedge |\text{FieldValue}_{Im_B}((o,f))| \leq \text{upper}_{Tm_{AB}}(\text{FieldSig}_{Tm_{AB}}(f))$.
- *If there exists a containment property in* Tm_{AB}*, the satisfaction formula for containment properties must be satisfied:* $\forall o \in Object_{Im_A} \cup Object_{Im_B} : \left|\left\{\left((fo, ff), fv\right) \mid \left((fo, ff), fv\right) \in \text{FieldValue}_{\text{combine}(Im_A,Im_B)} \wedge [\mathsf{obj}, o] = fv \wedge ff \in CR_{Tm_{AB}}\right\}\right| \leq 1$
- *There may be no cycles in the containment edges of the combined instance model:* $\left\{(fo, fv) \mid \left((fo, ff), fv\right) \in \text{FieldValue}_{\text{combine}(Im_A,Im_B)} \wedge ff \in CR_{Tm_{AB}}\right\}$ *is acyclic.*
- *The identity properties must remain satisfied when combining objects from different instance models:* $\forall [\mathsf{identity}, c, A] \in Prop_{Tm_A} \wedge [\mathsf{identity}, c, A] \in Prop_{Tm_B} \wedge o_1 \in Object_{Im_A} \setminus Object_{Im_B} \wedge o_2 \in Object_{Im_B} \setminus Object_{Im_A} \wedge \text{ObjectClass}_{Im_A}(o_1) = c \wedge \text{ObjectClass}_{Im_B}(o_2) = c \wedge a \in A : \text{FieldValue}_{Im_A}((o_1,a)) \equiv_{\text{combine}(Im_A,Im_B)} \text{FieldValue}_{Im_B}((o_2,a)) \implies o_1 = o_2$.
- *The opposite properties must remain satisfied when combining objects from different instance models:* $\forall [\mathsf{opposite}, r_1, r_2] \in Prop_{Tm_A} \wedge [\mathsf{opposite}, r_1, r_2] \in Prop_{Tm_B} \wedge o_1 \in Object_{Im_A} \wedge (o_1 \notin Object_{Im_B} \vee r_1 \notin \text{fields}_{Tm_B}(\text{ObjectClass}_{Im_B}(o_1))) \wedge o_2 \in Object_{Im_B} \wedge (o_2 \notin Object_{Im_A} \vee r_2 \notin \text{fields}_{Tm_A}(\text{ObjectClass}_{Im_A}(o_2))) \implies \text{edgeCount}_{Im_A}(o_1, r_1, o_2) = \text{edgeCount}_{Im_B}(o_2, r_2, o_1)$.

Then $\text{combine}(Im_A, Im_B)$ *is a valid instance model in the sense of Definition 3.2.19*

Also see `imod_combine_correct` *in* `Ecore.Instance_Model_Combination`

Proof. To proof that $\text{combine}(Im_A, Im_B)$ is a valid instance model, it needs to be shown that $\text{combine}(Im_A, Im_B)$ gives rise to a valid structure for an instance model and that Definition 3.2.19 holds. For readability, define Im_{AB} to be $\text{combine}(Im_A, Im_B)$.

Structural properties

- For each object o, $\text{ObjectClass}_{Im_{AB}}(o)$ must be an element of $Class_{Tm_{AB}}$.

  If $o \in Object_{Im_A} \setminus Object_{Im_B}$, then $\text{ObjectClass}_{Im_{AB}}(o) \in Class_{Tm_{AB}}$.

  Similarly, if $o \in Object_{Im_B} \setminus Object_{Im_A}$, then $\text{ObjectClass}_{Im_{AB}}(o) \in Class_{Tm_{AB}}$.

  If $o \in Object_{Im_A} \cap Object_{Im_B}$, then $\text{ObjectClass}_{Im_A}(o) = \text{ObjectClass}_{Im_B}(o)$ by assumption. Therefore $\text{ObjectClass}_{Im_{AB}}(o) \in Class_{Tm_{AB}}$.
- For each object o, $\text{ObjectId}_{Im_{AB}}(o)$ must be an element of $Name$.

  If $o \in Object_{Im_A} \setminus Object_{Im_B}$, then $\text{ObjectId}_{Im_{AB}}(o) \in Name$.

  Similarly, if $o \in Object_{Im_B} \setminus Object_{Im_A}$, then $\text{ObjectId}_{Im_{AB}}(o) \in Name$.

If $o \in Object_{Im_A} \cap Object_{Im_B}$, then $\text{ObjectId}_{Im_A}(o) = \text{ObjectId}_{Im_B}(o)$ by assumption. Therefore $\text{ObjectId}_{Im_{AB}}(o) \in Name$.

- For each object o, and $f \in \text{fields}_{Tm_{AB}}(\text{ObjectClass}_{Im_{AB}}(o))$, $\text{FieldValue}_{Im_{AB}}((o, f))$ must be an element of $Value_{Im_{AB}}$.

  First, note that $Value_{Im_A} \cup Value_{Im_B} \subseteq Value_{Im_{AB}}$
  (see `imod_combine_value` in `Ecore.Instance_Model_Combination`).

  If $o \in Object_{Im_A} \setminus Object_{Im_B}$, then $f \in \text{fields}_{Tm_A}(\text{ObjectClass}_{Im_A}(o))$ or $f \notin \text{fields}_{Tm_A}(\text{ObjectClass}_{Im_A}(o))$.

  - If $f \in \text{fields}_{Tm_A}(\text{ObjectClass}_{Im_A}(o))$, then $\text{FieldValue}_{Im_{AB}}((o, f)) = \text{FieldValue}_{Im_A}((o, f))$ and, therefore, $\text{FieldValue}_{Im_{AB}}((o, f)) \in Value_{Im_{AB}}$.
  - If $f \notin \text{fields}_{Tm_A}(\text{ObjectClass}_{Im_A}(o))$, then by assumption, $f \in Object_{Im_B}$. However, $f \notin Object_{Im_B}$, so this case is invalid.

  If $o \in Object_{Im_B} \setminus Object_{Im_A}$, then $f \in \text{fields}_{Tm_B}(\text{ObjectClass}_{Im_B}(o))$ or $f \notin \text{fields}_{Tm_B}(\text{ObjectClass}_{Im_B}(o))$.

  - If $f \in \text{fields}_{Tm_B}(\text{ObjectClass}_{Im_B}(o))$, then $\text{FieldValue}_{Im_{AB}}((o, f)) = \text{FieldValue}_{Im_B}((o, f))$ and, therefore, $\text{FieldValue}_{Im_{AB}}((o, f)) \in Value_{Im_{AB}}$.
  - If $f \notin \text{fields}_{Tm_B}(\text{ObjectClass}_{Im_B}(o))$, then by assumption, $f \in Object_{Im_A}$. However, $f \notin Object_{Im_A}$, so this case is invalid.

  If $o \in Object_{Im_A} \cap Object_{Im_B}$, then $f \in \text{fields}_{Tm_A}(\text{ObjectClass}_{Im_A}(o)) \cap \text{fields}_{Tm_B}(\text{ObjectClass}_{Im_B}(o))$ or $f \notin \text{fields}_{Tm_A}(\text{ObjectClass}_{Im_A}(o))$ or $f \notin \text{fields}_{Tm_B}(\text{ObjectClass}_{Im_B}(o))$.

  - If $f \in \text{fields}_{Tm_A}(\text{ObjectClass}_{Im_A}(o)) \cap \text{fields}_{Tm_B}(\text{ObjectClass}_{Im_B}(o))$, then $\text{FieldValue}_{Im_{AB}}((o, f)) = \text{FieldValue}_{Im_A}((o, f))$ and, therefore, $\text{FieldValue}_{Im_{AB}}((o, f)) \in Value_{Im_{AB}}$.
  - If $f \notin \text{fields}_{Tm_A}(\text{ObjectClass}_{Im_A}(o))$, then by assumption, $f \in \text{fields}_{Tm_B}(\text{ObjectClass}_{Im_B}(o))$. Therefore, $\text{FieldValue}_{Im_{AB}}((o, f)) = \text{FieldValue}_{Im_B}((o, f))$ which means that $\text{FieldValue}_{Im_{AB}}((o, f)) \in Value_{Im_{AB}}$.
  - If $f \notin \text{fields}_{Tm_B}(\text{ObjectClass}_{Im_B}(o))$, then by assumption, $f \in \text{fields}_{Tm_A}(\text{ObjectClass}_{Im_A}(o))$. Therefore, $\text{FieldValue}_{Im_{AB}}((o, f)) = \text{FieldValue}_{Im_A}((o, f))$ which means that $\text{FieldValue}_{Im_{AB}}((o, f)) \in Value_{Im_{AB}}$.

- For each constant c in $Constant_{Tm_{AB}}$, $\text{DefaultValue}_{Im_{AB}}(c)$ must be an element of $Value_{Im_{AB}}$.

  First, note that $Value_{Im_A} \cup Value_{Im_B} \subseteq Value_{Im_{AB}}$
  (see `imod_combine_value` in `Ecore.Instance_Model_Combination`).

  If $c \in Constant_{Tm_A} \setminus Constant_{Tm_B}$, then $\text{DefaultValue}_{Im_{AB}}(c) \in Value_{Im_{AB}}$.

  Similarly, if $c \in Constant_{Tm_B} \setminus Constant_{Tm_A}$, then $\text{DefaultValue}_{Im_{AB}}(c) \in Value_{Im_{AB}}$.

  If $c \in Constant_{Tm_A} \cap Constant_{Tm_B}$, then $\text{DefaultValue}_{Im_A}(c) = \text{DefaultValue}_{Im_B}(c)$ by assumption. Therefore $\text{DefaultValue}_{Im_{AB}}(c) \in Value_{Im_{AB}}$.

Validity properties

- $\forall((o, f), v) \in \text{FieldValue}_{Im_{AB}} : v :_{Im} \text{type}_{Tm_{AB}}(f)$

  If $o \in Object_{Im_A} \setminus Object_{Im_B}$, then $f \in \text{fields}_{Tm_A}(\text{ObjectClass}_{Im_A}(o))$ or $f \notin \text{fields}_{Tm_A}(\text{ObjectClass}_{Im_A}(o))$.

  - If $f \in \text{fields}_{Tm_A}(\text{ObjectClass}_{Im_A}(o))$, then $\text{FieldValue}_{Im_{AB}}((o, f)) = \text{FieldValue}_{Im_A}((o, f))$. In this case $\text{type}_{Tm_{AB}}(f) = \text{type}_{Tm_A}(f)$ because types are preserved by Tm_{AB}. Therefore, $v :_{Im} \text{type}_{Tm_{AB}}(f)$.
  - If $f \notin \text{fields}_{Tm_A}(\text{ObjectClass}_{Im_A}(o))$, then by assumption, $f \in Object_{Im_B}$. However, $f \notin Object_{Im_B}$, so this case is invalid.

  If $o \in Object_{Im_B} \setminus Object_{Im_A}$, then $f \in \text{fields}_{Tm_B}(\text{ObjectClass}_{Im_B}(o))$ or $f \notin \text{fields}_{Tm_B}(\text{ObjectClass}_{Im_B}(o))$.

  - If $f \in \text{fields}_{Tm_B}(\text{ObjectClass}_{Im_B}(o))$, then $\text{FieldValue}_{Im_{AB}}((o, f)) = \text{FieldValue}_{Im_B}((o, f))$. In this case $\text{type}_{Tm_{AB}}(f) = \text{type}_{Tm_B}(f)$ because types are preserved by Tm_{AB}. Therefore, $v :_{Im} \text{type}_{Tm_{AB}}(f)$.

– If $f \notin \text{fields}_{Tm_B}(\text{ObjectClass}_{Im_B}(o))$, then by assumption, $f \in Object_{Im_A}$. However, $f \notin Object_{Im_A}$, so this case is invalid.

If $o \in Object_{Im_A} \cap Object_{Im_B}$, then
$f \in \text{fields}_{Tm_A}(\text{ObjectClass}_{Im_A}(o)) \cap \text{fields}_{Tm_B}(\text{ObjectClass}_{Im_B}(o))$ or
$f \notin \text{fields}_{Tm_A}(\text{ObjectClass}_{Im_A}(o))$ or $f \notin \text{fields}_{Tm_B}(\text{ObjectClass}_{Im_B}(o))$.

– If $f \in \text{fields}_{Tm_A}(\text{ObjectClass}_{Im_A}(o)) \cap \text{fields}_{Tm_B}(\text{ObjectClass}_{Im_B}(o))$, then $\text{FieldValue}_{Im_{AB}}((o,f)) = \text{FieldValue}_{Im_A}((o,f))$. In this case $\text{type}_{Tm_{AB}}(f) = \text{type}_{Tm_A}(f)$ because types are preserved by Tm_{AB}. Therefore, $v :_{Im} \text{type}_{Tm_{AB}}(f)$.

– If $f \notin \text{fields}_{Tm_A}(\text{ObjectClass}_{Im_A}(o))$, then by assumption, $f \in \text{fields}_{Tm_B}(\text{ObjectClass}_{Im_B}(o))$. Therefore, $\text{FieldValue}_{Im_{AB}}((o,f)) = \text{FieldValue}_{Im_B}((o,f))$. Furthermore, $\text{type}_{Tm_{AB}}(f) = \text{type}_{Tm_A}(f)$ because types are preserved by Tm_{AB}. Therefore, $v :_{Im} \text{type}_{Tm_{AB}}(f)$.

– If $f \notin \text{fields}_{Tm_B}(\text{ObjectClass}_{Im_B}(o))$, then by assumption, $f \in \text{fields}_{Tm_A}(\text{ObjectClass}_{Im_A}(o))$. Therefore, $\text{FieldValue}_{Im_{AB}}((o,f)) = \text{FieldValue}_{Im_A}((o,f))$. Furthermore, $\text{type}_{Tm_{AB}}(f) = \text{type}_{Tm_A}(f)$ because types are preserved by Tm_{AB}. Therefore, $v :_{Im} \text{type}_{Tm_{AB}}(f)$.

- $\forall((o,f),v) \in \text{FieldValue}_{Im_{AB}} : \text{validMul}_{Im_{AB}}(v)$.

If $o \in Object_{Im_A} \setminus Object_{Im_B}$, then $f \in \text{fields}_{Tm_A}(\text{ObjectClass}_{Im_A}(o))$ or
$f \notin \text{fields}_{Tm_A}(\text{ObjectClass}_{Im_A}(o))$.

– If $f \in \text{fields}_{Tm_A}(\text{ObjectClass}_{Im_A}(o))$, then $\text{FieldValue}_{Im_{AB}}((o,f)) = \text{FieldValue}_{Im_A}((o,f))$. Because Im_A is valid, $\text{validMul}_{Im_A}(v)$ holds. If the multiplicity is preserved because $f \notin Field_{Tm_B}$, then $\text{validMul}_{Im_{AB}}(v)$. If the multiplicity is changed because $f \in Field_{Tm_B}$, then $\text{validMul}_{Im_{AB}}(v)$ is proven by assumption.

– If $f \notin \text{fields}_{Tm_A}(\text{ObjectClass}_{Im_A}(o))$, then by assumption, $f \in Object_{Im_B}$. However, $f \notin Object_{Im_B}$, so this case is invalid.

If $o \in Object_{Im_B} \setminus Object_{Im_A}$, then $f \in \text{fields}_{Tm_B}(\text{ObjectClass}_{Im_B}(o))$ or
$f \notin \text{fields}_{Tm_B}(\text{ObjectClass}_{Im_B}(o))$.

– If $f \in \text{fields}_{Tm_B}(\text{ObjectClass}_{Im_B}(o))$, then $\text{FieldValue}_{Im_{AB}}((o,f)) = \text{FieldValue}_{Im_B}((o,f))$. Because Im_B is valid, $\text{validMul}_{Im_B}(v)$ holds. If the multiplicity is preserved because $f \notin Field_{Tm_A}$, then $\text{validMul}_{Im_{AB}}(v)$. If the multiplicity is changed because $f \in Field_{Tm_A}$, then $\text{validMul}_{Im_{AB}}(v)$ is proven by assumption.

– If $f \notin \text{fields}_{Tm_B}(\text{ObjectClass}_{Im_B}(o))$, then by assumption, $f \in Object_{Im_A}$. However, $f \notin Object_{Im_A}$, so this case is invalid.

If $o \in Object_{Im_A} \cap Object_{Im_B}$, then
$f \in \text{fields}_{Tm_A}(\text{ObjectClass}_{Im_A}(o)) \cap \text{fields}_{Tm_B}(\text{ObjectClass}_{Im_B}(o))$ or
$f \notin \text{fields}_{Tm_A}(\text{ObjectClass}_{Im_A}(o))$ or $f \notin \text{fields}_{Tm_B}(\text{ObjectClass}_{Im_B}(o))$.

– If $f \in \text{fields}_{Tm_A}(\text{ObjectClass}_{Im_A}(o)) \cap \text{fields}_{Tm_B}(\text{ObjectClass}_{Im_B}(o))$, then $\text{FieldValue}_{Im_{AB}}((o,f)) = \text{FieldValue}_{Im_A}((o,f))$. In this case $\text{validMul}_{Im_{AB}}(v)$ is proven by assumption since the multiplicity of f has been combined into a new multiplicity.

– If $f \notin \text{fields}_{Tm_A}(\text{ObjectClass}_{Im_A}(o))$, then by assumption, $f \in \text{fields}_{Tm_B}(\text{ObjectClass}_{Im_B}(o))$. Therefore, $\text{FieldValue}_{Im_{AB}}((o,f)) = \text{FieldValue}_{Im_B}((o,f))$. Furthermore, because Im_B is valid, $\text{validMul}_{Im_B}(v)$ holds. If the multiplicity is preserved because $f \notin Field_{Tm_A}$, then $\text{validMul}_{Im_{AB}}(v)$. If the multiplicity is changed because $f \in Field_{Tm_A}$, then $\text{validMul}_{Im_{AB}}(v)$ is proven by assumption.

– If $f \notin \text{fields}_{Tm_B}(\text{ObjectClass}_{Im_B}(o))$, then by assumption, $f \in \text{fields}_{Tm_A}(\text{ObjectClass}_{Im_A}(o))$. Therefore, $\text{FieldValue}_{Im_{AB}}((o,f)) = \text{FieldValue}_{Im_A}((o,f))$. Furthermore, because Im_A is valid, $\text{validMul}_{Im_A}(v)$ holds. If the multiplicity is preserved because $f \notin Field_{Tm_B}$, then $\text{validMul}_{Im_{AB}}(v)$. If the multiplicity is changed because $f \in Field_{Tm_B}$, then $\text{validMul}_{Im_{AB}}(v)$ is proven by assumption.

- $\forall p \in Prop_{Tm_{AB}} : Im \models p$

Make a case distinction for the different possible properties.

– For $[\mathsf{abstract}, c] \in Prop_{Tm_{AB}}$, use the fact that Tm_{AB} is consistent to establish that abstract properties are only copied iff there are no instances of it in the combined instance graph (see Definition 4.3.4 for details). Therefore, $Im \models [\mathsf{abstract}, c]$.

– For $[\mathsf{containment}, r] \in Prop_{Tm_{AB}}$, use the assumptions to prove that $Im \models [\mathsf{containment}, r]$.

- For $[\mathsf{defaultValue}, f, v] \in Prop_{Tm_{AB}}$, there is no specific satisfaction formula, therefore $Im \models [\mathsf{defaultValue}, f, v]$.
- For $[\mathsf{identity}, c, A] \in Prop_{Tm_{AB}}$, if the property was copied over from only one of the type models, then $Im \models [\mathsf{identity}, c, A]$. This is the case because c could not have been part of the other type model and therefore not occur in its instance models.

  If $[\mathsf{identity}, c, A] \in Prop_{Tm_{AB}}$ was present in both type models, then the identity satisfaction formula will be correct for each pair of instances from Im_A and each pair of instances from Im_B. To ensure that it is correct for mixed instance pairs, use the assumption specified. Then establish that $Im \models [\mathsf{identity}, c, A]$.
- For $[\mathsf{keyset}, r, A] \in Prop_{Tm_{AB}}$, it is clear that each value of r in the combined instance graph is either the copied field value from Im_A or the copied field value from Im_B. Since values are preserved, there can be no new objects added to an existing relation. Therefore, $Im \models [\mathsf{keyset}, r, A]$.
- For $[\mathsf{opposite}, r, r'] \in Prop_{Tm_{AB}}$, if the property was copied over from only one of the type models, then $Im \models [\mathsf{opposite}, r, r']$. This is the case because r and r' could not have been part of the other type model and therefore not occur in its instance models.

  If $[\mathsf{opposite}, r, r'] \in Prop_{Tm_{AB}}$ was present in both type models, then the opposite satisfaction formula will be correct for each pair of instances from Im_A and each pair of instances from Im_B. To ensure that it is correct for mixed instance pairs, use the assumption specified. Then establish that $Im \models [\mathsf{opposite}, r, r']$.
- For $[\mathsf{readonly}, f] \in Prop_{Tm_{AB}}$, there is no specific satisfaction formula, therefore $Im \models [\mathsf{readonly}, f]$.

- $\forall c \in Constant_{Tm_{AB}} : \mathrm{DefaultValue}_{Im_{AB}}(c) :_{Im_{AB}} \mathrm{ConstType}_{Tm_{AB}}(c)$.

  If $c \in Constant_{Tm_A} \setminus Constant_{Tm_B}$, then $\mathrm{DefaultValue}_{Im_{AB}}(c) = \mathrm{DefaultValue}_{Im_A}(c)$. Furthermore, $\mathrm{ConstType}_{Tm_{AB}}(c) = \mathrm{ConstType}_{Tm_A}(c)$. Therefore,
  $\mathrm{DefaultValue}_{Im_{AB}}(c) :_{Im_{AB}} \mathrm{ConstType}_{Tm_{AB}}(c)$.

  Similarly, if $c \in Constant_{Tm_B} \setminus Constant_{Tm_A}$, then $\mathrm{DefaultValue}_{Im_{AB}}(c) = \mathrm{DefaultValue}_{Im_B}(c)$. Furthermore, $\mathrm{ConstType}_{Tm_{AB}}(c) = \mathrm{ConstType}_{Tm_B}(c)$. Therefore,
  $\mathrm{DefaultValue}_{Im_{AB}}(c) :_{Im_{AB}} \mathrm{ConstType}_{Tm_{AB}}(c)$.

  If $c \in Constant_{Tm_A} \cap Constant_{Tm_B}$, then $\mathrm{DefaultValue}_{Im_A}(c) = \mathrm{DefaultValue}_{Im_B}(c)$ by assumption. Furthermore, $\mathrm{ConstType}_{Tm_A}(c) = \mathrm{ConstType}_{Tm_B}(c)$, since Tm_{AB} is consistent. Therefore, $\mathrm{DefaultValue}_{Im_{AB}}(c) :_{Im_{AB}} \mathrm{ConstType}_{Tm_{AB}}(c)$.
- Tm_{AB} is consistent, as defined in Definition 3.2.11.

  This is specified to be true by assumption.

The proofs of all these individual properties complete the entire proof. □

As explained before, Theorem 4.4.12 does not take into account that the instance models are supposed to be distinct except for a set of objects. Furthermore, it does not take into account that the corresponding type models are supposed to be distinct except for a set of types. The following lemma is an alternation of the previous theorem, which takes these properties into account.

Lemma 4.4.13 (Consistency of the combination (mostly) distinct of instance models)
Assume that Im_A and Im_B are valid instance models in the sense of Definition 3.2.19. Assume that Im_A is typed by type model Tm_A. Furthermore, assume that Im_B is typed by type model Tm_B. Tm_A and Tm_B are consistent by definition. Also assume that $Tm_{AB} = \mathrm{combine}(Tm_A, Tm_B)$ is consistent in the sense of Definition 3.2.11. Moreover, assume that Tm_A and Tm_B are entirely distinct except for a set of types T. Also assume that Im_A and Im_B are entirely distinct except for a set of objects O. Finally, assume the following properties:

- *For all shared objects, the object class must be the same in both instance models: $\forall o \in Object_{Im_A} \cap Object_{Im_B} : \mathrm{ObjectClass}_{Im_A}(o) = \mathrm{ObjectClass}_{Im_B}(o)$.*
- *For all shared objects, the object id must be the same in both instance models: $\forall o \in Object_{Im_A} \cap Object_{Im_B} : \mathrm{ObjectId}_{Im_A}(o) = \mathrm{ObjectId}_{Im_B}(o)$.*
- *For all shared constants within the corresponding type graphs, the default value must be the same in both instance models: $\forall c \in Constant_{Tm_A} \cap Constant_{Tm_B} :$ $\mathrm{DefaultValue}_{Im_A}(c) = \mathrm{DefaultValue}_{Im_B}(c)$.*

- *The identifiers must be unique across both instance models:* $\forall o_1 \in Object_{Im_A} \setminus Object_{Im_B} \wedge o_2 \in Object_{Im_B} \setminus Object_{Im_A} : \text{ObjectId}_{Im_A}(o_1) = \text{ObjectId}_{Im_B}(o_2) \implies o_1 = o_2$.
- *If an object needs a field value in the combination of* Im_A *and* Im_B*, but this field value is not set in* Im_A*, then it must be set in* Im_B*:* $\forall o \in Object_{Im_A} \wedge f \notin \text{fields}_{Tm_A}(\text{ObjectClass}_{Im_A}(o)) : f \in \text{fields}_{Tm_{AB}}(\text{ObjectClass}_{\text{combine}(Im_A,Im_B)}(o)) \implies o \in Object_{Im_B} \wedge f \in \text{fields}_{Tm_B}(\text{ObjectClass}_{Im_B}(o))$.
- *If an object needs a field value in the combination of* Im_A *and* Im_B*, but this field value is not set in* Im_B*, then it must be set in* Im_A*:* $\forall o \in Object_{Im_B} \wedge f \notin \text{fields}_{Tm_B}(\text{ObjectClass}_{Im_B}(o)) : f \in \text{fields}_{Tm_{AB}}(\text{ObjectClass}_{\text{combine}(Im_A,Im_B)}(o)) \implies o \in Object_{Im_A} \wedge f \in \text{fields}_{Tm_A}(\text{ObjectClass}_{Im_A}(o))$.
- *If there exists a containment property in* Tm_{AB}*, the satisfaction formula for containment properties must be satisfied:* $\forall o \in Object_{Im_A} \cup Object_{Im_B} : \left|\left\{((fo, ff), fv) \mid ((fo, ff), fv) \in \text{FieldValue}_{\text{combine}(Im_A,Im_B)} \wedge [\mathsf{obj}, o] = fv \wedge ff \in CR_{Tm_{AB}}\right\}\right| \leq 1$
- *There may be no cycles in the containment edges of the combined instance model:* $\left\{(fo, fv) \mid ((fo, ff), fv) \in \text{FieldValue}_{\text{combine}(Im_A,Im_B)} \wedge ff \in CR_{Tm_{AB}}\right\}$ *is acyclic.*

Then $\text{combine}(Im_A, Im_B)$ *is a valid instance model in the sense of Definition 3.2.19.*

Also see `imod_combine_merge_correct` *in* `Ecore.Instance_Model_Combination`

Proof. Use Theorem 4.4.12 to show that $\text{combine}(Im_A, Im_B)$ is a consistent type model. Use the assumptions given. Some assumptions of Theorem 4.4.12 become irrelevant because Im_A and Im_B are mostly distinct. □

Finally, the concept of compatibility between two instance models is defined.

Definition 4.4.14 (Compatibility of instance models)
Assume instance models Im_A *and* Im_B*. We say that* Im_A *is compatible with* Im_B *if* $\text{combine}(Im_A, Im_B)$ *is a valid instance model in the sense of Definition 3.2.19.*

The notion of compatibility will be used later as a way to denote instance models that can be combined with other instance models without loss of validity.

4.4.2 Combining instance graphs

The structure of Figure 4.10 shows that the instance graphs IG_A and IG_B are combined into one instance graph IG_{AB}. This section provides the definition of this combination and its corresponding theorems. Please note that the definitions presented here, just like the previous section, are as generic as possible, and do not actively take into account that IG_A and IG_B are mostly distinct. This bit of information is added later as part of a theorem and proof.

Definition 4.4.15 (Combination function on instance graphs)
combine *is a binary function on two instance graphs which combines two instance graphs into one instance graph. Assume* IG_A *is an instance graph typed by type graph* TG_A *and* IG_B *is an instance graph typed by type graph* TG_B*, then* $\text{combine}(IG_A, IG_B)$ *is typed by* $\text{combine}(TG_A, TG_B)$ *and is defined as follows:*

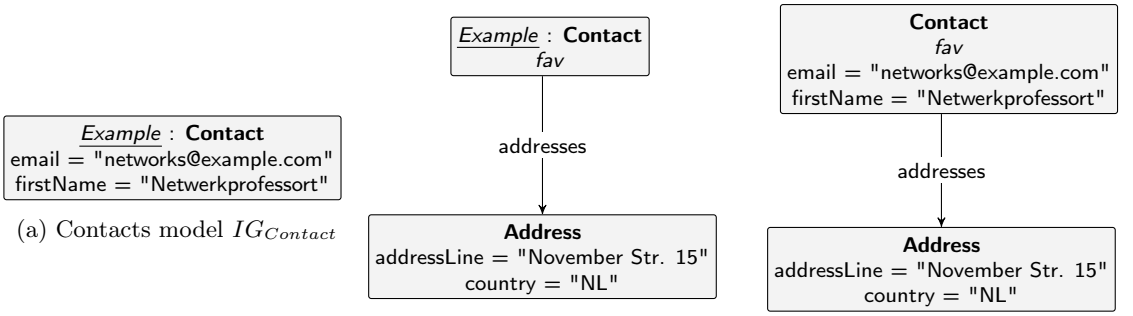
$$\begin{aligned}\text{combine}(IG_A, IG_B) = \langle N &= N_{IG_A} \cup N_{IG_B} \\ E &= E_{IG_A} \cup E_{IG_B} \\ \text{ident} &= \text{ident\_combine}(IG_A, IG_B)\rangle\end{aligned}$$

In which ident_combine *is given as part of Definition 4.4.16.*

Also see `ig_combine` *in* `GROOVE.Instance_Graph_Combination`

The definition of the combination of instance graphs is the easiest combination function to understand. Essentially, it does nothing more than combining the nodes and edges of the graph. The only thing that is done extra is the combination of the identification function, of which the definition is presented within the next definition.

Definition 4.4.16 (Combination function for identities of instance graphs)
$\text{ident\_combine}(IG_A, IG_B)$ *is a partial function on two type graphs which returns a new function*



(a) Contacts model $IG_{Contact}$

(b) Address extension IG_{Ext}

(c) Combined model $IG_{ContactExt}$

Figure 4.12: Example of the combination of instance graphs

$Id \Rightarrow (N_{IG_{AB}} \cap Node_t)$. *It is defined as follows:*

$$\text{ident\_combine}(IG_A, IG_B, i) = \begin{cases} \text{ident}_{IG_A}(i) & \text{if } i \in \text{dom ident}_{IG_A} \cap \text{dom ident}_{IG_B} \wedge \text{ident}_{IG_A}(i) = \text{ident}_{IG_B}(i) \\ \text{ident}_{IG_A}(i) & \text{if } i \in \text{dom ident}_{IG_A} \setminus \text{dom ident}_{IG_B} \\ \text{ident}_{IG_B}(i) & \text{if } i \in \text{dom ident}_{IG_B} \setminus \text{dom ident}_{IG_A} \end{cases}$$

Also see `ig_combine_ident` *in* `GROOVE.Instance_Graph_Combination`

The combination function for identifiers of the instance graphs is not very difficult either. If an identifier is only valid in one of the instance graphs, the identifier value is copied from that graph. Furthermore, if an identifier is valid in both instance graphs, it should be the case that the identifiers project on the same node. If this is the case, then the identifier is set for the combination.

Like the combination function of object identifiers of instance models (Definition 4.4.3), the behaviour of this function might seem strange. Theoretically, it might give rise to nodes with two identities, which is undesired. As will be shown later, the combination function and theorems assume that the identities of the graphs are already distinct. This assumption is fair, as it is possible to redefine two instance graphs to have distinct identities, without loss of significance.

With all definitions in place, it is possible to provide an example. Let us return to the contacts list example introduced in Figure 4.9 of Section 4.3.2. Suppose a simple instance of $TG_{Contact}$, which has one instance with a name and e-mail address. This could be formally defined as follows:

$$\begin{aligned} IG_{Contact} = \langle \quad N &= \{Contact_1, \text{“Netwerkprofessort”}, \text{“networks@example.com”}\} \\ \text{type}_\text{n} &= \{(Contact_1, \mathsf{Contact}), (\text{“networks@example.com”}, \mathsf{string}), \\ &\quad (\text{“Netwerkprofessort”}, \mathsf{string})\} \\ E &= \{(Contact_1, (\mathsf{Contact}, \mathsf{email}, \mathsf{string}), \text{“networks@example.com”}), \\ &\quad (Contact_1, (\mathsf{Contact}, \mathsf{firstName}, \mathsf{string}), \text{“Netwerkprofessort”})\} \\ \text{ident} &= \{(Example, Contact_1)\} \\ \rangle \end{aligned}$$

A visual representation of this model is included in Figure 4.12a. Now suppose that we want to extent this contact with an address. An instance of Tm_{Ext} represents this address, and is defined as follows:

$$\begin{aligned} IG_{Ext} = \langle \quad N &= \{Contact_1, Address_1, \text{“November Str. 15”}, \text{“NL”}\} \\ \text{type}_\text{n} &= \{(Contact_1, \mathsf{Contact}), (Address_1, \mathsf{Address}), \\ &\quad (\text{“November Str. 15”}, \mathsf{string}), (\text{“NL”}, \mathsf{string})\} \\ E &= \{(Contact_1, (\mathsf{Contact}, \mathsf{fav}, \mathsf{Contact}), Contact_1), \\ &\quad (Contact_1, (\mathsf{Contact}, \mathsf{addresses}, \mathsf{Address}), Address_1), \\ &\quad (Address_1, (\mathsf{Address}, \mathsf{addressLine}, \mathsf{string}), \text{“November Str. 15”}), \\ &\quad (Address_1, (\mathsf{Address}, \mathsf{country}, \mathsf{string}), \text{“NL”})\} \\ \text{ident} &= \{(Example, Contact_1)\} \\ \rangle \end{aligned}$$

The visual representation of IG_{Ext} is included in Figure 4.12b. With these instance models formally defined, it is possible to combine them using Definition 4.4.15. This will yield the following model:

$$
\begin{aligned}
IG_{ChatExt} = \langle \quad N &= \{Contact_1, Address_1, \text{“November Str. 15”}, \text{“NL”}\} \\
\text{type}_\text{n} &= \{(Contact_1, \mathsf{Contact}), (Address_1, \mathsf{Address}), \\
&\quad (\text{“networks@example.com”}, \mathsf{string}), (\text{“Netwerkprofessort”}, \mathsf{string}), \\
&\quad (\text{“November Str. 15”}, \mathsf{string}), (\text{“NL”}, \mathsf{string})\} \\
E &= \big\{\big(Contact_1, (\mathsf{Contact}, \mathsf{email}, \mathsf{string}), \text{“networks@example.com”}\big), \\
&\quad \big(Contact_1, (\mathsf{Contact}, \mathsf{firstName}, \mathsf{string}), \text{“Netwerkprofessort”}\big), \\
&\quad \big(Contact_1, (\mathsf{Contact}, \mathsf{fav}, \mathsf{Contact}), Contact_1\big), \\
&\quad \big(Contact_1, (\mathsf{Contact}, \mathsf{addresses}, \mathsf{Address}), Address_1\big), \\
&\quad \big(Address_1, (\mathsf{Address}, \mathsf{addressLine}, \mathsf{string}), \text{“November Str. 15”}\big), \\
&\quad \big(Address_1, (\mathsf{Address}, \mathsf{country}, \mathsf{string}), \text{“NL”}\big)\big\} \\
\text{ident} &= \{(Example, Contact_1)\} \\
\rangle &
\end{aligned}
$$

A visual representation of this combined model is included as Figure 4.12c. Like the example for the combination of type graphs, this example shows that the definition of the combination of instance graphs is useful. It allows to build larger graphs out of smaller building blocks. Furthermore, the example shows that the combination of the two instance graphs is typed by the combination of its corresponding type graphs.

Although the definitions of the combination of instance graphs are given, no mathematical properties or theorems are defined yet. Some mathematical properties hold for the combination of instance graphs, that will be presented in the following theorems.

Theorem 4.4.17 (Commutativity of the combination of instance graphs)
Assume that IG_A and IG_B are instance graphs, then the combine *function is commutative:*

$$\text{combine}(IG_A, IG_B) = \text{combine}(IG_B, IG_A)$$

Also see `ig_combine_commute` *in* `GROOVE.Instance_Graph_Combination`

Theorem 4.4.18 (Associativity of the combination of instance graphs)
Assume that IG_A, IG_B and IG_C are instance graphs, then the combine *function is associative:*

$$\text{combine}(\text{combine}(IG_A, IG_B), IG_C) = \text{combine}(IG_A, \text{combine}(IG_B, IG_C))$$

Also see `ig_combine_assoc` *in* `GROOVE.Instance_Graph_Combination`

Theorem 4.4.19 (Idempotence of the combination of instance graphs)
Assume that IG_A is an instance graph and that it is valid in the sense of Definition 3.3.10. Then the following property holds:

$$\text{combine}(IG_A, IG_A) = IG_A$$

Also see `ig_combine_idemp_alt` *in* `GROOVE.Instance_Graph_Combination`

These properties follow directly from Definition 4.4.15, but the corresponding proofs will not be included here. It should be noted that these properties are indeed proven correct as part of this thesis, and the corresponding proofs are validated within Isabelle.

Besides these properties, the combination of instance graphs also has an identity element. The empty instance graph represents this identity element, but it needs to be defined first:

Definition 4.4.20 (Empty instance graph)
Let IG_ϵ be the empty instance graph. It is typed by the empty type graph TG_ϵ. IG_ϵ is defined as:

$$
\begin{aligned}
IG_\epsilon = \langle N &= \{\} \\
E &= \{\} \\
\text{ident} &= undefined\rangle
\end{aligned}
$$

Theorem 4.4.21 (Correctness of the empty type model)
The empty instance graph, IG_ϵ, is valid with respect to Definition 3.3.10.

Also see `ig_empty_correct` *in* `GROOVE.Instance_Graph`

The proof for the correctness of the empty instance graph is trivial. Still, a validated version of this proof can be found within the Isabelle theories of this thesis.

As mentioned earlier, the empty instance graph acts as an identity element when combining two instance graphs. The following theorem specifies this behaviour.

Theorem 4.4.22 (Identity of the combination of instance models)
Assume that IG_A is an instance graph and that it is valid in the sense of Definition 3.3.10. Then IG_ϵ acts as an identity element in the combination function:

$$\text{combine}(IG_\epsilon, IG_A) = IG_A$$

Also see `ig_combine_identity_alt` *in* `GROOVE.Instance_Graph_Combination`

Once more, the proof of this theorem follows directly from the definition. Therefore, the corresponding proof will not be included here, but a validated version can be found within the Isabelle theories of this thesis.

A final desired property for the combination of instance models is a correctness property. Theorem 4.4.12 defines the theorem under which the combination of instance models is a valid instance model. Please note that this theorem is a generic theorem, which does not take into account that the instance models are mostly distinct.

Theorem 4.4.23 (Validity of the combination of instance graphs)
Assume that IG_A and IG_B are valid instance graphs in the sense of Definition 3.3.10. Assume that IG_A is typed by type model TG_A. Furthermore, assume that IG_B is typed by type model TG_B. TG_A and TG_B are valid by definition. Also assume that $TG_{AB} = \text{combine}(TG_A, TG_B)$ is valid in the sense of Definition 3.3.5. Finally, assume the following properties:

- *For all shared identities, the nodes belonging to the identities must be equal in both instance graphs:* $\forall i \in \text{dom ident}_{IG_A} \cap \text{dom ident}_{IG_B} : \text{ident}_{IG_A}(i) = \text{ident}_{IG_B}(i)$.
- *The outgoing multiplicity for edges must be valid:* $\forall et \in ET_{TG_A} \cup ET_{TG_B} \wedge n \in N_{IG_A} \cup N_{IG_B} :$ $(\text{type}_\text{n}(n), \text{src}(et)) \in \sqsubseteq_{TG_{AB}} \implies$ $|\{e \mid e \in E_{IG_A} \cup E_{IG_B} \wedge \text{src}(e) = n \wedge \text{type}_\text{e}(e) = et\}| \in \text{out}(\text{mult}_{TG_{AB}}(et))$.
- *The incoming multiplicity for edges must be valid:* $\forall et \in ET_{TG_A} \cup ET_{TG_B} \wedge n \in N_{IG_A} \cup N_{IG_B} :$ $(\text{type}_\text{n}(n), \text{tgt}(et)) \in \sqsubseteq_{TG_{AB}} \implies$ $|\{e \mid e \in E_{IG_A} \cup E_{IG_B} \wedge \text{tgt}(e) = n \wedge \text{type}_\text{e}(e) = et\}| \in \text{in}(\text{mult}_{TG_{AB}}(et))$.
- *Each node may only be contained by one other node:* $\forall n \in N_{IG_A} \cup N_{IG_B} : |\{e \mid e \in E_{IG_A} \cup E_{IG_B} \wedge \text{tgt}(e) = n \wedge \text{type}_\text{e}(e) \in \mathit{contains}_{TG_{AB}}\}| \leq 1$.
- *There may be no cycle in the containment edges of the combined instance graph:* $\{(\text{src}(e), \text{tgt}(e)) \mid e \in E_{IG_A} \cup E_{IG_B} \wedge \text{type}_\text{e}(e) \in \mathit{contains}_{TG_{AB}}\}$ *is acyclic.*

Then $\text{combine}(IG_A, IG_B)$ *is a valid instance graph in the sense of Definition 3.3.10.*

Also see `ig_combine_correct` *in* `GROOVE.Instance_Graph_Combination`

Proof. To proof that $\text{combine}(IG_A, IG_B)$ is a valid instance graph, it needs to be shown that $\text{combine}(IG_A, IG_B)$ gives rise to a valid structure for an instance graph and that Definition 3.3.10 holds. For readability, define IG_{AB} to be $\text{combine}(IG_A, IG_B)$.

Structural properties

- All elements of $N_{IG_{AB}}$ are elements of $Node_t \cup Node_v$.

  Follows from $N_{IG_A} \subseteq Node_t \cup Node_v$ and $N_{IG_B} \subseteq Node_t \cup Node_v$.

- All elements of $E_{IG_{AB}}$ are elements of $N_{IG_{AB}} \times ET_{TG_{AB}} \times N_{IG_{AB}}$.

  Follows from $E_{IG_A} \subseteq (N_{IG_A} \times ET_{TG_A} \times N_{IG_A})$ and $E_{IG_B} \subseteq (N_{IG_B} \times ET_{TG_B} \times N_{IG_B})$. To complete the proof, use that $ET_{TG_{AB}} = ET_{TG_A} \cup ET_{TG_B}$ and $N_{IG_{AB}} = N_{IG_A} \cup N_{IG_B}$.

- For each identity $i \in \text{dom ident}_{IG_{AB}}$, $\text{ident}_{IG_{AB}}(i)$ is an element of $N_{IG_{AB}} \cap Node_t$.

  First note that $\text{ident}_{IG_{AB}}(i) = \text{ident}_{IG_A}(i)$ or $\text{ident}_{IG_{AB}}(i) = \text{ident}_{IG_B}(i)$.

  If $\text{ident}_{IG_{AB}}(i) = \text{ident}_{IG_A}(i)$, then have $\text{ident}_{IG_A}(i) \in N_{IG_A} \cap Node_t$. Then use $N_{IG_{AB}} = N_{IG_A} \cup N_{IG_B}$ to have $\text{ident}_{IG_{AB}}(i) \in N_{IG_{AB}} \cap Node_t$.

  Similarly, if $\text{ident}_{IG_{AB}}(i) = \text{ident}_{IG_B}(i)$, then have $\text{ident}_{IG_B}(i) \in N_{IG_B} \cap Node_t$. Then use $N_{IG_{AB}} = N_{IG_A} \cup N_{IG_B}$ to have $\text{ident}_{IG_{AB}}(i) \in N_{IG_{AB}} \cap Node_t$.

Validity properties

- $\forall n \in N_{IG_{AB}} : \text{type}_n(n) \in NT_{TG_{AB}}$.

  Have that $\text{type}_n(n) \in NT_{TG_A}$ or $\text{type}_n(n) \in NT_{TG_B}$. Then have that $\text{type}_n(n) \in NT_{TG_{AB}}$ because $N_{IG_{AB}} = N_{IG_A} \cup N_{IG_B}$.

- $\forall e \in E_{IG_{AB}} : \text{type}_n\big(\text{src}(e)\big) \sqsubseteq_{TG_{AB}} \text{src}\big(\text{type}_e(e)\big)$.

  Since types of nodes and edges are preserved while merging, $\sqsubseteq_{TG_A} \subseteq \sqsubseteq_{TG_{AB}}$ and $\sqsubseteq_{TG_B} \subseteq \sqsubseteq_{TG_{AB}}$, it follows that $\text{type}_n\big(\text{src}(e)\big) \sqsubseteq_{TG_{AB}} \text{src}\big(\text{type}_e(e)\big)$.

- $\forall e \in E_{IG_{AB}} : \text{type}_n\big(\text{tgt}(e)\big) \sqsubseteq_{TG_{AB}} \text{tgt}\big(\text{type}_e(e)\big)$.

  Since types of nodes and edges are preserved while merging, $\sqsubseteq_{TG_A} \subseteq \sqsubseteq_{TG_{AB}}$ and $\sqsubseteq_{TG_B} \subseteq \sqsubseteq_{TG_{AB}}$, it follows that $\text{type}_n\big(\text{tgt}(e)\big) \sqsubseteq_{TG_{AB}} \text{tgt}\big(\text{type}_e(e)\big)$.

- $\forall n \in N_{IG_{AB}} : \text{type}_n(n) \notin abs_{TG_{AB}}$.

  Use that $N_{IG_{AB}} = N_{IG_A} \cup N_{IG_B}$. Then make a case distinction.

  If $n \in N_{IG_A}$, then $\text{type}_n(n) \notin abs_{TG_A}$. Furthermore, $\text{type}_n(n) \notin abs_{TG_B} \setminus NT_{TG_A}$. Then we have that $\text{type}_n(n) \notin abs_{TG_{AB}}$.

  Similarly, if $n \in N_{IG_B}$, then $\text{type}_n(n) \notin abs_{TG_B}$. Furthermore, $\text{type}_n(n) \notin abs_{TG_A} \setminus NT_{TG_B}$. Then we have that $\text{type}_n(n) \notin abs_{TG_{AB}}$.

- $\forall et \in ET_{TG_{AB}} : \forall n \in N_{IG_{AB}} : \text{type}_n(n) \sqsubseteq_{TG_{AB}} \text{src}(et) \implies |\{e \in E_{IG_{AB}} \mid \text{src}(e) = n \wedge \text{type}_e(e) = et\}| \in \text{out}(\text{mult}_{TG_{AB}}(et))$.

  This can be solved directly by expanding some definitions and using the assumption for outgoing multiplicities.

- $\forall et \in ET_{TG_{AB}} : \forall n \in N_{IG_{AB}} : \text{type}_n(n) \sqsubseteq_{TG_{AB}} \text{tgt}(et) \implies |\{e \in E_{IG_{AB}} \mid \text{tgt}(e) = n \wedge \text{type}_e(e) = et\}| \in \text{in}(\text{mult}_{TG_{AB}}(et))$.

  This can be solved directly by expanding some definitions and using the assumption for incoming multiplicities.

- $\forall n \in N_{IG_{AB}} : |\{e \in E_{IG_{AB}} \mid \text{tgt}(e) = n \wedge \text{type}_e(e) \in contains_{TG_{AB}}\}| \leq 1$.

  This can be solved directly by expanding some definitions and using the assumption for the containment of nodes.

- There may be no cycle between the containment edges in $E_{IG_{AB}}$.

  This is solved by assumption.

The proofs of all these individual properties complete the entire proof. □

As explained before, Theorem 4.4.23 does not take into account that the instance graphs are supposed to be distinct except for a set of nodes. Furthermore, it does not take into account that the corresponding type graphs are supposed to be distinct except for a set of node types. The following lemma is an alternation of the previous theorem, which takes these properties into account.

Lemma 4.4.24 (Consistency of the combination (mostly) distinct of instance graphs)
Assume that IG_A and IG_B are valid instance graphs in the sense of Definition 3.3.10. Assume that IG_A is typed by type model TG_A. Furthermore, assume that IG_B is typed by type model TG_B. TG_A and TG_B are consistent by definition. Also assume that $TG_{AB} = \text{combine}(TG_A, TG_B)$ is consistent in the sense of Definition 3.3.5. Moreover, assume that TG_A and TG_B are entirely distinct except for a set of node types NT. Also assume that IG_A and IG_B are entirely distinct except for a set of nodes N. Finally, assume the following properties:

- *For all shared identities, the nodes belonging to the identities must be equal in both instance graphs:* $\forall i \in \text{dom ident}_{IG_A} \cap \text{dom ident}_{IG_B} : \text{ident}_{IG_A}(i) = \text{ident}_{IG_B}(i)$.

- *The outgoing multiplicity for edges in IG_A must be valid:* $\forall et \in ET_{TG_A} \wedge n \in N_{IG_A} \cup N_{IG_B} : (\text{type}_\text{n}(n), \text{src}(et)) \in \sqsubseteq_{TG_{AB}} \implies |\{e \mid e \in E_{IG_A} \wedge \text{src}(e) = n \wedge \text{type}_\text{e}(e) = et\}| \in \text{out}(\text{mult}_{TG_A}(et))$.
- *The incoming multiplicity for edges in IG_A must be valid:* $\forall et \in ET_{TG_A} \wedge n \in N_{IG_A} \cup N_{IG_B} : (\text{type}_\text{n}(n), \text{tgt}(et)) \in \sqsubseteq_{TG_{AB}} \implies |\{e \mid e \in E_{IG_A} \wedge \text{tgt}(e) = n \wedge \text{type}_\text{e}(e) = et\}| \in \text{in}(\text{mult}_{TG_A}(et))$.
- *The outgoing multiplicity for edges in IG_B must be valid:* $\forall et \in ET_{TG_B} \wedge n \in N_{IG_A} \cup N_{IG_B} : (\text{type}_\text{n}(n), \text{src}(et)) \in \sqsubseteq_{TG_{AB}} \implies |\{e \mid e \in E_{IG_B} \wedge \text{src}(e) = n \wedge \text{type}_\text{e}(e) = et\}| \in \text{out}(\text{mult}_{TG_B}(et))$.
- *The incoming multiplicity for edges in IG_B must be valid:* $\forall et \in ET_{TG_B} \wedge n \in N_{IG_A} \cup N_{IG_B} : (\text{type}_\text{n}(n), \text{tgt}(et)) \in \sqsubseteq_{TG_{AB}} \implies |\{e \mid e \in E_{IG_B} \wedge \text{tgt}(e) = n \wedge \text{type}_\text{e}(e) = et\}| \in \text{in}(\text{mult}_{TG_B}(et))$.
- *Each node that is present in both instance graphs may only be contained by one other node:* $\forall n \in N_{IG_A} \cap N_{IG_B} : |\{e \mid e \in E_{IG_A} \cup E_{IG_B} \wedge \text{tgt}(e) = n \wedge \text{type}_\text{e}(e) \in \mathit{contains}_{TG_{AB}}\}| \leq 1$.
- *There may be no cycle in the containment edges of the combined instance graph:* $\{(\text{src}(e), \text{tgt}(e)) \mid e \in E_{IG_A} \cup E_{IG_B} \wedge \text{type}_\text{e}(e) \in \mathit{contains}_{TG_{AB}}\}$ *is acyclic.*

Then $\text{combine}(IG_A, IG_B)$ *is a valid instance graph in the sense of Definition 3.3.10.*

Also see `ig_combine_merge_correct` *in* `GROOVE.Instance_Graph_Combination`

Proof. Use Theorem 4.4.23 to show that $\text{combine}(IG_A, IG_B)$ is a consistent type model. Use the assumptions given. Some assumptions of Theorem 4.4.12 are solved using multiple assumptions because part of the assumption became irrelevant. □

Finally, the concept of compatibility between two instance graphs is defined.

Definition 4.4.25 (Compatibility of instance graphs)
Assume instance graphs IG_A and IG_B. We say that IG_A is compatible with IG_B if $\text{combine}(IG_A, IG_B)$ *is a valid instance graph in the sense of Definition 3.3.10.*

The notion of compatibility will be used later as a way to denote instance graphs that can be combined with other instance graphs without loss of validity.

4.4.3 Combining transformation functions

The previous sections discussed the combination of instance models and instance graphs. In this section, the combination of transformation functions between instance models and instance graphs is discussed. This combination is the last key element shown in Figure 4.10. If it is possible to combine f_A and f_B into $f_A \sqcup f_B$, then it is possible to build transformation functions between instance models and instance graphs iteratively.

Before it is possible to define a definition for the combination of two transformation functions, it is essential to define what functions are considered to be transformation functions.

Definition 4.4.26 (Transformation function from an instance model to an instance graph)
Let f be a function from instance models to instance graphs, Im be an instance model and IG the corresponding instance graph. f is a transformation function iff:

- *f projects Im onto IG:* $f(Im) = IG$;
- *After combination with another instance model, f preserves the type graph;*
- *After combination with another instance model, f preserves the nodes:* $\forall Im_x : N_{f(Im)} \subseteq N_{f(\text{combine}(Im, Im_x))}$;
- *After combination with another instance model, f preserves the edges:* $\forall Im_x : E_{f(Im)} \subseteq E_{f(\text{combine}(Im, Im_x))}$;
- *For all identities in the projected instance graph, f preserves the value of the identity if the instance model is combined with another instance model:* $\forall Im_x : \forall i \in \text{dom ident}_{f(Tm)} : \text{ident}_{f(Im)}(i) = \text{ident}_{f(\text{combine}(Im, Im_x))}(i)$.

Also see `ig_combine_mapping_function` *in* `Ecore-GROOVE-Mapping.Instance_Model_Graph_Mapping`

As expected, a transformation must project some instance model Im to its corresponding instance graph IG. Furthermore, it has to preserve properties of the projection, even after Im is combined with some other instance model. The rationale behind these properties is that after combining Im with some other instance model, there must still be a way to transform the elements that originated from Im. If that is possible, it is possible to use the transformation function as the basis for the combined transformation function, which can transform the combined instance model to a combined instance graph.

The following definition will describe how two transformation functions from instance models to instance graphs can be combined into a new transformation function, which projects the combination of two instance models onto the combination of the two corresponding instance graphs.

Definition 4.4.27 (Combination of transformation functions from an instance model to an instance graph)
Let f_A and f_B be a transformation functions in the sense of Definition 4.4.26. f_A projects an instance model Im_A onto instance graph IG_A. f_B projects an instance model Im_B onto instance graph IG_B. Then the combination of f_A and f_B is defined as:

$$\begin{aligned} f_A \sqcup f_B(Im) = \langle \quad & N = \{n \mid n \in N_{f_A(Im)} \wedge n \in N_{IG_A}\} \cup \{n \mid n \in N_{f_B(Im)} \wedge n \in N_{IG_B}\} \\ & E = \{e \mid e \in E_{f_A(Im)} \wedge e \in E_{IG_A}\} \cup \{e \mid e \in E_{f_B(Im)} \wedge e \in E_{IG_B}\} \\ & \text{ident} = \text{ident\_mapping}(f_A, IG_A, f_B, IG_B, Im)\rangle \end{aligned}$$

In which ident_mapping *is given as part of Definition 4.4.28*

Also see `ig_combine_mapping` *in* `Ecore-GROOVE-Mapping.Instance_Model_Graph_Mapping`

Like the combination of transformations functions from a type model to a type graph, the combination of transformations functions from an instance model to an instance graph follows the combination of instance graphs closely. The definition is an alternation of Definition 4.4.15. As shown for type models and type graphs, this is the desired behaviour, as the combination of the transformation functions should be able to transform the combination of two instance models to the combination of the two corresponding instance graphs.

Unsurprisingly, the definition of the identity function of $f_A \sqcup f_B$ is very similar to Definition 4.4.16.

Definition 4.4.28 (Combination of the identity function for two transformation functions)
$\text{ident\_mapping}(f_A, IG_A, f_B, IG_B, Im)$ *is a partial function on two transformation functions f_A and f_B, their corresponding projections IG_A and IG_B and an instance model Im which returns a new function $Id \Rightarrow (N_{f_A \sqcup f_B(Im)} \cap Node_t)$. It is defined as follows:*

$$\text{ident\_combine}(f_A, IG_A, f_B, IG_B, Im, i) = \begin{cases} \text{ident}_{f_A(Im)}(i) & \text{if } i \in \{i \mid i \in \text{dom ident}_{f_A(Im)} \wedge i \in \text{dom ident}_{IG_A}\} \cap \\ & \quad \{i \mid i \in \text{dom ident}_{f_A(Im)} \wedge i \in \text{dom ident}_{IG_A}\} \wedge \text{ident}_{f_A(Im)}(i) = \text{ident}_{IG_B}(i) \\ \text{ident}_{f_A(Im)}(i) & \text{if } i \in \{i \mid i \in \text{dom ident}_{f_A(Im)} \wedge i \in \text{dom ident}_{IG_A}\} \setminus \\ & \quad \{i \mid i \in \text{dom ident}_{f_A(Im)} \wedge i \in \text{dom ident}_{IG_A}\} \\ \text{ident}_{f_B(Im)}(i) & \text{if } i \in \{i \mid i \in \text{dom ident}_{f_A(Im)} \wedge i \in \text{dom ident}_{IG_A}\} \setminus \\ & \quad \{i \mid i \in \text{dom ident}_{f_A(Im)} \wedge i \in \text{dom ident}_{IG_A}\} \end{cases}$$

With these definitions in place, it is possible to provide the necessary theorems for the correctness of the combined function $f_A \sqcup f_B$.

Theorem 4.4.29 (The projection of a combined transformation function from an instance model to an instance graph)
Let f_A and f_B be a transformation functions in the sense of Definition 4.4.26. f_A projects an instance model Im_A onto instance graph IG_A. f_B projects an instance model Im_B onto instance graph IG_B. Then the combination of f_A and f_B, $f_A \sqcup f_B$ projects $\text{combine}(Im_A, Im_B)$ *onto* $\text{combine}(IG_A, IG_B)$*, so:*

$$f_A \sqcup f_B(\text{combine}(Im_A, Im_B)) = \text{combine}(IG_A, IG_B)$$

Also see `ig_combine_mapping_correct` *in* `Ecore-GROOVE-Mapping.Instance_Model_Graph_Mapping`

Proof. The corresponding proof follows directly from Definition 4.4.27 as well as Definition 4.4.26. Since the individual transformation functions f_A and f_B preserve the elements of their instance graphs when the

instance model is combined with another one, we can establish that the definition of $f_A \sqcup f_B$ is equal to the definition of combine(IG_A, IG_B). Therefore, $f_A \sqcup f_B(\text{combine}(Im_A, Im_B)) = \text{combine}(IG_A, IG_B)$. □

Although the presented theorem is a large step towards being able to build transformation functions from instance models to instance graphs iteratively, there is still one key element missing. It should be formally argued that $f_A \sqcup f_B$ is once again an transformation function in the sense of Definition 4.4.26. If this is formally argued, it becomes possible to easily combine $f_A \sqcup f_B$ with yet another transformation function. The following theorem states this property.

Theorem 4.4.30 (A combined transformation function from an instance model to an instance graph is a transformation function)
Let f_A and f_B be a transformation functions in the sense of Definition 4.4.26. f_A projects an instance model Im_A onto instance graph IG_A. f_B projects an instance model Im_B onto instance graph IG_B. Then the combination of f_A and f_B, $f_A \sqcup f_B$ is again a transformation function in the sense of Definition 4.4.26 which projects combine(Im_A, Im_B) *onto* combine(IG_A, IG_B).

Also see `ig_combine_mapping_function_correct` *in* `Ecore-GROOVE-Mapping.Instance_Model_Graph_Mapping`

Proof. Use Definition 4.4.26. Since the individual transformation functions f_A and f_B preserve the elements of their instance graphs when the instance model is combined with another one, we can establish that the definition of $f_A \sqcup f_B$ will also preserve these elements. This can be shown using the commutativity and associativity of the combination of instance models, see Theorem 4.4.6 and Theorem 4.4.7 respectively. □

This last theorem completes the recursive behaviour of combining transformation functions and therefore allows for building transformation functions from instance models to instance graphs iteratively.

The definitions and theorems that are presented so far only work in one direction: for transforming instance models into instance graphs. As visually shown in Figure 4.10, it must also be possible to transform instance graphs back into instance models. The definitions and theorems needed for this transformation are similar and will be presented in the remaining part of this section.

Definition 4.4.31 (Transformation function from an instance graph to an instance model)
Let f be a function from instance graphs to instance models, IG be an instance graph and Im the corresponding instance model. f is a transformation function iff:

- *f projects IG onto Im: $f(IG) = Im$;*
- *After combination with another instance graph, f preserves the type model;*
- *After combination with another instance graph, f preserves the objects:*
  $\forall IG_x : Object_{f(IG)} \subseteq Object_{f(\text{combine}(IG, IG_x))}$;
- *For all objects in the projected instance model, f preserves the object class if the instance graph is combined with another instance graph:*
  $\forall IG_x : \forall o \in Object_{f(IG)} : \text{ObjectClass}_{f(IG)}(o) = \text{ObjectClass}_{f(\text{combine}(IG, IG_x))}(o)$;
- *For all objects in the projected instance model, f preserves the object identifier if the instance graph is combined with another instance graph:*
  $\forall IG_x : \forall o \in Object_{f(IG)} : \text{ObjectId}_{f(IG)}(o) = \text{ObjectId}_{f(\text{combine}(IG, IG_x))}(o)$;
- *For all objects in the projected instance model and all fields in the type model corresponding to the projected instance model, f preserves the field value if the instance graph is combined with another instance graph:*
  $\forall IG_x : \forall o \in Object_{f(IG)} : \forall d \in Field_{Tm_{f(IG)}} :$
  $\text{FieldValue}_{f(IG)}((o, d)) = \text{FieldValue}_{f(\text{combine}(IG, IG_x))}((o, d))$;
- *For all constants in the type model corresponding to the projected instance model, f preserves the default value if the instance graph is combined with another instance graph:*
  $\forall IG_x : \forall c \in Constant_{Tm_{f(IG)}} : \text{DefaultValue}_{f(IG)}(c) = \text{DefaultValue}_{f(\text{combine}(IG, IG_x))}(c)$;

Also see `imod_combine_mapping_function` *in* `Ecore-GROOVE-Mapping.Instance_Model_Graph_Mapping`

Just like Definition 4.4.26, the definition of transformation functions from an instance graph to an instance model preserves all elements if the instance graph is combined with another instance graph. This will once more be the key to having the property of iterative building of transformation functions.

The following definition will describe how two transformation functions from instance graphs to instance models can be combined into a new transformation function, which projects the combination of two instance graphs onto the combination of the two corresponding instance models.

Definition 4.4.32 (Combination of transformation functions from an instance graph to an instance model)
Let f_A and f_B be a transformation functions in the sense of Definition 4.4.31. f_A projects an instance graph IG_A onto instance model Im_A. f_B projects an instance graph IG_B onto instance model Im_B. Then the combination of f_A and f_B is defined as:

$$
\begin{aligned}
f_A \sqcup f_B(IG) = \langle \quad & Object = \{o \mid o \in Object_{f_A(IG)} \wedge o \in Object_{Im_A}\} \cup \\
& \{o \mid o \in Object_{f_B(IG)} \wedge o \in Object_{Im_B}\} \\
& \text{ObjectClass} = \text{objectclass\_mapping}(f_A, Im_A, f_B, Im_B, IG) \\
& \text{ObjectId} = \text{objectid\_mapping}(f_A, Im_A, f_B, Im_B, IG) \\
& \text{FieldValue} = \text{fieldvalue\_mapping}(f_A, Im_A, f_B, Im_B, IG) \\
& \text{ConstType} = \text{consttype\_mapping}(f_A, Im_A, f_B, Im_B, IG) \\
\rangle &
\end{aligned}
$$

In which objectclass_mapping *is given as part of Definition 4.4.33,* objectid_mapping *as part of Definition 4.4.34,* fieldvalue_mapping *as part of Definition 4.4.36 and* defaultvalue_mapping *as part of Definition 4.4.35.*


Also see `imod_combine_mapping` *in* `Ecore-GROOVE-Mapping.Instance_Model_Graph_Mapping`

As expected, the definition for the combination of transformation functions from an instance graph to an instance model is an alternation of Definition 4.4.1. This alternation will once more allow the combined transformation function to project the combination of the instance graphs to the combination of the corresponding instance models.

The following four definitions will provide the remaining functions, which will closely follow their counterparts from Section 4.4.1.

Definition 4.4.33 (Combination of the object class function for two transformation functions)
objectclass_mapping$(f_A, Im_A, f_B, Im_B, IG)$ *is a partial function on two transformation functions f_A and f_B, their corresponding projections Im_A and Im_B and an instance model IG which returns a new function $Object_{f_A \sqcup f_B(IG)} \Rightarrow Class_{Tm_{f_A \sqcup f_B(IG)}}$. It is defined as follows:*

$$
\begin{aligned}
&\text{objectclass\_mapping}(f_A, Im_A, f_B, Im_B, IG, o) = \\
&\quad \begin{cases}
\text{ObjectClass}_{f_A(IG)}(o) & \text{if } o \in \{o \mid o \in Object_{f_A(IG)} \wedge o \in Object_{Im_A}\} \cap \\
& \quad \{o \mid o \in Object_{f_B(IG)} \wedge o \in Object_{Im_B}\} \wedge \\
& \quad \text{ObjectClass}_{f_A(IG)}(o) = \text{ObjectClass}_{f_B(IG)}(o) \\
\text{ObjectClass}_{f_A(IG)}(o) & \text{if } o \in \{o \mid o \in Object_{f_A(IG)} \wedge o \in Object_{Im_A}\} \setminus \\
& \quad \{o \mid o \in Object_{f_B(IG)} \wedge o \in Object_{Im_B}\} \\
\text{ObjectClass}_{f_B(IG)}(o) & \text{if } o \in \{o \mid o \in Object_{f_B(IG)} \wedge o \in Object_{Im_B}\} \setminus \\
& \quad \{o \mid o \in Object_{f_A(IG)} \wedge o \in Object_{Im_A}\}
\end{cases}
\end{aligned}
$$


Also see `imod_combine_object_class_mapping` *in* `Ecore-GROOVE-Mapping.Instance_Model_Graph_Mapping`

Definition 4.4.34 (Combination of the object identifier function for two transformation functions)
objectid_mapping$(f_A, Im_A, f_B, Im_B, IG)$ *is a partial function on two transformation functions f_A and f_B, their corresponding projections Im_A and Im_B and an instance model IG which returns a new*

function $Object_{f_A \sqcup f_B(IG)} \Rightarrow Name$. *It is defined as follows:*

$$\text{objectid\_mapping}(f_A, Im_A, f_B, Im_B, IG, o) =$$

$$\begin{cases} \text{ObjectId}_{f_A(IG)}(o) & \text{if } o \in \{o \mid o \in Object_{f_A(IG)} \wedge o \in Object_{Im_A}\} \cap \\ & \quad \{o \mid o \in Object_{f_B(IG)} \wedge o \in Object_{Im_B}\} \wedge \\ & \quad \text{ObjectId}_{f_A(IG)}(o) = \text{ObjectId}_{f_B(IG)}(o) \\ \text{ObjectId}_{f_A(IG)}(o) & \text{if } o \in \{o \mid o \in Object_{f_A(IG)} \wedge o \in Object_{Im_A}\} \setminus \\ & \quad \{o \mid o \in Object_{f_B(IG)} \wedge o \in Object_{Im_B}\} \\ \text{ObjectId}_{f_B(IG)}(o) & \text{if } o \in \{o \mid o \in Object_{f_B(IG)} \wedge o \in Object_{Im_B}\} \setminus \\ & \quad \{o \mid o \in Object_{f_A(IG)} \wedge o \in Object_{Im_A}\} \end{cases}$$

 *Also see* `imod_combine_object_id_mapping` *in* `Ecore-GROOVE-Mapping.Instance_Model_Graph_Mapping`

Definition 4.4.35 (Combination of the default value function for two transformation functions) defaultvalue_mapping$(f_A, Im_A, f_B, Im_B, IG)$ *is a partial function on two transformation functions* f_A *and* f_B*, their corresponding projections* Im_A *and* Im_B *and an instance model* IG *which returns a new function* $Constant_{Tm_{f_A \sqcup f_B(IG)}} \Rightarrow Value_{f_A \sqcup f_B(IG)}$. *It is defined as follows:*

$$\text{defaultvalue\_mapping}(f_A, Im_A, f_B, Im_B, IG, c) =$$

$$\begin{cases} \text{DefaultValue}_{f_A(IG)}(c) & \text{if } c \in \{c \mid c \in Constant_{Tm_{f_A(IG)}} \wedge c \in Constant_{Tm_A}\} \cap \\ & \quad \{c \mid c \in Constant_{Tm_{f_B(IG)}} \wedge c \in Constant_{Tm_B}\} \wedge \\ & \quad \text{DefaultValue}_{f_A(IG)}(c) = \text{DefaultValue}_{f_B(IG)}(c) \\ \text{DefaultValue}_{f_A(IG)}(c) & \text{if } c \in \{c \mid c \in Constant_{Tm_{f_A(IG)}} \wedge c \in Constant_{Tm_A}\} \setminus \\ & \quad \{c \mid c \in Constant_{Tm_{f_B(IG)}} \wedge c \in Constant_{Tm_B}\} \\ \text{DefaultValue}_{f_B(IG)}(c) & \text{if } c \in \{c \mid c \in Constant_{Tm_{f_B(IG)}} \wedge c \in Constant_{Tm_B}\} \setminus \\ & \quad \{c \mid c \in Constant_{Tm_{f_A(IG)}} \wedge c \in Constant_{Tm_A}\} \end{cases}$$

 Also see `imod_combine_default_value_mapping` *in* `Ecore-GROOVE-Mapping.Instance_Model_Graph_Mapping`

Definition 4.4.36 (Combination of the field value function for two transformation functions) fieldvalue_mapping$(f_A, Im_A, f_B, Im_B, IG)$ *is a partial function on two transformation functions* f_A *and* f_B*, their corresponding projections* Im_A *and* Im_B *and an instance model* IG *which returns a new function* $(Object_{f_A \sqcup f_B(IG)} \times Field_{Tm_{f_A \sqcup f_B(IG)}}) \Rightarrow Value_{f_A \sqcup f_B(IG)}$. *It is defined as follows:*

$$\text{fieldvalue\_mapping}(f_A, Im_A, f_B, Im_B, IG, (o, d)) =$$

$$\begin{cases} \text{FieldValue}_{f_A(IG)}((o,d)) & \text{if } o \in \{o \mid o \in Object_{f_A(IG)} \wedge o \in Object_{Im_A}\} \cap \\ & \quad \{o \mid o \in Object_{f_B(IG)} \wedge o \in Object_{Im_B}\} \wedge \\ & \quad d \in \{d \mid d \in \text{fields}_{Tm_{f_A(IG)}}(\text{ObjectClass}_{f_A(IG)}(o)) \wedge \\ & \quad\quad d \in \text{fields}_{Tm_A}(\text{ObjectClass}_{f_A(IG)}(o))\} \wedge \\ & \quad d \in \{d \mid d \in \text{fields}_{Tm_{f_B(IG)}}(\text{ObjectClass}_{f_B(IG)}(o)) \wedge \\ & \quad\quad d \in \text{fields}_{Tm_B}(\text{ObjectClass}_{f_B(IG)}(o))\} \wedge \\ & \quad \text{FieldValue}_{f_A(IG)}((o,d)) = \text{FieldValue}_{f_B(IG)}((o,d)) \\ \text{FieldValue}_{f_A(IG)}((o,d)) & \text{if } o \in \{o \mid o \in Object_{f_A(IG)} \wedge o \in Object_{Im_A}\} \wedge \\ & \quad d \in \{d \mid d \in \text{fields}_{Tm_{f_A(IG)}}(\text{ObjectClass}_{f_A(IG)}(o)) \wedge \\ & \quad\quad d \in \text{fields}_{Tm_A}(\text{ObjectClass}_{f_A(IG)}(o))\} \wedge \\ & \quad (o \notin \{o \mid o \in Object_{f_B(IG)} \wedge o \in Object_{Im_B}\} \vee \\ & \quad d \notin \{d \mid d \in \text{fields}_{Tm_{f_B(IG)}}(\text{ObjectClass}_{f_B(IG)}(o)) \wedge \\ & \quad\quad d \in \text{fields}_{Tm_B}(\text{ObjectClass}_{f_B(IG)}(o))\} \\ \text{FieldValue}_{f_B(IG)}((o,d)) & \text{if } o \in \{o \mid o \in Object_{f_B(IG)} \wedge o \in Object_{Im_B}\} \wedge \\ & \quad d \in \{d \mid d \in \text{fields}_{Tm_{f_B(IG)}}(\text{ObjectClass}_{f_B(IG)}(o)) \wedge \\ & \quad\quad d \in \text{fields}_{Tm_B}(\text{ObjectClass}_{f_B(IG)}(o))\} \wedge \\ & \quad (o \notin \{o \mid o \in Object_{f_A(IG)} \wedge o \in Object_{Im_A}\} \vee \\ & \quad d \notin \{d \mid d \in \text{fields}_{Tm_{f_A(IG)}}(\text{ObjectClass}_{f_A(IG)}(o)) \wedge \\ & \quad\quad d \in \text{fields}_{Tm_A}(\text{ObjectClass}_{f_A(IG)}(o))\} \end{cases}$$

Also see `imod_combine_field_value_mapping` *in* `Ecore-GROOVE-Mapping.Instance_Model_Graph_Mapping`

The definitions of most of these combination functions are straightforward. Only the fieldvalue_mapping seems significantly more complicated. However, a careful ready will still be able to see that the definition is an alternation of Definition 4.4.4. The definition seems more complicated because of the combination of objects and fields in the domain of the function.

With these definitions in place, it is possible to provide the necessary theorems for the correctness of the combined function $f_A \sqcup f_B$.

Theorem 4.4.37 (The projection of a combined transformation function from an instance graph to an instance model)
Let f_A and f_B be a transformation functions in the sense of Definition 4.4.31. f_A projects an instance graph IG_A onto instance model Im_A. f_B projects an instance model IG_B onto instance graph Im_B. Then the combination of f_A and f_B, $f_A \sqcup f_B$ projects combine(IG_A, IG_B) *onto* combine(Im_A, Im_B)*, so:*

$$f_A \sqcup f_B(\text{combine}(IG_A, IG_B)) = \text{combine}(Im_A, Im_B)$$

Also see `imod_combine_mapping_correct` *in* `Ecore-GROOVE-Mapping.Instance_Model_Graph_Mapping`

Proof. The corresponding proof follows directly from Definition 4.4.32 as well as Definition 4.4.31. Since the individual transformation functions f_A and f_B preserve the elements of their instance models when the instance graph is combined with another one, we can establish that the definition of $f_A \sqcup f_B$ is equal to the definition of combine(Im_A, Im_B). Therefore, $f_A \sqcup f_B(\text{combine}(IG_A, IG_B)) = \text{combine}(Im_A, Im_B)$. □

Like the combined transformation function from instance models to instance graphs, the combined transformation function from instance graphs to instance models is also a transformation function, but in the sense of Definition 4.4.31. This is stated in the following theorem.

Theorem 4.4.38 (A combined transformation function from an instance graph to an instance model is a transformation function)
Let f_A and f_B be a transformation functions in the sense of Definition 4.4.31. f_A projects an instance graph IG_A onto instance model Im_A. f_B projects an instance graph IG_B onto instance model Im_B. Then the combination of f_A and f_B, $f_A \sqcup f_B$ is again a transformation function in the sense of Definition 4.4.31 which projects combine(IG_A, IG_B) *onto* combine(Im_A, Im_B).

Also see `imod_combine_mapping_function_correct` *in* `Ecore-GROOVE-Mapping.Instance_Model_Graph_Mapping`

Proof. Use Definition 4.4.31. Since the individual transformation functions f_A and f_B preserve the elements of their instance models when the instance graph is combined with another one, we can establish that the definition of $f_A \sqcup f_B$ will also preserve these elements. This can be shown using the commutativity and associativity of the combination of instance graphs, see Theorem 4.4.17 and Theorem 4.4.18 respectively. □

Chapter 5

Library of transformations

The previous chapter introduced a framework to reason about composable model transformations formally. This framework does allow one to create models from scratch by iteratively adding different elements. Although the previous chapter did introduce the needed definitions and theorems to explain this framework, it did not discuss how to apply the framework in practice. In order to apply the framework, the framework needs two essential ingredients. First of all, the framework needs a set of proven model transformations that can be used to build models iteratively. Finally, these proven model transformations must be applied within the framework to create some larger model.

In this chapter, a set of small model transformations between GROOVE and Ecore is introduced. These model transformations can be used within the framework as 'building blocks', which are used to create larger models. The goal of this chapter is to show possible model transformations that can be applied with the framework from the previous chapter, in order to validate the framework using an application of these transformations. Therefore, the set of model transformations presented in this chapter will be non-exhaustive. It will only include transformations that are necessary for the application, as well as some transformations that show the potential of the framework.

This chapter is split into two parts. Within the first part, a set of model transformations between type models and type graphs is introduced. This set is used within the second part, where a set of model transformations between instance models and instance graphs is introduced. However, before any of these sets is discussed, some definitions used throughout this chapter are introduced.

5.1 Definitions

This section introduces some general definitions that are used within the model transformations of this chapter. They are introduced before the actual transformations for readability and to prevent repeating the same definitions as part of the transformations themselves.

Throughout this chapter, many sequences are used. Sequences, sometimes also called lists, are enumerated collections of objects in which repetitions are allowed. Each element in a sequence is a member of that sequence. Moreover, each element has a corresponding index that represents the position of the element within the sequence. For sequences, some definitions are defined to make it easier to reason about them.

Definition 5.1.1 (Prefix operator for sequences)
Assume $s = \langle m_1, m_2, ..., m_n \rangle$ to be a sequence. Then define $\#$ as the prefix operator on sequences, which adds an elements e to the beginning of the sequence s.

$$e \# s = e \# \langle m_1, m_2, ..., m_n \rangle = \langle e, m_1, m_2, ..., m_n \rangle$$

Please note that for every element belonging to index i, after adding e, the element belonging to index i will belong to index $i + 1$:

$$\begin{aligned} s_i &= m_i \\ (e \# s)_1 &= e \\ (e \# s)_{(i+1)} &= m_i \end{aligned}$$

Definition 5.1.2 (Append operator for sequences)
Assume $s = \langle m_1, m_2, ..., m_i \rangle$ *and* $t = \langle n_1, n_2, ..., n_j \rangle$ *to be sequences. Then define* $s@t$ *as an operator on two sequences, which appends sequence* t *to sequence* s*.*

$$s \mathbin{@} t = \langle m_1, m_2, ..., m_i, n_1, n_2, ..., n_j \rangle$$

The following holds for the indexes of $s \mathbin{@} t$*:*

$$(s \mathbin{@} t)_{1 \leq i \leq |s|} = m_i$$
$$(s \mathbin{@} t)_{|s|+1 \leq i \leq |s|+|t|} = n_i$$

Using the definitions on sequences, it becomes possible to define the transformation of identifiers and namespaces (see Definition 3.2.2). As explained earlier, Ecore uses the concepts of identifiers and namespaces to distinguish classes, enumeration types and user-defined data types. In GROOVE, these concepts do not exist, though it must be possible to express identifiers and namespaces in GROOVE. Therefore, a definition will be provided that allows for transforming identifiers and namespaces into sequences, and back. This definition will be used throughout this chapter to transform namespaces and identifiers.

Definition 5.1.3 (Transformation of namespaces to sequences)
Assume n *to be a valid namespace, in the sense of Definition 3.2.2. Then the recursive function* ns_to_list(n) *transforms a namespace into a sequence:*

$$\text{ns\_to\_list}(n) = \begin{cases} \text{ns\_to\_list}(ns) \mathbin{@} \langle name \rangle & \text{if } n = \langle ns, name \rangle \\ \langle\rangle & \text{if } n = \bot \end{cases}$$

Also see `ns_to_list` *in* `Ecore-GROOVE-Mapping.Namespace_List`

According to this definition .some.namespace.name is transformed into ⟨some, namespace, name⟩. Furthermore, the top namespace is transformed into the empty sequence, ⟨⟩, as directly defined as part of the definition.

Besides a definition to transform namespaces into sequences, there is also a transformation from sequences back to namespaces.

Definition 5.1.4 (Transformation of sequences to namespaces)
Assume s *to be a sequence of names. Then* list_to_ns(s) *is defined as the inverse function of* ns_to_list*, so it transforms sequences into their corresponding namespace.*

Also see `list_to_ns` *in* `Ecore-GROOVE-Mapping.Namespace_List`

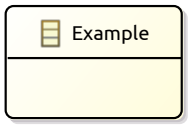
By definition, list_to_ns will convert the sequence ⟨some, namespace, name⟩) back into its corresponding namespace, ⟨some, namespace, name⟩. Furthermore, the empty sequence ⟨⟩ is converted to the top namespace ⊥.

5.2 Type level transformations

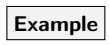
This section represents the first part of the library of transformations, as explained in the introduction of this chapter. Throughout this section, small transformations between type models and type graphs will be defined. In order for these transformations useful in the context of the transformation framework of Chapter 4, some properties must hold for each of them. For each transformation, the corresponding type model must be consistent in the sense of Definition 3.2.11 and the corresponding type graph must be valid in the sense of Definition 3.3.5. Furthermore, to be able to apply the transformation, the type model must be compatible with its counterpart in the transformation framework. In the same way, the type graph corresponding to the transformation must be compatible with its counterpart in the transformation framework. Moreover, it will be shown that the transformation function f that transforms the corresponding type model into a type graph is a valid transformation function in the sense of Definition 4.3.25. Finally, it will also be shown that the reverse transformation is a valid transformation function in the sense of Definition 4.3.30.

5.2.1 Regular classes

The first transformation that will be defined is a transformation of regular classes. A class without any additional properties is considered a regular class. The Ecore model for defining a regular class is given in the following definition:



(a) Tm_{Class} with $name$ = .Example



(b) TG_{Class} with $name$ = .Example

Figure 5.1: Visualisation of the transformation of regular classes

Definition 5.2.1 (Type model Tm_{Class})
Let Tm_{Class} be the type model containing a regular class with identifier name. Tm_{Class} is defined as:

$$\begin{aligned} Class &= \{name\} \\ Enum &= \{\} \\ UserDataType &= \{\} \\ Field &= \{\} \\ \text{FieldSig} &= \{\} \\ EnumValue &= \{\} \\ Inh &= \{\} \\ Prop &= \{\} \\ Constant &= \{\} \\ \text{ConstType} &= \{\} \end{aligned}$$

Also see `tmod_class` *in* `Ecore-GROOVE-Mapping-Library.ClassType`

Theorem 5.2.2 (Correctness of Tm_{Class})
Tm_{Class} (Definition 5.2.1) is a consistent type model in the sense of Definition 3.2.11.

Also see `tmod_class_correct` *in* `Ecore-GROOVE-Mapping-Library.ClassType`

A visual representation of Tm_{Class} with identifier .Example can be seen in Figure 5.1a. The correctness proof of Tm_{Class} is trivial, and therefore not included here. The proof can be found as part of the Isabelle validated proofs.

In order to make composing transformation functions possible, Tm_{Class} should be compatible with the type model it is combined with.

Theorem 5.2.3 (Correctness of combine(Tm, Tm_{Class}))
Assume a type model Tm that is consistent in the sense of Definition 3.2.11. Then Tm is compatible with Tm_{Class} (in the sense of Definition 4.3.13) if:

- *The identifier of the class in Tm_{Class} is not yet an identifier for a class, enumeration type or user-defined data type in Tm;*
- *The identifier of the class in Tm_{Class} is not in the namespace of any class, enumeration type or user-defined data type in Tm;*
- *None of the identifiers in any class, enumeration type or user-defined data type in Tm is in the namespace of the class in Tm_{Class}.*

Also see `tmod_class_combine_correct` *in* `Ecore-GROOVE-Mapping-Library.ClassType`

Proof. Use Lemma 4.3.12. It is possible to show that all assumptions hold. Now we have shown that combine(Tm, Tm_{Class}) is consistent in the sense of Definition 3.2.11. □

The definitions and theorems for a regular class within Ecore are now complete.

Encoding as node type

A possible encoding for regular classes in Ecore is using a node type in GROOVE. This node type will get a transformed identifier as name. The encoding corresponding to Tm_{Class} can then be represented as TG_{Class}, defined in the following definition:

Definition 5.2.4 (Type graph TG_{Class})
Let TG_{Class} be the type graph containing a single node type which encodes a regular class name. TG_{Class} is defined as:

$$\begin{aligned} NT &= \{\text{ns\_to\_list}(name)\} \\ ET &= \{\} \\ \sqsubseteq &= \{(\text{ns\_to\_list}(name), \text{ns\_to\_list}(name))\} \\ abs &= \{\} \\ \text{mult} &= \{\} \\ contains &= \{\} \end{aligned}$$

Also see `tg_class_as_node_type` *in* `Ecore-GROOVE-Mapping-Library.ClassType`

Theorem 5.2.5 (Correctness of TG_{Class})
TG_{Class} (Definition 5.2.4) is a valid type graph in the sense of Definition 3.3.5.

Also see `tg_class_as_node_type_correct` *in* `Ecore-GROOVE-Mapping-Library.ClassType`

A visual representation of TG_{Class} with identifier .Example can be seen in Figure 5.1b. The correctness proof of TG_{Class} is trivial, and therefore not included here. The proof can be found as part of the Isabelle validated proofs.

In order to make composing transformation functions possible, TG_{Class} should be compatible with the type graph it is combined with.

Theorem 5.2.6 (Correctness of combine(TG, TG_{Class}))
Assume a type graph TG that is valid in the sense of Definition 3.3.5. Then TG is compatible with TG_{Class} (in the sense of Definition 4.3.24) if:

- *The node type of the encoded class in TG_{Class} is not a node type in TG.*

Also see `tg_class_as_node_type_combine_correct` *in* `Ecore-GROOVE-Mapping-Library.ClassType`

Proof. Use Lemma 4.3.23. It is possible to show that all assumptions hold. Now we have shown that combine(TG, TG_{Class}) is valid in the sense of Definition 3.3.5. □

The next definitions define the transformation function from Tm_{Class} to TG_{Class}:

Definition 5.2.7 (Transformation function f_{Class})
The transformation function $f_{Class}(Tm)$ is defined as:

$$\begin{aligned} NT &= \{\text{ns\_to\_list}(c) \mid c \in Class_{Tm}\} \\ ET &= \{\} \\ \sqsubseteq &= \{(\text{ns\_to\_list}(c_1), \text{ns\_to\_list}(c_2)) \mid c_1 \in Class_{Tm} \wedge c_2 \in Class_{Tm}\} \\ abs &= \{\} \\ \text{mult} &= \{\} \\ contains &= \{\} \end{aligned}$$

Also see `tmod_class_to_tg_class_as_node_type` *in* `Ecore-GROOVE-Mapping-Library.ClassType`

Theorem 5.2.8 (Correctness of f_{Class})
$f_{Class}(Tm)$ (Definition 5.2.7) is a valid transformation function in the sense of Definition 4.3.25 transforming Tm_{Class} into TG_{Class}.

Also see `tmod_class_to_tg_class_as_node_type_func` *in* `Ecore-GROOVE-Mapping-Library.ClassType`

The proof of the correctness of f_{Class} will not be included here. Instead, it can be found in the validated Isabelle theories.

Finally, to complete the transformation, the transformation function that transforms TG_{Class} into Tm_{Class} is defined:

Definition 5.2.9 (Transformation function f'_{Class})
The transformation function $f'_{Class}(TG)$ is defined as:

$$\begin{aligned} Class &= \{\text{list\_to\_ns}(n) \mid n \in NT_{TG}\} \\ Enum &= \{\} \\ UserDataType &= \{\} \\ Field &= \{\} \\ \text{FieldSig} &= \{\} \\ EnumValue &= \{\} \\ Inh &= \{\} \\ Prop &= \{\} \\ Constant &= \{\} \\ \text{ConstType} &= \{\} \end{aligned}$$

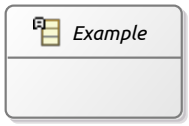
Also see `tg_class_as_node_type_to_tmod_class` *in* `Ecore-GROOVE-Mapping-Library.ClassType`

Theorem 5.2.10 (Correctness of f'_{Class})
$f'_{Class}(TG)$ (Definition 5.2.9) is a valid transformation function in the sense of Definition 4.3.30 transforming TG_{Class} into Tm_{Class}.

Also see `tg_class_as_node_type_to_tmod_class_func` *in* `Ecore-GROOVE-Mapping-Library.ClassType`

Once more, the correctness proof is not included here but can be found in the validated Isabelle proofs of this thesis.

5.2.2 Abstract classes



(a) $Tm_{AbsClass}$ with $name =$ .Example

(b) $TG_{AbsClass}$ with $name =$ .Example

Figure 5.2: Visualisation of the transformation of abstract classes

This section defines a transformation that is very close to the previous transformation. In this section, an abstract class without any additional properties is defined. The Ecore model for defining an abstract class is given in the following definition:

Definition 5.2.11 (Type model $Tm_{AbsClass}$)
Let $Tm_{AbsClass}$ be the type model containing an abstract class with identifier name. $Tm_{AbsClass}$ is defined as:

$$\begin{aligned} Class &= \{name\} \\ Enum &= \{\} \\ UserDataType &= \{\} \\ Field &= \{\} \\ \text{FieldSig} &= \{\} \\ EnumValue &= \{\} \\ Inh &= \{\} \\ Prop &= \{[\text{abstract}, name]\} \\ Constant &= \{\} \\ \text{ConstType} &= \{\} \end{aligned}$$

Also see `tmod_abstract_class` *in* `Ecore-GROOVE-Mapping-Library.AbstractClassType`

Theorem 5.2.12 (Correctness of $Tm_{AbsClass}$)
$Tm_{AbsClass}$ (Definition 5.2.11) is a consistent type model in the sense of Definition 3.2.11.

Also see `tmod_abstract_class_correct` *in* `Ecore-GROOVE-Mapping-Library.AbstractClassType`

A visual representation of $Tm_{AbsClass}$ with identifier .Example can be seen in Figure 5.2a. The correctness proof of $Tm_{AbsClass}$ is trivial, and therefore not included here. The proof can be found as part of the Isabelle validated proofs.

In order to make composing transformation functions possible, $Tm_{AbsClass}$ should be compatible with the type model it is combined with.

Theorem 5.2.13 (Correctness of combine$(Tm, Tm_{AbsClass})$)
Assume a type model Tm that is consistent in the sense of Definition 3.2.11. Then Tm is compatible with $Tm_{AbsClass}$ (in the sense of Definition 4.3.13) if:

- *The identifier of the class in $Tm_{AbsClass}$ is not yet an identifier for a class, enumeration type or user-defined data type in Tm;*
- *The identifier of the class in $Tm_{AbsClass}$ is not in the namespace of any class, enumeration type or user-defined data type in Tm;*
- *None of the identifiers in any class, enumeration type or user-defined data type in Tm is in the namespace of the class in $Tm_{AbsClass}$.*

Also see `tmod_abstract_class_combine_correct` *in* `Ecore-GROOVE-Mapping-Library.AbstractClassType`

Proof. Use Lemma 4.3.12. It is possible to show that all assumptions hold. Now we have shown that combine$(Tm, Tm_{AbsClass})$ is consistent in the sense of Definition 3.2.11. □

The definitions and theorems for a regular class within Ecore are now complete.

Encoding as node type

A possible encoding for abstract classes in Ecore is using a node type in GROOVE. This node type will get a transformed identifier as name. The encoding corresponding to $Tm_{AbsClass}$ can then be represented as $TG_{AbsClass}$, defined in the following definition:

Definition 5.2.14 (Type graph $TG_{AbsClass}$)
Let $TG_{AbsClass}$ be the type graph containing a single node type which encodes an abstract class name. $TG_{AbsClass}$ is defined as:

$$\begin{aligned} NT &= \{\text{ns\_to\_list}(name)\} \\ ET &= \{\} \\ \sqsubseteq &= \{(\text{ns\_to\_list}(name), \text{ns\_to\_list}(name))\} \\ abs &= \{\text{ns\_to\_list}(name)\} \\ \text{mult} &= \{\} \\ contains &= \{\} \end{aligned}$$

Also see `tg_abstract_class_as_node_type` *in* `Ecore-GROOVE-Mapping-Library.AbstractClassType`

Theorem 5.2.15 (Correctness of TG_{Class})
$TG_{AbsClass}$ (Definition 5.2.14) is a valid type graph in the sense of Definition 3.3.5.

Also see `tg_abstract_class_as_node_type_correct` *in* `Ecore-GROOVE-Mapping-Library.AbstractClassType`

A visual representation of $TG_{AbsClass}$ with identifier .Example can be seen in Figure 5.2b. The correctness proof of $TG_{AbsClass}$ is trivial, and therefore not included here. The proof can be found as part of the Isabelle validated proofs.

In order to make composing transformation functions possible, $TG_{AbsClass}$ should be compatible with the type graph it is combined with.

Theorem 5.2.16 (Correctness of combine$(TG, TG_{AbsClass})$)
Assume a type graph TG that is valid in the sense of Definition 3.3.5. Then TG is compatible with $TG_{AbsClass}$ (in the sense of Definition 4.3.24) if:

- *The node type of the encoded class in $TG_{AbsClass}$ is not a node type in TG.*

 *Also see* `tg_abstract_class_as_node_type_combine_correct` *in* `Ecore-GROOVE-Mapping-Library.AbstractClassType`

Proof. Use Lemma 4.3.23. It is possible to show that all assumptions hold. Now we have shown that combine$(TG, TG_{AbsClass})$ is valid in the sense of Definition 3.3.5. □

The next definitions define the transformation function from $Tm_{AbsClass}$ to $TG_{AbsClass}$:

Definition 5.2.17 (Transformation function $f_{AbsClass}$)
The transformation function $f_{AbsClass}(Tm)$ is defined as:

$$\begin{aligned} NT &= \{\text{ns\_to\_list}(c) \mid c \in Class_{Tm}\} \\ ET &= \{\} \\ \sqsubseteq &= \{(\text{ns\_to\_list}(c_1), \text{ns\_to\_list}(c_2)) \mid c_1 \in Class_{Tm} \wedge c_2 \in Class_{Tm}\} \\ abs &= \{\text{ns\_to\_list}(c) \mid c \in Class_{Tm}\} \\ \text{mult} &= \{\} \\ contains &= \{\} \end{aligned}$$

 *Also see* `tmod_abstract_class_to_tg_abstract_class_as_node_type` *in* `Ecore-GROOVE-Mapping-Library.AbstractClassType`

Theorem 5.2.18 (Correctness of $f_{AbsClass}$)
$f_{AbsClass}(Tm)$ (Definition 5.2.17) is a valid transformation function in the sense of Definition 4.3.25 transforming $Tm_{AbsClass}$ into $TG_{AbsClass}$.

 *Also see* `tmod_abstract_class_to_tg_abstract_class_as_node_type_func` *in* `Ecore-GROOVE-Mapping-Library.AbstractClassType`

The proof of the correctness of $f_{AbsClass}$ will not be included here. Instead, it can be found in the validated Isabelle theories.

Finally, to complete the transformation, the transformation function that transforms $TG_{AbsClass}$ into $Tm_{AbsClass}$ is defined:

Definition 5.2.19 (Transformation function $f'_{AbsClass}$)
The transformation function $f'_{AbsClass}(TG)$ is defined as:

$$\begin{aligned} Class &= \{\text{list\_to\_ns}(n) \mid n \in NT_{TG}\} \\ Enum &= \{\} \\ UserDataType &= \{\} \\ Field &= \{\} \\ \text{FieldSig} &= \{\} \\ EnumValue &= \{\} \\ Inh &= \{\} \\ Prop &= \{[\mathsf{abstract}, \text{ns\_to\_list}(c)] \mid c \in Class_{Tm}\} \\ Constant &= \{\} \\ \text{ConstType} &= \{\} \end{aligned}$$

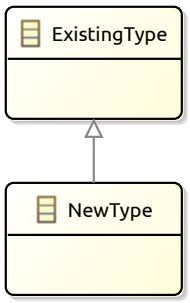
 *Also see* `tg_abstract_class_as_node_type_to_tmod_abstract_class` *in* `Ecore-GROOVE-Mapping-Library.AbstractClassType`

Theorem 5.2.20 (Correctness of $f'_{AbsClass}$)
$f'_{AbsClass}(TG)$ (Definition 5.2.19) is a valid transformation function in the sense of Definition 4.3.30 transforming $TG_{AbsClass}$ into $Tm_{AbsClass}$.

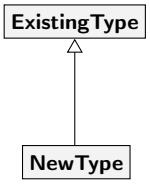
Also see `tg_abstract_class_as_node_type_to_tmod_abstract_class_func` *in* `Ecore-GROOVE-Mapping-Library.AbstractClassType`

Once more, the correctness proof is not included here but can be found in the validated Isabelle proofs of this thesis.

5.2.3 Regular subclasses



(a) $Tm_{Subclass}$ with $name$ = .NewType and $supertype$ = .ExistingType



(b) $TG_{Subclass}$ with $name$ = .NewType and $supertype$ = .ExistingType

Figure 5.3: Visualisation of the transformation of regular subclasses

This section will define the transformation of a regular subclass. Within this transformation, a newly introduced subclass is transformed, which extends an existing supertype. The Ecore type model that introduces such a subclass is defined as follows:

Definition 5.2.21 (Type model $Tm_{Subclass}$)
Let $Tm_{Subclass}$ be the type model containing a regular class with identifier name. The regular class name extends another regular class with identifier supertype. Furthermore, name $\neq$ supertype. $Tm_{Subclass}$ is defined as:

$$\begin{aligned} Class &= \{name, supertype\} \\ Enum &= \{\} \\ UserDataType &= \{\} \\ Field &= \{\} \\ \text{FieldSig} &= \{\} \\ EnumValue &= \{\} \\ Inh &= \{(name, supertype)\} \\ Prop &= \{\} \\ Constant &= \{\} \\ \text{ConstType} &= \{\} \end{aligned}$$

Also see `tmod_subclass` *in* `Ecore-GROOVE-Mapping-Library.SubclassType`

Theorem 5.2.22 (Correctness of $Tm_{Subclass}$)
$Tm_{Subclass}$ (Definition 5.2.21) is a consistent type model in the sense of Definition 3.2.11.

Also see `tmod_subclass_correct` *in* `Ecore-GROOVE-Mapping-Library.SubclassType`

A visual representation of $Tm_{Subclass}$ with the new subclass identified as .NewType and the existing supertype identified as .ExistingType can be seen in Figure 5.3a. The correctness proof of $Tm_{Subclass}$ is trivial, and therefore not included here. The proof can be found as part of the Isabelle validated proofs.

In order to make composing transformation functions possible, $Tm_{Subclass}$ should be compatible with the type model it is combined with.

Theorem 5.2.23 (Correctness of combine$(Tm, Tm_{Subclass})$)
Assume a type model Tm that is consistent in the sense of Definition 3.2.11. Then Tm is compatible with $Tm_{Subclass}$ (in the sense of Definition 4.3.13) if:

- *The only shared type is the supertype, so* $Class_{Tm} \cap Class_{Tm_{Subclass}} = \{supertype\}$*;*
- *The class name is not in the namespace of class supertype, and vice versa;*
- *name is not used as an identifier for an enumeration type or user-defined data type in* Tm*;*
- *The identifier of the class name in* Tm_{Class} *is not in the namespace of any class, enumeration type or user-defined data type in* Tm*;*
- *None of the identifiers in any class, enumeration type or user-defined data type in* Tm *is in the namespace of the class name in* Tm_{Class}*.*

Also see `tmod_subclass_combine_correct` *in* `Ecore-GROOVE-Mapping-Library.SubclassType`

Proof. Use Lemma 4.3.12. It is possible to show that all assumptions hold. For proving that the transitive closure of the inheritance relation is irreflexive, use the fact that *name* only appears in the domain of the relation. Now we have shown that combine(Tm, Tm_{Class}) is consistent in the sense of Definition 3.2.11. □

The definitions and theorems for a regular subclass within Ecore are now complete.

Encoding as node type

A possible encoding for regular subclasses in Ecore is using node types in GROOVE. The supertype and newly introduced subtype will both be node types with their corresponding identifiers transformed. The encoding corresponding to $Tm_{Subclass}$ can then be represented as $TG_{Subclass}$, defined in the following definition:

Definition 5.2.24 (Type graph $TG_{Subclass}$)
Let $TG_{Subclass}$ *be a type graph containing a two node types. The first node type encodes the regular class supertype. The second node type encodes a regular class name which extends the encoded class supertype. Furthermore* $name \neq supertype$*.* $TG_{Subclass}$ *is defined as:*

$$\begin{aligned} NT &= \{\text{ns\_to\_list}(name), \text{ns\_to\_list}(supertype)\} \\ ET &= \{\} \\ \sqsubseteq &= \{(\text{ns\_to\_list}(name), \text{ns\_to\_list}(name)), \\ &\quad (\text{ns\_to\_list}(supertype), \text{ns\_to\_list}(supertype)), \\ &\quad (\text{ns\_to\_list}(name), \text{ns\_to\_list}(supertype))\} \\ abs &= \{\} \\ \text{mult} &= \{\} \\ contains &= \{\} \end{aligned}$$

Also see `tg_subclass_as_node_type` *in* `Ecore-GROOVE-Mapping-Library.SubclassType`

Theorem 5.2.25 (Correctness of $TG_{Subclass}$)
$TG_{Subclass}$ *(Definition 5.2.24) is a valid type graph in the sense of Definition 3.3.5.*

Also see `tg_subclass_as_node_type_correct` *in* `Ecore-GROOVE-Mapping-Library.SubclassType`

A visual representation of $TG_{Subclass}$ with the new subclass identified as .NewType and the existing supertype identified as .ExistingType both encoded as node type, is shown in Figure 5.3b. The correctness proof of $TG_{Subclass}$ is trivial, and therefore not included here. The proof can be found as part of the Isabelle validated proofs.

In order to make composing transformation functions possible, $TG_{Subclass}$ should be compatible with the type graph it is combined with.

Theorem 5.2.26 (Correctness of combine$(TG, TG_{Subclass})$)
Assume a type graph TG that is valid in the sense of Definition 3.3.5. Then TG is compatible with $TG_{Subclass}$ *(in the sense of Definition 4.3.24) if:*

- *The only shared node type in TG and* $TG_{Subclass}$ *is the node type of the encoded supertype.*

Also see `tg_subclass_as_node_type_combine_correct` *in* `Ecore-GROOVE-Mapping-Library.SubclassType`

Proof. Use Lemma 4.3.23. It is possible to show that all assumptions hold. For proving the antisymmetry of the inheritance relation, use the fact that the node type which encodes class *name* only appears in the domain of the relation. Now we have shown that combine$(TG, TG_{Subclass})$ is valid in the sense of Definition 3.3.5. □

The next definitions define the transformation function from $Tm_{Sublass}$ to $TG_{Sublass}$:

Definition 5.2.27 (Transformation function $f_{Subclass}$)
The transformation function $f_{Subclass}(Tm)$ is defined as:

$$\begin{aligned} NT &= \{\text{ns\_to\_list}(c) \mid c \in Class_{Tm}\} \\ ET &= \{\} \\ \sqsubseteq &= \{(\text{ns\_to\_list}(c_1), \text{ns\_to\_list}(c_2)) \mid c_1 \in Class_{Tm} \wedge c_2 \in Class_{Tm} \wedge c_1 = c_2\} \cup \\ &\quad \{(\text{ns\_to\_list}(i), \text{ns\_to\_list}(j)) \mid (i,j) \in Inh_{Tm}\} \\ abs &= \{\} \\ \text{mult} &= \{\} \\ contains &= \{\} \end{aligned}$$

Also see `tmod_subclass_to_tg_subclass_as_node_type` *in* `Ecore-GROOVE-Mapping-Library.SubclassType`

Theorem 5.2.28 (Correctness of $f_{Subclass}$)
$f_{Subclass}(Tm)$ (Definition 5.2.27) is a valid transformation function in the sense of Definition 4.3.25 transforming $Tm_{Subclass}$ into $TG_{Subclass}$.

Also see `tmod_subclass_to_tg_subclass_as_node_type_func` *in* `Ecore-GROOVE-Mapping-Library.SubclassType`

The proof of the correctness of $f_{Subclass}$ will not be included here. Instead, it can be found in the validated Isabelle theories.

Finally, to complete the transformation, the transformation function that transforms $TG_{Subclass}$ into $Tm_{Subclass}$ is defined:

Definition 5.2.29 (Transformation function $f'_{Subclass}$)
The transformation function $f'_{Subclass}(TG)$ is defined as:

$$\begin{aligned} Class &= \{\text{list\_to\_ns}(n) \mid n \in NT_{TG}\} \\ Enum &= \{\} \\ UserDataType &= \{\} \\ Field &= \{\} \\ \text{FieldSig} &= \{\} \\ EnumValue &= \{\} \\ Inh &= \{(\text{list\_to\_ns}(i), \text{list\_to\_ns}(j)) \mid (i,j) \in \sqsubseteq_{TG} \wedge i \neq j\} \\ Prop &= \{\} \\ Constant &= \{\} \\ \text{ConstType} &= \{\} \end{aligned}$$

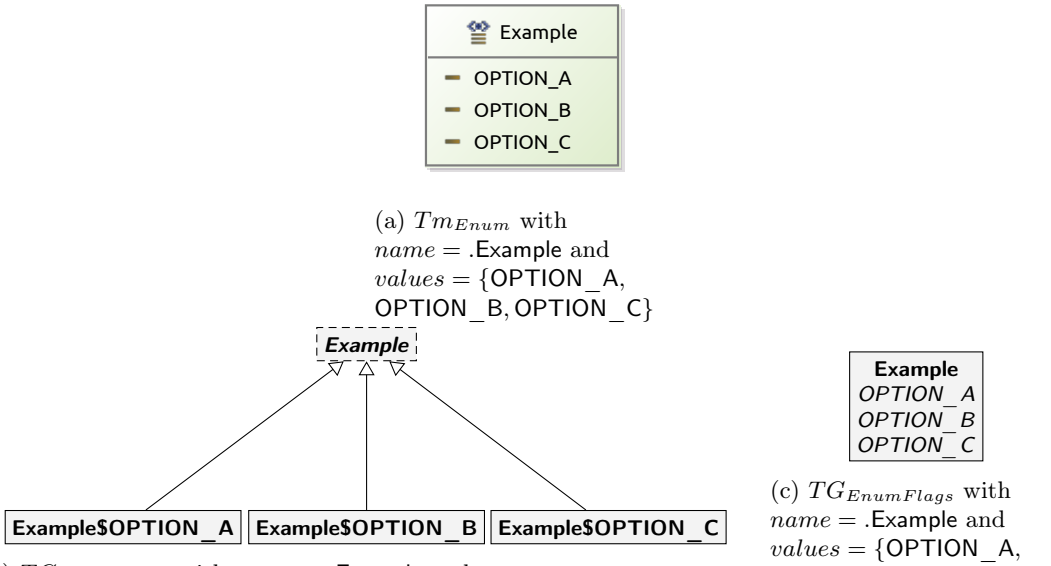
Also see `tg_subclass_as_node_type_to_tmod_subclass` *in* `Ecore-GROOVE-Mapping-Library.SubclassType`

Theorem 5.2.30 (Correctness of $f'_{Subclass}$)
$f'_{Subclass}(TG)$ (Definition 5.2.29) is a valid transformation function in the sense of Definition 4.3.30 transforming $TG_{Subclass}$ into $Tm_{Subclass}$.

Also see `tg_subclass_as_node_type_to_tmod_subclass_func` *in* `Ecore-GROOVE-Mapping-Library.SubclassType`

Once more, the correctness proof is not included here but can be found in the validated Isabelle proofs of this thesis.

5.2.4 Enumeration types



(a) Tm_{Enum} with $name$ = .Example and $values$ = {OPTION_A, OPTION_B, OPTION_C}

(b) $TG_{EnumNodes}$ with $name$ = .Example and $values$ = {OPTION_A, OPTION_B, OPTION_C}

(c) $TG_{EnumFlags}$ with $name$ = .Example and $values$ = {OPTION_A, OPTION_B, OPTION_C}

Figure 5.4: Visualisations of the transformations of enumeration types

This section will define the transformation of an enumeration type. Within this transformation, a new enumeration type is introduced, including its possible values. The Ecore type model that introduces such a subclass is defined as follows:

Definition 5.2.31 (Type model Tm_{Enum})
Let Tm_{Enum} be the type model containing a enumeration type with identifier name. The values of this enumeration type are defined as part of sequence values. Tm_{Enum} is defined as:

$$\begin{aligned} Class &= \{\} \\ Enum &= \{name\} \\ UserDataType &= \{\} \\ Field &= \{\} \\ \text{FieldSig} &= \{\} \\ EnumValue &= \{(name, v) \mid v \in values\} \\ Inh &= \{\} \\ Prop &= \{\} \\ Constant &= \{\} \\ \text{ConstType} &= \{\} \end{aligned}$$

Also see `tmod_enum` *in* `Ecore-GROOVE-Mapping-Library.EnumType`

Theorem 5.2.32 (Correctness of Tm_{Enum})
$Tm_{Subclass}$ (Definition 5.2.31) is a consistent type model in the sense of Definition 3.2.11.

Also see `tmod_enum_correct` *in* `Ecore-GROOVE-Mapping-Library.EnumType`

A visual representation of Tm_{Enum} with .Example as identifier for the new enumeration type and OPTION_A, OPTION_B and OPTION_C as its values can be seen in Figure 5.4a. The correctness proof of Tm_{Enum} is trivial, and therefore not included here. The proof can be found as part of the Isabelle validated proofs.

In order to make composing transformation functions possible, Tm_{Enum} should be compatible with the type model it is combined with.

Theorem 5.2.33 (Correctness of combine(Tm, Tm_{Enum}))
Assume a type model Tm that is consistent in the sense of Definition 3.2.11. Then Tm is compatible with Tm_{Enum} (in the sense of Definition 4.3.13) if:

- *The identifier of the enumeration type in Tm_{Enum} is not yet an identifier for a class, enumeration type or user-defined data type in Tm;*
- *The identifier of the enumeration type in Tm_{Enum} is not in the namespace of any class, enumeration type or user-defined data type in Tm;*
- *None of the identifiers in any class, enumeration type or user-defined data type in Tm is in the namespace of the enumeration type in Tm_{Enum}.*

 *Also see* `tmod_enum_combine_correct` *in* `Ecore-GROOVE-Mapping-Library.EnumType`

Proof. Use Lemma 4.3.12. It is possible to show that all assumptions hold. Now we have shown that combine(Tm, Tm_{Enum}) is consistent in the sense of Definition 3.2.11. □

The definitions and theorems for a regular subclass within Ecore are now complete.

Encoding as node type

A possible encoding for enumeration types in Ecore is using node types in GROOVE. In this case, the enumeration type itself is transformed into an abstract node type. Each value of the enumeration type is converted to its own node type, extending the abstract node type. The encoding corresponding to Tm_{Enum} can then be represented as $TG_{EnumNodes}$, defined in the following definition:

Definition 5.2.34 (Type graph $TG_{EnumNodes}$)
Let $TG_{EnumNodes}$ be a type graph containing multiple node types. The first node type encodes the enumeration type name. The other node types encode the values of enumeration type name. $TG_{EnumNodes}$ is defined as:

$$\begin{aligned}
NT &= \{\text{ns_to_list}(name)\} \cup \{\text{ns_to_list}(name) \mathbin{@} \langle v \rangle \mid v \in values\} \\
ET &= \{\} \\
\sqsubseteq &= \{(\text{ns_to_list}(name), \text{ns_to_list}(name))\} \cup \\
&\quad \{(\text{ns_to_list}(name) \mathbin{@} \langle v \rangle, \text{ns_to_list}(name) \mathbin{@} \langle v \rangle) \mid v \in values\} \cup \\
&\quad \{(\text{ns_to_list}(name) \mathbin{@} \langle v \rangle, \text{ns_to_list}(name)) \mid v \in values\} \\
abs &= \{\text{ns_to_list}(name)\} \\
\text{mult} &= \{\} \\
contains &= \{\}
\end{aligned}$$

 *Also see* `tg_enum_as_node_types` *in* `Ecore-GROOVE-Mapping-Library.EnumType`

Theorem 5.2.35 (Correctness of $TG_{EnumNodes}$)
$TG_{EnumNodes}$ (Definition 5.2.34) is a valid type graph in the sense of Definition 3.3.5.

 *Also see* `tg_enum_as_node_types_correct` *in* `Ecore-GROOVE-Mapping-Library.EnumType`

A visual representation of $TG_{EnumNodes}$ with .Example as identifier for the encoded enumeration type and OPTION_A, OPTION_B and OPTION_C as its values can be seen in Figure 5.4b. Please note that in this visualisation, the sequences are concatenated using the dollar sign $. The correctness proof of $TG_{EnumNodes}$ is trivial, and therefore not included here. The proof can be found as part of the Isabelle validated proofs.

In order to make composing transformation functions possible, $TG_{EnumNodes}$ should be compatible with the type graph it is combined with.

Theorem 5.2.36 (Correctness of combine$(TG, TG_{EnumNodes})$)
Assume a type graph TG that is valid in the sense of Definition 3.3.5. Then TG is compatible with $TG_{EnumNodes}$ (in the sense of Definition 4.3.24) if:

- *There are no shared node types between $TG_{EnumNodes}$ and TG.*

 *Also see* `tg_enum_as_node_types_combine_correct` *in* `Ecore-GROOVE-Mapping-Library.EnumType`

Proof. Use Lemma 4.3.23. It is possible to show that all assumptions hold. Now we have shown that combine$(TG, TG_{EnumNodes})$ is valid in the sense of Definition 3.3.5. □

The next definitions define the transformation function from Tm_{Enum} to $TG_{EnumNodes}$:

Definition 5.2.37 (Transformation function $f_{EnumNodes}$)
The transformation function $f_{EnumNodes}(Tm)$ is defined as:

$$\begin{aligned}
NT &= \{\text{ns_to_list}(e) \mid e \in Enum_{Tm}\} \cup \{\text{ns_to_list}(e) \mathbin{@} \langle v \rangle \mid (e, v) \in EnumValue_{Tm}\} \\
ET &= \{\} \\
\sqsubseteq &= \{(\text{ns_to_list}(e_1), \text{ns_to_list}(e_2)) \mid e_1 \in Enum_{Tm} \wedge e_2 \in Enum_{Tm}\} \cup \\
&\quad \{(\text{ns_to_list}(i) \mathbin{@} \langle j \rangle, \text{ns_to_list}(i) \mathbin{@} \langle j \rangle) \mid (i, j) \in EnumValue_{Tm}\} \cup \\
&\quad \{(\text{ns_to_list}(i) \mathbin{@} \langle j \rangle, \text{ns_to_list}(e)) \mid (i, j) \in EnumValue_{Tm} \wedge e \in Enum_{Tm}\} \\
abs &= \{\} \\
\text{mult} &= \{\} \\
contains &= \{\}
\end{aligned}$$

Also see `tmod_enum_to_tg_enum_as_node_types` *in* `Ecore-GROOVE-Mapping-Library.EnumType`

Theorem 5.2.38 (Correctness of $f_{EnumNodes}$)
$f_{EnumNodes}(Tm)$ (Definition 5.2.37) is a valid transformation function in the sense of Definition 4.3.25 transforming Tm_{Enum} into $TG_{EnumNodes}$.

Also see `tmod_enum_to_tg_enum_as_node_types_func` *in* `Ecore-GROOVE-Mapping-Library.EnumType`

The proof of the correctness of $f_{EnumNodes}$ will not be included here. Instead, it can be found in the validated Isabelle theories.

Finally, to complete the transformation, the transformation function that transforms $TG_{EnumNodes}$ into Tm_{Enum} is defined:

Definition 5.2.39 (Transformation function $f'_{EnumNodes}$)
The transformation function $f'_{EnumNodes}(TG, name)$ is defined as:

$$\begin{aligned}
Class &= \{\} \\
Enum &= \{\text{list_to_ns}(n) \mid n \in NT_{TG} \wedge n = \text{id_to_name}(name)\} \\
UserDataType &= \{\} \\
Field &= \{\} \\
\text{FieldSig} &= \{\} \\
EnumValue &= \{(\text{list_to_ns}(e), v) \mid e \mathbin{@} \langle v \rangle \in NT_{TG} \wedge e \mathbin{@} \langle v \rangle \neq \text{id_to_name}(name)\} \\
Inh &= \{\} \\
Prop &= \{\} \\
Constant &= \{\} \\
\text{ConstType} &= \{\}
\end{aligned}$$

Also see `tg_enum_as_node_types_to_tmod_enum` *in* `Ecore-GROOVE-Mapping-Library.EnumType`

Theorem 5.2.40 (Correctness of $f'_{EnumNodes}$)
$f'_{EnumNodes}(TG, name)$ (Definition 5.2.39) is a valid transformation function in the sense of Definition 4.3.30 transforming $TG_{EnumNodes}$ into Tm_{Enum}.

Also see `tg_enum_as_node_types_to_tmod_enum_func` *in* `Ecore-GROOVE-Mapping-Library.EnumType`

Once more, the correctness proof is not included here but can be found in the validated Isabelle proofs of this thesis.

Encoding as flags

Another possible encoding for enumeration types in Ecore is using flags in GROOVE. In this case, the enumeration type itself is transformed into a regular node type. Each value of the enumeration type is converted to a flag on this node type. The encoding corresponding to Tm_{Enum} can then be represented as $TG_{EnumFlags}$, defined in the following definition:

Definition 5.2.41 (Type graph $TG_{EnumFlags}$)
Let $TG_{EnumFlags}$ be a type graph containing a single node type which encodes the enumeration type name. The flags on the node type of name encode the different values. $TG_{EnumFlags}$ is defined as:

$$
\begin{aligned}
NT &= \{\text{ns\_to\_list}(name)\} \\
ET &= \{(\text{ns\_to\_list}(name), \langle v \rangle, \text{ns\_to\_list}(name)) \mid v \in values\} \\
\sqsubseteq &= \{(\text{ns\_to\_list}(name), \text{ns\_to\_list}(name))\} \\
abs &= \{\} \\
\text{mult}(e) &= \begin{cases} (0..1, 0..1) & \text{if } e \in ET_{TG_{EnumFlags}} \end{cases} \\
contains &= \{\}
\end{aligned}
$$

Also see `tg_enum_as_flags` *in* `Ecore-GROOVE-Mapping-Library.EnumType`

Theorem 5.2.42 (Correctness of $TG_{EnumFlags}$)
$TG_{EnumFlags}$ (Definition 5.2.41) is a valid type graph in the sense of Definition 3.3.5.

Also see `tg_enum_as_flags_correct` *in* `Ecore-GROOVE-Mapping-Library.EnumType`

A visual representation of $TG_{EnumFlags}$ with .Example as identifier for the encoded enumeration type and OPTION_A, OPTION_B and OPTION_C as its values can be seen in Figure 5.4c. The correctness proof of $TG_{EnumFlags}$ is trivial, and therefore not included here. The proof can be found as part of the Isabelle validated proofs.

In order to make composing transformation functions possible, $TG_{EnumFlags}$ should be compatible with the type graph it is combined with.

Theorem 5.2.43 (Correctness of combine$(TG, TG_{EnumFlags})$)
Assume a type graph TG that is valid in the sense of Definition 3.3.5. Then TG is compatible with $TG_{EnumFlags}$ (in the sense of Definition 4.3.24) if:

- *There are no shared node types between $TG_{EnumFlags}$ and TG.*

Also see `tg_enum_as_flags_combine_correct` *in* `Ecore-GROOVE-Mapping-Library.EnumType`

Proof. Use Lemma 4.3.23. It is possible to show that all assumptions hold. Now we have shown that combine$(TG, TG_{EnumFlags})$ is valid in the sense of Definition 3.3.5. □

The next definitions define the transformation function from Tm_{Enum} to $TG_{EnumFlags}$:

Definition 5.2.44 (Transformation function $f_{EnumFlags}$)
The transformation function $f_{EnumFlags}(Tm)$ is defined as:

$$
\begin{aligned}
NT &= \{\text{ns\_to\_list}(e) \mid e \in Enum_{Tm}\} \\
ET &= \{(\text{ns\_to\_list}(e), v, \text{ns\_to\_list}(e)) \mid (e, v) \in EnumValue_{Tm}\} \\
\sqsubseteq &= \{(\text{ns\_to\_list}(e_1), \text{ns\_to\_list}(e_2)) \mid e_1 \in Enum_{Tm} \wedge e_2 \in Enum_{Tm}\} \\
abs &= \{\} \\
\text{mult}(e) &= \begin{cases} (0..1, 0..1) & \text{if } e \in \{(\text{ns\_to\_list}(n), v, \text{ns\_to\_list}(n)) \mid (n, v) \in EnumValue_{Tm}\} \end{cases} \\
contains &= \{\}
\end{aligned}
$$

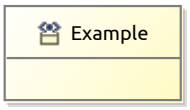
Also see `tmod_enum_to_tg_enum_as_flags` *in* `Ecore-GROOVE-Mapping-Library.EnumType`

Theorem 5.2.45 (Correctness of $f_{EnumFlags}$)
$f_{EnumFlags}(Tm)$ (Definition 5.2.44) is a valid transformation function in the sense of Definition 4.3.25 transforming Tm_{Enum} into $TG_{EnumFlags}$.

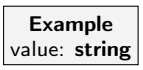
Also see `tmod_enum_to_tg_enum_as_flags_func` *in* `Ecore-GROOVE-Mapping-Library.EnumType`

The proof of the correctness of $f_{EnumFlags}$ will not be included here. Instead, it can be found in the validated Isabelle theories.

Finally, to complete the transformation, the transformation function that transforms $TG_{EnumFlags}$ into Tm_{Enum} is defined:



(a) $Tm_{UserType}$ with $name$ = .Example



(b) $TG_{UserType}$ with $name$ = .Example and $data\_edge$ = $value$

Figure 5.5: Visualisation of the transformation of user-defined data types

Definition 5.2.46 (Transformation function $f'_{EnumFlags}$)
The transformation function $f'_{EnumFlags}(TG)$ is defined as:

$$\begin{aligned}
Class &= \{\} \\
Enum &= \{\text{list_to_ns}(n) \mid n \in NT_{TG}\} \\
UserDataType &= \{\} \\
Field &= \{\} \\
\text{FieldSig} &= \{\} \\
EnumValue &= \{(\text{list_to_ns}(e), v) \mid (e, v, e) \in ET_{TG}\} \\
Inh &= \{\} \\
Prop &= \{\} \\
Constant &= \{\} \\
\text{ConstType} &= \{\}
\end{aligned}$$

Also see `tg_enum_as_flags_to_tmod_enum` *in* `Ecore-GROOVE-Mapping-Library.EnumType`

Theorem 5.2.47 (Correctness of $f'_{EnumNodes}$)
$f'_{EnumFlags}(TG)$ (Definition 5.2.46) is a valid transformation function in the sense of Definition 4.3.30 transforming $TG_{EnumFlags}$ into Tm_{Enum}.

Also see `tg_enum_as_flags_to_tmod_enum_func` *in* `Ecore-GROOVE-Mapping-Library.EnumType`

Once more, the correctness proof is not included here but can be found in the validated Isabelle proofs of this thesis.

5.2.5 User-defined data types

Within this section, the transformation of an user-defined data type will be defined. A user-defined data type in Ecore is a custom data type, of which a serialisation can be stored in the form of a string. The Ecore type model for introducing a user-defined data type is simple:

Definition 5.2.48 (Type model $Tm_{UserType}$)
Let $Tm_{UserType}$ be the type model containing a user-defined data type with identifier name. $Tm_{UserType}$ is defined as:

$$\begin{aligned}
Class &= \{\} \\
Enum &= \{\} \\
UserDataType &= \{name\} \\
Field &= \{\} \\
\text{FieldSig} &= \{\} \\
EnumValue &= \{\} \\
Inh &= \{\} \\
Prop &= \{\} \\
Constant &= \{\} \\
\text{ConstType} &= \{\}
\end{aligned}$$

Also see `tmod_userdatatype` *in* `Ecore-GROOVE-Mapping-Library.UserDataTypeType`

Theorem 5.2.49 (Correctness of $Tm_{UserType}$)
$Tm_{UserType}$ (Definition 5.2.48) is a consistent type model in the sense of Definition 3.2.11.

Also see `tmod_userdatatype_correct` *in* `Ecore-GROOVE-Mapping-Library.UserDataTypeType`

A visual representation of $Tm_{UserType}$ with identifier .Example can be seen in Figure 5.5a. The correctness proof of $Tm_{UserType}$ is trivial, and therefore not included here. The proof can be found as part of the Isabelle validated proofs.

In order to make composing transformation functions possible, $Tm_{UserType}$ should be compatible with the type model it is combined with.

Theorem 5.2.50 (Correctness of combine$(Tm, Tm_{UserType})$)
Assume a type model Tm that is consistent in the sense of Definition 3.2.11. Then Tm is compatible with $Tm_{UserType}$ (in the sense of Definition 4.3.13) if:

- *The identifier of the user-defined data type in $Tm_{UserType}$ is not yet an identifier for a class, enumeration type or user-defined data type in Tm;*
- *The identifier of the user-defined data type in $Tm_{UserType}$ is not in the namespace of any class, enumeration type or user-defined data type in Tm;*
- *None of the identifiers in any class, enumeration type or user-defined data type in Tm is in the namespace of the class in $Tm_{UserType}$.*

Also see `tmod_userdatatype_combine_correct` *in* `Ecore-GROOVE-Mapping-Library.UserDataTypeType`

Proof. Use Lemma 4.3.12. It is possible to show that all assumptions hold. Now we have shown that combine$(Tm, Tm_{UserType})$ is consistent in the sense of Definition 3.2.11. □

The definitions and theorems for a regular class within Ecore are now complete.

Encoding as node type

A possible encoding for user-defined data types in Ecore is using a node type in GROOVE. This node type will get a transformed identifier as name. In order to be able to store the serialised value, an edge type is created to a string node that represents the serialised value. The encoding corresponding to $Tm_{UserType}$ can then be represented as $TG_{UserType}$, defined in the following definition:

Definition 5.2.51 (Type graph $TG_{UserType}$)
Let $TG_{UserType}$ be the type graph containing a single node type which encodes a user-defined data type name. Furthermore, this node type has an edge type named data_edge to a string primitive to store its serialised value. $TG_{UserType}$ is defined as:

$$
\begin{aligned}
NT &= \{\text{ns\_to\_list}(name)\} \\
ET &= \{(\text{ns\_to\_list}(name), \langle data\_edge \rangle, \mathsf{string})\} \\
\sqsubseteq &= \{(\text{ns\_to\_list}(name), \text{ns\_to\_list}(name)), (\mathsf{string}, \mathsf{string})\} \\
abs &= \{\} \\
\text{mult}(e) &= \begin{cases} (0..*, 1..1) & \text{if } e \in ET_{TG_{UserType}} \end{cases} \\
contains &= \{\}
\end{aligned}
$$

Also see `tg_userdatatype_as_node_type` *in* `Ecore-GROOVE-Mapping-Library.UserDataTypeType`

Theorem 5.2.52 (Correctness of $TG_{UserType}$)
$TG_{UserType}$ (Definition 5.2.51) is a valid type graph in the sense of Definition 3.3.5.

Also see `tg_userdatatype_as_node_type_correct` *in* `Ecore-GROOVE-Mapping-Library.UserDataTypeType`

A visual representation of $TG_{UserType}$ with identifier .Example can be seen in Figure 5.5b. In this example, *value* was chosen as edge name. The correctness proof of $TG_{UserType}$ is trivial, and therefore not included here. The proof can be found as part of the Isabelle validated proofs.

In order to make composing transformation functions possible, $TG_{UserType}$ should be compatible with the type graph it is combined with.

Theorem 5.2.53 (Correctness of combine$(TG, TG_{UserType})$)
Assume a type graph TG that is valid in the sense of Definition 3.3.5. Then TG is compatible with $TG_{UserType}$ (in the sense of Definition 4.3.24) if:

- *The node type of the encoded class in $TG_{UserType}$ is not a node type in TG.*

 *Also see* `tg_userdatatype_as_node_type_combine_correct` *in* `Ecore-GROOVE-Mapping-Library.UserDataTypeType`

Proof. Use Lemma 4.3.23. It is possible to show that all assumptions hold. Now we have shown that combine$(TG, TG_{UserType})$ is valid in the sense of Definition 3.3.5. □

The next definitions define the transformation function from $Tm_{UserType}$ to $TG_{UserType}$:

Definition 5.2.54 (Transformation function $f_{UserType}$)
The transformation function $f_{UserType}(Tm)$ is defined as:

$$
\begin{aligned}
NT &= \{\text{ns\_to\_list}(u) \mid u \in UserDataType_{Tm}\} \cup \{\mathsf{string}\} \\
ET &= \{(\text{ns\_to\_list}(u), \langle data\_edge \rangle, \mathsf{string}) \mid u \in UserDataType_{Tm}\} \\
\sqsubseteq &= \{(\text{ns\_to\_list}(u_1), \text{ns\_to\_list}(u_2)) \mid u_1 \in UserDataType_{Tm} \wedge u_2 \in UserDataType_{Tm}\} \cup \\
&\quad \{(\mathsf{string}, \mathsf{string})\} \\
abs &= \{\} \\
\text{mult}(e) &= \begin{cases} (0..*, 1..1) & \text{if } e \in \{(\text{ns\_to\_list}(u), \langle data\_edge \rangle, \mathsf{string}) \mid u \in UserDataType_{Tm}\} \end{cases} \\
contains &= \{\}
\end{aligned}
$$

 *Also see* `tmod_userdatatype_to_tg_userdatatype_as_node_type` *in* `Ecore-GROOVE-Mapping-Library.UserDataTypeType`

Theorem 5.2.55 (Correctness of $f_{UserType}$)
$f_{UserType}(Tm)$ (Definition 5.2.54) is a valid transformation function in the sense of Definition 4.3.25 transforming $Tm_{UserType}$ into $TG_{UserType}$.

 *Also see* `tmod_userdatatype_to_tg_userdatatype_as_node_type_func` *in* `Ecore-GROOVE-Mapping-Library.UserDataTypeType`

The proof of the correctness of $f_{UserType}$ will not be included here. Instead, it can be found in the validated Isabelle theories.

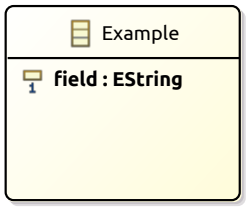
Finally, to complete the transformation, the transformation function that transforms $TG_{UserType}$ into $Tm_{UserType}$ is defined:

Definition 5.2.56 (Transformation function $f'_{UserType}$)
The transformation function $f'_{UserType}(TG)$ is defined as:

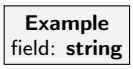
$$
\begin{aligned}
Class &= \{\} \\
Enum &= \{\} \\
UserDataType &= \{\text{list\_to\_ns}(n) \mid n \in NT_{TG} \cap Lab_t\} \\
Field &= \{\} \\
\text{FieldSig} &= \{\} \\
EnumValue &= \{\} \\
Inh &= \{\} \\
Prop &= \{\} \\
Constant &= \{\} \\
\text{ConstType} &= \{\}
\end{aligned}
$$

 *Also see* `tg_userdatatype_as_node_type_to_tmod_userdatatype` *in* `Ecore-GROOVE-Mapping-Library.UserDataTypeType`

Theorem 5.2.57 (Correctness of $f'_{UserType}$)
$f'_{UserType}(TG)$ (Definition 5.2.56) is a valid transformation function in the sense of Definition 4.3.30 transforming $TG_{UserType}$ into $Tm_{UserType}$.



(a) $Tm_{DataField}$ for a string with $name$ = field



(b) $TG_{DataField}$ for a string with $name$ = field

Figure 5.6: Visualisation of the transformation of a field typed by a data type

Also see `tg_userdatatype_as_node_type_to_tmod_userdatatype_func` *in* `Ecore-GROOVE-Mapping-Library.UserDataTypeType`

Once more, the correctness proof is not included here but can be found in the validated Isabelle proofs of this thesis.

5.2.6 Data fields

All transformations discussed so far have focused on introducing different kind of types. In the following transformations, these types will be enriched with fields. In this transformation specifically, a field typed by a data type will be introduced.

Definition 5.2.58 (Type model $Tm_{DataField}$)
Let $Tm_{DataField}$ be the type model containing a regular class with identifier classtype. Then $Tm_{DataField}$ defines a field named name with type fieldtype, in which fieldtype is either boolean, integer, real *or* string. *$Tm_{DataField}$ is defined as:*

$$
\begin{aligned}
Class &= \{classtype\} \\
Enum &= \{\} \\
UserDataType &= \{\} \\
Field &= \{(classtype, name)\} \\
\text{FieldSig} &= \begin{cases} (f, (fieldtype, 1..1)) & \text{if } f \in Field_{Tm_{DataField}} \end{cases} \\
EnumValue &= \{\} \\
Inh &= \{\} \\
Prop &= \{\} \\
Constant &= \{\} \\
\text{ConstType} &= \{\}
\end{aligned}
$$

Also see `tmod_data_field` *in* `Ecore-GROOVE-Mapping-Library.DataField`

Theorem 5.2.59 (Correctness of $Tm_{DataField}$)
$Tm_{DataField}$ (Definition 5.2.58) is a consistent type model in the sense of Definition 3.2.11.

Also see `tmod_data_field_correct` *in* `Ecore-GROOVE-Mapping-Library.DataField`

A visual representation of $Tm_{DataField}$ with field name field on class .Example can be seen in Figure 5.6a. In this example, the string type is chosen for the fieldtype, but any data type would have worked. The correctness proof of $Tm_{DataField}$ is more involved, it is not included here for conciseness. It can be found within the validated Isabelle proofs.

In order to make composing transformation functions possible, $Tm_{DataField}$ should be compatible with the type model it is combined with.

Theorem 5.2.60 (Correctness of combine$(Tm, Tm_{DataField})$)
Assume a type model Tm that is consistent in the sense of Definition 3.2.11. Then Tm is compatible with $Tm_{DataField}$ (in the sense of Definition 4.3.13) if:

- *The class type on which the field is defined, classtype, is already an existing class in Tm;*
- *The field named name is not already a field on classtype in Tm.*

Also see `tmod_data_field_combine_correct` *in* `Ecore-GROOVE-Mapping-Library.DataField`

Proof. Use Lemma 4.3.12. It is possible to show that all assumptions hold. Now we have shown that combine$(Tm, Tm_{DataField})$ is consistent in the sense of Definition 3.2.11. □

The definitions and theorems for defining a data field within Ecore are now complete.

Encoding as edge type

The most obvious encoding for an field in GROOVE would be using an edge type. The field is transformed into an edge type between an existing node type and the corresponding field type. The encoding corresponding to $Tm_{DataField}$ can then be represented as $TG_{DataField}$, defined in the following definition:

Definition 5.2.61 (Type graph $TG_{DataField}$)
Let $TG_{DataField}$ be the type graph containing a node type which encodes the class type classtype. Furthermore, define an edge type from classtype named name. This edge type targets a node of fieldtype. $TG_{DataField}$ is defined as:

$$
\begin{aligned}
NT &= \{\text{ns\_to\_list}(classtype), fieldtype\} \\
ET &= \{(\text{ns\_to\_list}(classtype), \langle name \rangle, fieldtype)\} \\
\sqsubseteq &= \{(\text{ns\_to\_list}(classtype), \text{ns\_to\_list}(classtype)), (fieldtype, fieldtype)\} \\
abs &= \{\} \\
\text{mult}(e) &= \begin{cases} (0..*, 1..1) & \text{if } e \in \{(\text{ns\_to\_list}(classtype), \langle name \rangle, fieldtype)\} \end{cases} \\
contains &= \{\}
\end{aligned}
$$

Also see `tg_data_field_as_edge_type` *in* `Ecore-GROOVE-Mapping-Library.DataField`

Theorem 5.2.62 (Correctness of $TG_{DataField}$)
$TG_{DataField}$ (Definition 5.2.61) is a valid type graph in the sense of Definition 3.3.5.

Also see `tg_data_field_as_edge_type_correct` *in* `Ecore-GROOVE-Mapping-Library.DataField`

A visual representation of $TG_{DataField}$ with edge name field on node type Example can be seen in Figure 5.6b. Like the previous example, a string has been chosen to be consequent, but any primitive type could have been used. The correctness proof of $TG_{DataField}$ is more involved, it is not included here for conciseness. It can be found within the validated Isabelle proofs.

In order to make composing transformation functions possible, $TG_{DataField}$ should be compatible with the type graph it is combined with.

Theorem 5.2.63 (Correctness of combine$(TG, TG_{DataField})$)
Assume a type graph TG that is valid in the sense of Definition 3.3.5. Then TG is compatible with $TG_{DataField}$ (in the sense of Definition 4.3.24) if:

- *The node type of the encoded class type in $TG_{DataField}$ is already an node type in TG;*
- *The node type of the encoded class type in $TG_{DataField}$ does not already have an edge type with the same name as the field in TG.*

Also see `tg_data_field_as_edge_type_combine_correct` *in* `Ecore-GROOVE-Mapping-Library.DataField`

Proof. Use Lemma 4.3.23. It is possible to show that all assumptions hold. Now we have shown that combine$(TG, TG_{DataField})$ is valid in the sense of Definition 3.3.5. □

The next definitions define the transformation function from $Tm_{DataField}$ to $TG_{DataField}$:

Definition 5.2.64 (Transformation function $f_{DataField}$)
The transformation function $f_{DataField}(Tm)$ is defined as:

$$
\begin{aligned}
NT &= \{\text{ns\_to\_list}(c) \mid c \in Class_{Tm}\} \cup \{fieldtype\} \\
ET &= \{(\text{ns\_to\_list}(c), \langle f \rangle, fieldtype) \mid (c, n) \in Field_{Tm}\} \\
\sqsubseteq &= \{(\text{ns\_to\_list}(c), \text{ns\_to\_list}(c)) \mid c \in Class_{Tm}\} \cup \{(fieldtype, fieldtype)\} \\
abs &= \{\} \\
\text{mult} &= \left\{(0..*, 1..1) \quad \text{if } e \in \{(\text{ns\_to\_list}(c), \langle f \rangle, fieldtype) \mid (c, n) \in Field_{Tm}\}\right. \\
contains &= \{\}
\end{aligned}
$$

Also see `tmod_data_field_to_tg_data_field_as_edge_type` *in* `Ecore-GROOVE-Mapping-Library.DataField`

Theorem 5.2.65 (Correctness of $f_{DataField}$)
$f_{DataField}(Tm)$ (Definition 5.2.64) is a valid transformation function in the sense of Definition 4.3.25 transforming $Tm_{DataField}$ into $TG_{DataField}$.

Also see `tmod_data_field_to_tg_data_field_as_edge_type_func` *in* `Ecore-GROOVE-Mapping-Library.DataField`

The proof of the correctness of $f_{DataField}$ will not be included here. Instead, it can be found in the validated Isabelle theories.

Finally, to complete the transformation, the transformation function that transforms $TG_{DataField}$ into $Tm_{DataField}$ is defined:

Definition 5.2.66 (Transformation function $f'_{DataField}$)
The transformation function $f'_{DataField}(TG)$ is defined as:

$$
\begin{aligned}
Class &= \{\text{list\_to\_ns}(n) \mid n \in NT_{TG} \cap Lab_t\} \\
Enum &= \{\} \\
UserDataType &= \{\} \\
Field &= \{(\text{list\_to\_ns}(\text{src}(e)), l) \mid e \in ET_{TG} \wedge \langle l \rangle = \text{lab}(e)\} \\
\text{FieldSig} &= \left\{(f, (fieldtype, 1..1)) \quad \text{if } f \in \{(\text{list\_to\_ns}(\text{src}(e)), l) \mid e \in ET_{TG} \wedge \langle l \rangle = \text{lab}(e)\}\right. \\
EnumValue &= \{\} \\
Inh &= \{\} \\
Prop &= \{\} \\
Constant &= \{\} \\
\text{ConstType} &= \{\}
\end{aligned}
$$

Also see `tg_data_field_as_edge_type_to_tmod_data_field` *in* `Ecore-GROOVE-Mapping-Library.DataField`

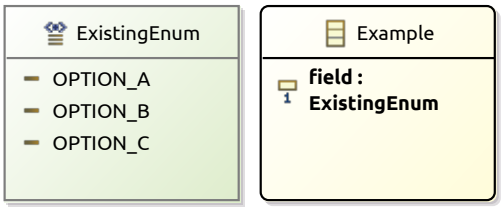
Theorem 5.2.67 (Correctness of $f'_{DataField}$)
$f'_{DataField}(TG)$ (Definition 5.2.66) is a valid transformation function in the sense of Definition 4.3.30 transforming $TG_{DataField}$ into $Tm_{DataField}$.

Also see `tg_data_field_as_edge_type_to_tmod_data_field_func` *in* `Ecore-GROOVE-Mapping-Library.DataField`

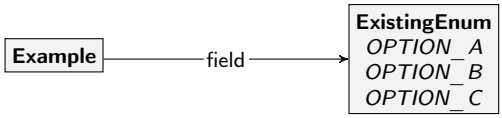
Once more, the correctness proof is not included here but can be found in the validated Isabelle proofs of this thesis.

5.2.7 Enumeration fields

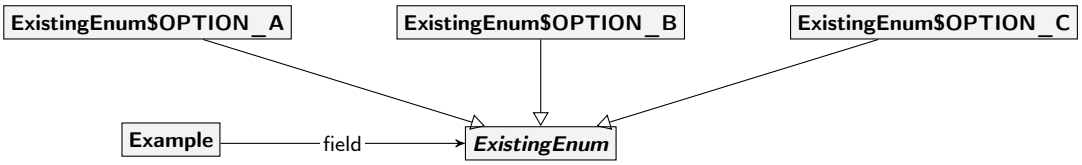
In this section, the transformation for a field typed by an enumeration type will be discussed. Since an enumeration type can be encoded in multiple ways, multiple encodings will be introduced for fields as well. First, the Ecore type model will be introduced.



(a) $Tm_{EnumField}$ for .ExistingEnum with *name* = field



(b) $TG_{EnumFieldFlags}$ for ExistingEnum with *name* = field



(c) $TG_{EnumFieldNodes}$ for ExistingEnum with *name* = field

Figure 5.7: Visualisation of the transformation of a field typed by an enumeration type

Definition 5.2.68 (Type model $Tm_{EnumField}$)
Let $Tm_{EnumField}$ be the type model containing a regular class with identifier classtype. Furthermore, it defines an enumeration type with identifier enumid, and corresponding values as set enumvalues. Then $Tm_{EnumField}$ defines a field named name with type enumid on class classtype. $Tm_{EnumField}$ is defined as:

$$
\begin{aligned}
Class &= \{classtype\} \\
Enum &= \{enumid\} \\
UserDataType &= \{\} \\
Field &= \{(classtype, name)\} \\
\text{FieldSig} &= \begin{cases} (f, (enumid, 1..1)) & \text{if } f \in Field_{Tm_{EnumField}} \end{cases} \\
EnumValue &= \{(enumid, v) \mid v \in enumvalues\} \\
Inh &= \{\} \\
Prop &= \{\} \\
Constant &= \{\} \\
\text{ConstType} &= \{\}
\end{aligned}
$$

Also see `tmod_enum_field` *in* `Ecore-GROOVE-Mapping-Library.EnumField`

Theorem 5.2.69 (Correctness of $Tm_{EnumField}$)
$Tm_{EnumField}$ (Definition 5.2.68) is a consistent type model in the sense of Definition 3.2.11.

Also see `tmod_enum_field_correct` *in* `Ecore-GROOVE-Mapping-Library.EnumField`

A visual representation of $Tm_{EnumField}$ with field name field on class .Example can be seen in Figure 5.7a. In this example, the field is typed by the .ExistingEnum enumeration type. The correctness proof of $Tm_{EnumField}$ is more involved, it is not included here for conciseness. It can be found within the validated Isabelle proofs.

In order to make composing transformation functions possible, $Tm_{EnumField}$ should be compatible with the type model it is combined with.

Theorem 5.2.70 (Correctness of combine$(Tm, Tm_{EnumField})$)
Assume a type model Tm that is consistent in the sense of Definition 3.2.11. Then Tm is compatible with $Tm_{EnumField}$ (in the sense of Definition 4.3.13) if:

- *The class type on which the field is defined, classtype, is already an existing class in Tm;*
- *The enumeration type by which the field is typed, enumid, is already an existing enumeration type in Tm;*
- *All the values for the enumeration type enumid are already enumeration values for enumid in Tm;*
- *The field named name is not already a field on classtype in Tm.*

Also see `tmod_enum_field_combine_correct` *in* `Ecore-GROOVE-Mapping-Library.EnumField`

Proof. Use Lemma 4.3.12. It is possible to show that all assumptions hold. Now we have shown that combine$(Tm, Tm_{EnumField})$ is consistent in the sense of Definition 3.2.11. □

The definitions and theorems for defining a field typed by an enumeration type within Ecore are now complete.

Encoding as edge type to an node type encoded enumeration type

As mentioned earlier, Section 5.2.4 defines multiple ways to encode an enumeration type. Each of these encodings needs a specialised field encoding. In principle, the encoding of the fields itself is the same, but since every transformation model needs to be valid on its own, the encodings need to be distinguished.

The first encoding for an enumeration type uses node types to encode the different values. The encoding corresponding to $Tm_{EnumField}$, in the case that the field references an enumeration type encoded as node types, can then be represented as $TG_{EnumFieldNodes}$, defined in the following definition:

Definition 5.2.71 (Type graph $TG_{EnumFieldNodes}$)
Let $TG_{EnumFieldNodes}$ be the type graph containing a node type which encodes the class type classtype. Furthermore, $TG_{EnumFieldNodes}$ contains an encoded version of enumeration type enumid with values from set enumvalues. This enumeration type is encoded using node types, as defined in $TG_{EnumNodes}$ (Definition 5.2.34). Finally, $TG_{EnumFieldNodes}$ defines an edge type from the encoded classtype named name to the encoded enumeration type enumid. $TG_{EnumFieldNodes}$ is defined as:

$$
\begin{aligned}
NT &= \{\text{ns\_to\_list}(\mathit{classtype}), \text{ns\_to\_list}(\mathit{enumid})\} \cup \\
&\quad \{\text{ns\_to\_list}(\mathit{enumid}) \mathbin{@} \langle v \rangle \mid v \in \mathit{enumvalues}\} \\
ET &= \{(\text{ns\_to\_list}(\mathit{classtype}), \langle \mathit{name} \rangle, \text{ns\_to\_list}(\mathit{enumid}))\} \\
\sqsubseteq &= \{(\text{ns\_to\_list}(\mathit{classtype}), \text{ns\_to\_list}(\mathit{classtype})), \\
&\quad (\text{ns\_to\_list}(\mathit{enumid}), \text{ns\_to\_list}(\mathit{enumid}))\} \cup \\
&\quad \{(\text{ns\_to\_list}(\mathit{enumid}) \mathbin{@} \langle v \rangle, \text{ns\_to\_list}(\mathit{enumid}) \mathbin{@} \langle v \rangle) \mid v \in \mathit{enumvalues}\} \cup \\
&\quad \{(\text{ns\_to\_list}(\mathit{enumid}) \mathbin{@} \langle v \rangle, \text{ns\_to\_list}(\mathit{enumid})) \mid v \in \mathit{enumvalues}\} \\
\mathit{abs} &= \{\text{ns\_to\_list}(\mathit{enumid})\} \\
\text{mult}(e) &= \begin{cases} (0..*, 1..1) & \text{if } e \in \{(\text{ns\_to\_list}(\mathit{classtype}), \langle \mathit{name} \rangle, \text{ns\_to\_list}(\mathit{enumid}))\} \end{cases} \\
\mathit{contains} &= \{\}
\end{aligned}
$$

Also see `tg_enum_as_node_types_field_as_edge_type` *in* `Ecore-GROOVE-Mapping-Library.EnumField`

Theorem 5.2.72 (Correctness of $TG_{EnumFieldNodes}$)
$TG_{EnumFieldNodes}$ (Definition 5.2.71) is a valid type graph in the sense of Definition 3.3.5.

Also see `tg_enum_as_node_types_field_as_edge_type_correct` *in* `Ecore-GROOVE-Mapping-Library.EnumField`

A visual representation of $TG_{EnumFieldNodes}$ with edge name field on node type Example can be seen in Figure 5.7c. The field references the encoded enumeration type ExistingEnum via the abstract type, such that its nodes can reference any of the concrete values. The correctness proof of $TG_{EnumFieldNodes}$ is more involved, it is not included here for conciseness. It can be found within the validated Isabelle proofs.

In order to make composing transformation functions possible, $TG_{EnumFieldNodes}$ should be compatible with the type graph it is combined with.

Theorem 5.2.73 (Correctness of combine$(TG, TG_{EnumFieldNodes})$)
Assume a type graph TG that is valid in the sense of Definition 3.3.5. Then TG is compatible with $TG_{EnumFieldNodes}$ (in the sense of Definition 4.3.24) if:

- *The node type of the encoded class type in $TG_{EnumFieldNodes}$ is already an node type in TG;*
- *All node types corresponding to the encoding of the enumeration type in $TG_{EnumFieldNodes}$ are already node types in TG;*

- *The node type of the encoded class type in $TG_{EnumFieldNodes}$ does not already have an edge type with the same name as the field in TG.*

Also see `tg_enum_as_node_types_field_as_edge_type_combine_correct` *in* `Ecore-GROOVE-Mapping-Library.EnumField`

Proof. Use Lemma 4.3.23. It is possible to show that all assumptions hold. Now we have shown that combine$(TG, TG_{EnumFieldNodes})$ is valid in the sense of Definition 3.3.5. □

The next definitions define the transformation function from $Tm_{EnumField}$ to $TG_{EnumFieldNodes}$:

Definition 5.2.74 (Transformation function $f_{EnumFieldNodes}$)
The transformation function $f_{EnumFieldNodes}(Tm)$ is defined as:

$$\begin{aligned}
NT &= \{\text{ns_to_list}(t) \mid t \in Class_{Tm} \cup Enum_{Tm}\} \cup \\
&\quad \{\text{ns_to_list}(e) \mathbin{@} \langle v \rangle \mid (e, v) \in EnumValue_{Tm}\} \\
ET &= \{(\text{ns_to_list}(c), \langle f \rangle, \text{ns_to_list}(e)) \mid (c, f) \in Field_{Tm} \wedge e \in Enum_{Tm}\} \\
\sqsubseteq &= \{(\text{ns_to_list}(x), \text{ns_to_list}(x)) \mid x \in Class_{Tm} \cup Enum_{Tm}\} \cup \\
&\quad \{(\text{ns_to_list}(e) \mathbin{@} \langle v \rangle, \text{ns_to_list}(e) \mathbin{@} \langle v \rangle) \mid (e, v) \in EnumValue_{Tm}\} \cup \\
&\quad \{(\text{ns_to_list}(e) \mathbin{@} \langle v \rangle, \text{ns_to_list}(e)) \mid (e, v) \in EnumValue_{Tm}\} \\
abs &= \{\text{ns_to_list}(t) \mid t \in Enum_{Tm}\} \\
\text{mult}(e) &= \begin{cases} (0..*, 1..1) & \text{if } e \in \{(\text{ns_to_list}(c), \langle f \rangle, \text{ns_to_list}(e)) \mid (c, f) \in Field_{Tm} \wedge e \in Enum_{Tm}\} \end{cases} \\
contains &= \{\}
\end{aligned}$$

Also see `tmod_enum_field_to_tg_enum_as_node_types_field_as_edge_type` *in* `Ecore-GROOVE-Mapping-Library.EnumField`

Theorem 5.2.75 (Correctness of $f_{EnumFieldNodes}$)
$f_{EnumFieldNodes}(Tm)$ (Definition 5.2.74) is a valid transformation function in the sense of Definition 4.3.25 transforming $Tm_{EnumField}$ into $TG_{EnumFieldNodes}$.

Also see `tmod_enum_field_to_tg_enum_as_node_types_field_as_edge_type_func` *in* `Ecore-GROOVE-Mapping-Library.EnumField`

The proof of the correctness of $f_{EnumFieldNodes}$ will not be included here. Instead, it can be found in the validated Isabelle theories.

Finally, to complete the transformation, the transformation function that transforms $TG_{EnumFieldNodes}$ into $Tm_{EnumField}$ is defined:

Definition 5.2.76 (Transformation function $f'_{EnumFieldNodes}$)
The transformation function $f'_{EnumFieldNodes}(TG)$ is defined as:

$$\begin{aligned}
Class &= \{\text{list_to_ns}(n) \mid n \in NT_{TG} \wedge n = \text{ns_to_list}(classtype)\} \\
Enum &= \{\text{list_to_ns}(n) \mid n \in NT_{TG} \wedge n = \text{ns_to_list}(enumid)\} \\
UserDataType &= \{\} \\
Field &= \{(\text{list_to_ns}(\text{src}(e)), l) \mid e \in ET_{TG} \wedge \langle l \rangle = \text{lab}(e)\} \\
\text{FieldSig} &= \begin{cases} (f, (fieldtype, 1..1)) & \text{if } f \in \{(\text{list_to_ns}(\text{src}(e)), l) \mid e \in ET_{TG} \wedge \langle l \rangle = \text{lab}(e)\} \end{cases} \\
EnumValue &= \{(enumid, v) \mid v \in enumvalues\} \\
Inh &= \{\} \\
Prop &= \{\} \\
Constant &= \{\} \\
\text{ConstType} &= \{\}
\end{aligned}$$

Also see `tg_enum_as_node_types_field_as_edge_type_to_tmod_enum_field` *in* `Ecore-GROOVE-Mapping-Library.EnumField`

Theorem 5.2.77 (Correctness of $f'_{EnumFieldNodes}$)
$f'_{EnumFieldNodes}(TG)$ (Definition 5.2.76) is a valid transformation function in the sense of Definition 4.3.30 transforming $TG_{EnumFieldNodes}$ into $Tm_{EnumField}$.

Also see `tg_enum_as_node_types_field_as_edge_type_to_tmod_enum_field_func` *in* `Ecore-GROOVE-Mapping-Library.EnumField`

Once more, the correctness proof is not included here but can be found in the validated Isabelle proofs of this thesis.

Encoding as edge type to an flag encoded enumeration type

The second encoding for an enumeration type uses flags to encode the different values. The encoding corresponding to $Tm_{EnumField}$, in the case that the field references an enumeration type encoded as flags, can then be represented as $TG_{EnumFieldFlags}$, defined in the following definition:

Definition 5.2.78 (Type graph $TG_{EnumFieldFlags}$)
Let $TG_{EnumFieldFlags}$ be the type graph containing a node type which encodes the class type classtype. Furthermore, $TG_{EnumFieldFlags}$ contains an encoded version of enumeration type enumid with values from set enumvalues. This enumeration type is encoded using flags, as defined in $TG_{EnumFlags}$ (Definition 5.2.41). Finally, $TG_{EnumFieldFlags}$ defines an edge type from the encoded classtype named name to the encoded enumeration type enumid. $TG_{EnumFieldFlags}$ is defined as:

$$
\begin{aligned}
NT &= \{\text{ns\_to\_list}(classtype), \text{ns\_to\_list}(enumid)\} \\
ET &= \{(\text{ns\_to\_list}(classtype), \langle name \rangle, \text{ns\_to\_list}(enumid))\} \\
\sqsubseteq &= \{(\text{ns\_to\_list}(classtype), \text{ns\_to\_list}(classtype)), \\
&\quad\ (\text{ns\_to\_list}(enumid), \text{ns\_to\_list}(enumid))\} \\
abs &= \{\} \\
\text{mult}(e) &= \begin{cases} (0..*, 1..1) & \text{if } e \in \{(\text{ns\_to\_list}(classtype), \langle name \rangle, \text{ns\_to\_list}(enumid))\} \end{cases} \\
contains &= \{\}
\end{aligned}
$$

Also see `tg_enum_as_flags_field_as_edge_type` *in* `Ecore-GROOVE-Mapping-Library.EnumField`

Theorem 5.2.79 (Correctness of $TG_{EnumFieldFlags}$)
$TG_{EnumFieldFlags}$ (Definition 5.2.78) is a valid type graph in the sense of Definition 3.3.5.

Also see `tg_enum_as_flags_field_as_edge_type_correct` *in* `Ecore-GROOVE-Mapping-Library.EnumField`

A visual representation of $TG_{EnumFieldFlags}$ with edge name field on node type Example can be seen in Figure 5.7b. The field references the encoded enumeration type ExistingEnum via the corresponding node type. The correctness proof of $TG_{EnumFieldFlags}$ is more involved, it is not included here for conciseness. It can be found within the validated Isabelle proofs.

In order to make composing transformation functions possible, $TG_{EnumFieldFlags}$ should be compatible with the type graph it is combined with.

Theorem 5.2.80 (Correctness of combine$(TG, TG_{EnumFieldFlags})$)
Assume a type graph TG that is valid in the sense of Definition 3.3.5. Then TG is compatible with $TG_{EnumFieldFlags}$ (in the sense of Definition 4.3.24) if:

- *The node type of the encoded class type in $TG_{EnumFieldFlags}$ is already an node type in TG;*
- *All node types corresponding to the encoding of the enumeration type in $TG_{EnumFieldFlags}$ are already node types in TG;*
- *The node type of the encoded class type in $TG_{EnumFieldFlags}$ does not already have an edge type with the same name as the field in TG.*

Also see `tg_enum_as_flags_field_as_edge_type_combine_correct` *in* `Ecore-GROOVE-Mapping-Library.EnumField`

Proof. Use Lemma 4.3.23. It is possible to show that all assumptions hold. Now we have shown that combine$(TG, TG_{EnumFieldFlags})$ is valid in the sense of Definition 3.3.5. □

The next definitions define the transformation function from $Tm_{EnumField}$ to $TG_{EnumFieldFlags}$:

Definition 5.2.81 (Transformation function $f_{EnumFieldFlags}$)
The transformation function $f_{EnumFieldFlags}(Tm)$ is defined as:

$$
\begin{aligned}
NT &= \{\text{ns\_to\_list}(t) \mid t \in Class_{Tm} \cup Enum_{Tm}\} \\
ET &= \{(\text{ns\_to\_list}(c), \langle f \rangle, \text{ns\_to\_list}(e)) \mid (c, f) \in Field_{Tm} \wedge e \in Enum_{Tm}\} \\
\sqsubseteq &= \{(\text{ns\_to\_list}(x), \text{ns\_to\_list}(x)) \mid x \in Class_{Tm} \cup Enum_{Tm}\} \\
abs &= \{\} \\
\text{mult}(e) &= \begin{cases} (0..*, 1..1) & \text{if } e \in \{(\text{ns\_to\_list}(c), \langle f \rangle, \text{ns\_to\_list}(e)) \mid (c, f) \in Field_{Tm} \wedge e \in Enum_{Tm}\} \end{cases} \\
contains &= \{\}
\end{aligned}
$$

Also see `tmod_enum_field_to_tg_enum_as_flags_field_as_edge_type` *in* `Ecore-GROOVE-Mapping-Library.EnumField`

Theorem 5.2.82 (Correctness of $f_{EnumFieldFlags}$)
$f_{EnumFieldFlags}(Tm)$ (Definition 5.2.81) is a valid transformation function in the sense of Definition 4.3.25 transforming $Tm_{EnumField}$ into $TG_{EnumFieldFlags}$.

Also see `tmod_enum_field_to_tg_enum_as_flags_field_as_edge_type_func` *in* `Ecore-GROOVE-Mapping-Library.EnumField`

The proof of the correctness of $f_{EnumFieldFlags}$ will not be included here. Instead, it can be found in the validated Isabelle theories.

Finally, to complete the transformation, the transformation function that transforms $TG_{EnumFieldFlags}$ into $Tm_{EnumField}$ is defined:

Definition 5.2.83 (Transformation function $f'_{EnumFieldFlags}$)
The transformation function $f'_{EnumFieldFlags}(TG)$ is defined as:

$$
\begin{aligned}
Class &= \{\text{list\_to\_ns}(n) \mid n \in NT_{TG} \wedge n = \text{ns\_to\_list}(classtype)\} \\
Enum &= \{\text{list\_to\_ns}(n) \mid n \in NT_{TG} \wedge n = \text{ns\_to\_list}(enumid)\} \\
UserDataType &= \{\} \\
Field &= \{(\text{list\_to\_ns}(\text{src}(e)), l) \mid e \in ET_{TG} \wedge \langle l \rangle = \text{lab}(e)\} \\
\text{FieldSig} &= \begin{cases} (f, (fieldtype, 1..1)) & \text{if } f \in \{(\text{list\_to\_ns}(\text{src}(e)), l) \mid e \in ET_{TG} \wedge \langle l \rangle = \text{lab}(e)\} \end{cases} \\
EnumValue &= \{(enumid, v) \mid v \in enumvalues\} \\
Inh &= \{\} \\
Prop &= \{\} \\
Constant &= \{\} \\
\text{ConstType} &= \{\}
\end{aligned}
$$

Also see `tg_enum_as_flags_field_as_edge_type_to_tmod_enum_field` *in* `Ecore-GROOVE-Mapping-Library.EnumField`

Theorem 5.2.84 (Correctness of $f'_{EnumFieldFlags}$)
$f'_{EnumFieldFlags}(TG)$ (Definition 5.2.83) is a valid transformation function in the sense of Definition 4.3.30 transforming $TG_{EnumFieldFlags}$ into $Tm_{EnumField}$.

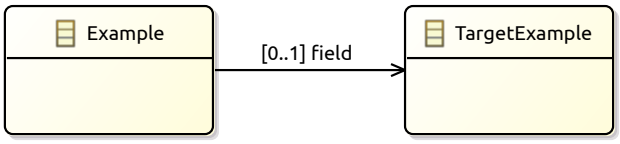
Also see `tg_enum_as_flags_field_as_edge_type_to_tmod_enum_field_func` *in* `Ecore-GROOVE-Mapping-Library.EnumField`

Once more, the correctness proof is not included here but can be found in the validated Isabelle proofs of this thesis.

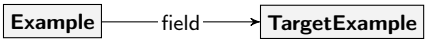
5.2.8 Nullable class fields

The previous sections have shown transformations of fields to attribute types. It has not shown any relations between objects yet. This transformation defines the transformation of a nullable class field.

Definition 5.2.85 (Type model $Tm_{NullableClassField}$)
Let $Tm_{NullableClassField}$ be the type model containing a regular class with identifier classtype. Then



(a) $Tm_{NullableClassField}$ to a class type .TargetExample with $name$ = field



(b) $TG_{NullableClassField}$ to a class type TargetExample with $name$ = field

Figure 5.8: Visualisation of the transformation of a field typed by an optional class type

$Tm_{NullableClassField}$ defines a field named name with type $fieldtype$, in which $fieldtype$ is the identifier of another class type in $Tm_{NullableClassField}$. $Tm_{NullableClassField}$ is defined as:

$$
\begin{aligned}
Class &= \{classtype, fieldtype\} \\
Enum &= \{\} \\
UserDataType &= \{\} \\
Field &= \{(classtype, name)\} \\
\text{FieldSig} &= \begin{cases} (f, (?fieldtype, 0..1)) & \text{if } f \in Field_{Tm_{NullableClassField}} \end{cases} \\
EnumValue &= \{\} \\
Inh &= \{\} \\
Prop &= \{\} \\
Constant &= \{\} \\
\text{ConstType} &= \{\}
\end{aligned}
$$

Also see `tmod_nullable_class_field` *in* `Ecore-GROOVE-Mapping-Library.NullableClassField`

Theorem 5.2.86 (Correctness of $Tm_{NullableClassField}$)
$Tm_{NullableClassField}$ (Definition 5.2.85) is a consistent type model in the sense of Definition 3.2.11.

Also see `tmod_nullable_class_field_correct` *in* `Ecore-GROOVE-Mapping-Library.NullableClassField`

A visual representation of $Tm_{NullableClassField}$ with field name field on class .Example can be seen in Figure 5.8a. In this example, field references a class of .TargetExample. Please note that the lower bound of the multiplicity is 0 as the class is considered nullable. That means that setting a value for field on class .Example is optional. The correctness proof of $Tm_{NullableClassField}$ is more involved, it is not included here for conciseness. It can be found within the validated Isabelle proofs.

In order to make composing transformation functions possible, $Tm_{NullableClassField}$ should be compatible with the type model it is combined with.

Theorem 5.2.87 (Correctness of combine$(Tm, Tm_{NullableClassField})$)
Assume a type model Tm that is consistent in the sense of Definition 3.2.11. Then Tm is compatible with $Tm_{NullableClassField}$ (in the sense of Definition 4.3.13) if:

- *The class type on which the field is defined, $classtype$, is already an existing class in Tm;*
- *The class type which the field targets, $fieldtype$, is already an existing class in Tm;*
- *The field named name is not already a field on $classtype$ in Tm.*

Also see `tmod_nullable_class_field_combine_correct` *in* `Ecore-GROOVE-Mapping-Library.NullableClassField`

Proof. Use Lemma 4.3.12. It is possible to show that all assumptions hold. Now we have shown that combine$(Tm, Tm_{NullableClassField})$ is consistent in the sense of Definition 3.2.11. □

The definitions and theorems for defining a data field within Ecore are now complete.

Encoding as edge type

Like any field definition so far, an edge type will be used within GROOVE to encode the field. The field is transformed into an edge type from the encoded class type to the encoded target type. The encoding corresponding to $Tm_{NullableClassField}$ can then be represented as $TG_{NullableClassField}$, defined in the following definition:

Definition 5.2.88 (Type graph $TG_{NullableClassField}$)
Let $TG_{NullableClassField}$ be the type graph containing a node type which encodes the class type $classtype$. Furthermore, define an edge type from $classtype$ named $name$. This edge type targets a node of $fieldtype$. $TG_{NullableClassField}$ is defined as:

$$\begin{aligned}
NT &= \{\text{ns_to_list}(classtype), \text{ns_to_list}(fieldtype)\} \\
ET &= \{(\text{ns_to_list}(classtype), \langle name\rangle, \text{ns_to_list}(fieldtype))\} \\
\sqsubseteq &= \{(\text{ns_to_list}(classtype), \text{ns_to_list}(classtype)), \\
&\quad\ (\text{ns_to_list}(fieldtype), \text{ns_to_list}(fieldtype))\} \\
abs &= \{\} \\
\text{mult}(e) &= \begin{cases} (0..*, 0..1) & \text{if } e \in \{(\text{ns_to_list}(classtype), \langle name\rangle, \text{ns_to_list}(fieldtype))\} \end{cases} \\
contains &= \{\}
\end{aligned}$$

Also see `tg_nullable_class_field_as_edge_type` *in* `Ecore-GROOVE-Mapping-Library.NullableClassField`

Theorem 5.2.89 (Correctness of $TG_{NullableClassField}$)
$TG_{NullableClassField}$ (Definition 5.2.88) is a valid type graph in the sense of Definition 3.3.5.

Also see `tg_nullable_class_field_as_edge_type_correct` *in* `Ecore-GROOVE-Mapping-Library.NullableClassField`

A visual representation of $TG_{NullableClassField}$ with edge name field on node type Example can be seen in Figure 5.8b. Like the previous example, field references the .TargetExample class, but in this case the encoded node type of TargetExample. The correctness proof of $TG_{NullableClassField}$ is more involved, it is not included here for conciseness. It can be found within the validated Isabelle proofs.

In order to make composing transformation functions possible, $TG_{NullableClassField}$ should be compatible with the type graph it is combined with.

Theorem 5.2.90 (Correctness of combine$(TG, TG_{NullableClassField})$)
Assume a type graph TG that is valid in the sense of Definition 3.3.5. Then TG is compatible with $TG_{NullableClassField}$ (in the sense of Definition 4.3.24) if:

- *The node types of the encoded class types in $TG_{NullableClassField}$ are already node types in TG;*
- *The node type of the encoded class type in $TG_{NullableClassField}$ does not already have an edge type with the same name as the field in TG.*

Also see `tg_nullable_class_field_as_edge_type_combine_correct` *in* `Ecore-GROOVE-Mapping-Library.NullableClassField`

Proof. Use Lemma 4.3.23. It is possible to show that all assumptions hold. Now we have shown that combine$(TG, TG_{NullableClassField})$ is valid in the sense of Definition 3.3.5. □

The next definitions define the transformation function from $Tm_{NullableClassField}$ to $TG_{NullableClassField}$:

Definition 5.2.91 (Transformation function $f_{NullableClassField}$)

The transformation function $f_{NullableClassField}(Tm)$ is defined as:

$$
\begin{aligned}
NT &= \{\text{ns\_to\_list}(c) \mid c \in Class_{Tm}\} \\
ET &= \{(\text{ns\_to\_list}(c), \langle f \rangle, \text{ns\_to\_list}(fieldtype)) \mid (c, n) \in Field_{Tm}\} \\
\sqsubseteq &= \{(\text{ns\_to\_list}(c), \text{ns\_to\_list}(c)) \mid c \in Class_{Tm}\} \\
abs &= \{\} \\
\text{mult} &= \begin{cases} (0..*, 0..1) & \text{if } e \in \{(\text{ns\_to\_list}(c), \langle f \rangle, \text{ns\_to\_list}(fieldtype)) \mid (c, n) \in Field_{Tm}\} \end{cases} \\
contains &= \{\}
\end{aligned}
$$

Also see `tmod_nullable_class_field_to_tg_nullable_class_field_as_edge_type` *in* `Ecore-GROOVE-Mapping-Library.NullableClassField`

Theorem 5.2.92 (Correctness of $f_{NullableClassField}$)
$f_{NullableClassField}(Tm)$ (Definition 5.2.91) is a valid transformation function in the sense of Definition 4.3.25 transforming $Tm_{NullableClassField}$ into $TG_{NullableClassField}$.

Also see `tmod_nullable_class_field_to_tg_nullable_class_field_as_edge_type_func` *in* `Ecore-GROOVE-Mapping-Library.NullableClassField`

The proof of the correctness of $f_{NullableClassField}$ will not be included here. Instead, it can be found in the validated Isabelle theories.

Finally, to complete the transformation, the transformation function that transforms $TG_{NullableClassField}$ into $Tm_{NullableClassField}$ is defined:

Definition 5.2.93 (Transformation function $f'_{NullableClassField}$)
The transformation function $f'_{NullableClassField}(TG)$ is defined as:

$$
\begin{aligned}
Class &= \{\text{list\_to\_ns}(n) \mid n \in NT_{TG}\} \\
Enum &= \{\} \\
UserDataType &= \{\} \\
Field &= \{(\text{list\_to\_ns}(\text{src}(e)), l) \mid e \in ET_{TG} \wedge \langle l \rangle = \text{lab}(e)\} \\
\text{FieldSig} &= \begin{cases} (f, (?fieldtype, 0..1)) & \text{if } f \in \{(\text{list\_to\_ns}(\text{src}(e)), l) \mid e \in ET_{TG} \wedge \langle l \rangle = \text{lab}(e)\} \end{cases} \\
EnumValue &= \{\} \\
Inh &= \{\} \\
Prop &= \{\} \\
Constant &= \{\} \\
\text{ConstType} &= \{\}
\end{aligned}
$$

Also see `tg_nullable_class_field_as_edge_type_to_tmod_nullable_class_field` *in* `Ecore-GROOVE-Mapping-Library.NullableClassField`

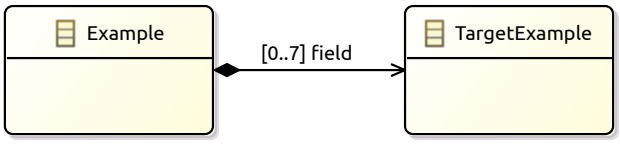
Theorem 5.2.94 (Correctness of $f'_{NullableClassField}$)
$f'_{NullableClassField}(TG)$ (Definition 5.2.93) is a valid transformation function in the sense of Definition 4.3.30 transforming $TG_{NullableClassField}$ into $Tm_{NullableClassField}$.

Also see `tg_nullable_class_field_as_edge_type_to_tmod_nullable_class_field_func` *in* `Ecore-GROOVE-Mapping-Library.NullableClassField`

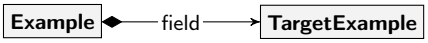
Once more, the correctness proof is not included here but can be found in the validated Isabelle proofs of this thesis.

5.2.9 Contained class set fields

In this section, the transformation of a containment field typed by a set of a proper class type is shown. This transformation defines a field which has a value a set of objects of a specified class type, that is not nullable. Furthermore, the objects referenced by the field are contained by the source class. First, the Ecore type model is defined.



(a) $Tm_{ContainedClassSetField}$ to a class type .TargetExample with $name$ = field



(b) $TG_{ContainedClassSetField}$ to a class type TargetExample with $name$ = field

Figure 5.9: Visualisation of the transformation of a containment field typed by a set of a proper class type

Definition 5.2.95 (Type model $Tm_{ContainedClassSetField}$)
Let $Tm_{ContainedClassSetField}$ be the type model containing a regular class with identifier classtype. Then $Tm_{ContainedClassSetField}$ defines a field named name with type containedtype, in which containedtype is the identifier of another class type in $Tm_{ContainedClassSetField}$. Furthermore, define mul to be a valid multiplicity for the field name. $Tm_{ContainedClassSetField}$ is defined as:

$$
\begin{aligned}
Class &= \{classtype, containedtype\} \\
Enum &= \{\} \\
UserDataType &= \{\} \\
Field &= \{(classtype, name)\} \\
\text{FieldSig} &= \begin{cases} (f, (\mathsf{setof}, !containedtype), mul)) & \text{if } f \in Field_{Tm_{ContainedClassSetField}} \end{cases} \\
EnumValue &= \{\} \\
Inh &= \{\} \\
Prop &= \{(\mathsf{containment}, (classtype, name))\} \\
Constant &= \{\} \\
\text{ConstType} &= \{\}
\end{aligned}
$$

Also see `tmod_contained_class_set_field` *in* `Ecore-GROOVE-Mapping-Library.ContainedClassSetField`

Theorem 5.2.96 (Correctness of $Tm_{ContainedClassSetField}$)
$Tm_{ContainedClassSetField}$ (Definition 5.2.95) is a consistent type model in the sense of Definition 3.2.11.

Also see `tmod_contained_class_set_field_correct` *in* `Ecore-GROOVE-Mapping-Library.ContainedClassSetField`

A visual representation of $Tm_{ContainedClassSetField}$ with field name field on class .Example can be seen in Figure 5.9a. In this example, field references a class of .TargetExample. Please note that the multiplicity 0..7 has been chosen here as an example, any valid multiplicity could have been used. Also notice that the field is in fact a containment relation. The correctness proof of $Tm_{ContainedClassSetField}$ is more involved, it is not included here for conciseness. It can be found within the validated Isabelle proofs.

In order to make composing transformation functions possible, $Tm_{ContainedClassSetField}$ should be compatible with the type model it is combined with.

Theorem 5.2.97 (Correctness of combine$(Tm, Tm_{ContainedClassSetField})$)
Assume a type model Tm that is consistent in the sense of Definition 3.2.11. Then Tm is compatible with $Tm_{ContainedClassSetField}$ (in the sense of Definition 4.3.13) if:

- *The class type on which the field is defined, classtype, is already an existing class in Tm;*
- *The class type which the field targets, containedtype, is already an existing class in Tm;*
- *The field named name is not already a field on classtype in Tm;*
- *The multiplicity mul is valid.*

Also see `tmod_contained_class_set_field_combine_correct` *in* `Ecore-GROOVE-Mapping-Library.ContainedClassSetField`

Proof. Use Lemma 4.3.12. It is possible to show that all assumptions hold. Now we have shown that combine($Tm, Tm_{ContainedClassSetField}$) is consistent in the sense of Definition 3.2.11. □

The definitions and theorems for defining a data field within Ecore are now complete.

Encoding as edge type

Like the previous encodings of fields, an edge type in GROOVE will be used as encoding for the field. The field is transformed into an edge type from the encoded class type to the encoded target type. Furthermore, the edge type will be a containment edge, to support the fact that the target nodes are contained by the source nodes. The encoding corresponding to $Tm_{ContainedClassSetField}$ can then be represented as $TG_{ContainedClassSetField}$, defined in the following definition:

Definition 5.2.98 (Type graph $TG_{ContainedClassSetField}$)
Let $TG_{ContainedClassSetField}$ be the type graph containing a node type which encodes the class type classtype. Furthermore, define an edge type from classtype named name. This edge type targets a node of containedtype. Finally, define multiplicity mul to be $TG_{ContainedClassSetField}$ is defined as:

$$
\begin{aligned}
NT &= \{\text{ns\_to\_list}(\mathit{classtype}), \text{ns\_to\_list}(\mathit{containedtype})\} \\
ET &= \{(\text{ns\_to\_list}(\mathit{classtype}), \langle \mathit{name} \rangle, \text{ns\_to\_list}(\mathit{containedtype}))\} \\
\sqsubseteq &= \{(\text{ns\_to\_list}(\mathit{classtype}), \text{ns\_to\_list}(\mathit{classtype})), \\
&\quad\ (\text{ns\_to\_list}(\mathit{containedtype}), \text{ns\_to\_list}(\mathit{containedtype}))\} \\
\mathit{abs} &= \{\} \\
\text{mult}(e) &= \begin{cases} (0..1, \mathit{mul}) & \text{if } e \in \{(\text{ns\_to\_list}(\mathit{classtype}), \langle \mathit{name} \rangle, \text{ns\_to\_list}(\mathit{containedtype}))\} \end{cases} \\
\mathit{contains} &= \{(\text{ns\_to\_list}(\mathit{classtype}), \langle \mathit{name} \rangle, \text{ns\_to\_list}(\mathit{containedtype}))\}
\end{aligned}
$$

Also see `tg_contained_class_set_field_as_edge_type` *in* `Ecore-GROOVE-Mapping-Library.ContainedClassSetField`

Theorem 5.2.99 (Correctness of $TG_{ContainedClassSetField}$)
$TG_{ContainedClassSetField}$ (Definition 5.2.98) is a valid type graph in the sense of Definition 3.3.5.

Also see `tg_contained_class_set_field_as_edge_type_correct` *in* `Ecore-GROOVE-Mapping-Library.ContainedClassSetField`

A visual representation of $TG_{ContainedClassSetField}$ with edge name field on node type Example can be seen in Figure 5.9b. Like the previous example, field references the .TargetExample class, but in this case the encoded node type of TargetExample. Furthermore, it is visable that the introduced edge type for the field is indeed a containment edge. The correctness proof of $TG_{ContainedClassSetField}$ is more involved, it is not included here for conciseness. It can be found within the validated Isabelle proofs.

In order to make composing transformation functions possible, $TG_{ContainedClassSetField}$ should be compatible with the type graph it is combined with.

Theorem 5.2.100 (Correctness of combine($TG, TG_{ContainedClassSetField}$))
Assume a type graph TG that is valid in the sense of Definition 3.3.5. Then TG is compatible with $TG_{ContainedClassSetField}$ (in the sense of Definition 4.3.24) if:

- *The node types of the encoded class types in $TG_{ContainedClassSetField}$ are already node types in TG.*
- *The node type of the encoded class type in $TG_{ContainedClassSetField}$ does not already have an edge type with the same name as the field in TG;*
- *The multiplicity mul is valid.*

Also see `tg_contained_class_set_field_as_edge_type_combine_correct` *in* `Ecore-GROOVE-Mapping-Library.ContainedClassSetField`

Proof. Use Lemma 4.3.23. It is possible to show that all assumptions hold. Now we have shown that combine($TG, TG_{ContainedClassSetField}$) is valid in the sense of Definition 3.3.5. □

The next definitions define the transformation function from $Tm_{ContainedClassSetField}$ to $TG_{ContainedClassSetField}$:

Definition 5.2.101 (Transformation function $f_{ContainedClassSetField}$)
The transformation function $f_{ContainedClassSetField}(Tm)$ *is defined as:*

$$
\begin{aligned}
NT &= \{\text{ns\_to\_list}(c) \mid c \in Class_{Tm}\} \\
ET &= \{(\text{ns\_to\_list}(c), \langle f \rangle, \text{ns\_to\_list}(containedtype)) \mid (c, n) \in Field_{Tm}\} \\
\sqsubseteq &= \{(\text{ns\_to\_list}(c), \text{ns\_to\_list}(c)) \mid c \in Class_{Tm}\} \\
abs &= \{\} \\
\text{mult} &= \begin{cases} (0..1, mul) & \text{if } e \in \{(\text{ns\_to\_list}(c), \langle f \rangle, \text{ns\_to\_list}(containedtype)) \mid (c, n) \in Field_{Tm}\} \end{cases} \\
contains &= \{(\text{ns\_to\_list}(c), \langle f \rangle, \text{ns\_to\_list}(containedtype)) \mid (c, n) \in Field_{Tm}\}
\end{aligned}
$$

Also see `tmod_contained_class_set_field_to_tg_contained_class_set_field_as_edge_type` *in* `Ecore-GROOVE-Mapping-Library.ContainedClassSetField`

Theorem 5.2.102 (Correctness of $f_{ContainedClassSetField}$)
$f_{ContainedClassSetField}(Tm)$ *(Definition 5.2.101) is a valid transformation function in the sense of Definition 4.3.25 transforming* $Tm_{ContainedClassSetField}$ *into* $TG_{ContainedClassSetField}$.

Also see `tmod_contained_class_set_field_to_tg_contained_class_set_field_as_edge_type_func` *in* `Ecore-GROOVE-Mapping-Library.ContainedClassSetField`

The proof of the correctness of $f_{ContainedClassSetField}$ will not be included here. Instead, it can be found in the validated Isabelle theories.

Finally, to complete the transformation, the transformation function that transforms $TG_{ContainedClassSetField}$ into $Tm_{ContainedClassSetField}$ is defined:

Definition 5.2.103 (Transformation function $f'_{ContainedClassSetField}$)
The transformation function $f'_{ContainedClassSetField}(TG)$ *is defined as:*

$$
\begin{aligned}
Class &= \{\text{list\_to\_ns}(n) \mid n \in NT_{TG}\} \\
Enum &= \{\} \\
UserDataType &= \{\} \\
Field &= \{(\text{list\_to\_ns}(\text{src}(e)), l) \mid e \in ET_{TG} \wedge \langle l \rangle = \text{lab}(e)\} \\
\text{FieldSig} &= \begin{cases} (f, ((\mathsf{setof}, !containedtype), mul)) & \text{if } f \in \{(\text{list\_to\_ns}(\text{src}(e)), l) \mid e \in ET_{TG} \wedge \\ & \qquad \langle l \rangle = \text{lab}(e)\} \end{cases} \\
EnumValue &= \{\} \\
Inh &= \{\} \\
Prop &= \{(\mathsf{containment}, (\text{list\_to\_ns}(\text{src}(e)), l)) \mid e \in ET_{TG} \wedge \langle l \rangle = \text{lab}(e)\} \\
Constant &= \{\} \\
\text{ConstType} &= \{\}
\end{aligned}
$$

Also see `tg_contained_class_set_field_as_edge_type_to_tmod_contained_class_set_field` *in* `Ecore-GROOVE-Mapping-Library.ContainedClassSetField`

Theorem 5.2.104 (Correctness of $f'_{ContainedClassSetField}$)
$f'_{ContainedClassSetField}(TG)$ *(Definition 5.2.103) is a valid transformation function in the sense of Definition 4.3.30 transforming* $TG_{ContainedClassSetField}$ *into* $Tm_{ContainedClassSetField}$.

Also see `tg_contained_class_set_field_as_edge_type_to_tmod_contained_class_set_field_func` *in* `Ecore-GROOVE-Mapping-Library.ContainedClassSetField`

Once more, the correctness proof is not included here but can be found in the validated Isabelle proofs of this thesis.

5.3 Instance level transformations

The previous section has presented a set of model transformations between type models and type graphs. In this section, these transformations are used to define model transformations between instance model

and instance graphs. Therefore, this section provides the second set of model transformations introduced within the introduction of this chapter.

Throughout this section, small transformations between instance models and instance graphs will be defined. In order for these transformations useful in the context of the transformation framework of Chapter 4, some properties must hold for each of them. For each transformation, the corresponding instance model and instance graph must be valid in the sense of Definition 3.2.19 and Definition 3.3.10 respectively. Furthermore, the instance model corresponding to the transformation must be compatible with its counterpart in the transformation framework. In the same way, the corresponding instance graph must be compatible with its counterpart in the transformation framework. Moreover, it will be shown that the transformation function f that transforms the corresponding instance model into an instance graph is a valid transformation function in the sense of Definition 4.4.26. Finally, it will also be shown that the reverse transformation is a valid transformation function in the sense of Definition 4.4.31.

5.3.1 Plain objects

The first transformation that will be defined is a transformation of plain objects of a regular class type. The corresponding type level transformation can be found in Section 5.2.1. This transformation introduces an arbitrary amount of instances of the class introduced on the type level. First, the definition of the corresponding instance model is given.

Definition 5.3.1 (Instance model Im_{Class})
Let Im_{Class} be the instance model containing a set of objects objects which are all typed by class name. Furthermore, an injective function fid is defined which maps every object in the set to its corresponding identifier. Im_{Class} is typed by Tm_{Class} (Definition 5.2.1) and is defined as:

$$\begin{aligned} Object &= objects \\ \text{ObjectClass} &= \{(ob, name) \quad \text{if } ob \in objects \\ \text{ObjectId} &= \{(ob, fid(ob)) \quad \text{if } ob \in objects \\ \text{FieldValue} &= \{\} \\ \text{DefaultValue} &= \{\} \end{aligned}$$

Also see `imod_class` *in* `Ecore-GROOVE-Mapping-Library.ClassInstance`

Theorem 5.3.2 (Correctness of Im_{Class})
Im_{Class} (Definition 5.3.1) is a valid instance model in the sense of Definition 3.2.19.

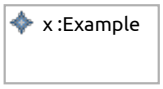
Also see `imod_class_correct` *in* `Ecore-GROOVE-Mapping-Library.ClassInstance`

A visual representation of Im_{Class} with $objects = \{ob\}$ and $fid(ob) = x$ can be seen in Figure 5.10a. Although this visualisation only shows one object, it is possible to have an arbitrary amount of objects in Im_{Class}, as long as they are all typed by the corresponding class introduced on the type level. In the visualisation, the identifier .Example is used for the class, in correspondence with Figure 5.1a The correctness proof of Im_{Class} is trivial, and therefore not included here. The proof can be found as part of the Isabelle validated proofs.

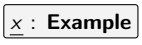
In order to make composing transformation functions possible, Im_{Class} should be compatible with the instance model it is combined with.

Theorem 5.3.3 (Correctness of combine(Im, Im_{Class}))
Assume an instance model Im that is valid in the sense of Definition 3.2.19. Then Im is compatible with Im_{Class} (in the sense of Definition 4.4.14) if:

- *All requirements of Theorem 5.2.3 are met, to ensure the combination of the corresponding type models is valid;*



(a) Im_{Class} with one object identified as x



(b) IG_{Class} with one node identified as x

Figure 5.10: Visualisation of the transformation of plain objects typed by regular classes

- *All the objects in Im_{Class} have an (internal and explicit) identity that is not yet used in Im.*

Also see `imod_class_combine_correct` *in* `Ecore-GROOVE-Mapping-Library.ClassInstance`

Proof. Use Lemma 4.4.13. It is possible to show that all assumptions hold. Now we have shown that combine(Im, Im_{Class}) is consistent in the sense of Definition 3.2.19. □

Please note that this combination is quite trivial, as the newly introduced objects cannot have fields. This is because they are all typed by the new class type introduced in Tm_{Class}. Since this new class type is new by assumption, the existing model cannot have fields defined for the class type.

The definitions and theorems for introducing plain objects of regular classes within Ecore are now complete.

Encoding as nodes

A possible encoding for plain objects in Ecore is using nodes in GROOVE. Each node is typed by the node type that was introduced in TG_{Class}, and copies the identifiers set of the objects to the corresponding nodes. The encoding corresponding to Im_{Class} can then be represented as IG_{Class}, defined in the following definition:

Definition 5.3.4 (Instance graph IG_{Class})
Let IG_{Class} be the instance graph with as nodes the converted objects of Im_{Class} (Definition 5.3.1). Furthermore, reuse the injective function fid that maps every object to its identifier. Finally, use the node type introduced in TG_{Class} (Definition 5.2.4). IG_{Class} is defined typed by TG_{Class} and is defined as:

$$\begin{aligned} N &= objects \\ E &= \{\} \\ \text{ident} &= \begin{cases} (fid(ob), ob) & \text{if } ob \in objects \end{cases} \end{aligned}$$

with

$$\text{type}_n = \begin{cases} (ob, \text{ns\_to\_list}(name)) & \text{if } ob \in objects \end{cases}$$

Also see `ig_class_as_node_type` *in* `Ecore-GROOVE-Mapping-Library.ClassInstance`

Theorem 5.3.5 (Correctness of IG_{Class})
IG_{Class} (Definition 5.3.4) is a valid instance graph in the sense of Definition 3.3.10.

Also see `ig_class_as_node_type_correct` *in* `Ecore-GROOVE-Mapping-Library.ClassInstance`

A visual representation of IG_{Class} with $objects = \{ob\}$ and $fid(ob) = x$ can be seen in Figure 5.10b. Like the previous example for the Ecore instance model, only one node is shown here, but multiple nodes can be introduced at once if there are more objects in the *objects* set. As shown in the definition, the node type identified by Example is used to type all the nodes, in correspondence with Figure 5.1b. The correctness proof of IG_{Class} is trivial, and therefore not included here. The proof can be found as part of the Isabelle validated proofs.

In order to make composing transformation functions possible, IG_{Class} should be compatible with the instance graph it is combined with.

Theorem 5.3.6 (Correctness of combine(IG, IG_{Class}))
Assume an instance graph IG that is valid in the sense of Definition 3.3.10. Then IG is compatible with IG_{Class} (in the sense of Definition 4.4.25) if:

- *All requirements of Theorem 5.2.6 are met, to ensure the combination of the corresponding type graphs is valid;*
- *All the nodes in IG_{Class} have an (internal and explicit) identity that is not yet used in IG.*

Also see `ig_class_as_node_type_combine_correct` *in* `Ecore-GROOVE-Mapping-Library.ClassInstance`

Proof. Use Lemma 4.4.24. It is possible to show that all assumptions hold. Now we have shown that combine(IG, IG_{Class}) is valid in the sense of Definition 3.3.10. □

The next definitions define the transformation function from Im_{Class} to IG_{Class}:

Definition 5.3.7 (Transformation function f_{Class})
The transformation function $f_{Class}(Im)$ is defined as:

$$\begin{aligned} N &= Object_{Im} \\ E &= \{\} \\ \text{ident} &= \begin{cases} (fid(ob), ob) & \text{if } ob \in Object_{Im} \end{cases} \end{aligned}$$

with

$$\text{type}_n = \begin{cases} (ob, \text{ns\_to\_list}(name)) & \text{if } ob \in Object_{Im} \end{cases}$$

Also see `imod_class_to_ig_class_as_node_type` *in* `Ecore-GROOVE-Mapping-Library.ClassInstance`

Theorem 5.3.8 (Correctness of f_{Class})
$f_{Class}(Im)$ (Definition 5.3.7) is a valid transformation function in the sense of Definition 4.4.26 transforming Im_{Class} into IG_{Class}.

Also see `imod_class_to_ig_class_as_node_type_func` *in* `Ecore-GROOVE-Mapping-Library.ClassInstance`

The proof of the correctness of f_{Class} will not be included here. Instead, it can be found in the validated Isabelle theories. The proof is quite trivial, as extending Im can only add extra objects, but not remove the existing ones.

Finally, to complete the transformation, the transformation function that transforms IG_{Class} into Im_{Class} is defined:

Definition 5.3.9 (Transformation function f'_{Class})
The transformation function $f'_{Class}(IG)$ is defined as:

$$\begin{aligned} Object &= N_{IG} \\ \text{ObjectClass} &= \begin{cases} (ob, name) & \text{if } ob \in N_{IG} \end{cases} \\ \text{ObjectId} &= \begin{cases} (ob, fid(ob)) & \text{if } ob \in N_{IG} \end{cases} \\ \text{FieldValue} &= \{\} \\ \text{DefaultValue} &= \{\} \end{aligned}$$

Also see `ig_class_as_node_type_to_imod_class` *in* `Ecore-GROOVE-Mapping-Library.ClassInstance`

Theorem 5.3.10 (Correctness of f'_{Class})
$f'_{Class}(IG)$ (Definition 5.3.9) is a valid transformation function in the sense of Definition 4.4.31 transforming IG_{Class} into Im_{Class}.

Also see `ig_class_as_node_type_to_imod_class_func` *in* `Ecore-GROOVE-Mapping-Library.ClassInstance`

Once more, the correctness proof is not included here but can be found in the validated Isabelle proofs of this thesis.

5.3.2 Abstract classes

In this section, the instance level transformation corresponding to the type level transformation of abstract classes is discussed. The type level transformation of abstract class types can be found in Section 5.2.2.

Informally speaking, it is quite weird to think about the transformation of abstract classes on the instance level, as abstract classes cannot have instances. The transformation here is included for completeness, to allow for adding abstract types while working with transformations on the type level. In practice, the

instance level will consist of the empty instance model and empty instance graph, showing that after adding an abstract class on the type level, instances of the type model will still be valid.

First, the corresponding instance model is introduced.

Definition 5.3.11 (Instance model $Im_{AbsClass}$)
Let $Im_{AbsClass}$ be the empty instance model Im_ϵ (Definition 4.4.9), except that it is typed by the type model $Tm_{AbsClass}$ (Definition 5.2.11).

Also see `imod_abstract_class` *in* `Ecore-GROOVE-Mapping-Library.AbstractClassInstance`

Theorem 5.3.12 (Correctness of $Im_{AbsClass}$)
$Im_{AbsClass}$ (Definition 5.3.11) is a valid instance model in the sense of Definition 3.2.19.

Also see `imod_abstract_class_correct` *in* `Ecore-GROOVE-Mapping-Library.AbstractClassInstance`

Since $Im_{AbsClass}$ does not define any objects, there is no need for a visual representation. However, in order to make composing transformation functions possible, $Im_{AbsClass}$ should still be compatible with the instance model it is combined with.

Theorem 5.3.13 (Correctness of combine$(Im, Im_{AbsClass})$)
Assume an instance model Im that is valid in the sense of Definition 3.2.19. Then Im is compatible with $Im_{AbsClass}$ (in the sense of Definition 4.4.14) if:

- *All requirements of Theorem 5.2.13 are met, to ensure the combination of the corresponding type models is valid.*

Also see `imod_abstract_class_combine_correct` *in* `Ecore-GROOVE-Mapping-Library.AbstractClassInstance`

Proof. Use Lemma 4.4.13. It is possible to show that all assumptions hold. Now we have shown that combine$(Im, Im_{AbsClass})$ is consistent in the sense of Definition 3.2.19. □

The definitions and theorems for the Ecore instance model corresponding to $Tm_{AbsClass}$ are now complete.

The node type encoding

As has been shown earlier, an possible encoding for abstract class types is by introducing an abstract node type. This has been done in $TG_{AbsClass}$. Like the Ecore instance model, the GROOVE instance graph is also empty, because abstract node types cannot be instantiated. This gives rise to $IG_{AbsClass}$, which is defined as follows:

Definition 5.3.14 (Instance graph $IG_{AbsClass}$)
Let $IG_{AbsClass}$ be the empty instance graph IG_ϵ (Definition 4.4.20), except that it is typed by the type graph $TG_{AbsClass}$ (Definition 5.2.14).

Also see `ig_abstract_class_as_node_type` *in* `Ecore-GROOVE-Mapping-Library.AbstractClassInstance`

Theorem 5.3.15 (Correctness of $IG_{AbsClass}$)
$IG_{AbsClass}$ (Definition 5.3.14) is a valid instance graph in the sense of Definition 3.3.10.

Also see `ig_abstract_class_as_node_type_correct` *in* `Ecore-GROOVE-Mapping-Library.AbstractClassInstance`

In order to make composing transformation functions possible, $IG_{AbsClass}$ should be compatible with the instance graph it is combined with.

Theorem 5.3.16 (Correctness of combine$(IG, IG_{AbsClass})$)
Assume an instance graph IG that is valid in the sense of Definition 3.3.10. Then IG is compatible with $IG_{AbsClass}$ (in the sense of Definition 4.4.25) if:

(a) $Im_{Subclass}$ with one object identified as x

(b) $IG_{Subclass}$ with one node identified as x

Figure 5.11: Visualisation of the transformation of objects typed by a subclass

- *All requirements of Theorem 5.2.16 are met, to ensure the combination of the corresponding type graphs is valid.*

Also see `ig_abstract_class_as_node_type_combine_correct` *in* `Ecore-GROOVE-Mapping-Library.AbstractClassInstance`

Proof. Use Lemma 4.4.24. It is possible to show that all assumptions hold. Now we have shown that combine$(IG, IG_{AbsClass})$ is valid in the sense of Definition 3.3.10. □

The next definitions define the transformation function from $Im_{AbsClass}$ to $IG_{AbsClass}$:

Definition 5.3.17 (Transformation function $f_{AbsClass}$)
The transformation function $f_{AbsClass}(Im)$ is defined as the function that always outputs the empty instance graph IG_ϵ (Definition 4.4.20), except that it is typed by $TG_{AbsClass}$.

Also see `imod_abstract_class_to_ig_abstract_class_as_node_type` *in* `Ecore-GROOVE-Mapping-Library.AbstractClassInstance`

Theorem 5.3.18 (Correctness of $f_{AbsClass}$)
$f_{AbsClass}(Im)$ (Definition 5.3.17) is a valid transformation function in the sense of Definition 4.4.26 transforming $Im_{AbsClass}$ into $IG_{AbsClass}$.

Also see `imod_abstract_class_to_ig_abstract_class_as_node_type_func` *in* `Ecore-GROOVE-Mapping-Library.AbstractClassInstance`

The proof of the correctness of $f_{AbsClass}$ will not be included here. Instead, it can be found in the validated Isabelle theories. Obviously, the proof is trivial, as the function does not do any conversion. It does just output the empty instance model.

Finally, to complete the transformation, the transformation function that transforms $IG_{AbsClass}$ into $Im_{AbsClass}$ is defined:

Definition 5.3.19 (Transformation function $f'_{AbsClass}$)
The transformation function $f'_{AbsClass}(IG)$ is defined as the function that always outputs the empty instance model Im_ϵ (Definition 4.4.9), except that it is typed by $Tm_{AbsClass}$.

Also see `ig_abstract_class_as_node_type_to_imod_abstract_class` *in* `Ecore-GROOVE-Mapping-Library.AbstractClassInstance`

Theorem 5.3.20 (Correctness of $f'_{AbsClass}$)
$f'_{AbsClass}(IG)$ (Definition 5.3.19) is a valid transformation function in the sense of Definition 4.4.31 transforming $IG_{AbsClass}$ into $Im_{AbsClass}$.

Also see `ig_abstract_class_as_node_type_to_imod_abstract_class_func` *in* `Ecore-GROOVE-Mapping-Library.AbstractClassInstance`

Once more, the correctness proof is not included here but can be found in the validated Isabelle proofs of this thesis.

5.3.3 Plain objects typed by a subclass

In this section, the transformation of plain objects typed by a regular subclass is discussed. The corresponding type level transformation can be found in Section 5.2.3. This transformation is very similar to the the transformation of plain objects discussed in Section 5.3.1. Like that transformation, it is possible to introduce an arbitrary amount of instances of the subclass introduced on the type level. First, the definition of the corresponding instance model is given.

Definition 5.3.21 (Instance model $Im_{Subclass}$)
Let $Im_{Subclass}$ be the instance model containing a set of objects objects which are all typed by subclass name, which extends class supertype. Furthermore, an injective function fid is defined which maps every object in the set to its corresponding identifier. $Im_{Subclass}$ is typed by $Tm_{Subclass}$ (Definition 5.2.21) and is defined as:

$$\begin{aligned} Object &= objects \\ \text{ObjectClass} &= \Big\{(ob, name) \quad \text{if } ob \in objects \\ \text{ObjectId} &= \Big\{(ob, fid(ob)) \quad \text{if } ob \in objects \\ \text{FieldValue} &= \{\} \\ \text{DefaultValue} &= \{\} \end{aligned}$$

Also see `imod_subclass` *in* `Ecore-GROOVE-Mapping-Library.SubclassInstance`

Theorem 5.3.22 (Correctness of $Im_{Subclass}$)
$Im_{Subclass}$ (Definition 5.3.21) is a valid instance model in the sense of Definition 3.2.19.

Also see `imod_subclass_correct` *in* `Ecore-GROOVE-Mapping-Library.SubclassInstance`

A visual representation of $Im_{Subclass}$ with $objects = \{ob\}$ and $fid(ob) = x$ can be seen in Figure 5.11a. Although this visualisation only shows one object, it is possible to have an arbitrary amount of objects in $Im_{Subclass}$, as long as they are all typed by the corresponding class introduced on the type level. In the visualisation, the identifier .NewType is used for the class, in correspondence with Figure 5.3a The correctness proof of $Im_{Subclass}$ is trivial, and therefore not included here. The proof can be found as part of the Isabelle validated proofs.

In order to make composing transformation functions possible, $Im_{Subclass}$ should be compatible with the instance model it is combined with.

Theorem 5.3.23 (Correctness of combine$(Im, Im_{Subclass})$)
Assume an instance model Im that is valid in the sense of Definition 3.2.19. Then Im is compatible with $Im_{Subclass}$ (in the sense of Definition 4.4.14) if:

- *All requirements of Theorem 5.2.23 are met, to ensure the combination of the corresponding type models is valid;*
- *All the objects in $Im_{Subclass}$ have an (internal and explicit) identity that is not yet used in Im;*
- *Im is not typed by a type model that defines any fields for the supertype class.*

Also see `imod_subclass_combine_correct` *in* `Ecore-GROOVE-Mapping-Library.SubclassInstance`

Proof. Use Lemma 4.4.13. It is possible to show that all assumptions hold. Now we have shown that combine$(Im, Im_{Subclass})$ is consistent in the sense of Definition 3.2.19. □

Please note that in this case, it has been made explicit that the new objects introduced do not have any fields defined. This is by ensuring the supertype does not define any fields. The new subclass does not have fields itself, as it cannot have existed in the combined type model.

The definitions and theorems for introducing plain objects of regular subclasses within Ecore are now complete.

Encoding as nodes

As was the case with plain objects of regular classes, a possible encoding for plain objects of subclasses in Ecore is using nodes in GROOVE. Each node is typed by the node type that was introduced in $TG_{Subclass}$, and copies the identifiers set of the objects to the corresponding nodes. The encoding corresponding to $Im_{Subclass}$ can then be represented as $IG_{Subclass}$, defined in the following definition:

Definition 5.3.24 (Instance graph $IG_{Subclass}$)
Let $IG_{Subclass}$ be the instance graph with as nodes the converted objects of $Im_{Subclass}$ (Definition 5.3.21). Furthermore, reuse the injective function fid that maps every object to its identifier. Finally, use the

node type name introduced in $TG_{Subclass}$, that extends the supertype node type. (Definition 5.2.4). $IG_{Subclass}$ is defined typed by $TG_{Subclass}$ and is defined as:

$$\begin{aligned} N &= objects \\ E &= \{\} \\ \text{ident} &= \begin{cases} (fid(ob), ob) & \text{if } ob \in objects \end{cases} \end{aligned}$$

with

$$\text{type}_n = \begin{cases} (ob, \text{ns\_to\_list}(name)) & \text{if } ob \in objects \end{cases}$$

Also see `ig_subclass_as_node_type` *in* `Ecore-GROOVE-Mapping-Library.SubclassInstance`

Theorem 5.3.25 (Correctness of $IG_{Subclass}$)
$IG_{Subclass}$ (Definition 5.3.24) is a valid instance graph in the sense of Definition 3.3.10.

Also see `ig_subclass_as_node_type_correct` *in* `Ecore-GROOVE-Mapping-Library.SubclassInstance`

A visual representation of $IG_{Subclass}$ with $objects = \{ob\}$ and $fid(ob) = x$ can be seen in Figure 5.11b. Like the previous example for the Ecore instance model, only one node is shown here, but multiple nodes can be introduced at once if there are more objects in the *objects* set. As shown in the definition, the node type identified by NewType is used to type all the nodes, in correspondence with Figure 5.3b. The correctness proof of $IG_{Subclass}$ is trivial, and therefore not included here. The proof can be found as part of the Isabelle validated proofs.

In order to make composing transformation functions possible, $IG_{Subclass}$ should be compatible with the instance graph it is combined with.

Theorem 5.3.26 (Correctness of combine$(IG, IG_{Subclass})$)
Assume an instance graph IG that is valid in the sense of Definition 3.3.10. Then IG is compatible with $IG_{Subclass}$ (in the sense of Definition 4.4.25) if:

- *All requirements of Theorem 5.2.26 are met, to ensure the combination of the corresponding type graphs is valid;*
- *All the nodes in $IG_{Subclass}$ have an (internal and explicit) identity that is not yet used in IG;*
- *There are no edge types from or to the supertype node type, this includes edges from and to types that supertype inherits from.*

Also see `ig_subclass_as_node_type_combine_correct` *in* `Ecore-GROOVE-Mapping-Library.SubclassInstance`

Proof. Use Lemma 4.4.24. It is possible to show that all assumptions hold. Now we have shown that combine$(IG, IG_{Subclass})$ is valid in the sense of Definition 3.3.10. □

Like the correctness for the Ecore instance model, validity is guaranteed here by assuming there exist no edge types from and to the *supertype* node type.

The next definitions define the transformation function from $Im_{Subclass}$ to $IG_{Subclass}$:

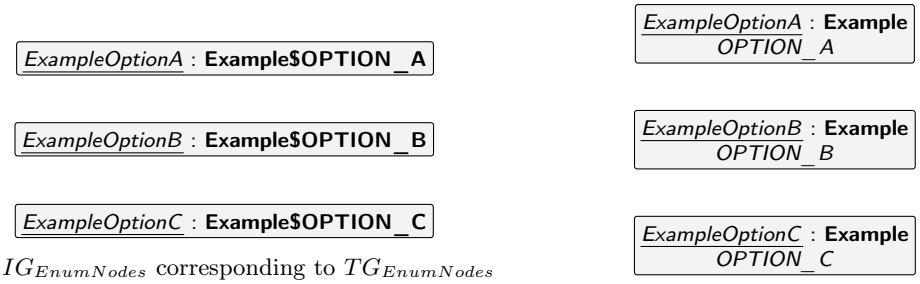
Definition 5.3.27 (Transformation function $f_{Subclass}$)
The transformation function $f_{Subclass}(Im)$ is defined as:

$$\begin{aligned} N &= Object_{Im} \\ E &= \{\} \\ \text{ident} &= \begin{cases} (fid(ob), ob) & \text{if } ob \in Object_{Im} \end{cases} \end{aligned}$$

with

$$\text{type}_n = \begin{cases} (ob, \text{ns\_to\_list}(name)) & \text{if } ob \in Object_{Im} \end{cases}$$

Also see `imod_subclass_to_ig_subclass_as_node_type` *in* `Ecore-GROOVE-Mapping-Library.SubclassInstance`



(a) $IG_{EnumNodes}$ corresponding to $TG_{EnumNodes}$

(b) $IG_{EnumFlags}$ corresponding to $TG_{EnumFlags}$

Figure 5.12: Visualisation of the transformation of enumeration values corresponding to an enumeration type

Theorem 5.3.28 (Correctness of $f_{Subclass}$)
$f_{Subclass}(Im)$ (Definition 5.3.27) is a valid transformation function in the sense of Definition 4.4.26 transforming $Im_{Subclass}$ into $IG_{Subclass}$.

Also see `imod_subclass_to_ig_subclass_as_node_type_func` *in* `Ecore-GROOVE-Mapping-Library.SubclassInstance`

The proof of the correctness of $f_{Subclass}$ will not be included here. Instead, it can be found in the validated Isabelle theories. The proof is quite trivial, as extending Im can only add extra objects, but not remove the existing ones.

Finally, to complete the transformation, the transformation function that transforms $IG_{Subclass}$ into $Im_{Subclass}$ is defined:

Definition 5.3.29 (Transformation function $f'_{Subclass}$)
The transformation function $f'_{Subclass}(IG)$ is defined as:

$$\begin{aligned} Object &= N_{IG} \\ \text{ObjectClass} &= \left\{(ob, name) \quad \text{if } ob \in N_{IG} \right. \\ \text{ObjectId} &= \left\{(ob, fid(ob)) \quad \text{if } ob \in N_{IG} \right. \\ \text{FieldValue} &= \{\} \\ \text{DefaultValue} &= \{\} \end{aligned}$$

Also see `ig_subclass_as_node_type_to_imod_subclass` *in* `Ecore-GROOVE-Mapping-Library.SubclassInstance`

Theorem 5.3.30 (Correctness of $f'_{Subclass}$)
$f'_{Subclass}(IG)$ (Definition 5.3.29) is a valid transformation function in the sense of Definition 4.4.31 transforming $IG_{Subclass}$ into $Im_{Subclass}$.

Also see `ig_subclass_as_node_type_to_imod_subclass_func` *in* `Ecore-GROOVE-Mapping-Library.SubclassInstance`

Once more, the correctness proof is not included here but can be found in the validated Isabelle proofs of this thesis.

5.3.4 Enumeration values

This section defines the transformation of enumeration values belonging to an enumeration type on the type level. The corresponding type level transformation can be found in Section 5.2.4. This transformation introduces new nodes in an instance graph that correspond to the values of an enumeration type. Within an instance model, nothing new needs to be introduced, and there the empty instance model is used once more.

First, the instance model corresponding to Tm_{Enum} is defined.

Definition 5.3.31 (Instance model Im_{Enum})
Let Im_{Enum} be the empty instance model Im_ϵ (Definition 4.4.9), except that it is typed by the type model Tm_{Enum} (Definition 5.2.31).

Also see `imod_enum` *in* `Ecore-GROOVE-Mapping-Library.EnumInstance`

Theorem 5.3.32 (Correctness of Im_{Enum})
Im_{Enum} (Definition 5.3.31) is a valid instance model in the sense of Definition 3.2.19.

Also see `imod_enum_correct` *in* `Ecore-GROOVE-Mapping-Library.EnumInstance`

Since the instance model corresponding to the transformation of enumeration values does not define any objects, there is no visual representation needed. Moreover, the correctness proof of Im_{Enum} is trivial, and therefore not included here. The proof can be found as part of the Isabelle validated proofs.

In order to make composing transformation functions possible, Im_{Enum} should be compatible with the instance model it is combined with.

Theorem 5.3.33 (Correctness of combine(Im, Im_{Enum}))
Assume an instance model Im that is valid in the sense of Definition 3.2.19. Then Im is compatible with Im_{Enum} (in the sense of Definition 4.4.14) if:

- *All requirements of Theorem 5.2.33 are met, to ensure the combination of the corresponding type models is valid.*

  Also see `imod_enum_combine_correct` *in* `Ecore-GROOVE-Mapping-Library.EnumInstance`

Proof. Use Lemma 4.4.13. It is possible to show that all assumptions hold. Now we have shown that combine(Im, Im_{Enum}) is consistent in the sense of Definition 3.2.19. □

Please note that this combination is trivial, as the instance model is empty. However, on the instance graph level, more complex definitions are used.

The definitions and theorems for introducing plain objects of regular classes within Ecore are now complete.

Encoding of node type values as nodes

Section Section 5.2.4 has shown two possible encodings of an enumeration type in GROOVE. Both encodings require a different definition on the instance level. In the case that an enumeration type is encoded as node types, the enumeration values will be nodes typed by these node types, one node for each value of the enumeration type. This gives rise to an instance graph $IG_{EnumNodes}$, defined in the following definition:

Definition 5.3.34 (Instance graph $IG_{EnumNodes}$)
Let $IG_{EnumNodes}$ be the instance graph corresponding $TG_{EnumNodes}$ (Definition 5.2.34). $IG_{EnumNodes}$ defines a node for each possible value of the enumeration type encoded by $TG_{EnumNodes}$. Each of these nodes is typed by its corresponding node type. Furthermore, the function fob and fib are defined. fob converts each value of the enumeration type to its internal node identity. fid maps each value of the enumeration type to an explicit identity. $IG_{EnumNodes}$ is defined typed by $TG_{EnumNodes}$ and is defined as:

$$N = \{fob(v) \mid v \in values\}$$
$$E = \{\}$$
$$\text{ident} = \begin{cases} (fid(v), fob(v)) & \text{if } v \in values \end{cases}$$

with

$$\text{type}_n = \begin{cases} (v, \text{ns\_to\_list}(name) \mathbin{@} \langle v \rangle) & \text{if } v \in values \end{cases}$$

Also see `ig_enum_as_node_types` *in* `Ecore-GROOVE-Mapping-Library.EnumInstance`

Theorem 5.3.35 (Correctness of $IG_{EnumNodes}$)
$IG_{EnumNodes}$ (Definition 5.3.34) is a valid instance graph in the sense of Definition 3.3.10.

Also see `ig_enum_as_node_types_correct` *in* `Ecore-GROOVE-Mapping-Library.EnumInstance`

A visual representation of $IG_{EnumNodes}$ with .Example as identifier for the encoded enumeration type and OPTION_A, OPTION_B and OPTION_C as its values can be seen in Figure 5.12a. In this representation, it can also be seen that each values has its corresponding identifier, with $fid(\text{OPTION\_A}) = ExampleOptionA$, $fid(\text{OPTION\_B}) = ExampleOptionB$ and $fid(\text{OPTION\_C}) = ExampleOptionC$. Furthermore, the instances shown here correspond to the visual representation shown in Figure 5.4b. The correctness proof of $IG_{EnumNodes}$ is trivial, and therefore not included here. The proof can be found as part of the Isabelle validated proofs.

In order to make composing transformation functions possible, $IG_{EnumNodes}$ should be compatible with the instance graph it is combined with.

Theorem 5.3.36 (Correctness of combine$(IG, IG_{EnumNodes})$)
Assume an instance graph IG that is valid in the sense of Definition 3.3.10. Then IG is compatible with $IG_{EnumNodes}$ (in the sense of Definition 4.4.25) if:

- *All requirements of Theorem 5.2.36 are met, to ensure the combination of the corresponding type graphs is valid;*
- *All the nodes in $IG_{EnumNodes}$ have an (internal and explicit) identity that is not yet used in IG.*

Also see `ig_enum_as_node_types_combine_correct` *in* `Ecore-GROOVE-Mapping-Library.EnumInstance`

Proof. Use Lemma 4.4.24. It is possible to show that all assumptions hold. Now we have shown that combine$(IG, IG_{EnumNodes})$ is valid in the sense of Definition 3.3.10. □

The next definitions define the transformation function from Im_{Enum} to $IG_{EnumNodes}$:

Definition 5.3.37 (Transformation function $f_{EnumNodes}$)
The transformation function $f_{EnumNodes}(Im)$ is defined as:

$$\begin{aligned} N &= \{fob(v) \mid v \in values\} \\ E &= \{\} \\ \text{ident} &= \begin{cases} (fid(v), fob(v)) & \text{if } v \in values \end{cases} \end{aligned}$$

with

$$\text{type}_n = \begin{cases} (v, \text{ns\_to\_list}(name) \mathbin{@} \langle v \rangle) & \text{if } v \in values \end{cases}$$

Also see `imod_enum_to_ig_enum_as_node_types` *in* `Ecore-GROOVE-Mapping-Library.EnumInstance`

Theorem 5.3.38 (Correctness of $f_{EnumNodes}$)
$f_{EnumNodes}(Im)$ (Definition 5.3.37) is a valid transformation function in the sense of Definition 4.4.26 transforming Im_{Enum} into $IG_{EnumNodes}$.

Also see `imod_enum_to_ig_enum_as_node_types_func` *in* `Ecore-GROOVE-Mapping-Library.EnumInstance`

The proof of the correctness of $f_{EnumNodes}$ will not be included here. Instead, it can be found in the validated Isabelle theories. It should be noted that the proof is trivial, as the function has to introduce all nodes a new nodes. There is nothing to convert from Im_{Enum}.

Finally, to complete the transformation, the transformation function that transforms $IG_{EnumNodes}$ into Im_{Enum} is defined:

Definition 5.3.39 (Transformation function $f'_{EnumNodes}$)
The transformation function $f'_{EnumNodes}(IG)$ is defined as the function that always outputs the empty instance model Im_ϵ (Definition 4.4.9), except that it is typed by Tm_{Enum}.

Also see `ig_enum_as_node_types_to_imod_enum` *in* `Ecore-GROOVE-Mapping-Library.EnumInstance`

Theorem 5.3.40 (Correctness of $f'_{EnumNodes}$)
$f'_{EnumNodes}(IG)$ (Definition 5.3.39) is a valid transformation function in the sense of Definition 4.4.31 transforming $IG_{EnumNodes}$ into Im_{Enum}.

Also see `ig_enum_as_node_types_to_imod_enum_func` *in* `Ecore-GROOVE-Mapping-Library.EnumInstance`

Once more, the correctness proof is not included here but can be found in the validated Isabelle proofs of this thesis.

Encoding of flag values as nodes

The previous subsection discussed how to encode the enumeration values when the enumeration type is encoded as different node types. In this subsection, the transformation of the enumeration values is discussed in the case that the enumeration type is encoded using flags in GROOVE.

In the case that an enumeration type is encoded using flags for the values, a node is introduced for each value, all typed by the enumeration node type. Each of the nodes has a single flag, corresponding to the value the node represents. This gives rise to an instance graph $IG_{EnumFlags}$, defined in the following definition:

Definition 5.3.41 (Instance graph $IG_{EnumFlags}$)
Let $IG_{EnumFlags}$ be the instance graph corresponding $TG_{EnumFlags}$ (Definition 5.2.41). $IG_{EnumFlags}$ defines a node for each possible value of the enumeration type encoded by $TG_{EnumFlags}$. Each of these nodes is typed by the corresponding node type and has one of the flags set, the flag corresponding to the value the node represents. Furthermore, the function fob and fib are defined. fob converts each value of the enumeration type to its internal node identity. fid maps each value of the enumeration type to an explicit identity. $IG_{EnumFlags}$ is defined typed by $TG_{EnumFlags}$ and is defined as:

$$
\begin{aligned}
N &= \{fob(v) \mid v \in values\} \\
E &= \left\{ \Big(fob(v), (\text{ns\_to\_list}(name), \langle v \rangle, \text{ns\_to\_list}(name)), fob(v) \Big) \mid v \in values \right\} \\
\text{ident} &= \Big\{ (fid(v), fob(v)) \quad \text{if } v \in values
\end{aligned}
$$

with

$$
\text{type}_n = \Big\{ (v, \text{ns\_to\_list}(name)) \quad \text{if } v \in values
$$

Also see `ig_enum_as_flags` *in* `Ecore-GROOVE-Mapping-Library.EnumInstance`

Theorem 5.3.42 (Correctness of $IG_{EnumFlags}$)
$IG_{EnumFlags}$ (Definition 5.3.41) is a valid instance graph in the sense of Definition 3.3.10.

Also see `ig_enum_as_flags_correct` *in* `Ecore-GROOVE-Mapping-Library.EnumInstance`

A visual representation of $IG_{EnumFlags}$ with .Example as identifier for the encoded enumeration type and OPTION_A, OPTION_B and OPTION_C as its values can be seen in Figure 5.12b. In this representation, it can also be seen that each values has its corresponding identifier, with $fid(\text{OPTION\_A}) = ExampleOptionA$, $fid(\text{OPTION\_B}) = ExampleOptionB$ and $fid(\text{OPTION\_C}) = ExampleOptionC$. Furthermore, the instances shown here correspond to the visual representation shown in Figure 5.4c. The correctness proof of $IG_{EnumFlags}$ is trivial, and therefore not included here. The proof can be found as part of the Isabelle validated proofs.

In order to make composing transformation functions possible, $IG_{EnumFlags}$ should be compatible with the instance graph it is combined with.

Theorem 5.3.43 (Correctness of combine$(IG, IG_{EnumFlags})$)
Assume an instance graph IG that is valid in the sense of Definition 3.3.10. Then IG is compatible with $IG_{EnumFlags}$ (in the sense of Definition 4.4.25) if:

- *All requirements of Theorem 5.2.43 are met, to ensure the combination of the corresponding type graphs is valid;*
- *All the nodes in $IG_{EnumFlags}$ have an (internal and explicit) identity that is not yet used in IG.*

Also see `ig_enum_as_flags_combine_correct` *in* `Ecore-GROOVE-Mapping-Library.EnumInstance`

Proof. Use Lemma 4.4.24. It is possible to show that all assumptions hold. Now we have shown that combine$(IG, IG_{EnumFlags})$ is valid in the sense of Definition 3.3.10. □

The next definitions define the transformation function from Im_{Enum} to $IG_{EnumFlags}$:

Definition 5.3.44 (Transformation function $f_{EnumFlags}$)
The transformation function $f_{EnumFlags}(Im)$ is defined as:

$$N = \{fob(v) \mid v \in values\}$$
$$E = \left\{\Big(fob(v), (\text{ns\_to\_list}(name), \langle v\rangle, \text{ns\_to\_list}(name)), fob(v)\Big) \mid v \in values\right\}$$
$$\text{ident} = \Big\{(fid(v), fob(v)) \quad \text{if } v \in values$$

with

$$\text{type}_n = \Big\{(v, \text{ns\_to\_list}(name)) \quad \text{if } v \in values$$

Also see `imod_enum_to_ig_enum_as_flags` *in* `Ecore-GROOVE-Mapping-Library.EnumInstance`

Theorem 5.3.45 (Correctness of $f_{EnumFlags}$)
$f_{EnumFlags}(Im)$ (Definition 5.3.44) is a valid transformation function in the sense of Definition 4.4.26 transforming Im_{Enum} into $IG_{EnumFlags}$.

Also see `imod_enum_to_ig_enum_as_flags_func` *in* `Ecore-GROOVE-Mapping-Library.EnumInstance`

The proof of the correctness of $f_{EnumFlags}$ will not be included here. Instead, it can be found in the validated Isabelle theories. It should be noted that the proof is trivial, as the function has to introduce all nodes a new nodes. There is nothing to convert from Im_{Enum}.

Finally, to complete the transformation, the transformation function that transforms $IG_{EnumFlags}$ into Im_{Enum} is defined:

Definition 5.3.46 (Transformation function $f'_{EnumFlags}$)
The transformation function $f'_{EnumFlags}(IG)$ is defined as the function that always outputs the empty instance model Im_ϵ (Definition 4.4.9), except that it is typed by Tm_{Enum}.

Also see `ig_enum_as_flags_to_imod_enum` *in* `Ecore-GROOVE-Mapping-Library.EnumInstance`

Theorem 5.3.47 (Correctness of $f'_{EnumFlags}$)
$f'_{EnumFlags}(IG)$ (Definition 5.3.46) is a valid transformation function in the sense of Definition 4.4.31 transforming $IG_{EnumFlags}$ into Im_{Enum}.

Also see `ig_enum_as_flags_to_imod_enum_func` *in* `Ecore-GROOVE-Mapping-Library.EnumInstance`

Once more, the correctness proof is not included here but can be found in the validated Isabelle proofs of this thesis.

5.3.5 User-defined data types

In this section, the instance level transformation corresponding to the type level transformation of user-defined data types is discussed. The type level transformation of user-defined data types can be found in Section 5.2.5.

This definition does not actually introduce values for user-defined data types. This is done upon instantiating the type via a field. Therefore, an empty instance model and empty instance graph will be used for completeness.

First, the corresponding instance model is introduced.

Definition 5.3.48 (Instance model $Im_{UserType}$)
Let $Im_{UserType}$ be the empty instance model Im_ϵ (Definition 4.4.9), except that it is typed by the type model $Tm_{UserType}$ (Definition 5.2.48).

Also see `imod_userdatatype` *in* `Ecore-GROOVE-Mapping-Library.UserDataTypeInstance`

Theorem 5.3.49 (Correctness of $Im_{UserType}$)
$Im_{UserType}$ (Definition 5.3.48) is a valid instance model in the sense of Definition 3.2.19.

Also see `imod_userdatatype_correct` *in* `Ecore-GROOVE-Mapping-Library.UserDataTypeInstance`

Since $Im_{UserType}$ does not define any objects, there is no need for a visual representation. However, in order to make composing transformation functions possible, $Im_{UserType}$ should still be compatible with the instance model it is combined with.

Theorem 5.3.50 (Correctness of combine$(Im, Im_{UserType})$)
Assume an instance model Im that is valid in the sense of Definition 3.2.19. Then Im is compatible with $Im_{UserType}$ (in the sense of Definition 4.4.14) if:

- *All requirements of Theorem 5.2.50 are met, to ensure the combination of the corresponding type models is valid.*

Also see `imod_userdatatype_combine_correct` *in* `Ecore-GROOVE-Mapping-Library.UserDataTypeInstance`

Proof. Use Lemma 4.4.13. It is possible to show that all assumptions hold. Now we have shown that combine$(Im, Im_{UserType})$ is consistent in the sense of Definition 3.2.19. □

The definitions and theorems for the Ecore instance model corresponding to $Tm_{UserType}$ are now complete.

The node type encoding

As has been shown earlier, an possible encoding for user-defined data types is by introducing a node type. This has been done in $TG_{UserType}$. Like the Ecore instance model, the GROOVE instance graph is also empty, because the values for the type are not instantiated now. This gives rise to $IG_{UserType}$, which is defined as follows:

Definition 5.3.51 (Instance graph $IG_{UserType}$)
Let $IG_{UserType}$ be the empty instance graph IG_ϵ (Definition 4.4.20), except that it is typed by the type graph $TG_{UserType}$ (Definition 5.2.51).

Also see `ig_userdatatype_as_node_type` *in* `Ecore-GROOVE-Mapping-Library.UserDataTypeInstance`

Theorem 5.3.52 (Correctness of $IG_{UserType}$)
$IG_{UserType}$ (Definition 5.3.51) is a valid instance graph in the sense of Definition 3.3.10.

Also see `ig_userdatatype_as_node_type_correct` *in* `Ecore-GROOVE-Mapping-Library.UserDataTypeInstance`

In order to make composing transformation functions possible, $IG_{UserType}$ should be compatible with the instance graph it is combined with.

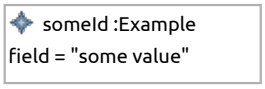
Theorem 5.3.53 (Correctness of combine$(IG, IG_{UserType})$)
Assume an instance graph IG that is valid in the sense of Definition 3.3.10. Then IG is compatible with $IG_{UserType}$ (in the sense of Definition 4.4.25) if:

- *All requirements of Theorem 5.2.53 are met, to ensure the combination of the corresponding type graphs is valid.*

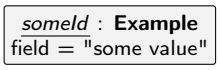
Also see `ig_userdatatype_as_node_type_combine_correct` *in* `Ecore-GROOVE-Mapping-Library.UserDataTypeInstance`

Proof. Use Lemma 4.4.24. It is possible to show that all assumptions hold. Now we have shown that combine$(IG, IG_{UserType})$ is valid in the sense of Definition 3.3.10. □

The next definitions define the transformation function from $Im_{UserType}$ to $IG_{UserType}$:



(a) $Im_{DataField}$ with one object and string value "some value"



(b) $IG_{DataField}$ with one node and string value "some value"

Figure 5.13: Visualisation of the transformation of field values from fields typed by data types

Definition 5.3.54 (Transformation function $f_{UserType}$)
The transformation function $f_{UserType}(Im)$ is defined as the function that always outputs the empty instance graph IG_ϵ (Definition 4.4.20), except that it is typed by $TG_{UserType}$.

Also see `imod_userdatatype_to_ig_userdatatype_as_node_type` *in* `Ecore-GROOVE-Mapping-Library.UserDataTypeInstance`

Theorem 5.3.55 (Correctness of $f_{UserType}$)
$f_{UserType}(Im)$ (Definition 5.3.54) is a valid transformation function in the sense of Definition 4.4.26 transforming $Im_{UserType}$ into $IG_{UserType}$.

Also see `imod_userdatatype_to_ig_userdatatype_as_node_type_func` *in* `Ecore-GROOVE-Mapping-Library.UserDataTypeInstance`

The proof of the correctness of $f_{UserType}$ will not be included here. Instead, it can be found in the validated Isabelle theories. Obviously, the proof is trivial, as the function does not do any conversion. It does just output the empty instance model.

Finally, to complete the transformation, the transformation function that transforms $IG_{UserType}$ into $Im_{UserType}$ is defined:

Definition 5.3.56 (Transformation function $f'_{UserType}$)
The transformation function $f'_{UserType}(IG)$ is defined as the function that always outputs the empty instance model Im_ϵ (Definition 4.4.9), except that it is typed by $Tm_{UserType}$.

Also see `ig_userdatatype_as_node_type_to_imod_userdatatype` *in* `Ecore-GROOVE-Mapping-Library.UserDataTypeInstance`

Theorem 5.3.57 (Correctness of $f'_{UserType}$)
$f'_{UserType}(IG)$ (Definition 5.3.56) is a valid transformation function in the sense of Definition 4.4.31 transforming $IG_{UserType}$ into $Im_{UserType}$.

Also see `ig_userdatatype_as_node_type_to_imod_userdatatype_func` *in* `Ecore-GROOVE-Mapping-Library.UserDataTypeInstance`

Once more, the correctness proof is not included here but can be found in the validated Isabelle proofs of this thesis.

5.3.6 Data field values

The previous sections have shown the instance level transformations of the introduction of all kinds of types and their instances. From this section onward, these types and their instances will be enriched by introducing fields. In this section, the instance level transformation belonging to the transformation of a data field is discussed. The type level transformation for data fields can be found in Section 5.2.6. On the instance level, values for the data fields are introduced.

Definition 5.3.58 (Instance model $Im_{DataField}$)
Let $Im_{DataField}$ be an instance model typed by $Tm_{DataField}$ (Definition 5.2.58). Define a set objects, which represent the objects that will get a value for the field introduced by $Tm_{DataField}$. Furthermore, define a function obids which maps each of these objects to their corresponding identifier and a function values, which maps each of these objects to its value for the field introduced by $Tm_{DataField}$. $Im_{DataField}$

is defined as:

$$
\begin{aligned}
Object &= objects \\
\text{ObjectClass} &= \{(ob, classtype) \quad \text{if } ob \in objects \\
\text{ObjectId} &= \{(ob, obids(ob)) \quad \text{if } ob \in objects \\
\text{FieldValue} &= \{((ob, (classtype, name)), values(ob)) \quad \text{if } ob \in objects \\
\text{DefaultValue} &= \{\}
\end{aligned}
$$

Also see `imod_data_field` *in* `Ecore-GROOVE-Mapping-Library.DataFieldValue`

Theorem 5.3.59 (Correctness of $Im_{DataField}$)
$Im_{DataField}$ (Definition 5.3.58) is a valid instance model in the sense of Definition 3.2.19.

Also see `imod_data_field_correct` *in* `Ecore-GROOVE-Mapping-Library.DataFieldValue`

A visual representation of $Im_{DataField}$ with $objects = \{ob\}$ and $obids(ob) = someId$ can be seen in Figure 5.13a. In this visualisation, the field value for ob is defined as $values(ob) =$ "some value". Although this visualisation only shows one object, it is required to define a value for all objects that contain the field. Failing to do so would result in an invalid instance model after it is combined with another model, as the next definition will show. The correctness proof of $Im_{DataField}$ only is already quite involved, but not be included here for conciseness. It can be found as part of the validated Isabelle proofs.

In order to make composing transformation functions possible, $Im_{DataField}$ should be compatible with the instance model it is combined with.

Theorem 5.3.60 (Correctness of combine$(Im, Im_{DataField})$)
Assume an instance model Im that is valid in the sense of Definition 3.2.19. Then Im is compatible with $Im_{DataField}$ (in the sense of Definition 4.4.14) if:

- *All requirements of Theorem 5.2.60 are met, to ensure the combination of the corresponding type models is valid;*
- *The class type on which the field is defined by $Tm_{DataField}$ may not be extended by another class type in the type model corresponding to Im;*
- *All of the objects in the set objects must already be objects in Im;*
- *All objects typed by the class type on which the field is defined must occur in the set objects and thus have a value in $Im_{DataField}$;*
- *For all of the objects in the set objects, the identifier set by obids must be the same identifier as set by Im for that object;*
- *For all objects in set objects, the value set by the values function must be valid.*

Also see `imod_data_field_combine_correct` *in* `Ecore-GROOVE-Mapping-Library.DataFieldValue`

Proof. Use Lemma 4.4.13. It is possible to show that all assumptions hold. Now we have shown that combine$(Im, Im_{DataField})$ is consistent in the sense of Definition 3.2.19. □

As explained earlier, $Im_{DataField}$ needs to introduce values for all objects that are typed by the class type on which the field is defined. This is enforced by the requirements of Theorem 5.3.60. The proof is not included here for conciseness, but can be found as part of the validated proofs in Isabelle.

The definitions and theorems for introducing values for fields of data types within Ecore are now complete.

Encoding as edges and nodes

In the type level transformation of data fields, data fields were encoded in GROOVE as edge types to an primitive type. On the instance level, this edge type will be used and edges will be created to give a value to each node type that has the field defined. The encoding corresponding to $Im_{DataField}$ can then be represented as $IG_{DataField}$, defined in the following definition:

Definition 5.3.61 (Instance graph $IG_{DataField}$)
Let $IG_{DataField}$ be the instance graph typed by type graph $TG_{DataField}$ (Definition 5.2.61). Reuse the set objects from $Im_{DataField}$. Moreover, reuse the functions obids and values from $Im_{DataField}$. The objects in the set objects are converted to nodes in $Im_{DataField}$. For each of these objects, an edge of the encoded field is created. This edge targets a node that corresponds to the value set by values for the corresponding object. Finally, the identity of the objects is defined using obids. $IG_{DataField}$ is defined as:

$$\begin{aligned} N &= objects \cup \{values(ob) \mid ob \in objects\} \\ E &= \{(ob, (\text{ns\_to\_list}(classtype), \langle name \rangle, fieldtype), values(ob)) \mid ob \in objects\} \\ \text{ident} &= \{(obids(ob), ob) \quad \text{if } ob \in objects \end{aligned}$$

with

$$\text{type}_n = \{(ob, \text{ns\_to\_list}(classtype)) \quad \text{if } ob \in objects$$

Also see `ig_data_field_as_edge_type` *in* `Ecore-GROOVE-Mapping-Library.DataFieldValue`

Theorem 5.3.62 (Correctness of $IG_{DataField}$)
$IG_{DataField}$ (Definition 5.3.61) is a valid instance graph in the sense of Definition 3.3.10.

Also see `ig_data_field_as_edge_type_correct` *in* `Ecore-GROOVE-Mapping-Library.DataFieldValue`

A visual representation of $IG_{DataField}$ with $objects = \{ob\}$ and $obids(ob) = someId$ can be seen in Figure 5.13b. Like the previous visualisation, the field value for ob is defined as $values(ob) =$ "some value". Although this visualisation only shows one node, it is required to define a value for all nodes typed by the node type corresponding to the field. Failing to do so would result in an invalid instance graph after it is combined with another graph, as the next definition will show. The correctness proof of $IG_{DataField}$ only is already quite involved, but not be included here for conciseness. It can be found as part of the validated Isabelle proofs.

In order to make composing transformation functions possible, $IG_{DataField}$ should be compatible with the instance graph it is combined with.

Theorem 5.3.63 (Correctness of combine$(IG, IG_{DataField})$)
Assume an instance graph IG that is valid in the sense of Definition 3.3.10. Then IG is compatible with $IG_{DataField}$ (in the sense of Definition 4.4.25) if:

- *All requirements of Theorem 5.2.63 are met, to ensure the combination of the corresponding type graphs is valid;*
- *The node type on which the corresponding field is defined is not extended by other node types within the type graph corresponding to IG;*
- *All nodes in IG that are typed by the node type on which the field is defined are also nodes in $IG_{DataField}$;*
- *For all nodes shared between IG and $IG_{DataField}$, each node must have the same identifier in both IG and $IG_{DataField}$;*
- *For all nodes for which the field is set, the values function must define a valid value;*
- *If an primitive type has incoming or outgoing edge types in the type graph corresponding to IG, then the lower multiplicity of these edge types must be 0.*

Also see `ig_data_field_as_edge_type_combine_correct` *in* `Ecore-GROOVE-Mapping-Library.DataFieldValue`

Proof. Use Lemma 4.4.24. It is possible to show that all assumptions hold. Now we have shown that combine$(IG, IG_{DataField})$ is valid in the sense of Definition 3.3.10. □

Like the definition for the combination of instance models, the combination of instance graphs also requires the user to set a value for all nodes that are typed by the node type that corresponds to the field type. This is to keep the graph valid.

The next definitions define the transformation function from $Im_{DataField}$ to $IG_{DataField}$:

Definition 5.3.64 (Transformation function $f_{DataField}$)
The transformation function $f_{DataField}(Im)$ is defined as:

$$
\begin{aligned}
N &= Object_{Im} \cup \{values(ob) \mid ob \in Object_{Im}\} \\
E &= \left\{\left(ob, (\text{ns\_to\_list}(classtype), \langle name \rangle, fieldtype), values(ob)\right) \mid ob \in Object_{Im}\right\} \\
\text{ident} &= \left\{(obids(ob), ob) \quad \text{if } ob \in Object_{Im}\right.
\end{aligned}
$$

with

$$
\text{type}_n = \left\{(ob, \text{ns\_to\_list}(name)) \quad \text{if } ob \in Object_{Im}\right.
$$

Also see `imod_data_field_to_ig_data_field_as_edge_type` *in* `Ecore-GROOVE-Mapping-Library.DataFieldValue`

Theorem 5.3.65 (Correctness of $f_{DataField}$)
$f_{DataField}(Im)$ (Definition 5.3.64) is a valid transformation function in the sense of Definition 4.4.26 transforming $Im_{DataField}$ into $IG_{DataField}$.

Also see `imod_data_field_to_ig_data_field_as_edge_type_func` *in* `Ecore-GROOVE-Mapping-Library.DataFieldValue`

The proof of the correctness of $f_{DataField}$ will not be included here. Instead, it can be found in the validated Isabelle theories.

Finally, to complete the transformation, the transformation function that transforms $IG_{DataField}$ into $Im_{DataField}$ is defined:

Definition 5.3.66 (Transformation function $f'_{DataField}$)
The transformation function $f'_{DataField}(IG)$ is defined as:

$$
\begin{aligned}
Object &= \{\text{src}(e) \mid e \in E_{IG}\} \\
\text{ObjectClass} &= \left\{(ob, classtype) \quad \text{if } ob \in \{\text{src}(e) \mid e \in E_{IG}\}\right. \\
\text{ObjectId} &= \left\{(ob, obids(ob)) \quad \text{if } ob \in \{\text{src}(e) \mid e \in E_{IG}\}\right. \\
\text{FieldValue} &= \left\{((ob, (classtype, name)), values(ob)) \quad \text{if } ob \in \{\text{src}(e) \mid e \in E_{IG}\}\right. \\
\text{DefaultValue} &= \{\}
\end{aligned}
$$

Also see `ig_data_field_as_edge_type_to_imod_data_field` *in* `Ecore-GROOVE-Mapping-Library.DataFieldValue`

Theorem 5.3.67 (Correctness of $f'_{DataField}$)
$f'_{DataField}(IG)$ (Definition 5.3.66) is a valid transformation function in the sense of Definition 4.4.31 transforming $IG_{DataField}$ into $Im_{DataField}$.

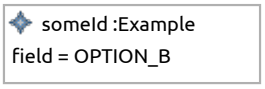
Also see `ig_data_field_as_edge_type_to_imod_data_field_func` *in* `Ecore-GROOVE-Mapping-Library.DataFieldValue`

Once more, the correctness proof is not included here but can be found in the validated Isabelle proofs of this thesis.

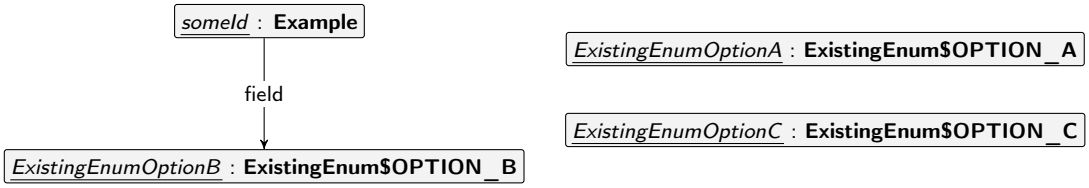
5.3.7 Enumeration field values

In this section, the instance level transformation belonging to the transformation of an enumeration field is discussed. The type level transformation for enumeration fields can be found in Section 5.2.7. On the instance level, values for the enumeration fields are introduced.

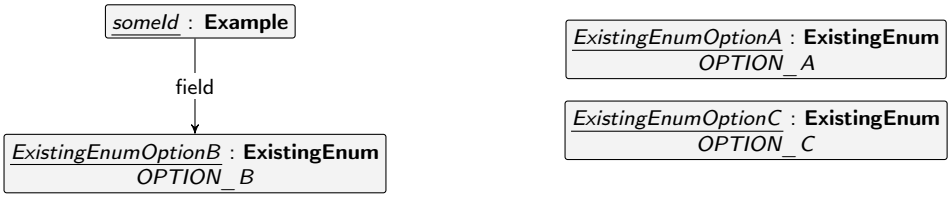
Definition 5.3.68 (Instance model $Im_{EnumField}$)
Let $Im_{EnumField}$ be an instance model typed by $Tm_{EnumField}$ (Definition 5.2.68). Define a set objects, which represent the objects that will get a value for the field introduced by $Tm_{EnumField}$. Furthermore, define a function obids which maps each of these objects to their corresponding identifier and a function values, which maps each of these objects to its value for the field introduced by $Tm_{EnumField}$.



(a) $Im_{EnumField}$ with one object and its field referencing enumeration option $OPTION\_B$



(b) $IG_{EnumFieldNodes}$ with one node and and its edge referencing enumeration option $OPTION\_B$



(c) $IG_{EnumFieldFlags}$ with one node and its edge referencing enumeration option $OPTION\_B$

Figure 5.14: Visualisation of the transformation of field values from fields typed by enumeration types

$Im_{EnumField}$ is defined as:

$$
\begin{aligned}
Object &= objects \\
\text{ObjectClass} &= \left\{(ob, classtype) \quad \text{if } ob \in objects \right. \\
\text{ObjectId} &= \left\{(ob, obids(ob)) \quad \text{if } ob \in objects \right. \\
\text{FieldValue} &= \left\{\Big((ob, (classtype, name)), \big[\mathsf{enum}, (enumid, values(ob))\big]\Big) \quad \text{if } ob \in objects \right. \\
\text{DefaultValue} &= \{\}
\end{aligned}
$$

Also see `imod_enum_field` *in* `Ecore-GROOVE-Mapping-Library.EnumFieldValue`

Theorem 5.3.69 (Correctness of $Im_{EnumField}$)
$Im_{EnumField}$ (Definition 5.3.68) is a valid instance model in the sense of Definition 3.2.19.

Also see `imod_enum_field_correct` *in* `Ecore-GROOVE-Mapping-Library.EnumFieldValue`

A visual representation of $Im_{EnumField}$ with $objects = \{ob\}$ and $obids(ob) = someId$ can be seen in Figure 5.14a. In this visualisation, the field value for ob is defined as $values(ob) = OPTION\_B$. Although this visualisation only shows one object, it is required to define a value for all objects that contain the field. Failing to do so would result in an invalid instance model after it is combined with another model, as the next definition will show. The correctness proof of $Im_{EnumField}$ only is quite involved, but not be included here for conciseness. It can be found as part of the validated Isabelle proofs.

In order to make composing transformation functions possible, $Im_{EnumField}$ should be compatible with the instance model it is combined with.

Theorem 5.3.70 (Correctness of combine$(Im, Im_{EnumField})$)
Assume an instance model Im that is valid in the sense of Definition 3.2.19. Then Im is compatible with $Im_{EnumField}$ (in the sense of Definition 4.4.14) if:

- *All requirements of Theorem 5.2.70 are met, to ensure the combination of the corresponding type models is valid;*
- *The class type on which the field is defined by $Tm_{EnumField}$ may not be extended by another class type in the type model corresponding to Im;*
- *All of the objects in the set objects must already be objects in Im;*
- *All objects typed by the class type on which the field is defined must occur in the set objects and thus have a value in $Im_{EnumField}$;*

- *For all of the objects in the set objects, the identifier set by obids must be the same identifier as set by Im for that object;*
- *For all objects in set objects, the value set by the values function must be valid.*

Also see `imod_enum_field_combine_correct` *in* `Ecore-GROOVE-Mapping-Library.EnumFieldValue`

Proof. Use Lemma 4.4.13. It is possible to show that all assumptions hold. Now we have shown that combine$(Im, Im_{EnumField})$ is consistent in the sense of Definition 3.2.19. □

As explained earlier, $Im_{EnumField}$ needs to introduce values for all objects that are typed by the class type on which the field is defined. This is enforced by the requirements of Theorem 5.3.70. The proof is not included here for conciseness, but can be found as part of the validated proofs in Isabelle.

The definitions and theorems for introducing values for fields of data types within Ecore are now complete.

Encoding as edges and nodes with a node type encoded enumeration type

As discussed in Section 5.2.7, there are two different encodings for a field typed by an enumeration type. These correspond to the two different encodings of the enumeration type itself. On the instance level, these encodings also need to be distinguished. The first encoding of the values assumes that the enumeration is encoded using node types. The encoding corresponding to $Im_{EnumField}$ is then represented as $IG_{EnumFieldNodes}$, defined in the following definition:

Definition 5.3.71 (Instance graph $IG_{EnumFieldNodes}$)
Let $IG_{EnumFieldNodes}$ be the instance graph typed by type graph $TG_{EnumFieldNodes}$ (Definition 5.2.71). Reuse the set objects from $Im_{EnumField}$. Moreover, reuse the functions obids and values from $Im_{EnumField}$. Furthermore, define enumob to be the function that maps an enumeration value to an internal node identity. Similarly, define enumids as the function that maps an enumeration value to its explicit node id.

Within $IG_{EnumFieldNodes}$, the objects in the set objects are converted to nodes in $Im_{EnumField}$. For each of these objects, an edge of the encoded field is created. This edge targets a node that corresponds to the value set by values for the corresponding object. Furthermore, the identity of the objects is defined using obids. Finally, ensure that the instances of the enumeration values exist and encode them in the same way as $IG_{EnumNodes}$ (Definition 5.3.34). $IG_{EnumFieldNodes}$ is defined as:

$$N = objects \cup \{enumob(v) \mid v \in enumvalues\}$$

$$E = \big\{\big(ob, (\text{ns\_to\_list}(classtype), \langle name\rangle, \text{ns\_to\_list}(enumid)), enumob(values(ob))\big) \mid ob \in objects\big\}$$

$$\text{ident} = \begin{cases} (obids(ob), ob) & \text{if } ob \in objects \\ (enumids(v), enumob(v)) & \text{if } v \in enumvalues \end{cases}$$

with

$$\text{type}_n = \begin{cases} (ob, \text{ns\_to\_list}(classtype)) & \text{if } ob \in objects \\ (enumob(v), \text{ns\_to\_list}(enumid) \mathbin{@} \langle v\rangle) & \text{if } v \in enumvalues \end{cases}$$

Also see `ig_enum_as_node_types_field_as_edge_type` *in* `Ecore-GROOVE-Mapping-Library.EnumFieldValue`

Theorem 5.3.72 (Correctness of $IG_{EnumFieldNodes}$)
$IG_{EnumFieldNodes}$ (Definition 5.3.71) is a valid instance graph in the sense of Definition 3.3.10.

Also see `ig_enum_as_node_types_field_as_edge_type_correct` *in* `Ecore-GROOVE-Mapping-Library.EnumFieldValue`

A visual representation of $IG_{EnumFieldNodes}$ with $objects = \{ob\}$ and $obids(ob) = someId$ can be seen in Figure 5.14b. Like the previous visualisation, the field value for ob is defined as $values(ob) = OPTION\_B$. Although this visualisation only shows one node, it is required to define a value for all nodes that are typed by the node type corresponding to the field. Failing to do so would result in an invalid instance graph after it is combined with another graph, as the next definition will show. The correctness proof of $IG_{EnumFieldNodes}$ only is quite involved, but not be included here for conciseness. It can be found as part of the validated Isabelle proofs.

In order to make composing transformation functions possible, $IG_{EnumFieldNodes}$ should be compatible with the instance graph it is combined with.

Theorem 5.3.73 (Correctness of combine$(IG, IG_{EnumFieldNodes})$)
Assume an instance graph IG that is valid in the sense of Definition 3.3.10. Then IG is compatible with $IG_{EnumFieldNodes}$ (in the sense of Definition 4.4.25) if:

- *All requirements of Theorem 5.2.73 are met, to ensure the combination of the corresponding type graphs is valid;*
- *The node type on which the corresponding field is defined is not extended by other node types within the type graph corresponding to IG;*
- *All nodes in IG that are typed by the node type on which the field is defined are also nodes in $IG_{EnumFieldNodes}$;*
- *All nodes in $IG_{EnumFieldNodes}$ that encode the values of the corresponding enumeration type are also nodes in IG;*
- *For all nodes shared between IG and $IG_{EnumFieldNodes}$, each node must have the same identifier in both IG and $IG_{EnumFieldNodes}$;*
- *For all nodes for which the field is set, the values function must define a valid value.*

Also see `ig_enum_as_node_types_field_as_edge_type_combine_correct` *in* `Ecore-GROOVE-Mapping-Library.EnumFieldValue`

Proof. Use Lemma 4.4.24. It is possible to show that all assumptions hold. Now we have shown that combine$(IG, IG_{EnumFieldNodes})$ is valid in the sense of Definition 3.3.10. □

Like the definition for the combination of instance models, the combination of instance graphs also requires the user to set a value for all nodes that are typed by the node type that corresponds to the field type. This is to keep the graph valid.

The next definitions define the transformation function from $Im_{EnumField}$ to $IG_{EnumFieldNodes}$:

Definition 5.3.74 (Transformation function $f_{EnumFieldNodes}$)
The transformation function $f_{EnumFieldNodes}(Im)$ is defined as:

$$N = Object_{Im} \cup \{enumob(ob) \mid v \in enumvalues\}$$
$$E = \{(ob, (\text{ns\_to\_list}(classtype), \langle name \rangle, \text{ns\_to\_list}(enumid)), enumob(values(ob))) \mid ob \in Object_{Im}\}$$
$$\text{ident} = \begin{cases} (obids(ob), ob) & \text{if } ob \in Object_{Im} \\ (enumids(v), enumob(v)) & \text{if } v \in enumvalues \end{cases}$$

with

$$\text{type}_n = \begin{cases} (ob, \text{ns\_to\_list}(classtype)) & \text{if } ob \in Object_{Im} \\ (enumob(v), \text{ns\_to\_list}(enumid) \mathbin{@} \langle v \rangle) & \text{if } v \in enumvalues \end{cases}$$

Also see `imod_enum_field_to_ig_enum_as_node_types_field_as_edge_type` *in* `Ecore-GROOVE-Mapping-Library.EnumFieldValue`

Theorem 5.3.75 (Correctness of $f_{EnumFieldNodes}$)
$f_{EnumFieldNodes}(Im)$ (Definition 5.3.74) is a valid transformation function in the sense of Definition 4.4.26 transforming $Im_{EnumField}$ into $IG_{EnumFieldNodes}$.

Also see `imod_enum_field_to_ig_enum_as_node_types_field_as_edge_type_func` *in* `Ecore-GROOVE-Mapping-Library.EnumFieldValue`

The proof of the correctness of $f_{EnumFieldNodes}$ will not be included here. Instead, it can be found in the validated Isabelle theories.

Finally, to complete the transformation, the transformation function that transforms $IG_{EnumFieldNodes}$ into $Im_{EnumField}$ is defined:

Definition 5.3.76 (Transformation function $f'_{EnumFieldNodes}$)
The transformation function $f'_{EnumFieldNodes}(IG)$ is defined as:

$$
\begin{aligned}
Object &= \{\mathrm{src}(e) \mid e \in E_{IG}\} \\
\mathrm{ObjectClass} &= \Big\{(ob, name) \quad \text{if } ob \in \{\mathrm{src}(e) \mid e \in E_{IG}\} \\
\mathrm{ObjectId} &= \Big\{(ob, obids(ob)) \quad \text{if } ob \in \{\mathrm{src}(e) \mid e \in E_{IG}\} \\
\mathrm{FieldValue} &= \Big\{\Big((ob, (classtype, name)), \big(\mathsf{enum}, (enumid, values(ob))\big)\Big) \quad \text{if } ob \in \{\mathrm{src}(e) \mid e \in E_{IG}\} \\
\mathrm{DefaultValue} &= \{\}
\end{aligned}
$$

Also see `ig_enum_as_node_types_field_as_edge_type_to_imod_enum_field` *in* `Ecore-GROOVE-Mapping-Library.EnumFieldValue`

Theorem 5.3.77 (Correctness of $f'_{EnumFieldNodes}$)
$f'_{EnumFieldNodes}(IG)$ (Definition 5.3.76) is a valid transformation function in the sense of Definition 4.4.31 transforming $IG_{EnumFieldNodes}$ into $Im_{EnumField}$.

Also see `ig_enum_as_node_types_field_as_edge_type_to_imod_enum_field_func` *in* `Ecore-GROOVE-Mapping-Library.EnumFieldValue`

Once more, the correctness proof is not included here but can be found in the validated Isabelle proofs of this thesis.

Encoding as edges and nodes with a flag encoded enumeration type

The second possible encoding of the values assumes that the enumeration is encoded using flags. The encoding corresponding to $Im_{EnumField}$ is then represented as $IG_{EnumFieldFlags}$, defined in the following definition:

Definition 5.3.78 (Instance graph $IG_{EnumFieldFlags}$)
Let $IG_{EnumFieldFlags}$ be the instance graph typed by type graph $TG_{EnumFieldFlags}$ (Definition 5.2.78). Reuse the set objects from $Im_{EnumField}$. Moreover, reuse the functions obids and values from $Im_{EnumField}$. Furthermore, define enumob to be the function that maps an enumeration value to an internal node identity. Similarly, define enumids as the function that maps an enumeration value to its explicit node id.

Within $IG_{EnumFieldFlags}$, the objects in the set objects are converted to nodes in $Im_{EnumField}$. For each of these objects, an edge of the encoded field is created. This edge targets a node that corresponds to the value set by values for the corresponding object. Furthermore, the identity of the objects is defined using obids. Finally, ensure that the instances of the enumeration values exist and encode them in the same way as $IG_{EnumFlags}$ (Definition 5.3.41). $IG_{EnumFieldFlags}$ is defined as:

$$
\begin{aligned}
N &= objects \cup \{enumob(v) \mid v \in enumvalues\} \\
E &= \{\big(ob, (\mathrm{ns\_to\_list}(classtype), \langle name \rangle, \mathrm{ns\_to\_list}(enumid)), enumob(values(ob))\big) \mid ob \in objects\} \\
\mathrm{ident} &= \begin{cases} (obids(ob), ob) & \text{if } ob \in objects \\ (enumids(v), enumob(v)) & \text{if } v \in enumvalues \end{cases}
\end{aligned}
$$

with

$$
\mathrm{type}_n = \begin{cases} (ob, \mathrm{ns\_to\_list}(classtype)) & \text{if } ob \in objects \\ (enumob(v), \mathrm{ns\_to\_list}(enumid)) & \text{if } v \in enumvalues \end{cases}
$$

Also see `ig_enum_as_flags_field_as_edge_type` *in* `Ecore-GROOVE-Mapping-Library.EnumFieldValue`

Theorem 5.3.79 (Correctness of $IG_{EnumFieldFlags}$)
$IG_{EnumFieldFlags}$ (Definition 5.3.78) is a valid instance graph in the sense of Definition 3.3.10.

Also see `ig_enum_as_flags_field_as_edge_type_correct` *in* `Ecore-GROOVE-Mapping-Library.EnumFieldValue`

A visual representation of $IG_{EnumFieldFlags}$ with $objects = \{ob\}$ and $obids(ob) = someId$ can be seen in Figure 5.14c. It does not differ much from the previous encoding, except that the values of the

enumeration type are shown as flags on the nodes instead of using seperate types. The formal definition is therefore very similar, except for the definition of type_n. Although this visualisation only shows one node, it is required to define a value for all nodes that are typed by the node type corresponding to the field. Failing to do so would, once more, result in an invalid instance graph after it is combined with another graph. The correctness proof of $IG_{EnumFieldFlags}$ only is quite involved, but not be included here for conciseness. It can be found as part of the validated Isabelle proofs.

In order to make composing transformation functions possible, $IG_{EnumFieldFlags}$ should be compatible with the instance graph it is combined with.

Theorem 5.3.80 (Correctness of combine$(IG, IG_{EnumFieldFlags})$)
Assume an instance graph IG that is valid in the sense of Definition 3.3.10. Then IG is compatible with $IG_{EnumFieldFlags}$ (in the sense of Definition 4.4.25) if:

- *All requirements of Theorem 5.2.80 are met, to ensure the combination of the corresponding type graphs is valid;*
- *The node type on which the corresponding field is defined is not extended by other node types within the type graph corresponding to IG;*
- *All nodes in IG that are typed by the node type on which the field is defined are also nodes in $IG_{EnumFieldFlags}$;*
- *All nodes in $IG_{EnumFieldFlags}$ that encode the values of the corresponding enumeration type are also nodes in IG;*
- *For all nodes shared between IG and $IG_{EnumFieldFlags}$, each node must have the same identifier in both IG and $IG_{EnumFieldFlags}$;*
- *For all nodes for which the field is set, the values function must define a valid value.*

Also see `ig_enum_as_flags_field_as_edge_type_combine_correct` *in* `Ecore-GROOVE-Mapping-Library.EnumFieldValue`

Proof. Use Lemma 4.4.24. It is possible to show that all assumptions hold. Now we have shown that combine$(IG, IG_{EnumFieldFlags})$ is valid in the sense of Definition 3.3.10. □

The next definitions define the transformation function from $Im_{EnumField}$ to $IG_{EnumFieldFlags}$:

Definition 5.3.81 (Transformation function $f_{EnumFieldFlags}$)
The transformation function $f_{EnumFieldFlags}(Im)$ is defined as:

$$
\begin{aligned}
N &= Object_{Im} \cup \{enumob(ob) \mid v \in enumvalues\} \\
E &= \big\{\big(ob, (\text{ns\_to\_list}(classtype), \langle name \rangle, \text{ns\_to\_list}(enumid)), enumob(values(ob))\big) \mid \\
&\quad ob \in Object_{Im}\big\} \\
\text{ident} &= \begin{cases} (obids(ob), ob) & \text{if } ob \in Object_{Im} \\ (enumids(v), enumob(v)) & \text{if } v \in enumvalues \end{cases}
\end{aligned}
$$

with

$$
\text{type}_n = \begin{cases} (ob, \text{ns\_to\_list}(classtype)) & \text{if } ob \in Object_{Im} \\ (enumob(v), \text{ns\_to\_list}(enumid)) & \text{if } v \in enumvalues \end{cases}
$$

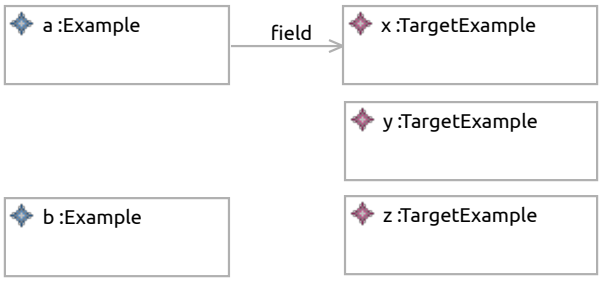
Also see `imod_enum_field_to_ig_enum_as_flags_field_as_edge_type` *in* `Ecore-GROOVE-Mapping-Library.EnumFieldValue`

Theorem 5.3.82 (Correctness of $f_{EnumFieldFlags}$)
$f_{EnumFieldFlags}(Im)$ (Definition 5.3.81) is a valid transformation function in the sense of Definition 4.4.26 transforming $Im_{EnumField}$ into $IG_{EnumFieldFlags}$.

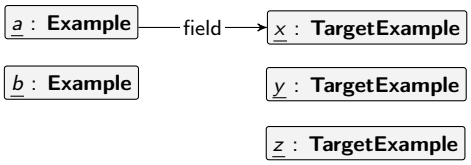
Also see `imod_enum_field_to_ig_enum_as_flags_field_as_edge_type_func` *in* `Ecore-GROOVE-Mapping-Library.EnumFieldValue`

The proof of the correctness of $f_{EnumFieldFlags}$ will not be included here. Instead, it can be found in the validated Isabelle theories.

Finally, to complete the transformation, the transformation function that transforms $IG_{EnumFieldFlags}$ into $Im_{EnumField}$ is defined:



(a) $Im_{NullableClassField}$ with examples of different nodes with different values for field



(b) $IG_{NullableClassField}$ with examples of different nodes with different values for field

Figure 5.15: Visualisation of the transformation of field values from fields typed by nullable class types

Definition 5.3.83 (Transformation function $f'_{EnumFieldFlags}$)
The transformation function $f'_{EnumFieldFlags}(IG)$ is defined as:

$$
\begin{aligned}
Object &= \{\text{src}(e) \mid e \in E_{IG}\} \\
\text{ObjectClass} &= \left\{(ob, classtype) \quad \text{if } ob \in \{\text{src}(e) \mid e \in E_{IG}\}\right. \\
\text{ObjectId} &= \left\{(ob, obids(ob)) \quad \text{if } ob \in \{\text{src}(e) \mid e \in E_{IG}\}\right. \\
\text{FieldValue} &= \left\{\Big((ob, (classtype, name)), \big[\mathsf{enum}, (enumid, values(ob))\big]\Big) \quad \text{if } ob \in \{\text{src}(e) \mid e \in E_{IG}\}\right. \\
\text{DefaultValue} &= \{\}
\end{aligned}
$$

Also see `ig_enum_as_flags_field_as_edge_type_to_imod_enum_field` *in* `Ecore-GROOVE-Mapping-Library.EnumFieldValue`

Theorem 5.3.84 (Correctness of $f'_{EnumFieldFlags}$)
$f'_{EnumFieldFlags}(IG)$ (Definition 5.3.83) is a valid transformation function in the sense of Definition 4.4.31 transforming $IG_{EnumFieldFlags}$ into $Im_{EnumField}$.

Also see `ig_enum_as_flags_field_as_edge_type_to_imod_enum_field_func` *in* `Ecore-GROOVE-Mapping-Library.EnumFieldValue`

Once more, the correctness proof is not included here but can be found in the validated Isabelle proofs of this thesis.

5.3.8 Nullable class field values

This section introduces the instance level transformation belonging to the transformation of a nullable class field. The type level transformation for nullable class fields can be found in Section 5.2.8. On the instance level, values for these fields are introduced.

Definition 5.3.85 (Instance model $Im_{NullableClassField}$)
Let $Im_{NullableClassField}$ be an instance model typed by $Tm_{NullableClassField}$ (Definition 5.2.85). Define disjoint sets valobjects and nilobjects. The objects in valobjects will get a proper class value for the field introduced by $Tm_{NullableClassField}$, while the objects in nilobjects get a nil *value for the same field. Furthermore, define a function obids which maps each of these objects to their corresponding identifier and a function values, which maps the objects in valobjects to its value for the field introduced by*

$Tm_{NullableClassField}$. $Im_{NullableClassField}$ *is defined as:*

$$
\begin{aligned}
Object &= nilobjects \cup valobjects \cup \{values(ob) \mid ob \in valobjects\} \\
\text{ObjectClass} &= \begin{cases} (ob, classtype) & \text{if } ob \in nilobjects \cup valobjects \\ (ob, fieldtype) & \text{if } ob \in \{values(ob) \mid ob \in valobjects\} \end{cases} \\
\text{ObjectId} &= \begin{cases} (ob, obids(ob)) & \text{if } ob \in objects \end{cases} \\
\text{FieldValue} &= \begin{cases} ((ob, (classtype, name)), \mathsf{nil}) & \text{if } ob \in nilobjects \\ ((ob, (classtype, name)), [\mathsf{obj}, values(ob)]) & \text{if } ob \in valobjects \end{cases} \\
\text{DefaultValue} &= \{\}
\end{aligned}
$$

Also see `imod_nullable_class_field` *in* `Ecore-GROOVE-Mapping-Library.NullableClassFieldValue`

Theorem 5.3.86 (Correctness of $Im_{NullableClassField}$)
$Im_{NullableClassField}$ (Definition 5.3.85) is a valid instance model in the sense of Definition 3.2.19.

Also see `imod_nullable_class_field_correct` *in* `Ecore-GROOVE-Mapping-Library.NullableClassFieldValue`

A visual representation of $Im_{NullableClassField}$ with $valobjects = \{ob_a\}$ and $nilobjects = \{ob_b\}$ can be seen in Figure 5.15a. In this visualisation, the field value for ob_a is defined as $values(ob_a) = ob_x$. Furthermore, the value for ob_b is nil, because it occurs within the set *nilobjects*. Like the previous transformations for field values, the value needs to be set for all objects that are typed by the class type corresponding to the field. Failing to do so would result in an invalid instance model after it is combined with another model, as the next definition will show. The correctness proof of $Im_{NullableClassField}$ only is already quite involved, but not be included here for conciseness. It can be found as part of the validated Isabelle proofs.

In order to make composing transformation functions possible, $Im_{NullableClassField}$ should be compatible with the instance model it is combined with.

Theorem 5.3.87 (Correctness of combine$(Im, Im_{NullableClassField})$)
Assume an instance model Im that is valid in the sense of Definition 3.2.19. Then Im is compatible with $Im_{NullableClassField}$ (in the sense of Definition 4.4.14) if:

- *All requirements of Theorem 5.2.87 are met, to ensure the combination of the corresponding type models is valid;*
- *The class type on which the field is defined by $Tm_{NullableClassField}$ may not be extended by another class type in the type model corresponding to Im;*
- *All of the objects in the sets nilobjects and valobjects must already be objects in Im;*
- *All of the objects referenced by the objects in the set valobjects must already be objects in Im;*
- *All objects typed by the class type on which the field is defined must occur in the set nilobjects ∪ valobjects and thus have a value in $Im_{NullableClassField}$;*
- *For all of the objects in the set objects, the identifier set by obids must be the same identifier as set by Im for that object;*
- *The sets valobjects and nilobjects must be disjoint, each object only gets a proper class value or a nil value, not both;*
- *For all objects in set valobjects, the value set by the values function must be valid.*

Also see `imod_nullable_class_field_combine_correct` *in* `Ecore-GROOVE-Mapping-Library.NullableClassFieldValue`

Proof. Use Lemma 4.4.13. It is possible to show that all assumptions hold. Now we have shown that combine$(Im, Im_{NullableClassField})$ is consistent in the sense of Definition 3.2.19. □

As explained earlier, $Im_{NullableClassField}$ needs to introduce values for all objects that are typed by the class type on which the field is defined. This is enforced by the requirements of Theorem 5.3.87. The proof is not included here for conciseness, but can be found as part of the validated proofs in Isabelle.

The definitions and theorems for introducing values for fields of data types within Ecore are now complete.

Encoding as edges and nodes

In the type level transformation of nullable class fields, nullable class fields were encoded in GROOVE as edge types to a corresponding encoded node type. On the instance level, this edge type will be used and edges will be created to give a value to each node type that has the field defined. The encoding corresponding to $Im_{NullableClassField}$ can then be represented as $IG_{NullableClassField}$, defined in the following definition:

Definition 5.3.88 (Instance graph $IG_{NullableClassField}$)
Let $IG_{NullableClassField}$ be the instance graph typed by type graph $TG_{NullableClassField}$ (Definition 5.2.88). Reuse the sets nilobjects and valobjects from $Im_{NullableClassField}$. Moreover, reuse the functions obids and values from $Im_{NullableClassField}$.

The objects in the sets nilobjects and valobjects are converted to nodes in $Im_{NullableClassField}$. For each of these objects, an edge of the encoded field is created. This edge targets a node that corresponds to the value set by values for the corresponding object in valobjects. The outgoing multiplicity of the edge type created by $TG_{NullableClassField}$ is 0..1, such that the objects in nilobjects do not need to have this edge, representing the absence of a value. Finally, the identity of the objects is defined using obids. $IG_{NullableClassField}$ is defined as:

$$N = nilobjects \cup valobjects \cup \{values(ob) \mid ob \in valobjects\}$$
$$E = \left\{\left(ob, (\text{ns\_to\_list}(classtype), \langle name \rangle, \text{ns\_to\_list}(fieldtype)), values(ob)\right) \mid ob \in valobjects\right\}$$
$$\text{ident} = \left\{(obids(ob), ob) \quad \text{if } ob \in objects\right.$$

with

$$\text{type}_n = \begin{cases} (ob, \text{ns\_to\_list}(classtype)) & \text{if } ob \in nilobjects \cup valobjects \\ (ob, \text{ns\_to\_list}(fieldtype)) & \text{if } ob \in \{values(ob) \mid ob \in valobjects\} \end{cases}$$

Also see `ig_nullable_class_field_as_edge_type` *in* `Ecore-GROOVE-Mapping-Library.NullableClassFieldValue`

Theorem 5.3.89 (Correctness of $IG_{NullableClassField}$)
$IG_{NullableClassField}$ (Definition 5.3.88) is a valid instance graph in the sense of Definition 3.3.10.

Also see `ig_nullable_class_field_as_edge_type_correct` *in* `Ecore-GROOVE-Mapping-Library.NullableClassFieldValue`

A visual representation of $IG_{NullableClassField}$ with $valobjects = \{ob_a\}$ and $nilobjects = \{ob_b\}$ can be seen in Figure 5.15b. Like the previous visualisation, the field value for ob_a is defined as $values(ob_a) = ob_x$. Since ob_b was in the set $nilobjects$, no edge has been created for this node. Like the previous field encodings, one needs to set the values for the field for all objects of the encoded class type at once. Failing to do so would result in an invalid instance graph after it is combined with another graph, as the next definition will show. The correctness proof of $IG_{NullableClassField}$ only is already quite involved, but not be included here for conciseness. It can be found as part of the validated Isabelle proofs.

In order to make composing transformation functions possible, $IG_{NullableClassField}$ should be compatible with the instance graph it is combined with.

Theorem 5.3.90 (Correctness of combine$(IG, IG_{NullableClassField})$)
Assume an instance graph IG that is valid in the sense of Definition 3.3.10. Then IG is compatible with $IG_{NullableClassField}$ (in the sense of Definition 4.4.25) if:

- *All requirements of Theorem 5.2.90 are met, to ensure the combination of the corresponding type graphs is valid;*
- *The node type on which the corresponding field is defined is not extended by other node types within the type graph corresponding to IG;*
- *All nodes in IG are also nodes in $IG_{NullableClassField}$;*
- *For all nodes shared between IG and $IG_{NullableClassField}$, each node must have the same identifier in both IG and $IG_{NullableClassField}$;*

- *The sets valobjects and nilobjects must be disjoint, each node gets either one edge to another node, or no edge at all;*
- *For all nodes for which the field is set, the values function must define a valid value.*

Also see `ig_nullable_class_field_as_edge_type_combine_correct` *in* `Ecore-GROOVE-Mapping-Library.NullableClassFieldValue`

Proof. Use Lemma 4.4.24. It is possible to show that all assumptions hold. Now we have shown that combine$(IG, IG_{NullableClassField})$ is valid in the sense of Definition 3.3.10. □

The next definitions define the transformation function from $Im_{NullableClassField}$ to $IG_{NullableClassField}$:

Definition 5.3.91 (Transformation function $f_{NullableClassField}$)
The transformation function $f_{NullableClassField}(Im)$ is defined as:

$$\begin{aligned} N &= Object_{Im} \\ E &= \{(ob, (\text{ns\_to\_list}(classtype), \langle name \rangle, \text{ns\_to\_list}(fieldtype)), values(ob)) \mid \\ &\quad ob \in Object_{Im} \wedge ob \in valobjects\} \\ \text{ident} &= \{(obids(ob), ob) \quad \text{if } ob \in Object_{Im} \end{aligned}$$

with

$$\text{type}_n = \begin{cases} (ob, \text{ns\_to\_list}(name)) & \text{if } ob \in Object_{Im} \wedge ob \in nilobjects \cup valobjects \\ (ob, \text{ns\_to\_list}(name)) & \text{if } ob \in Object_{Im} \wedge ob \in \{values(ob) \mid ob \in valobjects\} \end{cases}$$

Also see `imod_nullable_class_field_to_ig_nullable_class_field_as_edge_type` *in* `Ecore-GROOVE-Mapping-Library.NullableClassFieldValue`

Theorem 5.3.92 (Correctness of $f_{NullableClassField}$)
$f_{NullableClassField}(Im)$ (Definition 5.3.91) is a valid transformation function in the sense of Definition 4.4.26 transforming $Im_{NullableClassField}$ into $IG_{NullableClassField}$.

Also see `imod_nullable_class_field_to_ig_nullable_class_field_as_edge_type_func` *in* `Ecore-GROOVE-Mapping-Library.NullableClassFieldValue`

The proof of the correctness of $f_{NullableClassField}$ will not be included here. Instead, it can be found in the validated Isabelle theories.

Finally, to complete the transformation, the transformation function that transforms $IG_{NullableClassField}$ into $Im_{NullableClassField}$ is defined:

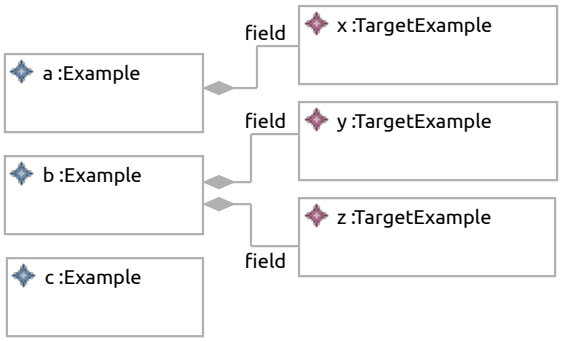
Definition 5.3.93 (Transformation function $f'_{NullableClassField}$)
The transformation function $f'_{NullableClassField}(IG)$ is defined as:

$$\begin{aligned} Object &= N_{IG} \\ \text{ObjectClass} &= \begin{cases} (ob, classtype) & \text{if } ob \in N_{IG} \wedge ob \in nilobjects \cup valobjects \\ (ob, fieldtype) & \text{if } ob \in N_{IG} \wedge ob \in \{values(ob) \mid ob \in valobjects\} \end{cases} \\ \text{ObjectId} &= \{(ob, obids(ob)) \quad \text{if } ob \in N_{IG} \\ \text{FieldValue} &= \begin{cases} ((ob, (classtype, name)), \mathsf{nil}) & \text{if } ob \in N_{IG} \wedge ob \in nilobjects \\ ((ob, (classtype, name)), [\mathsf{obj}, values(ob)]) & \text{if } ob \in N_{IG} \wedge ob \in valobjects \end{cases} \\ \text{DefaultValue} &= \{\} \end{aligned}$$

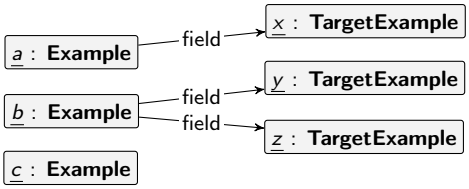
Also see `ig_nullable_class_field_as_edge_type_to_imod_nullable_class_field` *in* `Ecore-GROOVE-Mapping-Library.NullableClassFieldValue`

Theorem 5.3.94 (Correctness of $f'_{NullableClassField}$)
$f'_{NullableClassField}(IG)$ (Definition 5.3.93) is a valid transformation function in the sense of Definition 4.4.31 transforming $IG_{NullableClassField}$ into $Im_{NullableClassField}$.

Also see `ig_nullable_class_field_as_edge_type_to_imod_nullable_class_field_func` *in* `Ecore-GROOVE-Mapping-Library.NullableClassFieldValue`



(a) $Im_{ContainedClassSetField}$ with examples of different nodes with different values for field



(b) $IG_{ContainedClassSetField}$ with examples of different nodes with different values for field

Figure 5.16: Visualisation of the transformation of field values from containment fields typed by a set of a proper class type

Once more, the correctness proof is not included here but can be found in the validated Isabelle proofs of this thesis.

5.3.9 Contained class set field values

This section introduces the instance level transformation belonging to the transformation of a containment field of a set of a proper class type. The type level transformation belonging to these fields can be found in Section 5.2.9. On the instance level, values for these fields are introduced.

Definition 5.3.95 (Instance model $Im_{ContainedClassSetField}$)
Let $Im_{ContainedClassSetField}$ be an instance model typed by $Tm_{ContainedClassSetField}$ (Definition 5.2.95). Define a set objects, which represent the objects that will get a value for the field introduced by $Tm_{DataField}$. Furthermore, define a function obids which maps each of these objects to their corresponding identifier and a function values, which maps each of these objects to its value for the field introduced by $Tm_{DataField}$. Please note that values returns a set of objects, as the field allows for this. $Im_{DataField}$ is defined as:

$$Object = objects \cup \left(\bigcup_{ob \in objects} values(ob) \right)$$

$$\text{ObjectClass} = \begin{cases} (ob, classtype) & \text{if } ob \in objects \\ (ob, containedtype) & \text{if } ob \in \bigcup_{ob \in objects} values(ob) \end{cases}$$

$$\text{ObjectId} = \left\{ (ob, obids(ob)) \quad \text{if } ob \in objects \right.$$

$$\text{FieldValue} = \left\{ \left((ob, (classtype, name)), \left[\mathsf{setof}, \langle [\mathsf{obj}, ob] \mid ob \in values(ob) \rangle \right] \right) \quad \text{if } ob \in objects \right.$$

$$\text{DefaultValue} = \{\}$$

Also see `imod_contained_class_set_field` *in* `Ecore-GROOVE-Mapping-Library.ContainedClassSetFieldValue`

Theorem 5.3.96 (Correctness of $Im_{ContainedClassSetField}$)
$Im_{ContainedClassSetField}$ (Definition 5.3.95) is a valid instance model in the sense of Definition 3.2.19.

Also see `imod_contained_class_set_field_correct` *in* `Ecore-GROOVE-Mapping-Library.ContainedClassSetFieldValue`

A visual representation of $Im_{ContainedClassSetField}$ with $objects = \{ob_a, ob_b, ob_c\}$ can be seen in Figure 5.16a. This example is typed by $Tm_{ContainedClassSetField}$ in Figure 5.9a. In this visualisation, the field value for ob_a is defined as $values(ob_a) = \{ob_x\}$. Furthermore, the value for ob_b is $values(ob_a) = \{ob_y, ob_z\}$.

Finally, the value for ob_c is $values(ob_c) = \{\}$, which is allowed because the lower bound of the multiplicity is set 0 by the example. Like the previous transformations for field values, the value needs to be set for all objects that are typed by the class type corresponding to the field. Failing to do so would result in an invalid instance model after it is combined with another model, as the next definition will show. The correctness proof of $Im_{ContainedClassSetField}$ only is already quite involved, but not be included here for conciseness. It can be found as part of the validated Isabelle proofs.

In order to make composing transformation functions possible, $Im_{ContainedClassSetField}$ should be compatible with the instance model it is combined with.

Theorem 5.3.97 (Correctness of combine$(Im, Im_{ContainedClassSetField})$)
Assume an instance model Im that is valid in the sense of Definition 3.2.19. Then Im is compatible with $Im_{ContainedClassSetField}$ (in the sense of Definition 4.4.14) if:

- *All requirements of Theorem 5.2.97 are met, to ensure the combination of the corresponding type models is valid;*
- *The class type on which the field is defined by $Tm_{ContainedClassSetField}$ may not be extended by another class type in the type model corresponding to Im;*
- *The contained type and the class type cannot be the same, e.g. classtype ≠ containedtype.*
- *All of the objects in the set objects must already be objects in Im;*
- *All of the referenced objects cannot be objects in Im, they are newly introduced by $Im_{ContainedClassSetField}$;*
- *All objects typed by the class type on which the field is defined must occur in the set objects and thus have a value in $Im_{ContainedClassSetField}$;*
- *For all of the objects in the set objects, the identifier set by obids must be the same identifier as set by Im for that object;*
- *The object ids for the newly introduced objects must be unique with respect to each other and all other objects within Im;*
- *For all objects in set valobjects, the value set by the values function must be valid and the amount of elements in each value must be within the multiplicity mul.*

 *Also see* `imod_contained_class_set_field_combine_correct` *in* `Ecore-GROOVE-Mapping-Library.ContainedClassSetFieldValue`

Proof. Use Lemma 4.4.13. It is possible to show that all assumptions hold. Now we have shown that combine$(Im, Im_{ContainedClassSetField})$ is consistent in the sense of Definition 3.2.19. □

Please note that all objects referenced by any objects via this field are newly created. They may not exist on the existing model. This is enforced to ensure that the containment relations of objects remain acyclic, which is needed to keep the instance model valid. The proof is not included here for conciseness, but can be found as part of the validated proofs in Isabelle.

The definitions and theorems for introducing values for fields of data types within Ecore are now complete.

Encoding as edges and nodes

In the type level transformation of contained class set fields, a single containment edge type was introduced to encode the values for the containment field. On the instance level, the values for each object will be encoded using this edge type. The encoding corresponding to $Im_{ContainedClassSetField}$ can then be represented as $IG_{ContainedClassSetField}$, defined in the following definition:

Definition 5.3.98 (Instance graph $IG_{ContainedClassSetField}$)
Let $IG_{ContainedClassSetField}$ be the instance graph typed by type graph $TG_{ContainedClassSetField}$ (Definition 5.2.98). Reuse the set objects from $Im_{ContainedClassSetField}$. Moreover, reuse the functions obids and values from $Im_{ContainedClassSetField}$.

The objects in the set objects are converted to nodes in $Im_{ContainedClassSetField}$. For each of these objects, a edge is created for each referenced object within the value of that field. Each of these edges targets an node that encodes an object that was referenced by the value. Finally, the identity of the objects

is defined using obids. $IG_{ContainedClassSetField}$ *is defined as:*

$$N = objects \cup \left(\bigcup_{ob \in objects} values(ob) \right)$$

$$E = \bigcup_{ob \in objects} \{ (ob, (\text{ns\_to\_list}(classtype), \langle name \rangle, \text{ns\_to\_list}(containedtype)), v) \mid v \in values(ob) \}$$

$$\text{ident} = \left\{ (obids(ob), ob) \quad \text{if } ob \in objects \cup \left(\bigcup_{ob \in objects} values(ob) \right) \right.$$

with

$$\text{type}_n = \begin{cases} (ob, \text{ns\_to\_list}(classtype)) & \text{if } ob \in objects \\ (v, \text{ns\_to\_list}(containedtype)) & \text{if } v \in \bigcup_{ob \in objects} values(ob) \end{cases}$$

Also see `ig_contained_class_set_field_as_edge_type` *in* `Ecore-GROOVE-Mapping-Library.ContainedClassSetFieldValue`

Theorem 5.3.99 (Correctness of $IG_{ContainedClassSetField}$)
$IG_{ContainedClassSetField}$ *(Definition 5.3.98) is a valid instance graph in the sense of Definition 3.3.10.*

Also see `ig_contained_class_set_field_as_edge_type_correct` *in* `Ecore-GROOVE-Mapping-Library.ContainedClassSetFieldValue`

A visual representation of $IG_{ContainedClassSetField}$ with $objects = \{ob_a, ob_b, ob_c\}$ can be seen in Figure 5.16b. This example is typed by $TG_{ContainedClassSetField}$ in Figure 5.9b. In this visualisation, the field value for ob_a is defined as $values(ob_a) = \{ob_x\}$. Furthermore, the value for ob_b is $values(ob_a) = \{ob_y, ob_z\}$. Finally, the value for ob_c is $values(ob_c) = \{\}$. Like the previous field encodings, one needs to set the values for the field for all objects of the encoded class type at once. Failing to do so would result in an invalid instance graph after it is combined with another graph, as the next definition will show. The correctness proof of $IG_{ContainedClassSetField}$ only is already quite involved, but not be included here for conciseness. It can be found as part of the validated Isabelle proofs.

In order to make composing transformation functions possible, $IG_{ContainedClassSetField}$ should be compatible with the instance graph it is combined with.

Theorem 5.3.100 (Correctness of combine$(IG, IG_{ContainedClassSetField})$)
Assume an instance graph IG that is valid in the sense of Definition 3.3.10. Then IG is compatible with $IG_{ContainedClassSetField}$ *(in the sense of Definition 4.4.25) if:*

- *All requirements of Theorem 5.2.100 are met, to ensure the combination of the corresponding type graphs is valid;*
- *The node type on which the corresponding field is defined is not extended by other node types within the type graph corresponding to IG;*
- *The contained type and the class type cannot be the same, e.g.* $classtype \neq containedtype$.
- *All nodes in objects are also nodes in* $IG_{ContainedClassSetField}$;
- *All nodes referenced by the nodes in objects are not already nodes in* $IG_{ContainedClassSetField}$, *e.g. the nodes referenced by values are newly introduced;*
- *All nodes typed by the node type on which the field is defined must occur in the set objects and thus have a value in* $IG_{ContainedClassSetField}$;
- *The object ids for the newly introduced objects must be unique with respect to each other and all other objects within IG;*
- *For all nodes shared between IG and* $IG_{ContainedClassSetField}$, *each node must have the same identifier in both IG and* $IG_{ContainedClassSetField}$;
- *For all nodes in set objects, the value set by the values function must be valid and the amount of elements in each value must be within the multiplicity mul.*

Also see `ig_contained_class_set_field_as_edge_type_combine_correct` *in* `Ecore-GROOVE-Mapping-Library.ContainedClassSetFieldValue`

Proof. Use Lemma 4.4.24. It is possible to show that all assumptions hold. Now we have shown that combine$(IG, IG_{ContainedClassSetField})$ is valid in the sense of Definition 3.3.10. □

The next definitions define the transformation function from $Im_{ContainedClassSetField}$ to $IG_{ContainedClassSetField}$:

Definition 5.3.101 (Transformation function $f_{ContainedClassSetField}$)
The transformation function $f_{ContainedClassSetField}(Im)$ is defined as:

$$
\begin{aligned}
N &= Object_{Im} \\
E &= \bigcup_{ob \in Object_{Im} \wedge ob \in objects} \{(ob, (\text{ns\_to\_list}(classtype), \langle name \rangle, \text{ns\_to\_list}(containedtype)), v) \mid \\
&\qquad v \in values(ob)\} \\
\text{ident} &= \{(obids(ob), ob) \quad \text{if } ob \in Object_{Im}
\end{aligned}
$$

with

$$
\text{type}_n = \begin{cases} (ob, \text{ns\_to\_list}(classtype)) & \text{if } ob \in Object_{Im} \wedge ob \in objects \\ (v, \text{ns\_to\_list}(containedtype)) & \text{if } v \in \bigcup_{ob \in Object_{Im} \wedge ob \in objects} values(ob) \end{cases}
$$

Also see `imod_contained_class_set_field_to_ig_contained_class_set_field_as_edge_type` *in* `Ecore-GROOVE-Mapping-Library.ContainedClassSetFieldValue`

Theorem 5.3.102 (Correctness of $f_{ContainedClassSetField}$)
$f_{ContainedClassSetField}(Im)$ (Definition 5.3.101) is a valid transformation function in the sense of Definition 4.4.26 transforming $Im_{ContainedClassSetField}$ into $IG_{ContainedClassSetField}$.

Also see `imod_contained_class_set_field_to_ig_contained_class_set_field_as_edge_type_func` *in* `Ecore-GROOVE-Mapping-Library.ContainedClassSetFieldValue`

The proof of the correctness of $f_{ContainedClassSetField}$ will not be included here. Instead, it can be found in the validated Isabelle theories.

Finally, to complete the transformation, the transformation function that transforms $IG_{ContainedClassSetField}$ into $Im_{ContainedClassSetField}$ is defined:

Definition 5.3.103 (Transformation function $f'_{ContainedClassSetField}$)
The transformation function $f'_{ContainedClassSetField}(IG)$ is defined as:

$$
\begin{aligned}
Object &= N_{IG} \\
\text{ObjectClass} &= \begin{cases} (ob, classtype) & \text{if } ob \in N_{IG} \wedge ob \in objects \\ (ob, containedtype) & \text{if } ob \in N_{IG} \wedge ob \in \bigcup_{ob \in objects} values(ob) \end{cases} \\
\text{ObjectId} &= \{(ob, obids(ob)) \quad \text{if } ob \in N_{IG} \\
\text{FieldValue} &= \begin{cases} \Big((ob, (classtype, name)), \big[\mathsf{setof}, \langle [\mathsf{obj}, ob] \mid ob \in values(ob) \rangle\big]\Big) & \text{if } ob \in N_{IG} \wedge \\ & \qquad ob \in objects \end{cases} \\
\text{DefaultValue} &= \{\}
\end{aligned}
$$

Also see `ig_contained_class_set_field_as_edge_type_to_imod_contained_class_set_field` *in* `Ecore-GROOVE-Mapping-Library.ContainedClassSetFieldValue`

Theorem 5.3.104 (Correctness of $f'_{ContainedClassSetField}$)
$f'_{ContainedClassSetField}(IG)$ (Definition 5.3.103) is a valid transformation function in the sense of Definition 4.4.31 transforming $IG_{ContainedClassSetField}$ into $Im_{ContainedClassSetField}$.

Also see `ig_contained_class_set_field_as_edge_type_to_imod_contained_class_set_field_func` *in* `Ecore-GROOVE-Mapping-Library.ContainedClassSetFieldValue`

Once more, the correctness proof is not included here but can be found in the validated Isabelle proofs of this thesis.

Chapter 6

Application

This chapter will provide the necessary steps to apply the work presented in this thesis. So far, the thesis has introduced formalisations of Ecore and GROOVE, discussed in Section 3.2 and Section 3.3 respectively. Furthermore, an transformation framework for reasoning about composable model transformations has been introduced as part of Chapter 4. Finally, the previous chapter introduced some small transformations that represent the 'building blocks' to use within the framework. In this chapter, the transformation framework is applied by building an example model from scratch, showing the necessary steps to apply the framework. Building the model will be done using the 'building blocks' of the library of transformations. Furthermore, each of the steps is described formally using the formalisations presented earlier.

Contrary to the previous chapters, no distinction will be made between the type level and instance level. This distinction is unneeded because the different levels are ultimately tied to each other. When building an instance model or instance graph iteratively, a corresponding type model or type graph is build as well. Therefore, within this chapter, an instance model will be built. For each step in building the instance model, the corresponding step on the level of a type model will be discussed as well. Therefore, the steps for building both models are presented in a mixed fashion.

6.1 The model

Throughout this chapter, the steps for building the same model are shown. The model that will be built is a model that represents student housing. A visualisation of the final Ecore model is included in Figure 6.1. Within the model, student houses are represented through the .House type. A house can have 0 till 9 rooms, which are represented by the .Room type. Each room can have one tenant, represented by the .Tenant. Possibly, the tenant can have another subtenant, which is modelled through the subtenant relation.

Since many student houses have a name, this is reflected as such in the model. The .House type has a string attribute name which represents the name of the house. Furthermore, some houses do have a common living room, while others do not. This is modelled through the boolean attribute living_room.

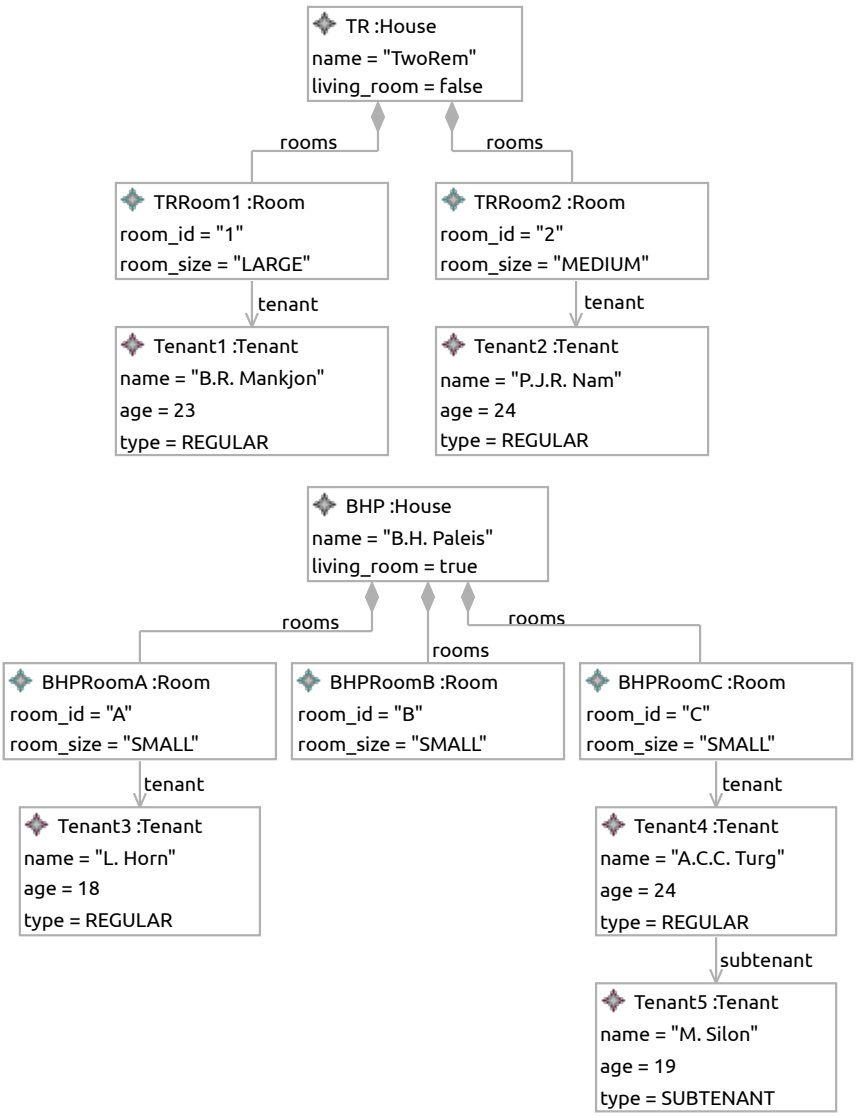
Each room within the house is identified by some identifier. Some houses number their rooms, and others use letters for the same purpose. The identifier of a room is reflected by the string attribute room_id. Furthermore, a room can either be small, medium or large. The size is used to determine the price of the room. This size of the room is represented using an enumeration type .RoomSize. Each room sets one size using the room_size attribute, which is typed by the .RoomSize enumeration type.

For tenants, the model stores the name and the age through the string attribute name and int attribute age respectively. Furthermore, a tenant can be a regular tenant or a subtenant, which is represented using an another enumeration type .TenantType. Each tenant is either a regular tenant of a subtenant, which is modelled using the type attribute, which is typed by the .TenantType enumeration type.

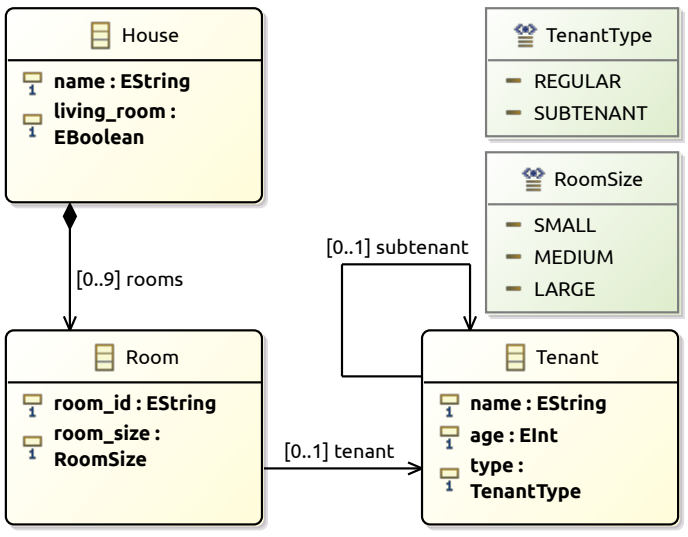
In the next section, this model will be built in 15 steps. Furthermore, a corresponding GROOVE encoding will be built at the same time. At the end of the next section, the model of Figure 6.1 is obtained, as well as the corresponding GROOVE encoding.

6.2 Building the model

Within this section, the 15 steps are provided to build the model represented by Figure 6.1. For each of these 15 steps, the corresponding Ecore model and GROOVE graphs are shown.



(a) Instance model



(b) Type model

Figure 6.1: The final model of student housing

Before the model is built, it is necessary to initialize the initial models. The initial models are empty and are used as a starting point. Each step will then add more elements to the model until the final model is obtained. Therefore, define the following models:

Initial models	
$Tm_0 =$	Tm_ϵ (Definition 4.3.8)
$Im_0 =$	Im_ϵ (Definition 4.4.9)
$TG_0 =$	TG_ϵ (Definition 4.3.19)
$IG_0 =$	IG_ϵ (Definition 4.4.20)
$f_0(Im_0) =$	IG_ϵ (Definition 4.4.20)
$f'_0(IG_0) =$	Im_ϵ (Definition 4.4.9)

Essentially, every model is defined to be the empty model, and every graph is defined to be the empty graph. Furthermore, f_0 is the mapping function which projects Im_0 (and Tm_0) onto IG_0 (and TG_0). f'_0 is the inverse function which maps IG_0 (and TG_0) onto Im_0 (and Tm_0).

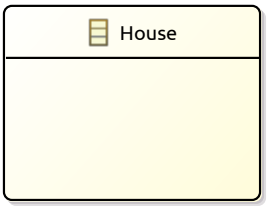
6.2.1 Houses

The first step of building the model is to add a type. Without any types, nothing interesting can be produced. The first type will be the class type for houses. Section 5.2.1 is used to introduce the class type, while on the instance level, Section 5.3.1 is used to introduce the house objects.

The *name* of the new house type is .House. Furthermore, 2 house objects are introduced, $objects = \{TR, BHP\}$. Furthermore, we assume that the object identifiers are equal to the internal node id, so $fid(TR) = TR$ and $fid(BHP) = BHP$. The following model is obtained:

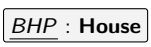
Models after step 1	
$Tm_1 =$	combine$(Tm_0, Tm_{Class}) =$ $Class = \{$.House$\}$ $Enum = \{\}$ $UserDataType = \{\}$ $Field = \{\}$ FieldSig $= \{\}$ $EnumValue = \{\}$ $Inh = \{\}$ $Prop = \{\}$ $Constant = \{\}$ ConstType $= \{\}$
$Im_1 =$	combine$(Tm_0, Im_{Class}) =$ $Object = \{TR, BHP\}$ ObjectClass$(ob) = \{(TR,$ .House$), (BHP,$ .House$)\}$ ObjectId $= \{(TR, TR), (BHP, BHP)\}$ FieldValue $= \{\}$ DefaultValue $= \{\}$
$TG_1 =$	combine$(TG_0, TG_{Class}) =$ $NT = \{\langle$House$\rangle\}$ $ET = \{\}$ $\sqsubseteq = \{(\langle$House$\rangle, \langle$House$\rangle)\}$ $abs = \{\}$ mult $= \{\}$ $contains = \{\}$
$IG_1 =$	combine$(TG_0, IG_{Class}) =$ $N = \{TR, BHP\}$ $E = \{\}$ ident $= \{(TR, TR), (BHP, BHP)\}$ type$_n = \{(TR, \langle$House$\rangle), (BHP, \langle$House$\rangle)\}$
$f_1(Im_1) =$	$f_0(Im_0) \sqcup f_{Class}(Im_{Class})$ (Definition 4.4.27)
$f'_1(IG_1) =$	$f'_0(IG_0) \sqcup f'_{Class}(IG_{Class})$ (Definition 4.4.32)

(a) Instance Model Im_1



(b) Type Model Tm_1

Figure 6.2: The Ecore model after step 1



(a) Instance Graph IG_1

(b) Type Graph TG_1

Figure 6.3: The GROOVE graphs after step 1

A visual representation of Tm_1 and Im_1 can be found in Figure 6.2. Similarly, a visual representation of TG_1 and IG_1 can be found in Figure 6.3. Please note that because of the definitions of $f_1(Im_1)$ and $f'_1(IG_1)$, we have that $f_1(Im_1) = IG_1$ and $f'_1(IG_1) = Im_1$. Furthermore, $f_1(Im_1)$ and $f'_1(IG_1)$ are valid mapping functions themselves, such that they can be combined with another mapping function in the next step.

The models itself are not very special as of yet, but that is expected. Each step is only a small building block, and introducing a type is not very special.

6.2.2 The Room class

The second step is very similar to the previous step, as yet another type will be introduced. This time around, the class type for rooms is introduced. Section 5.2.1 is used to introduce the class type, while on the instance level, Section 5.3.1 is used to introduce the house objects.

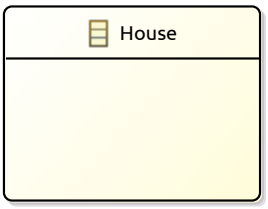
The *name* of the new room type is .Room. However, this time, no objects of the room type will be introduced just yet, therefore, $objects = \{\}$. Moreover, there is no need to define fid. The following model is obtained:

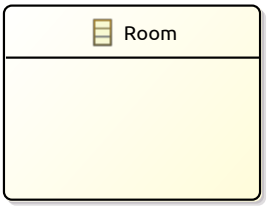
Models after step 2	
$Tm_2 =$	combine$(Tm_1, Tm_{Class}) =$
	$Class = \{.\mathsf{House}, .\mathsf{Room}\}$
	$Enum = \{\}$
	$UserDataType = \{\}$
	$Field = \{\}$
	FieldSig $= \{\}$
	$EnumValue = \{\}$
	$Inh = \{\}$
	$Prop = \{\}$
	$Constant = \{\}$
	ConstType $= \{\}$
$Im_2 =$	combine$(Tm_1, Im_{Class}) =$
	$Object = \{TR, BHP\}$
	ObjectClass$(ob) = \{(TR, .\mathsf{House}), (BHP, .\mathsf{House})\}$
	ObjectId $= \{(TR, TR), (BHP, BHP)\}$
	FieldValue $= \{\}$
	DefaultValue $= \{\}$
$TG_2 =$	combine$(TG_1, TG_{Class}) =$
	$NT = \{\langle\mathsf{House}\rangle, \langle\mathsf{Room}\rangle\}$
	$ET = \{\}$
	$\sqsubseteq = \{(\langle\mathsf{House}\rangle, \langle\mathsf{House}\rangle), (\langle\mathsf{Room}\rangle, \langle\mathsf{Room}\rangle)\}$
	$abs = \{\}$
	mult $= \{\}$
	$contains = \{\}$
$IG_2 =$	combine$(TG_1, IG_{Class}) =$
	$N = \{TR, BHP\}$
	$E = \{\}$
	ident $= \{(TR, TR), (BHP, BHP)\}$
	type$_n$ $= \{(TR, \langle\mathsf{House}\rangle), (BHP, \langle\mathsf{House}\rangle)\}$
$f_2(Im_2) =$	$f_1(Im_1) \sqcup f_{Class}(Im_{Class})$ (Definition 4.4.27)
$f'_2(IG_2) =$	$f'_1(IG_1) \sqcup f'_{Class}(IG_{Class})$ (Definition 4.4.32)

A visual representation of Tm_2 and Im_2 can be found in Figure 6.4. Similarly, a visual representation of TG_2 and IG_2 can be found in Figure 6.5. Please note that because of the definitions of $f_2(Im_2)$ and $f'_2(IG_2)$, we have that $f_2(Im_2) = IG_2$ and $f'_2(IG_2) = Im_2$. Furthermore, $f_2(Im_2)$ and $f'_2(IG_2)$ are valid mapping functions themselves, such that they can be combined with another mapping function in the next step.

The visual representation of the models is still not stunning, mainly because the instance model and instance graph have not changed during this step. These are unchanged because the room objects will

(a) Instance Model Im_2





(b) Type Model Tm_2

Figure 6.4: The Ecore model after step 2

(a) Instance Graph IG_2

(b) Type Graph TG_2

Figure 6.5: The GROOVE graphs after step 2

all be contained by houses, and therefore need to be introduced later to keep the model and graphs valid.

6.2.3 House names

In the third step, something interesting finally happens. This step introduces the names for houses, or in model terms, the name attribute on the .House class is introduced, including its values. Section 5.2.6 is used to introduce the field, while on the instance level, Section 5.3.6 is used to introduce the values.

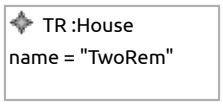
The *classtype* of the new field is .House, as the field will be defined for houses. The *name* of the new field is name and the $fieldtype$ is string. The set of objects of which the value is set is is equal to all house objects, so $objects = \{TR, BHP\}$. The function for *obids* returns the existing identifier of each of these objects. The *values* function is defined as follows:

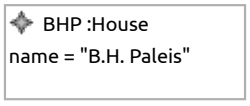
$$values = \{(TR, \text{“TwoRem”}), (BHP, \text{“B.H. Paleis”})\}$$

The following model is obtained:

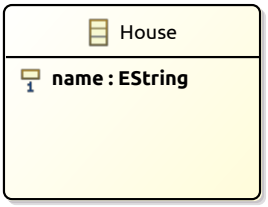
Models after step 3	
$Tm_3 =$	$\text{combine}(Tm_2, Tm_{DataField}) =$
	$Class = \{\text{.House}, \text{.Room}\}$
	$Enum = \{\}$
	$UserDataType = \{\}$
	$Field = \{(\text{.House}, \text{name})\}$
	$\text{FieldSig} = \{((\text{.House}, \text{name}), (\text{string}, 1..1))\}$
	$EnumValue = \{\}$
	$Inh = \{\}$
	$Prop = \{\}$
	$Constant = \{\}$
	$\text{ConstType} = \{\}$
$Im_3 =$	$\text{combine}(Tm_2, Im_{DataField}) =$
	$Object = \{TR, BHP\}$
	$\text{ObjectClass}(ob) = \{(TR, \text{.House}), (BHP, \text{.House})\}$
	$\text{ObjectId} = \{(TR, TR), (BHP, BHP)\}$
	$\text{FieldValue} = \{((TR, (\text{.House}, \text{name})), [\text{string}, \text{“TwoRem”}]),$
	$((BHP, (\text{.House}, \text{name})), [\text{string}, \text{“B.H. Paleis”}])\}$
	$\text{DefaultValue} = \{\}$
$TG_3 =$	$\text{combine}(TG_2, TG_{DataField}) =$
	$NT = \{\langle\text{House}\rangle, \langle\text{Room}\rangle, \text{string}\}$
	$ET = \{(\langle\text{House}\rangle, \langle\text{name}\rangle, \text{string})\}$
	$\sqsubseteq = \{(\langle\text{House}\rangle, \langle\text{House}\rangle), (\langle\text{Room}\rangle, \langle\text{Room}\rangle), (\text{string}, \text{string})\}$
	$abs = \{\}$
	$\text{mult} = \{((\langle\text{House}\rangle, \langle\text{name}\rangle, \text{string}), (0..*, 1..1))\}$
	$contains = \{\}$
$IG_3 =$	$\text{combine}(TG_2, IG_{DataField}) =$
	$N = \{TR, BHP, \text{“TwoRem”}, \text{“B.H. Paleis”}\}$
	$E = \{(TR, (\langle\text{House}\rangle, \langle\text{name}\rangle, \text{string}), \text{“TwoRem”}),$
	$(BHP, (\langle\text{House}\rangle, \langle\text{name}\rangle, \text{string}), \text{“B.H. Paleis”})\}$
	$\text{ident} = \{(TR, TR), (BHP, BHP)\}$
	$\text{type}_n = \{(TR, \langle\text{House}\rangle), (BHP, \langle\text{House}\rangle), (\text{“TwoRem”}, \text{string}), (\text{“B.H. Paleis”}, \text{string})\}$
$f_3(Im_3) =$	$f_2(Im_2) \sqcup f_{DataField}(Im_{DataField})$ (Definition 4.4.27)
$f'_3(IG_3) =$	$f'_2(IG_2) \sqcup f'_{DataField}(IG_{DataField})$ (Definition 4.4.32)

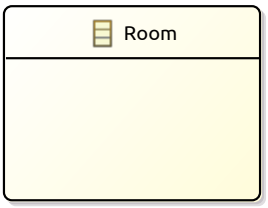
A visual representation of Tm_3 and Im_3 can be found in Figure 6.6. Similarly, a visual representation of TG_3 and IG_3 can be found in Figure 6.7. Please note that because of the definitions of $f_3(Im_3)$ and





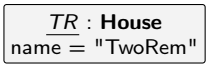
(a) Instance Model Im_3

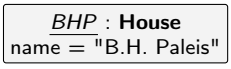




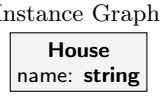
(b) Type Model Tm_3

Figure 6.6: The Ecore model after step 3





(a) Instance Graph IG_3



(b) Type Graph TG_3

Figure 6.7: The GROOVE graphs after step 3

$f'_3(IG_3)$, we have that $f_3(Im_3) = IG_3$ and $f'_3(IG_3) = Im_3$. Furthermore, $f_3(Im_3)$ and $f'_3(IG_3)$ are valid mapping functions themselves, such that they can be combined with another mapping function in the next step.

Although visually the models are still not very advanced, formally they already from quite a definition. This definition will only get more substantial as more fields and objects are added in the next steps.

6.2.4 Rooms

In the fourth step, the room objects will be introduced as part of the introduction of the rooms containment relation. This step introduces the rooms relation on the .House class, including its values. Section 5.2.9 is used to introduce the field, while on the instance level, Section 5.3.9 is used to introduce the values.

The *classtype* of the new field is .House, as the field will be defined for houses. The *name* of the new field is rooms and the *containedtype* is .Room. The set of objects of which the value is set is equal to all house objects, so $objects = \{TR, BHP\}$. The function for *obids* returns the existing identifier of each of these objects. The multiplicity is set to 0..9 for the new field, and the *values* function is defined as follows:

$$values = \big\{\big(TR, \{TRRoom1, TRRoom2\}\big), \big(BHP, \{BHPRoomA, BHPRoomB, BHPRoomC\}\big)\big\}$$

Please note that the referenced objects are all new. For these new objects *obids* returns as identifier the internal node label, just as was done for the houses. The following model is obtained:

Models after step 4

$Tm_4 =$ combine$(Tm_3, Tm_{ContainedClassSetField}) =$

$Class = \{$.House, .Room$\}$

$Enum = \{\}$

$UserDataType = \{\}$

$Field = \{$(.House, name), (.House, rooms)$\}$

FieldSig $= \big\{\big((.\text{House}, \text{name}), (\text{string}, 1..1)\big),$
$\big((.\text{House}, \text{rooms}), ([\text{setof}, !.\text{Room}], 0..9)\big)\big\}$

$EnumValue = \{\}$

$Inh = \{\}$

$Prop = \big\{\big(\text{containment}, (.\text{House}, \text{rooms})\big)\big\}$

$Constant = \{\}$

ConstType $= \{\}$

$Im_4 =$ combine$(Tm_3, Im_{ContainedClassSetField}) =$

$Object = \{TR, BHP, TRRoom1, TRRoom2, BHPRoomA, BHPRoomB, BHPRoomC\}$

ObjectClass$(ob) = \{(TR, .\text{House}), (BHP, .\text{House}), (TRRoom1, .\text{Room}), (TRRoom2, .\text{Room}), (BHPRoomA, .\text{Room}), (BHPRoomB, .\text{Room}), (BHPRoomC, .\text{Room})\}$

ObjectId $= \{(TR, TR), (BHP, BHP), (TRRoom1, TRRoom1), (TRRoom2, TRRoom2), (BHPRoomA, BHPRoomA), (BHPRoomB, BHPRoomB), (BHPRoomC, BHPRoomC)\}$

FieldValue $= \Big\{\Big(\big(TR, (.\text{House}, \text{name})\big), \big[\text{string}, \text{“TwoRem”}\big]\Big),$
$\Big(\big(BHP, (.\text{House}, \text{name})\big), \big[\text{string}, \text{“B.H. Paleis”}\big]\Big),$
$\Big(\big(TR, (.\text{House}, \text{rooms})\big), \big[\text{setof}, \big\langle[\text{obj}, TRRoom1],$
$[\text{obj}, TRRoom2]\big\rangle\big]\Big),$
$\Big(\big(BHP, (.\text{House}, \text{rooms})\big) \big[\text{setof}, \big\langle[\text{obj}, BHPRoomA],$
$[\text{obj}, BHPRoomB], (\text{obj}, BHPRoomC)\big\rangle\big]\Big)\Big\}$

DefaultValue $= \{\}$

Models after step 4

$TG_4 =$ $\text{combine}(TG_3, TG_{ContainedClassSetField}) =$

$$NT = \{\langle \mathsf{House}\rangle, \langle \mathsf{Room}\rangle, \mathsf{string}\}$$

$$ET = \big\{\big(\langle \mathsf{House}\rangle, \langle \mathsf{name}\rangle, \mathsf{string}\big), \big(\langle \mathsf{House}\rangle, \langle \mathsf{rooms}\rangle, \langle \mathsf{Room}\rangle\big)\big\}$$

$$\sqsubseteq = \big\{\big(\langle \mathsf{House}\rangle, \langle \mathsf{House}\rangle\big), \big(\langle \mathsf{Room}\rangle, \langle \mathsf{Room}\rangle\big), \big(\mathsf{string}, \mathsf{string}\big)\big\}$$

$$abs = \{\}$$

$$\text{mult} = \Big\{\Big(\big(\langle \mathsf{House}\rangle, \langle \mathsf{name}\rangle, \mathsf{string}\big), \big(0..*, 1..1\big)\Big), \Big(\big(\langle \mathsf{House}\rangle, \langle \mathsf{rooms}\rangle, \langle \mathsf{Room}\rangle\big), \big(0..1, 0..9\big)\Big)\Big\}$$

$$contains = \big\{\big(\langle \mathsf{House}\rangle, \langle \mathsf{rooms}\rangle, \langle \mathsf{Room}\rangle\big)\big\}$$

$IG_4 =$ $\text{combine}(TG_3, IG_{ContainedClassSetField}) =$

$$N = \{TR, BHP, TRRoom1, TRRoom2, BHPRoomA, BHPRoomB, BHPRoomC, \text{“TwoRem”}, \text{“B.H. Paleis”}\}$$

$$E = \Big\{\Big(TR, \big(\langle \mathsf{House}\rangle, \langle \mathsf{name}\rangle, \mathsf{string}\big), \text{“TwoRem”}\Big), \Big(BHP, \big(\langle \mathsf{House}\rangle, \langle \mathsf{name}\rangle, \mathsf{string}\big), \text{“B.H. Paleis”}\Big), \Big(TR, \big(\langle \mathsf{House}\rangle, \langle \mathsf{rooms}\rangle, \langle \mathsf{Room}\rangle\big), TRRoom1\Big), \Big(TR, \big(\langle \mathsf{House}\rangle, \langle \mathsf{rooms}\rangle, \langle \mathsf{Room}\rangle\big), TRRoom2\Big), \Big(BHP, \big(\langle \mathsf{House}\rangle, \langle \mathsf{rooms}\rangle, \langle \mathsf{Room}\rangle\big), BHPRoomA\Big), \Big(BHP, \big(\langle \mathsf{House}\rangle, \langle \mathsf{rooms}\rangle, \langle \mathsf{Room}\rangle\big), BHPRoomB\Big), \Big(BHP, \big(\langle \mathsf{House}\rangle, \langle \mathsf{rooms}\rangle, \langle \mathsf{Room}\rangle\big), BHPRoomC\Big)\Big\}$$

$$\text{ident} = \{(TR, TR), (BHP, BHP), (TRRoom1, TRRoom1), (TRRoom2, TRRoom2), (BHPRoomA, BHPRoomA), (BHPRoomB, BHPRoomB), (BHPRoomC, BHPRoomC)\}$$

$$\text{type}_n = \{(TR, \langle \mathsf{House}\rangle), (BHP, \langle \mathsf{House}\rangle), (TRRoom1, \langle \mathsf{Room}\rangle), (TRRoom2, \langle \mathsf{Room}\rangle), (BHPRoomA, \langle \mathsf{Room}\rangle), (BHPRoomB, \langle \mathsf{Room}\rangle), (BHPRoomC, \langle \mathsf{Room}\rangle), (\text{“TwoRem”}, \mathsf{string}), (\text{“B.H. Paleis”}, \mathsf{string})\}$$

$f_4(Im_4) =$ $f_3(Im_3) \sqcup f_{ContainedClassSetField}(Im_{ContainedClassSetField})$ (Definition 4.4.27)

$f'_4(IG_4) =$ $f'_3(IG_3) \sqcup f'_{ContainedClassSetField}(IG_{ContainedClassSetField})$ (Definition 4.4.32)

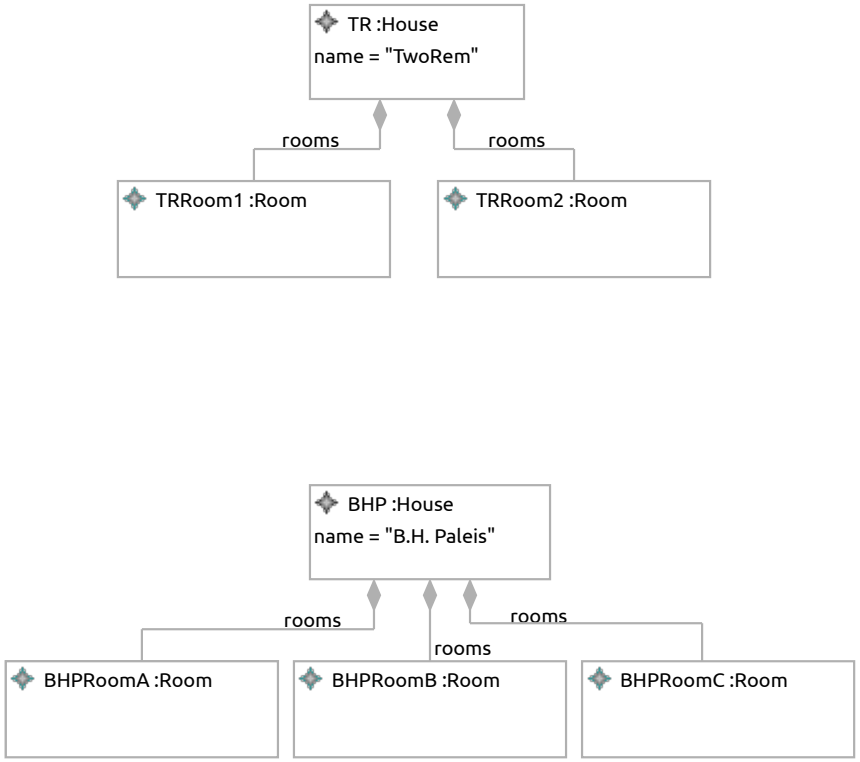
A visual representation of Tm_4 and Im_4 can be found in Figure 6.8. Similarly, a visual representation of TG_4 and IG_4 can be found in Figure 6.9. Please note that because of the definitions of $f_4(Im_4)$ and $f'_4(IG_4)$, we have that $f_4(Im_4) = IG_4$ and $f'_4(IG_4) = Im_4$. Furthermore, $f_4(Im_4)$ and $f'_4(IG_4)$ are valid mapping functions themselves, such that they can be combined with another mapping function in the next step.

The introduction of the room objects makes that the model represents something. A house has rooms that it contains, and each house has a different set of rooms. Still, there is room to enlarge the model more, such that more details are included.

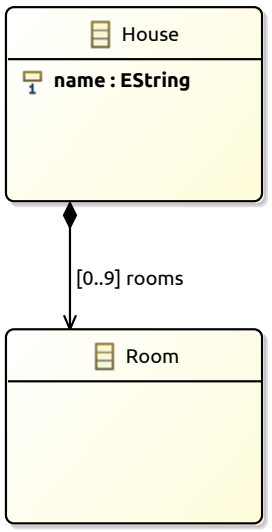
6.2.5 Room identifiers

In the fifth step, the room identifiers are introduced by introducing another string field. This step introduces gives each room a size, or in model terms, the room_size attribute on the .Room class is introduced, including its values. Section 5.2.7 is used to introduce the field on the type level, while on the instance level, Section 5.3.7 is used to introduce the values.

The *classtype* of the new field is .Room, as the field will be defined for rooms. The *name* of the new field is room_id and the $fieldtype$ is .string. The set of objects of which the value is set is equal to all room objects, so $objects = \{TRRoom1, TRRoom2, BHPRoomA, BHPRoomB, BHPRoomC\}$. The function for *obids* returns the existing identifier of each of these objects. The *values* function is defined

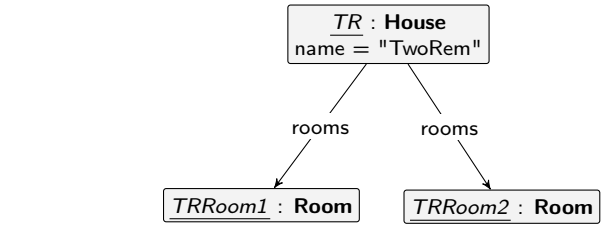


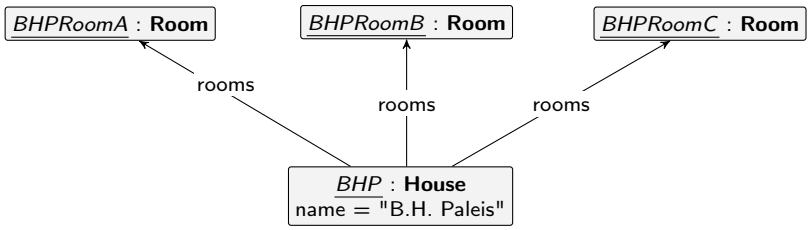
(a) Instance Model Im_4



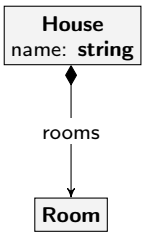
(b) Type Model Tm_4

Figure 6.8: The Ecore model after step 4





(a) Instance Graph IG_4



(b) Type Graph TG_4

Figure 6.9: The GROOVE graphs after step 4

as follows:

$$values = \{(TRRoom1, \text{“1”}), (TRRoom2, \text{“2”}), (BHPRoomA, \text{“A”}), (BHPRoomB, \text{“B”}), (BHPRoomC, \text{“C”})\}$$

The following model is obtained:

Models after step 5

$Tm_5 =$ combine$(Tm_4, Tm_{DataField}) =$

$$Class = \{\text{.House}, \text{.Room}\}$$
$$Enum = \{\}$$
$$UserDataType = \{\}$$
$$Field = \{(\text{.House}, \text{name}), (\text{.House}, \text{rooms}), (\text{.Room}, \text{room\_id})\}$$
$$\text{FieldSig} = \big\{\big((\text{.House}, \text{name}), (\text{string}, 1..1)\big), \big((\text{.House}, \text{rooms}), ([\text{setof}, \text{!.Room}], 0..9)\big), \big((\text{.Room}, \text{room\_id}), (\text{string}, 1..1)\big)\big\}$$
$$EnumValue = \{\}$$
$$Inh = \{\}$$
$$Prop = \big\{\big(\text{containment}, (\text{.House}, \text{rooms})\big)\big\}$$
$$Constant = \{\}$$
$$\text{ConstType} = \{\}$$

$Im_5 =$ combine$(Tm_4, Im_{DataField}) =$

$$Object = \{TR, BHP, TRRoom1, TRRoom2, BHPRoomA, BHPRoomB, BHPRoomC\}$$
$$\text{ObjectClass}(ob) = \{(TR, \text{.House}), (BHP, \text{.House}), (TRRoom1, \text{.Room}), (TRRoom2, \text{.Room}), (BHPRoomA, \text{.Room}), (BHPRoomB, \text{.Room}), (BHPRoomC, \text{.Room})\}$$
$$\text{ObjectId} = \{(TR, TR), (BHP, BHP), (TRRoom1, TRRoom1), (TRRoom2, TRRoom2), (BHPRoomA, BHPRoomA), (BHPRoomB, BHPRoomB), (BHPRoomC, BHPRoomC)\}$$
$$\text{FieldValue} = \Big\{\Big(\big(TR, (\text{.House}, \text{name})\big), \big[\text{string}, \text{“TwoRem”}\big]\Big), \Big(\big(BHP, (\text{.House}, \text{name})\big), \big[\text{string}, \text{“B.H. Paleis”}\big]\Big), \Big(\big(TR, (\text{.House}, \text{rooms})\big), \big[\text{setof}, \langle[\text{obj}, TRRoom1], [\text{obj}, TRRoom2]\rangle\big]\Big), \Big(\big(BHP, (\text{.House}, \text{rooms})\big) \big[\text{setof}, \langle[\text{obj}, BHPRoomA], [\text{obj}, BHPRoomB], (\text{obj}, BHPRoomC)\rangle\big]\Big), \Big(\big(TRRoom1, (\text{.Room}, \text{room\_id})\big), \big[\text{string}, \text{“1”}\big]\Big), \Big(\big(TRRoom2, (\text{.Room}, \text{room\_id})\big), \big[\text{string}, \text{“2”}\big]\Big), \Big(\big(BHPRoomA, (\text{.Room}, \text{room\_id})\big), \big[\text{string}, \text{“A”}\big]\Big), \Big(\big(BHPRoomB, (\text{.Room}, \text{room\_id})\big), \big[\text{string}, \text{“B”}\big]\Big), \Big(\big(BHPRoomC, (\text{.Room}, \text{room\_id})\big), \big[\text{string}, \text{“C”}\big]\Big)\Big\}$$
$$\text{DefaultValue} = \{\}$$

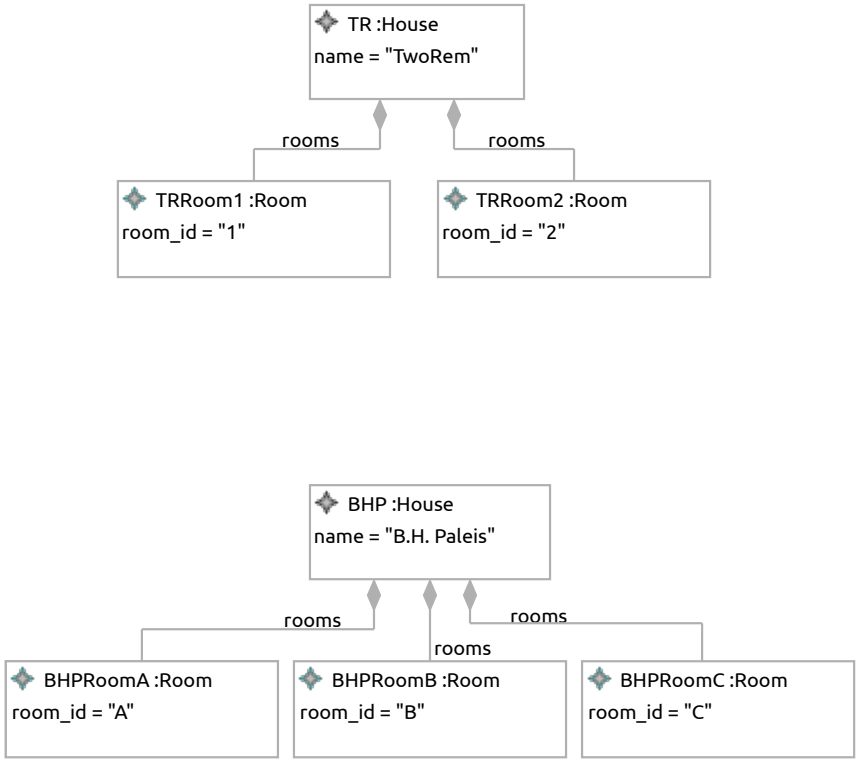
Models after step 5

$$
\begin{aligned}
TG_5 = \ & \text{combine}(TG_4, TG_{DataField}) = \\
& NT = \{\langle\mathsf{House}\rangle, \langle\mathsf{Room}\rangle, \mathsf{string}\} \\
& ET = \big\{\big(\langle\mathsf{House}\rangle, \langle\mathsf{name}\rangle, \mathsf{string}\big), \big(\langle\mathsf{House}\rangle, \langle\mathsf{rooms}\rangle, \langle\mathsf{Room}\rangle\big), \\
& \qquad \big(\langle\mathsf{Room}\rangle, \langle\mathsf{room\_id}\rangle, \mathsf{string}\big)\big\} \\
& \sqsubseteq = \big\{\big(\langle\mathsf{House}\rangle, \langle\mathsf{House}\rangle\big), \big(\langle\mathsf{Room}\rangle, \langle\mathsf{Room}\rangle\big), \big(\mathsf{string}, \mathsf{string}\big)\big\} \\
& abs = \{\} \\
& \text{mult} = \Big\{\Big(\big(\langle\mathsf{House}\rangle, \langle\mathsf{name}\rangle, \mathsf{string}\big), \big(0..*, 1..1\big)\Big), \\
& \qquad \Big(\big(\langle\mathsf{House}\rangle, \langle\mathsf{rooms}\rangle, \langle\mathsf{Room}\rangle\big), \big(0..1, 0..9\big)\Big), \\
& \qquad \Big(\big(\langle\mathsf{Room}\rangle, \langle\mathsf{room\_id}\rangle, \mathsf{string}\big), \big(0..*, 1..1\big)\Big)\Big\} \\
& contains = \big\{\big(\langle\mathsf{House}\rangle, \langle\mathsf{rooms}\rangle, \langle\mathsf{Room}\rangle\big)\big\} \\
IG_5 = \ & \text{combine}(TG_4, IG_{DataField}) = \\
& N = \{TR, BHP, TRRoom1, TRRoom2, BHPRoomA, BHPRoomB, BHPRoomC, \\
& \quad \text{“TwoRem”}, \text{“B.H. Paleis”}, \text{“1”}, \text{“2”}, \text{“A”}, \text{“B”}, \text{“C”}\} \\
& E = \Big\{\Big(TR, \big(\langle\mathsf{House}\rangle, \langle\mathsf{name}\rangle, \mathsf{string}\big), \text{“TwoRem”}\Big), \\
& \qquad \Big(BHP, \big(\langle\mathsf{House}\rangle, \langle\mathsf{name}\rangle, \mathsf{string}\big), \text{“B.H. Paleis”}\Big), \\
& \qquad \Big(TR, \big(\langle\mathsf{House}\rangle, \langle\mathsf{rooms}\rangle, \langle\mathsf{Room}\rangle\big), TRRoom1\Big), \\
& \qquad \Big(TR, \big(\langle\mathsf{House}\rangle, \langle\mathsf{rooms}\rangle, \langle\mathsf{Room}\rangle\big), TRRoom2\Big), \\
& \qquad \Big(BHP, \big(\langle\mathsf{House}\rangle, \langle\mathsf{rooms}\rangle, \langle\mathsf{Room}\rangle\big), BHPRoomA\Big), \\
& \qquad \Big(BHP, \big(\langle\mathsf{House}\rangle, \langle\mathsf{rooms}\rangle, \langle\mathsf{Room}\rangle\big), BHPRoomB\Big), \\
& \qquad \Big(BHP, \big(\langle\mathsf{House}\rangle, \langle\mathsf{rooms}\rangle, \langle\mathsf{Room}\rangle\big), BHPRoomC\Big), \\
& \qquad \Big(TRRoom1, \big(\langle\mathsf{Room}\rangle, \langle\mathsf{room\_id}\rangle, \mathsf{string}\big), \text{“1”}\Big), \\
& \qquad \Big(TRRoom2, \big(\langle\mathsf{Room}\rangle, \langle\mathsf{room\_id}\rangle, \mathsf{string}\big), \text{“2”}\Big), \\
& \qquad \Big(BHPRoomA, \big(\langle\mathsf{Room}\rangle, \langle\mathsf{room\_id}\rangle, \mathsf{string}\big), \text{“A”}\Big), \\
& \qquad \Big(BHPRoomB, \big(\langle\mathsf{Room}\rangle, \langle\mathsf{room\_id}\rangle, \mathsf{string}\big), \text{“B”}\Big), \\
& \qquad \Big(BHPRoomC, \big(\langle\mathsf{Room}\rangle, \langle\mathsf{room\_id}\rangle, \mathsf{string}\big), \text{“C”}\Big)\Big\} \\
& \text{ident} = \{(TR, TR), (BHP, BHP), (TRRoom1, TRRoom1), (TRRoom2, TRRoom2), \\
& \qquad (BHPRoomA, BHPRoomA), (BHPRoomB, BHPRoomB), \\
& \qquad (BHPRoomC, BHPRoomC)\} \\
& \text{type}_n = \{(TR, \langle\mathsf{House}\rangle), (BHP, \langle\mathsf{House}\rangle), (TRRoom1, \langle\mathsf{Room}\rangle), (TRRoom2, \langle\mathsf{Room}\rangle), \\
& \qquad (BHPRoomA, \langle\mathsf{Room}\rangle), (BHPRoomB, \langle\mathsf{Room}\rangle), (BHPRoomC, \langle\mathsf{Room}\rangle), \\
& \qquad (\text{“TwoRem”}, \mathsf{string}), (\text{“B.H. Paleis”}, \mathsf{string}), (\text{“1”}, \mathsf{string}), (\text{“2”}, \mathsf{string}), (\text{“A”}, \mathsf{string}), \\
& \qquad (\text{“B”}, \mathsf{string}), (\text{“C”}, \mathsf{string})\}
\end{aligned}
$$

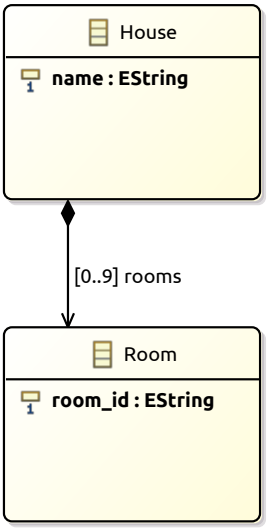
$f_5(Im_5) = f_4(Im_4) \sqcup f_{DataField}(Im_{DataField})$ (Definition 4.4.27)
$f'_5(IG_5) = f'_4(IG_4) \sqcup f'_{DataField}(IG_{DataField})$ (Definition 4.4.32)

A visual representation of Tm_5 and Im_5 can be found in Figure 6.10. Similarly, a visual representation of TG_5 and IG_5 can be found in Figure 6.11. Please note that because of the definitions of $f_5(Im_5)$ and $f'_5(IG_5)$, we have that $f_5(Im_5) = IG_5$ and $f'_5(IG_5) = Im_5$. Furthermore, $f_5(Im_5)$ and $f'_5(IG_5)$ are valid mapping functions themselves, such that they can be combined with another mapping function in the next step.

The visualisation unveils no surprising details. A string field was already added earlier to the house objects and introducing such a field on the room only shows that it is indeed possible to introduce fields on contained objects.

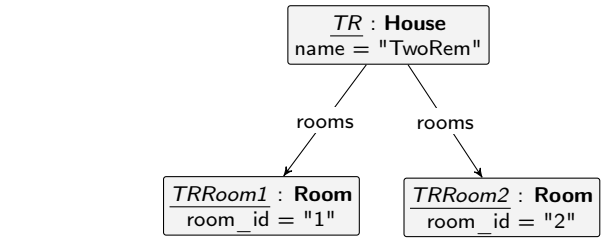


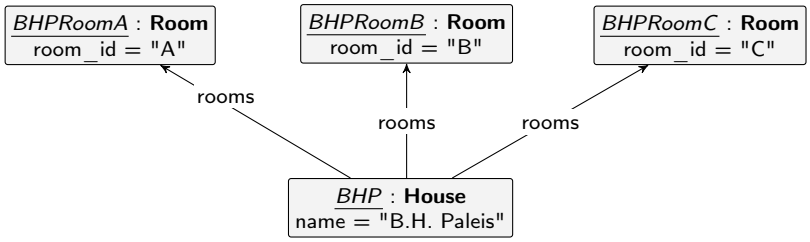
(a) Instance Model Im_5



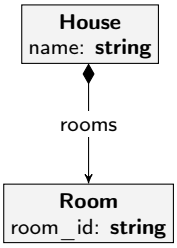
(b) Type Model Tm_5

Figure 6.10: The Ecore model after step 5





(a) Instance Graph IG_5



(b) Type Graph TG_5

Figure 6.11: The GROOVE graphs after step 5

6.2.6 The room size enumeration type

In the sixth step, a new type will be introduced. This time around, it will not be a class type, but an enumeration type. It is the .RoomSize enumeration type used to specify the room sizes. Section 5.2.4 is used to introduce the enumeration type on the type level, while on the instance level, Section 5.3.4 is used to introduce the values of the enumeration.

The *name* of the new enumeration type is .RoomSize, and it has $values = \{\mathsf{SMALL}, \mathsf{MEDIUM}, \mathsf{LARGE}\}$. Please note that there are multiple encodings possible to encode an enumeration type in GROOVE. The encoding using flags is chosen for this enumeration type. The function fob which maps the enumeration values to internal node ids is defined as follows:

$$fob = \{(\mathsf{SMALL}, \text{SmallSize}), (\mathsf{MEDIUM}, MediumSize), (\mathsf{LARGE}, LargeSize)\}$$

The function fid which maps the enumeration values to explicit node identifiers is equal to the definition of fob, so $fid = fob$. The following model is obtained:

Models after step 6

$$
\begin{aligned}
Tm_6 = \quad & \text{combine}(Tm_5, Tm_{Enum}) = \\
& Class = \{\text{.House}, \text{.Room}\} \\
& Enum = \{\text{.RoomSize}\} \\
& UserDataType = \{\} \\
& Field = \{(\text{.House}, \mathsf{name}), (\text{.House}, \mathsf{rooms}), (\text{.Room}, \mathsf{room\_id})\} \\
& \text{FieldSig} = \big\{\big((\text{.House}, \mathsf{name}), (\mathsf{string}, 1..1)\big), \\
& \qquad \big((\text{.House}, \mathsf{rooms}), ([\mathsf{setof}, \text{!.Room}], 0..9)\big), \\
& \qquad \big((\text{.Room}, \mathsf{room\_id}), (\mathsf{string}, 1..1)\big)\big\} \\
& EnumValue = \{(\text{.RoomSize}, \mathsf{SMALL}), (\text{.RoomSize}, \mathsf{MEDIUM}), (\text{.RoomSize}, \mathsf{LARGE})\} \\
& Inh = \{\} \\
& Prop = \big\{\big(\mathsf{containment}, (\text{.House}, \mathsf{rooms})\big)\big\} \\
& Constant = \{\} \\
& \text{ConstType} = \{\}
\end{aligned}
$$

Models after step 6

$Im_6 =$ combine$(Tm_5, Im_{Enum}) =$

$$Object = \{TR, BHP, TRRoom1, TRRoom2, BHPRoomA, BHPRoomB, BHPRoomC\}$$

$$\text{ObjectClass}(ob) = \{(TR, .\mathsf{House}), (BHP, .\mathsf{House}), (TRRoom1, .\mathsf{Room}), (TRRoom2, .\mathsf{Room}), (BHPRoomA, .\mathsf{Room}), (BHPRoomB, .\mathsf{Room}), (BHPRoomC, .\mathsf{Room})\}$$

$$\text{ObjectId} = \{(TR, TR), (BHP, BHP), (TRRoom1, TRRoom1), (TRRoom2, TRRoom2), (BHPRoomA, BHPRoomA), (BHPRoomB, BHPRoomB), (BHPRoomC, BHPRoomC)\}$$

$$\text{FieldValue} = \Big\{\Big((TR, (.\mathsf{House}, \mathsf{name})), [\mathsf{string}, \text{“TwoRem”}]\Big), \Big((BHP, (.\mathsf{House}, \mathsf{name})), [\mathsf{string}, \text{“B.H. Paleis”}]\Big), \Big((TR, (.\mathsf{House}, \mathsf{rooms})), [\mathsf{setof}, \langle [\mathsf{obj}, TRRoom1], [\mathsf{obj}, TRRoom2]\rangle]\Big), \Big((BHP, (.\mathsf{House}, \mathsf{rooms}))\, [\mathsf{setof}, \langle [\mathsf{obj}, BHPRoomA], [\mathsf{obj}, BHPRoomB], (\mathsf{obj}, BHPRoomC)\rangle]\Big), \Big((TRRoom1, (.\mathsf{Room}, \mathsf{room\_id})), [\mathsf{string}, \text{“1”}]\Big), \Big((TRRoom2, (.\mathsf{Room}, \mathsf{room\_id})), [\mathsf{string}, \text{“2”}]\Big), \Big((BHPRoomA, (.\mathsf{Room}, \mathsf{room\_id})), [\mathsf{string}, \text{“A”}]\Big), \Big((BHPRoomB, (.\mathsf{Room}, \mathsf{room\_id})), [\mathsf{string}, \text{“B”}]\Big), \Big((BHPRoomC, (.\mathsf{Room}, \mathsf{room\_id})), [\mathsf{string}, \text{“C”}]\Big)\Big\}$$

$$\text{DefaultValue} = \{\}$$

$TG_6 =$ combine$(TG_5, TG_{EnumFlags}) =$

$$NT = \{\langle \mathsf{House}\rangle, \langle \mathsf{Room}\rangle, \langle \mathsf{RoomSize}\rangle, \mathsf{string}\}$$

$$ET = \{(\langle \mathsf{House}\rangle, \langle \mathsf{name}\rangle, \mathsf{string}), (\langle \mathsf{House}\rangle, \langle \mathsf{rooms}\rangle, \langle \mathsf{Room}\rangle), (\langle \mathsf{Room}\rangle, \langle \mathsf{room\_id}\rangle, \mathsf{string}), (\langle \mathsf{RoomSize}\rangle, \langle \mathsf{SMALL}\rangle, \langle \mathsf{RoomSize}\rangle), (\langle \mathsf{RoomSize}\rangle, \langle \mathsf{MEDIUM}\rangle, \langle \mathsf{RoomSize}\rangle), (\langle \mathsf{RoomSize}\rangle, \langle \mathsf{LARGE}\rangle, \langle \mathsf{RoomSize}\rangle)\}$$

$$\sqsubseteq = \{(\langle \mathsf{House}\rangle, \langle \mathsf{House}\rangle), (\langle \mathsf{Room}\rangle, \langle \mathsf{Room}\rangle), (\langle \mathsf{RoomSize}\rangle, \langle \mathsf{RoomSize}\rangle), (\mathsf{string}, \mathsf{string})\}$$

$$abs = \{\}$$

$$\text{mult} = \Big\{\Big((\langle \mathsf{House}\rangle, \langle \mathsf{name}\rangle, \mathsf{string}), (0..*, 1..1)\Big), \Big((\langle \mathsf{House}\rangle, \langle \mathsf{rooms}\rangle, \langle \mathsf{Room}\rangle), (0..1, 0..9)\Big), \Big((\langle \mathsf{Room}\rangle, \langle \mathsf{room\_id}\rangle, \mathsf{string}), (0..*, 1..1)\Big), \Big((\langle \mathsf{RoomSize}\rangle, \langle \mathsf{SMALL}\rangle, \langle \mathsf{RoomSize}\rangle), (0..1, 0..1)\Big), \Big((\langle \mathsf{RoomSize}\rangle, \langle \mathsf{MEDIUM}\rangle, \langle \mathsf{RoomSize}\rangle), (0..1, 0..1)\Big), \Big((\langle \mathsf{RoomSize}\rangle, \langle \mathsf{LARGE}\rangle, \langle \mathsf{RoomSize}\rangle), (0..1, 0..1)\Big)\Big\}$$

$$contains = \{(\langle \mathsf{House}\rangle, \langle \mathsf{rooms}\rangle, \langle \mathsf{Room}\rangle)\}$$

Models after step 6

$$
\begin{aligned}
IG_6 = \quad & \text{combine}(TG_5, IG_{EnumFlags}) = \\
& N = \{TR, BHP, TRRoom1, TRRoom2, BHPRoomA, BHPRoomB, BHPRoomC, \\
& \quad SmallSize, MediumSize, LargeSize, \text{“TwoRem”}, \text{“B.H. Paleis”}, \text{“1”}, \text{“2”}, \text{“A”}, \text{“B”}, \\
& \quad \text{“C”}\} \\
& E = \Big\{\Big(TR, \big(\langle \mathsf{House}\rangle, \langle \mathsf{name}\rangle, \mathsf{string}\big), \text{“TwoRem”}\Big), \\
& \quad \Big(BHP, \big(\langle \mathsf{House}\rangle, \langle \mathsf{name}\rangle, \mathsf{string}\big), \text{“B.H. Paleis”}\Big), \\
& \quad \Big(TR, \big(\langle \mathsf{House}\rangle, \langle \mathsf{rooms}\rangle, \langle \mathsf{Room}\rangle\big), TRRoom1\Big), \\
& \quad \Big(TR, \big(\langle \mathsf{House}\rangle, \langle \mathsf{rooms}\rangle, \langle \mathsf{Room}\rangle\big), TRRoom2\Big), \\
& \quad \Big(BHP, \big(\langle \mathsf{House}\rangle, \langle \mathsf{rooms}\rangle, \langle \mathsf{Room}\rangle\big), BHPRoomA\Big), \\
& \quad \Big(BHP, \big(\langle \mathsf{House}\rangle, \langle \mathsf{rooms}\rangle, \langle \mathsf{Room}\rangle\big), BHPRoomB\Big), \\
& \quad \Big(BHP, \big(\langle \mathsf{House}\rangle, \langle \mathsf{rooms}\rangle, \langle \mathsf{Room}\rangle\big), BHPRoomC\Big), \\
& \quad \Big(TRRoom1, \big(\langle \mathsf{Room}\rangle, \langle \mathsf{room\_id}\rangle, \mathsf{string}\big), \text{“1”}\Big), \\
& \quad \Big(TRRoom2, \big(\langle \mathsf{Room}\rangle, \langle \mathsf{room\_id}\rangle, \mathsf{string}\big), \text{“2”}\Big), \\
& \quad \Big(BHPRoomA, \big(\langle \mathsf{Room}\rangle, \langle \mathsf{room\_id}\rangle, \mathsf{string}\big), \text{“A”}\Big), \\
& \quad \Big(BHPRoomB, \big(\langle \mathsf{Room}\rangle, \langle \mathsf{room\_id}\rangle, \mathsf{string}\big), \text{“B”}\Big), \\
& \quad \Big(BHPRoomC, \big(\langle \mathsf{Room}\rangle, \langle \mathsf{room\_id}\rangle, \mathsf{string}\big), \text{“C”}\Big), \\
& \quad \Big(SmallSize, \big(\langle \mathsf{RoomSize}\rangle, \langle \mathsf{SMALL}\rangle, \langle \mathsf{RoomSize}\rangle\big), SmallSize\Big), \\
& \quad \Big(MediumSize, \big(\langle \mathsf{RoomSize}\rangle, \langle \mathsf{MEDIUM}\rangle, \langle \mathsf{RoomSize}\rangle\big), MediumSize\Big), \\
& \quad \Big(LargeSize, \big(\langle \mathsf{RoomSize}\rangle, \langle \mathsf{LARGE}\rangle, \langle \mathsf{RoomSize}\rangle\big), LargeSize\Big)\Big\} \\
& \text{ident} = \{(TR, TR), (BHP, BHP), (TRRoom1, TRRoom1), (TRRoom2, TRRoom2), \\
& \quad (BHPRoomA, BHPRoomA), (BHPRoomB, BHPRoomB), \\
& \quad (BHPRoomC, BHPRoomC), (SmallSize, SmallSize), \\
& \quad (MediumSize, MediumSize), (LargeSize, LargeSize)\} \\
& \text{type}_n = \{(TR, \langle \mathsf{House}\rangle), (BHP, \langle \mathsf{House}\rangle), (TRRoom1, \langle \mathsf{Room}\rangle), (TRRoom2, \langle \mathsf{Room}\rangle), \\
& \quad (BHPRoomA, \langle \mathsf{Room}\rangle), (BHPRoomB, \langle \mathsf{Room}\rangle), (BHPRoomC, \langle \mathsf{Room}\rangle), \\
& \quad (SmallSize, \langle \mathsf{RoomSize}\rangle), (MediumSize, \langle \mathsf{RoomSize}\rangle), (LargeSize, \langle \mathsf{RoomSize}\rangle), \\
& \quad (\text{“TwoRem”}, \mathsf{string}), (\text{“B.H. Paleis”}, \mathsf{string}), (\text{“1”}, \mathsf{string}), (\text{“2”}, \mathsf{string}), (\text{“A”}, \mathsf{string}), \\
& \quad (\text{“B”}, \mathsf{string}), (\text{“C”}, \mathsf{string})\}
\end{aligned}
$$

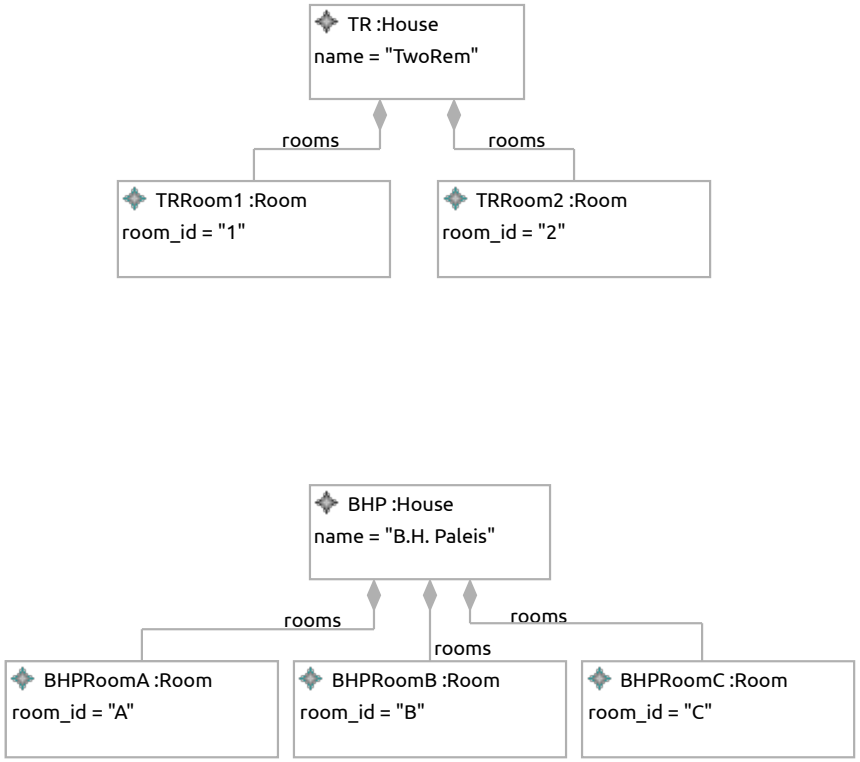
$f_6(Im_6) = f_5(Im_5) \sqcup f_{EnumFlags}(Im_{Enum})$ (Definition 4.4.27)
$f'_6(IG_6) = f'_5(IG_5) \sqcup f'_{EnumFlags}(IG_{EnumFlags})$ (Definition 4.4.32)

A visual representation of Tm_6 and Im_6 can be found in Figure 6.12. Similarly, a visual representation of TG_6 and IG_6 can be found in Figure 6.13. Please note that because of the definitions of $f_6(Im_6)$ and $f'_6(IG_6)$, we have that $f_6(Im_6) = IG_6$ and $f'_6(IG_6) = Im_6$. Furthermore, $f_6(Im_6)$ and $f'_6(IG_6)$ are valid mapping functions themselves, such that they can be combined with another mapping function in the next step.

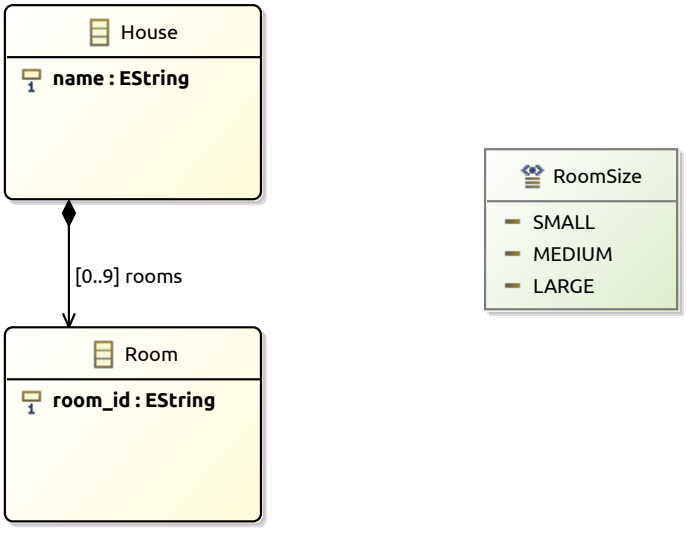
The introduction of the enumeration type shows the encoding of an enumeration type as GROOVE flags in practice. It should be noted that although the instance model has not changed, the instance graph obtained instances of the enumeration values. These instances will be referenced by an enumeration field in the next step.

6.2.7 Room sizes

In the seventh step, a field referencing the new enumeration type is introduced. Section 5.2.7 is used to introduce the enumeration field on the type level, while on the instance level, Section 5.3.7 is used to introduce the values.

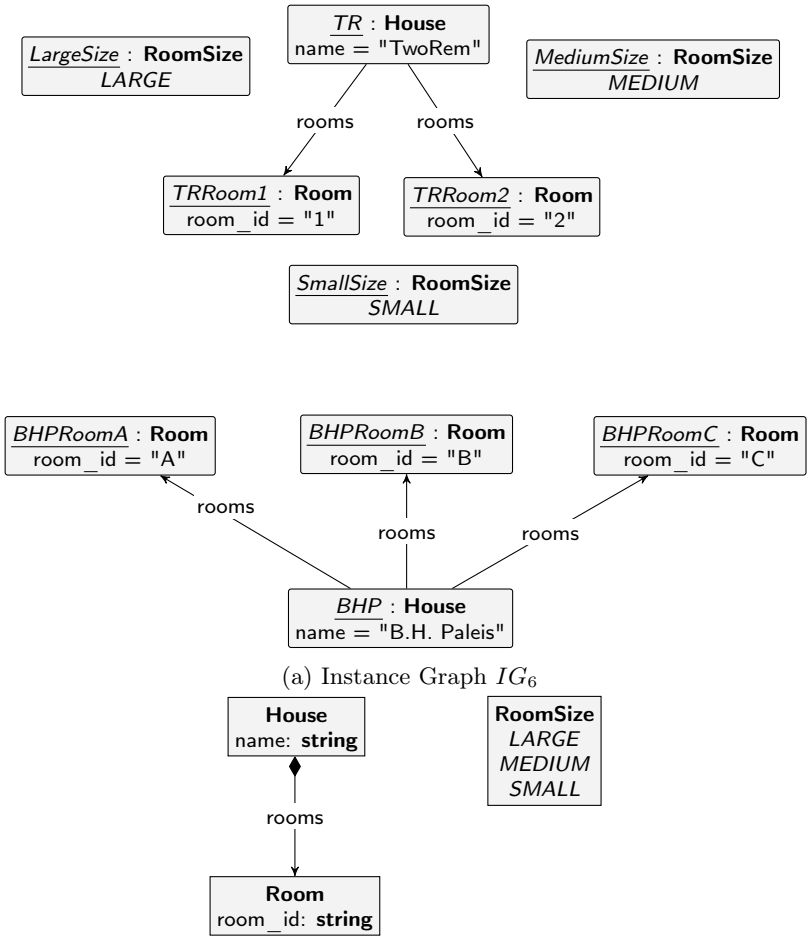


(a) Instance Model Im_6



(b) Type Model Tm_6

Figure 6.12: The Ecore model after step 6



(a) Instance Graph IG_6

(b) Type Graph TG_6

Figure 6.13: The GROOVE graphs after step 6

The *classtype* of the new field is .Room, as the field will be defined for rooms. The *name* of the new field is room_size and the *enumid* is .RoomSize. Furthermore, the set of *enumvalues* is equal to the set of values for the .RoomSize enumeration type, so $enumvalues = \{\mathsf{SMALL}, \mathsf{MEDIUM}, \mathsf{LARGE}\}$. Then, *enumids* returns for each enumeration value the corresponding node identifier used in the GROOVE graph, while *enumob* lists the corresponding internal node id.

The set of objects of which the value is set is equal to all room objects, so $objects = \{TRRoom1, TRRoom2, BHPRoomA,$ The *values* function is defined as follows:

$$values = \{(TRRoom1, \mathsf{LARGE}), (TRRoom2, \mathsf{MEDIUM}), (BHPRoomA, \mathsf{SMALL}),\\ (BHPRoomB, \mathsf{SMALL}), (BHPRoomC, \mathsf{SMALL})\}$$

The following model is obtained:

Models after step 7

$$\begin{aligned}
Tm_7 = \quad & \text{combine}(Tm_6, Tm_{EnumField}) = \\
& Class = \{\text{.House}, \text{.Room}\} \\
& Enum = \{\text{.RoomSize}\} \\
& UserDataType = \{\} \\
& Field = \{(\text{.House}, \mathsf{name}), (\text{.House}, \mathsf{rooms}), (\text{.Room}, \mathsf{room_id}),\\
& \qquad (\text{.Room}, \mathsf{room_size})\} \\
& \text{FieldSig} = \big\{\big((\text{.House}, \mathsf{name}), (\mathsf{string}, 1..1)\big),\\
& \qquad \big((\text{.House}, \mathsf{rooms}), ([\mathsf{setof}, \text{!.Room}], 0..9)\big),\\
& \qquad \big((\text{.Room}, \mathsf{room_id}), (\mathsf{string}, 1..1)\big),\\
& \qquad \big((\text{.Room}, \mathsf{room_size}), (\text{.RoomSize}, 1..1)\big)\big\} \\
& EnumValue = \{(\text{.RoomSize}, \mathsf{SMALL}), (\text{.RoomSize}, \mathsf{MEDIUM}), (\text{.RoomSize}, \mathsf{LARGE})\} \\
& Inh = \{\} \\
& Prop = \big\{\big(\mathsf{containment}, (\text{.House}, \mathsf{rooms})\big)\big\} \\
& Constant = \{\} \\
& \text{ConstType} = \{\}
\end{aligned}$$

Models after step 7

$Im_7 =$ combine$(Tm_6, Im_{EnumField}) =$

$$Object = \{TR, BHP, TRRoom1, TRRoom2, BHPRoomA, BHPRoomB, BHPRoomC\}$$

$$\text{ObjectClass}(ob) = \{(TR, .\mathsf{House}), (BHP, .\mathsf{House}), (TRRoom1, .\mathsf{Room}), (TRRoom2, .\mathsf{Room}), (BHPRoomA, .\mathsf{Room}), (BHPRoomB, .\mathsf{Room}), (BHPRoomC, .\mathsf{Room})\}$$

$$\text{ObjectId} = \{(TR, TR), (BHP, BHP), (TRRoom1, TRRoom1), (TRRoom2, TRRoom2), (BHPRoomA, BHPRoomA), (BHPRoomB, BHPRoomB), (BHPRoomC, BHPRoomC)\}$$

$$\text{FieldValue} = \Big\{\Big((TR, (.\mathsf{House}, \mathsf{name})), [\mathsf{string}, \text{“TwoRem”}]\Big),$$
$$\Big((BHP, (.\mathsf{House}, \mathsf{name})), [\mathsf{string}, \text{“B.H. Paleis”}]\Big),$$
$$\Big((TR, (.\mathsf{House}, \mathsf{rooms})), [\mathsf{setof}, \langle [\mathsf{obj}, TRRoom1], [\mathsf{obj}, TRRoom2]\rangle]\Big),$$
$$\Big((BHP, (.\mathsf{House}, \mathsf{rooms}))\, [\mathsf{setof}, \langle [\mathsf{obj}, BHPRoomA], [\mathsf{obj}, BHPRoomB], (\mathsf{obj}, BHPRoomC)\rangle]\Big),$$
$$\Big((TRRoom1, (.\mathsf{Room}, \mathsf{room\_id})), [\mathsf{string}, \text{“1”}]\Big),$$
$$\Big((TRRoom2, (.\mathsf{Room}, \mathsf{room\_id})), [\mathsf{string}, \text{“2”}]\Big),$$
$$\Big((BHPRoomA, (.\mathsf{Room}, \mathsf{room\_id})), [\mathsf{string}, \text{“A”}]\Big),$$
$$\Big((BHPRoomB, (.\mathsf{Room}, \mathsf{room\_id})), [\mathsf{string}, \text{“B”}]\Big),$$
$$\Big((BHPRoomC, (.\mathsf{Room}, \mathsf{room\_id})), [\mathsf{string}, \text{“C”}]\Big),$$
$$\Big((TRRoom1, (.\mathsf{Room}, \mathsf{room\_size})), [\mathsf{enum}, (.\mathsf{RoomSize}, \mathsf{LARGE})]\Big),$$
$$\Big((TRRoom2, (.\mathsf{Room}, \mathsf{room\_size})), [\mathsf{enum}, (.\mathsf{RoomSize}, \mathsf{MEDIUM})]\Big),$$
$$\Big((BHPRoomA, (.\mathsf{Room}, \mathsf{room\_size})), [\mathsf{enum}, (.\mathsf{RoomSize}, \mathsf{SMALL})]\Big),$$
$$\Big((BHPRoomB, (.\mathsf{Room}, \mathsf{room\_size})), [\mathsf{enum}, (.\mathsf{RoomSize}, \mathsf{SMALL})]\Big),$$
$$\Big((BHPRoomC, (.\mathsf{Room}, \mathsf{room\_size})), [\mathsf{enum}, (.\mathsf{RoomSize}, \mathsf{SMALL})]\Big)\Big\}$$

$$\text{DefaultValue} = \{\}$$

Models after step 7

$$
\begin{aligned}
TG_7 = \quad & \text{combine}(TG_6, TG_{EnumFieldFlags}) = \\
& NT = \{\langle\mathsf{House}\rangle, \langle\mathsf{Room}\rangle, \langle\mathsf{RoomSize}\rangle, \mathsf{string}\} \\
& ET = \big\{\big(\langle\mathsf{House}\rangle, \langle\mathsf{name}\rangle, \mathsf{string}\big), \big(\langle\mathsf{House}\rangle, \langle\mathsf{rooms}\rangle, \langle\mathsf{Room}\rangle\big), \\
& \quad \big(\langle\mathsf{Room}\rangle, \langle\mathsf{room\_id}\rangle, \mathsf{string}\big), \big(\langle\mathsf{Room}\rangle, \langle\mathsf{room\_size}\rangle, \langle\mathsf{RoomSize}\rangle\big), \\
& \quad \big(\langle\mathsf{RoomSize}\rangle, \langle\mathsf{SMALL}\rangle, \langle\mathsf{RoomSize}\rangle\big), \big(\langle\mathsf{RoomSize}\rangle, \langle\mathsf{MEDIUM}\rangle, \langle\mathsf{RoomSize}\rangle\big), \\
& \quad \big(\langle\mathsf{RoomSize}\rangle, \langle\mathsf{LARGE}\rangle, \langle\mathsf{RoomSize}\rangle\big)\big\} \\
& \sqsubseteq = \big\{\big(\langle\mathsf{House}\rangle, \langle\mathsf{House}\rangle\big), \big(\langle\mathsf{Room}\rangle, \langle\mathsf{Room}\rangle\big), \big(\langle\mathsf{RoomSize}\rangle, \langle\mathsf{RoomSize}\rangle\big), \\
& \quad \big(\mathsf{string}, \mathsf{string}\big)\big\} \\
& abs = \{\} \\
& \text{mult} = \Big\{\Big(\big(\langle\mathsf{House}\rangle, \langle\mathsf{name}\rangle, \mathsf{string}\big), \big(0..*, 1..1\big)\Big), \\
& \quad \Big(\big(\langle\mathsf{House}\rangle, \langle\mathsf{rooms}\rangle, \langle\mathsf{Room}\rangle\big), \big(0..1, 0..9\big)\Big), \\
& \quad \Big(\big(\langle\mathsf{Room}\rangle, \langle\mathsf{room\_id}\rangle, \mathsf{string}\big), \big(0..*, 1..1\big)\Big), \\
& \quad \Big(\big(\langle\mathsf{Room}\rangle, \langle\mathsf{room\_size}\rangle, \langle\mathsf{RoomSize}\rangle\big), \big(0..*, 1..1\big)\Big), \\
& \quad \Big(\big(\langle\mathsf{RoomSize}\rangle, \langle\mathsf{SMALL}\rangle, \langle\mathsf{RoomSize}\rangle\big), \big(0..1, 0..1\big)\Big), \\
& \quad \Big(\big(\langle\mathsf{RoomSize}\rangle, \langle\mathsf{MEDIUM}\rangle, \langle\mathsf{RoomSize}\rangle\big), \big(0..1, 0..1\big)\Big), \\
& \quad \Big(\big(\langle\mathsf{RoomSize}\rangle, \langle\mathsf{LARGE}\rangle, \langle\mathsf{RoomSize}\rangle\big), \big(0..1, 0..1\big)\Big)\Big\} \\
& contains = \big\{\big(\langle\mathsf{House}\rangle, \langle\mathsf{rooms}\rangle, \langle\mathsf{Room}\rangle\big)\big\}
\end{aligned}
$$

Models after step 7

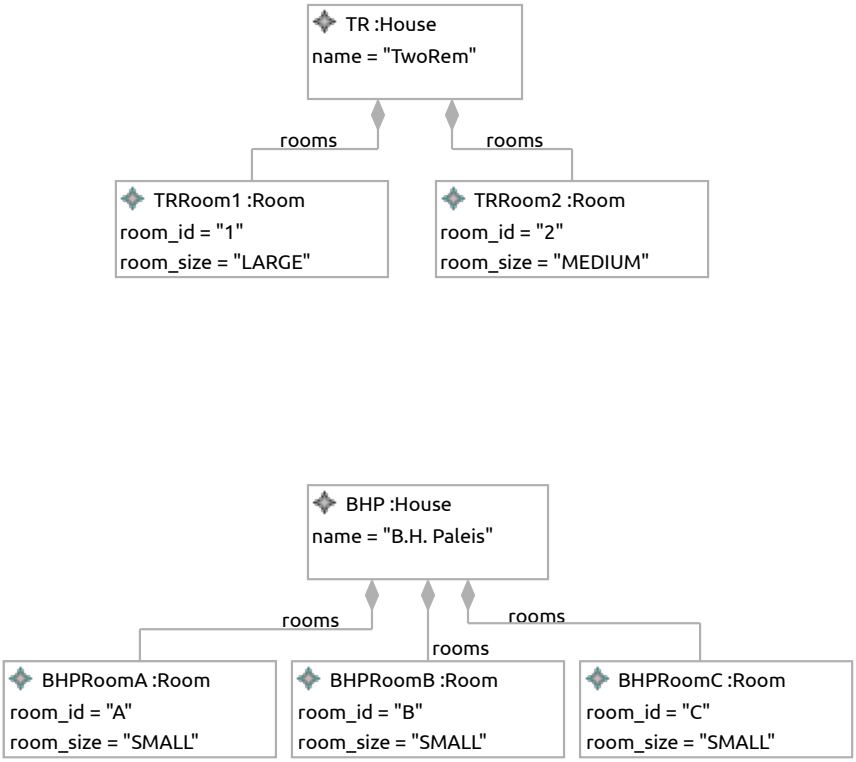
$$
\begin{aligned}
IG_7 = \quad & \text{combine}(TG_6, IG_{EnumFieldFlags}) = \\
& N = \{TR, BHP, TRRoom1, TRRoom2, BHPRoomA, BHPRoomB, BHPRoomC, \\
& \quad SmallSize, MediumSize, LargeSize, \text{“TwoRem”}, \text{“B.H. Paleis”}, \text{“1”}, \text{“2”}, \text{“A”}, \text{“B”}, \\
& \quad \text{“C”}\} \\
& E = \Big\{\Big(TR, \big(\langle\mathsf{House}\rangle, \langle\mathsf{name}\rangle, \mathsf{string}\big), \text{“TwoRem”}\Big), \\
& \quad \Big(BHP, \big(\langle\mathsf{House}\rangle, \langle\mathsf{name}\rangle, \mathsf{string}\big), \text{“B.H. Paleis”}\Big), \\
& \quad \Big(TR, \big(\langle\mathsf{House}\rangle, \langle\mathsf{rooms}\rangle, \langle\mathsf{Room}\rangle\big), TRRoom1\Big), \\
& \quad \Big(TR, \big(\langle\mathsf{House}\rangle, \langle\mathsf{rooms}\rangle, \langle\mathsf{Room}\rangle\big), TRRoom2\Big), \\
& \quad \Big(BHP, \big(\langle\mathsf{House}\rangle, \langle\mathsf{rooms}\rangle, \langle\mathsf{Room}\rangle\big), BHPRoomA\Big), \\
& \quad \Big(BHP, \big(\langle\mathsf{House}\rangle, \langle\mathsf{rooms}\rangle, \langle\mathsf{Room}\rangle\big), BHPRoomB\Big), \\
& \quad \Big(BHP, \big(\langle\mathsf{House}\rangle, \langle\mathsf{rooms}\rangle, \langle\mathsf{Room}\rangle\big), BHPRoomC\Big), \\
& \quad \Big(TRRoom1, \big(\langle\mathsf{Room}\rangle, \langle\mathsf{room\_id}\rangle, \mathsf{string}\big), \text{“1”}\Big), \\
& \quad \Big(TRRoom2, \big(\langle\mathsf{Room}\rangle, \langle\mathsf{room\_id}\rangle, \mathsf{string}\big), \text{“2”}\Big), \\
& \quad \Big(BHPRoomA, \big(\langle\mathsf{Room}\rangle, \langle\mathsf{room\_id}\rangle, \mathsf{string}\big), \text{“A”}\Big), \\
& \quad \Big(BHPRoomB, \big(\langle\mathsf{Room}\rangle, \langle\mathsf{room\_id}\rangle, \mathsf{string}\big), \text{“B”}\Big), \\
& \quad \Big(BHPRoomC, \big(\langle\mathsf{Room}\rangle, \langle\mathsf{room\_id}\rangle, \mathsf{string}\big), \text{“C”}\Big), \\
& \quad \Big(TRRoom1, \big(\langle\mathsf{Room}\rangle, \langle\mathsf{room\_size}\rangle, \langle\mathsf{RoomSize}\rangle\big), LargeSize\Big), \\
& \quad \Big(TRRoom2, \big(\langle\mathsf{Room}\rangle, \langle\mathsf{room\_size}\rangle, \langle\mathsf{RoomSize}\rangle\big), MediumSize\Big), \\
& \quad \Big(BHPRoomA, \big(\langle\mathsf{Room}\rangle, \langle\mathsf{room\_size}\rangle, \langle\mathsf{RoomSize}\rangle\big), SmallSize\Big), \\
& \quad \Big(BHPRoomB, \big(\langle\mathsf{Room}\rangle, \langle\mathsf{room\_size}\rangle, \langle\mathsf{RoomSize}\rangle\big), SmallSize\Big), \\
& \quad \Big(BHPRoomC, \big(\langle\mathsf{Room}\rangle, \langle\mathsf{room\_size}\rangle, \langle\mathsf{RoomSize}\rangle\big), SmallSize\Big), \\
& \quad \Big(SmallSize, \big(\langle\mathsf{RoomSize}\rangle, \langle\mathsf{SMALL}\rangle, \langle\mathsf{RoomSize}\rangle\big), SmallSize\Big), \\
& \quad \Big(MediumSize, \big(\langle\mathsf{RoomSize}\rangle, \langle\mathsf{MEDIUM}\rangle, \langle\mathsf{RoomSize}\rangle\big), MediumSize\Big), \\
& \quad \Big(LargeSize, \big(\langle\mathsf{RoomSize}\rangle, \langle\mathsf{LARGE}\rangle, \langle\mathsf{RoomSize}\rangle\big), LargeSize\Big)\Big\} \\
& \text{ident} = \{(TR, TR), (BHP, BHP), (TRRoom1, TRRoom1), (TRRoom2, TRRoom2), \\
& \quad (BHPRoomA, BHPRoomA), (BHPRoomB, BHPRoomB), \\
& \quad (BHPRoomC, BHPRoomC), (SmallSize, SmallSize), \\
& \quad (MediumSize, MediumSize), (LargeSize, LargeSize)\} \\
& \text{type}_n = \{(TR, \langle\mathsf{House}\rangle), (BHP, \langle\mathsf{House}\rangle), (TRRoom1, \langle\mathsf{Room}\rangle), (TRRoom2, \langle\mathsf{Room}\rangle), \\
& \quad (BHPRoomA, \langle\mathsf{Room}\rangle), (BHPRoomB, \langle\mathsf{Room}\rangle), (BHPRoomC, \langle\mathsf{Room}\rangle), \\
& \quad (SmallSize, \langle\mathsf{RoomSize}\rangle), (MediumSize, \langle\mathsf{RoomSize}\rangle), (LargeSize, \langle\mathsf{RoomSize}\rangle), \\
& \quad (\text{“TwoRem”}, \mathsf{string}), (\text{“B.H. Paleis”}, \mathsf{string}), (\text{“1”}, \mathsf{string}), (\text{“2”}, \mathsf{string}), (\text{“A”}, \mathsf{string}), \\
& \quad (\text{“B”}, \mathsf{string}), (\text{“C”}, \mathsf{string})\}
\end{aligned}
$$

$f_7(Im_7) = f_6(Im_6) \sqcup f_{EnumFieldFlags}(Im_{EnumField})$ (Definition 4.4.27)

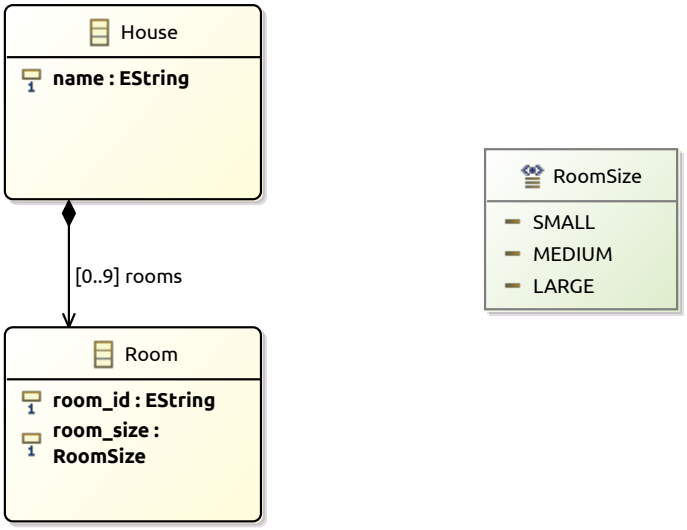
$f'_7(IG_7) = f'_6(IG_6) \sqcup f'_{EnumFieldFlags}(IG_{EnumFieldFlags})$ (Definition 4.4.32)

A visual representation of Tm_7 and Im_7 can be found in Figure 6.14. Similarly, a visual representation of TG_7 and IG_7 can be found in Figure 6.15. Please note that because of the definitions of $f_7(Im_7)$ and $f'_7(IG_7)$, we have that $f_7(Im_7) = IG_7$ and $f'_7(IG_7) = Im_7$. Furthermore, $f_7(Im_7)$ and $f'_7(IG_7)$ are valid mapping functions themselves, such that they can be combined with another mapping function in the next step.

On the instance model level, no surprising elements are introduced. An enumeration field acts like most

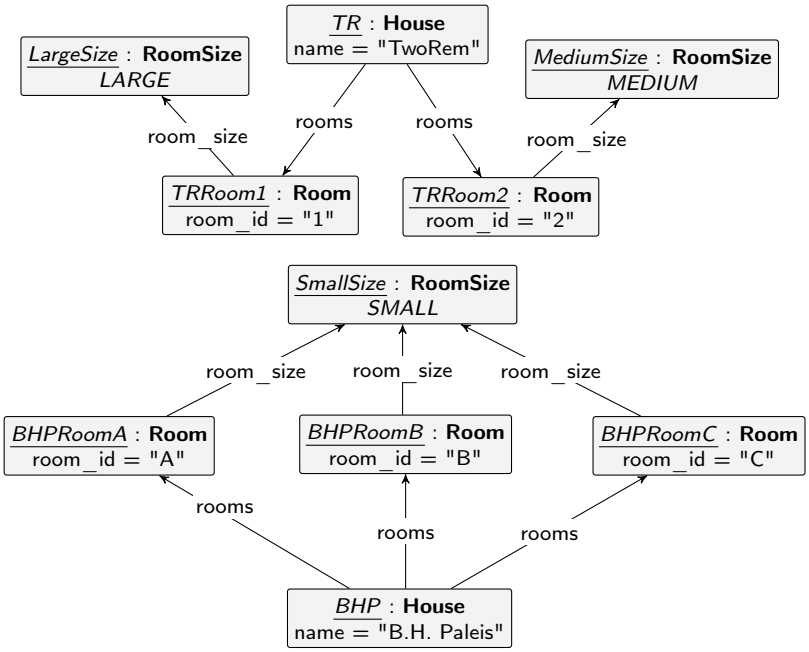


(a) Instance Model Im_7

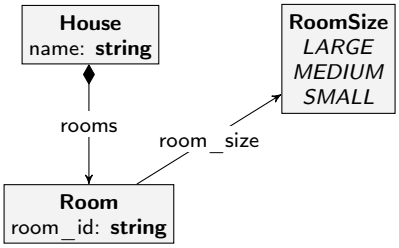


(b) Type Model Tm_7

Figure 6.14: The Ecore model after step 7



(a) Instance Graph IG_7



(b) Type Graph TG_7

Figure 6.15: The GROOVE graphs after step 7

other attributes within Ecore. However, the visualisation shows how enumeration values are referenced by their instance nodes in the instance graph. The encoding of the field makes use of the presented encoding to emulate enumeration types within GROOVE.

6.2.8 Tenants

The eighth step introduces another class type. This time, the .Tenant type is introduced to enrich the model and graphs even further. Section 5.2.1 is used to introduce the class type, while on the instance level, Section 5.3.1 is used to introduce the house objects.

The *name* of the new tenant type is .Tenant. Furthermore, 5 tenant objects are introduced, $objects = \{Tenant1, Tenant2, Tenant3, Tenant4, Tenant5\}$. Furthermore, we assume that the object identifiers are equal to the internal node id, so $fid(Tenant1) = Tenant1$ and $fid(Tenant2) = Tenant2$, etc. The following model is obtained:

Models after step 8

$$
\begin{aligned}
Tm_8 = \ & \text{combine}(Tm_7, Tm_{Class}) = \\
& Class = \{\text{.House}, \text{.Room}, \text{.Tenant}\} \\
& Enum = \{\text{.RoomSize}\} \\
& UserDataType = \{\} \\
& Field = \{(\text{.House}, \text{name}), (\text{.House}, \text{rooms}), (\text{.Room}, \text{room\_id}), \\
& \qquad (\text{.Room}, \text{room\_size})\} \\
& \text{FieldSig} = \big\{\big((\text{.House}, \text{name}), (\text{string}, 1..1)\big), \\
& \qquad \big((\text{.House}, \text{rooms}), ([\text{setof}, \text{!.Room}], 0..9)\big), \\
& \qquad \big((\text{.Room}, \text{room\_id}), (\text{string}, 1..1)\big), \\
& \qquad \big((\text{.Room}, \text{room\_size}), (\text{.RoomSize}, 1..1)\big)\big\} \\
& EnumValue = \{(\text{.RoomSize}, \text{SMALL}), (\text{.RoomSize}, \text{MEDIUM}), (\text{.RoomSize}, \text{LARGE})\} \\
& Inh = \{\} \\
& Prop = \big\{\big(\text{containment}, (\text{.House}, \text{rooms})\big)\big\} \\
& Constant = \{\} \\
& \text{ConstType} = \{\}
\end{aligned}
$$

Models after step 8

$$
\begin{aligned}
Im_8 = \quad & \text{combine}(Tm_7, Im_{Class}) = \\
& Object = \{TR, BHP, TRRoom1, TRRoom2, BHPRoomA, BHPRoomB, \\
& \quad BHPRoomC, Tenant1, Tenant2, Tenant3, Tenant4, Tenant5\} \\
& \text{ObjectClass}(ob) = \{(TR, .\mathsf{House}), (BHP, .\mathsf{House}), (TRRoom1, .\mathsf{Room}), \\
& \quad (TRRoom2, .\mathsf{Room}), (BHPRoomA, .\mathsf{Room}), (BHPRoomB, .\mathsf{Room}), \\
& \quad (BHPRoomC, .\mathsf{Room}, (Tenant1, .\mathsf{Tenant}), (Tenant2, .\mathsf{Tenant}), \\
& \quad (Tenant3, .\mathsf{Tenant}), (Tenant4, .\mathsf{Tenant}), (Tenant5, .\mathsf{Tenant})\} \\
& \text{ObjectId} = \{(TR, TR), (BHP, BHP), (TRRoom1, TRRoom1), \\
& \quad (TRRoom2, TRRoom2), (BHPRoomA, BHPRoomA), \\
& \quad (BHPRoomB, BHPRoomB), (BHPRoomC, BHPRoomC), \\
& \quad (Tenant1, Tenant1), (Tenant2, Tenant2), (Tenant3, Tenant3), \\
& \quad (Tenant4, Tenant4), (Tenant5, Tenant5)\} \\
& \text{FieldValue} = \Big\{\Big((TR, (.\mathsf{House}, \mathsf{name})), [\mathsf{string}, \text{“TwoRem”}]\Big), \\
& \quad \Big((BHP, (.\mathsf{House}, \mathsf{name})), [\mathsf{string}, \text{“B.H. Paleis”}]\Big), \\
& \quad \Big((TR, (.\mathsf{House}, \mathsf{rooms})), [\mathsf{setof}, \langle [\mathsf{obj}, TRRoom1], \\
& \qquad [\mathsf{obj}, TRRoom2]\rangle]\Big), \\
& \quad \Big((BHP, (.\mathsf{House}, \mathsf{rooms}))\, [\mathsf{setof}, \langle [\mathsf{obj}, BHPRoomA], \\
& \qquad [\mathsf{obj}, BHPRoomB], (\mathsf{obj}, BHPRoomC)\rangle]\Big), \\
& \quad \Big((TRRoom1, (.\mathsf{Room}, \mathsf{room\_id})), [\mathsf{string}, \text{“1”}]\Big), \\
& \quad \Big((TRRoom2, (.\mathsf{Room}, \mathsf{room\_id})), [\mathsf{string}, \text{“2”}]\Big), \\
& \quad \Big((BHPRoomA, (.\mathsf{Room}, \mathsf{room\_id})), [\mathsf{string}, \text{“A”}]\Big), \\
& \quad \Big((BHPRoomB, (.\mathsf{Room}, \mathsf{room\_id})), [\mathsf{string}, \text{“B”}]\Big), \\
& \quad \Big((BHPRoomC, (.\mathsf{Room}, \mathsf{room\_id})), [\mathsf{string}, \text{“C”}]\Big), \\
& \quad \Big((TRRoom1, (.\mathsf{Room}, \mathsf{room\_size})), [\mathsf{enum}, (.\mathsf{RoomSize}, \mathsf{LARGE})]\Big), \\
& \quad \Big((TRRoom2, (.\mathsf{Room}, \mathsf{room\_size})), [\mathsf{enum}, (.\mathsf{RoomSize}, \mathsf{MEDIUM})]\Big), \\
& \quad \Big((BHPRoomA, (.\mathsf{Room}, \mathsf{room\_size})), [\mathsf{enum}, (.\mathsf{RoomSize}, \mathsf{SMALL})]\Big), \\
& \quad \Big((BHPRoomB, (.\mathsf{Room}, \mathsf{room\_size})), [\mathsf{enum}, (.\mathsf{RoomSize}, \mathsf{SMALL})]\Big), \\
& \quad \Big((BHPRoomC, (.\mathsf{Room}, \mathsf{room\_size})), [\mathsf{enum}, (.\mathsf{RoomSize}, \mathsf{SMALL})]\Big)\Big\} \\
& \text{DefaultValue} = \{\}
\end{aligned}
$$

Models after step 8

$$
\begin{aligned}
TG_8 = \quad & \text{combine}(TG_7, TG_{Class}) = \\
& NT = \{\langle\mathsf{House}\rangle, \langle\mathsf{Room}\rangle, \langle\mathsf{Tenant}\rangle, \langle\mathsf{RoomSize}\rangle, \mathsf{string}\} \\
& ET = \big\{\big(\langle\mathsf{House}\rangle, \langle\mathsf{name}\rangle, \mathsf{string}\big), \big(\langle\mathsf{House}\rangle, \langle\mathsf{rooms}\rangle, \langle\mathsf{Room}\rangle\big), \\
& \quad \big(\langle\mathsf{Room}\rangle, \langle\mathsf{room\_id}\rangle, \mathsf{string}\big), \big(\langle\mathsf{Room}\rangle, \langle\mathsf{room\_size}\rangle, \langle\mathsf{RoomSize}\rangle\big), \\
& \quad \big(\langle\mathsf{RoomSize}\rangle, \langle\mathsf{SMALL}\rangle, \langle\mathsf{RoomSize}\rangle\big), \big(\langle\mathsf{RoomSize}\rangle, \langle\mathsf{MEDIUM}\rangle, \langle\mathsf{RoomSize}\rangle\big), \\
& \quad \big(\langle\mathsf{RoomSize}\rangle, \langle\mathsf{LARGE}\rangle, \langle\mathsf{RoomSize}\rangle\big)\big\} \\
& \sqsubseteq = \big\{\big(\langle\mathsf{House}\rangle, \langle\mathsf{House}\rangle\big), \big(\langle\mathsf{Room}\rangle, \langle\mathsf{Room}\rangle\big), \big(\langle\mathsf{Tenant}\rangle, \langle\mathsf{Tenant}\rangle\big), \\
& \quad \big(\langle\mathsf{RoomSize}\rangle, \langle\mathsf{RoomSize}\rangle\big), \big(\mathsf{string}, \mathsf{string}\big)\big\} \\
& abs = \{\} \\
& \text{mult} = \Big\{\Big(\big(\langle\mathsf{House}\rangle, \langle\mathsf{name}\rangle, \mathsf{string}\big), \big(0..*, 1..1\big)\Big), \\
& \quad \Big(\big(\langle\mathsf{House}\rangle, \langle\mathsf{rooms}\rangle, \langle\mathsf{Room}\rangle\big), \big(0..1, 0..9\big)\Big), \\
& \quad \Big(\big(\langle\mathsf{Room}\rangle, \langle\mathsf{room\_id}\rangle, \mathsf{string}\big), \big(0..*, 1..1\big)\Big), \\
& \quad \Big(\big(\langle\mathsf{Room}\rangle, \langle\mathsf{room\_size}\rangle, \langle\mathsf{RoomSize}\rangle\big), \big(0..*, 1..1\big)\Big), \\
& \quad \Big(\big(\langle\mathsf{RoomSize}\rangle, \langle\mathsf{SMALL}\rangle, \langle\mathsf{RoomSize}\rangle\big), \big(0..1, 0..1\big)\Big), \\
& \quad \Big(\big(\langle\mathsf{RoomSize}\rangle, \langle\mathsf{MEDIUM}\rangle, \langle\mathsf{RoomSize}\rangle\big), \big(0..1, 0..1\big)\Big), \\
& \quad \Big(\big(\langle\mathsf{RoomSize}\rangle, \langle\mathsf{LARGE}\rangle, \langle\mathsf{RoomSize}\rangle\big), \big(0..1, 0..1\big)\Big)\Big\} \\
& contains = \big\{\big(\langle\mathsf{House}\rangle, \langle\mathsf{rooms}\rangle, \langle\mathsf{Room}\rangle\big)\big\}
\end{aligned}
$$

Models after step 8

$IG_8 =$ combine$(TG_7, IG_{Class}) =$

$$N = \{TR, BHP, TRRoom1, TRRoom2, BHPRoomA, BHPRoomB, BHPRoomC, Tenant1, Tenant2, Tenant3, Tenant4, Tenant5, SmallSize, MediumSize, LargeSize, \text{“TwoRem”}, \text{“B.H. Paleis”}, \text{“1”}, \text{“2”}, \text{“A”}, \text{“B”}, \text{“C”}\}$$

$$E = \Big\{\Big(TR, \big(\langle\mathsf{House}\rangle, \langle\mathsf{name}\rangle, \mathsf{string}\big), \text{“TwoRem”}\Big), \Big(BHP, \big(\langle\mathsf{House}\rangle, \langle\mathsf{name}\rangle, \mathsf{string}\big), \text{“B.H. Paleis”}\Big), \Big(TR, \big(\langle\mathsf{House}\rangle, \langle\mathsf{rooms}\rangle, \langle\mathsf{Room}\rangle\big), TRRoom1\Big), \Big(TR, \big(\langle\mathsf{House}\rangle, \langle\mathsf{rooms}\rangle, \langle\mathsf{Room}\rangle\big), TRRoom2\Big), \Big(BHP, \big(\langle\mathsf{House}\rangle, \langle\mathsf{rooms}\rangle, \langle\mathsf{Room}\rangle\big), BHPRoomA\Big), \Big(BHP, \big(\langle\mathsf{House}\rangle, \langle\mathsf{rooms}\rangle, \langle\mathsf{Room}\rangle\big), BHPRoomB\Big), \Big(BHP, \big(\langle\mathsf{House}\rangle, \langle\mathsf{rooms}\rangle, \langle\mathsf{Room}\rangle\big), BHPRoomC\Big), \Big(TRRoom1, \big(\langle\mathsf{Room}\rangle, \langle\mathsf{room\_id}\rangle, \mathsf{string}\big), \text{“1”}\Big), \Big(TRRoom2, \big(\langle\mathsf{Room}\rangle, \langle\mathsf{room\_id}\rangle, \mathsf{string}\big), \text{“2”}\Big), \Big(BHPRoomA, \big(\langle\mathsf{Room}\rangle, \langle\mathsf{room\_id}\rangle, \mathsf{string}\big), \text{“A”}\Big), \Big(BHPRoomB, \big(\langle\mathsf{Room}\rangle, \langle\mathsf{room\_id}\rangle, \mathsf{string}\big), \text{“B”}\Big), \Big(BHPRoomC, \big(\langle\mathsf{Room}\rangle, \langle\mathsf{room\_id}\rangle, \mathsf{string}\big), \text{“C”}\Big), \Big(TRRoom1, \big(\langle\mathsf{Room}\rangle, \langle\mathsf{room\_size}\rangle, \langle\mathsf{RoomSize}\rangle\big), LargeSize\Big), \Big(TRRoom2, \big(\langle\mathsf{Room}\rangle, \langle\mathsf{room\_size}\rangle, \langle\mathsf{RoomSize}\rangle\big), MediumSize\Big), \Big(BHPRoomA, \big(\langle\mathsf{Room}\rangle, \langle\mathsf{room\_size}\rangle, \langle\mathsf{RoomSize}\rangle\big), SmallSize\Big), \Big(BHPRoomB, \big(\langle\mathsf{Room}\rangle, \langle\mathsf{room\_size}\rangle, \langle\mathsf{RoomSize}\rangle\big), SmallSize\Big), \Big(BHPRoomC, \big(\langle\mathsf{Room}\rangle, \langle\mathsf{room\_size}\rangle, \langle\mathsf{RoomSize}\rangle\big), SmallSize\Big), \Big(SmallSize, \big(\langle\mathsf{RoomSize}\rangle, \langle\mathsf{SMALL}\rangle, \langle\mathsf{RoomSize}\rangle\big), SmallSize\Big), \Big(MediumSize, \big(\langle\mathsf{RoomSize}\rangle, \langle\mathsf{MEDIUM}\rangle, \langle\mathsf{RoomSize}\rangle\big), MediumSize\Big), \Big(LargeSize, \big(\langle\mathsf{RoomSize}\rangle, \langle\mathsf{LARGE}\rangle, \langle\mathsf{RoomSize}\rangle\big), LargeSize\Big)\Big\}$$

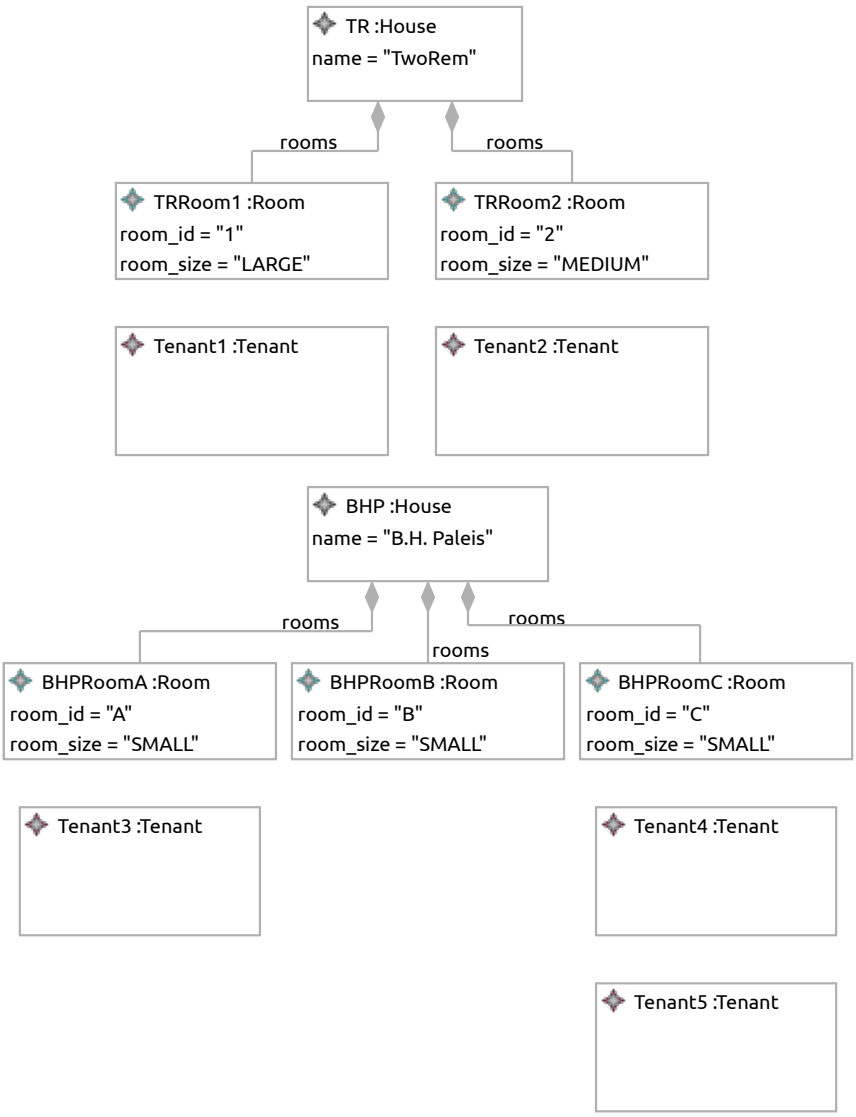
$$\text{ident} = \{(TR, TR), (BHP, BHP), (TRRoom1, TRRoom1), (TRRoom2, TRRoom2), (BHPRoomA, BHPRoomA), (BHPRoomB, BHPRoomB), (BHPRoomC, BHPRoomC), (Tenant1, Tenant1), (Tenant2, Tenant2), (Tenant3, Tenant3), (Tenant4, Tenant4), (Tenant5, Tenant5), (SmallSize, SmallSize), (MediumSize, MediumSize), (LargeSize, LargeSize)\}$$

$$\text{type}_n = \{(TR, \langle\mathsf{House}\rangle), (BHP, \langle\mathsf{House}\rangle), (TRRoom1, \langle\mathsf{Room}\rangle), (TRRoom2, \langle\mathsf{Room}\rangle), (BHPRoomA, \langle\mathsf{Room}\rangle), (BHPRoomB, \langle\mathsf{Room}\rangle), (BHPRoomC, \langle\mathsf{Room}\rangle), (Tenant1, \langle\mathsf{Tenant}\rangle), (Tenant2, \langle\mathsf{Tenant}\rangle), (Tenant3, \langle\mathsf{Tenant}\rangle), (Tenant4, \langle\mathsf{Tenant}\rangle), (Tenant5, \langle\mathsf{Tenant}\rangle), (SmallSize, \langle\mathsf{RoomSize}\rangle), (MediumSize, \langle\mathsf{RoomSize}\rangle), (LargeSize, \langle\mathsf{RoomSize}\rangle), (\text{“TwoRem”}, \mathsf{string}), (\text{“B.H. Paleis”}, \mathsf{string}), (\text{“1”}, \mathsf{string}), (\text{“2”}, \mathsf{string}), (\text{“A”}, \mathsf{string}), (\text{“B”}, \mathsf{string}), (\text{“C”}, \mathsf{string})\}$$

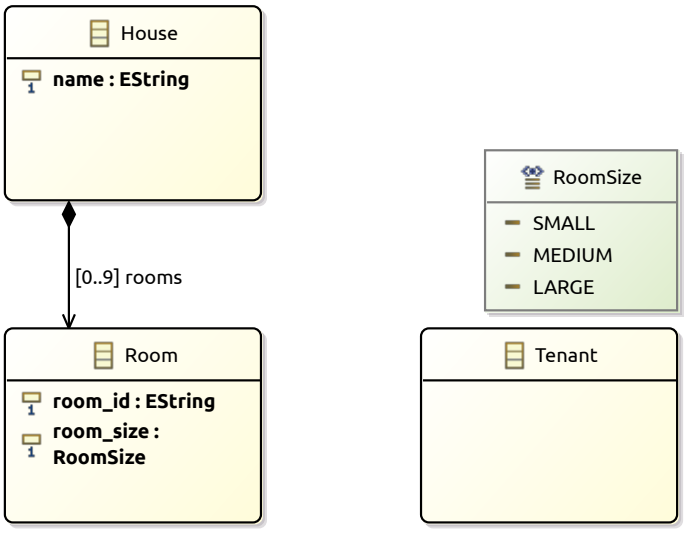
$f_8(Im_8) =$ $f_7(Im_7) \sqcup f_{Class}(Im_{Class})$ (Definition 4.4.27)

$f'_8(IG_8) =$ $f'_7(IG_7) \sqcup f'_{Class}(IG_{Class})$ (Definition 4.4.32)

A visual representation of Tm_8 and Im_8 can be found in Figure 6.16. Similarly, a visual representation of TG_8 and IG_8 can be found in Figure 6.17. Please note that because of the definitions of $f_8(Im_8)$

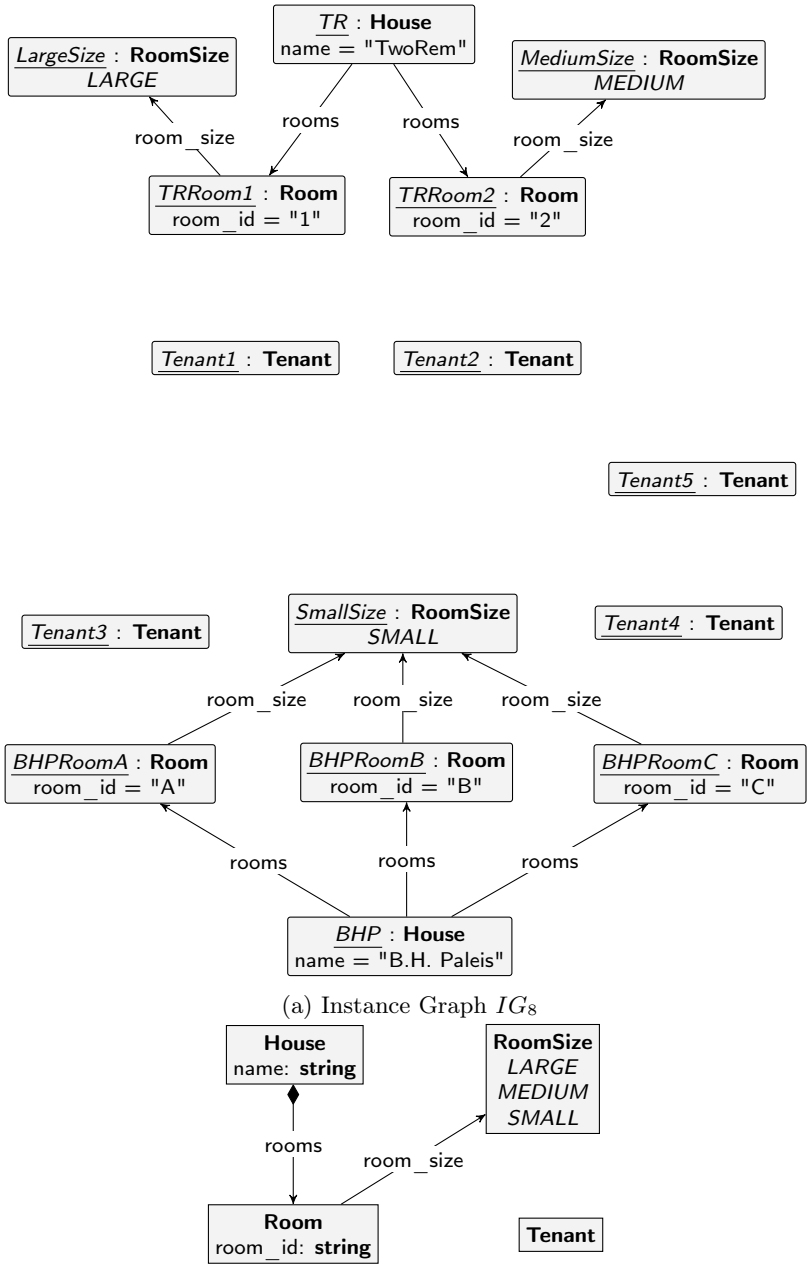


(a) Instance Model Im_8



(b) Type Model Tm_8

Figure 6.16: The Ecore model after step 8



(a) Instance Graph IG_8

(b) Type Graph TG_8

Figure 6.17: The GROOVE graphs after step 8

and $f'_8(IG_8)$, we have that $f_8(Im_8) = IG_8$ and $f'_8(IG_8) = Im_8$. Furthermore, $f_8(Im_8)$ and $f'_8(IG_8)$ are valid mapping functions themselves, such that they can be combined with another mapping function in the next step.

The introduction of the tenant class shows that models can be extended if there is already much information in the model. Introducing types can be done at any moment, so there is no need to introduce all types at the beginning. In that sense, the transformation framework presented by this thesis is not deterministic; there are multiple ways to encode the same model.

6.2.9 Tenant names

In the ninth step, the tenants are named. Formally, the name attribute on the .Tenant class is introduced, including its values. As before, Section 5.2.6 is used to introduce the field, while on the instance level, Section 5.3.6 is used to introduce the values.

The *classtype* of the new field is .Tenant, as the field will be defined for tenants. The *name* of the new field is name and the $fieldtype$ is string. The set of objects of which the value is set is is equal to all tenant objects, so $objects = \{Tenant1, Tenant2, Tenant3, Tenant4, Tenant5\}$. The function for *obids* returns the existing identifier of each of these objects. The *values* function is defined as follows:

$$values = values = \{(Tenant1, \text{“B.R. Mankjon”}), (Tenant2, \text{“P.J.R. Nam”}), (Tenant3, \text{“L. Horn”}), (Tenant4, \text{“A.C.C. Turg”}), (Tenant5, \text{“M. Silon”})\}$$

The following model is obtained:

Models after step 9

$$
\begin{aligned}
Tm_9 = \quad & \text{combine}(Tm_8, Tm_{DataField}) = \\
& Class = \{\text{.House}, \text{.Room}, \text{.Tenant}\} \\
& Enum = \{\text{.RoomSize}\} \\
& UserDataType = \{\} \\
& Field = \{(\text{.House}, \text{name}), (\text{.House}, \text{rooms}), (\text{.Room}, \text{room\_id}), \\
& \qquad (\text{.Room}, \text{room\_size}), (\text{.Tenant}, \text{name})\} \\
& \text{FieldSig} = \big\{\big((\text{.House}, \text{name}), (\text{string}, 1..1)\big), \\
& \qquad \big((\text{.House}, \text{rooms}), ([\text{setof}, \text{!.Room}], 0..9)\big), \\
& \qquad \big((\text{.Room}, \text{room\_id}), (\text{string}, 1..1)\big), \\
& \qquad \big((\text{.Room}, \text{room\_size}), (\text{.RoomSize}, 1..1)\big), \\
& \qquad \big((\text{.Tenant}, \text{name}), (\text{string}, 1..1)\big)\big\} \\
& EnumValue = \{(\text{.RoomSize}, \text{SMALL}), (\text{.RoomSize}, \text{MEDIUM}), (\text{.RoomSize}, \text{LARGE})\} \\
& Inh = \{\} \\
& Prop = \big\{\big(\text{containment}, (\text{.House}, \text{rooms})\big)\big\} \\
& Constant = \{\} \\
& \text{ConstType} = \{\}
\end{aligned}
$$

Models after step 9

$$
\begin{aligned}
Im_9 = \quad & \text{combine}(Tm_8, Im_{DataField}) = \\
& Object = \{TR, BHP, TRRoom1, TRRoom2, BHPRoomA, BHPRoomB, \\
& \quad BHPRoomC, Tenant1, Tenant2, Tenant3, Tenant4, Tenant5\} \\
& \text{ObjectClass}(ob) = \{(TR, .\mathsf{House}), (BHP, .\mathsf{House}), (TRRoom1, .\mathsf{Room}), \\
& \quad (TRRoom2, .\mathsf{Room}), (BHPRoomA, .\mathsf{Room}), (BHPRoomB, .\mathsf{Room}), \\
& \quad (BHPRoomC, .\mathsf{Room}, (Tenant1, .\mathsf{Tenant}), (Tenant2, .\mathsf{Tenant}), \\
& \quad (Tenant3, .\mathsf{Tenant}), (Tenant4, .\mathsf{Tenant}), (Tenant5, .\mathsf{Tenant})\} \\
& \text{ObjectId} = \{(TR, TR), (BHP, BHP), (TRRoom1, TRRoom1), \\
& \quad (TRRoom2, TRRoom2), (BHPRoomA, BHPRoomA), \\
& \quad (BHPRoomB, BHPRoomB), (BHPRoomC, BHPRoomC), \\
& \quad (Tenant1, Tenant1), (Tenant2, Tenant2), (Tenant3, Tenant3), \\
& \quad (Tenant4, Tenant4), (Tenant5, Tenant5)\} \\
& \text{FieldValue} = \Big\{\Big((TR, (.\mathsf{House}, \mathsf{name})), [\mathsf{string}, \text{“TwoRem”}]\Big), \\
& \quad \Big((BHP, (.\mathsf{House}, \mathsf{name})), [\mathsf{string}, \text{“B.H. Paleis”}]\Big), \\
& \quad \Big((TR, (.\mathsf{House}, \mathsf{rooms})), [\mathsf{setof}, \langle [\mathsf{obj}, TRRoom1], \\
& \qquad [\mathsf{obj}, TRRoom2]\rangle]\Big), \\
& \quad \Big((BHP, (.\mathsf{House}, \mathsf{rooms}))\, [\mathsf{setof}, \langle [\mathsf{obj}, BHPRoomA], \\
& \qquad [\mathsf{obj}, BHPRoomB], (\mathsf{obj}, BHPRoomC)\rangle]\Big), \\
& \quad \Big((TRRoom1, (.\mathsf{Room}, \mathsf{room\_id})), [\mathsf{string}, \text{“1”}]\Big), \\
& \quad \Big((TRRoom2, (.\mathsf{Room}, \mathsf{room\_id})), [\mathsf{string}, \text{“2”}]\Big), \\
& \quad \Big((BHPRoomA, (.\mathsf{Room}, \mathsf{room\_id})), [\mathsf{string}, \text{“A”}]\Big), \\
& \quad \Big((BHPRoomB, (.\mathsf{Room}, \mathsf{room\_id})), [\mathsf{string}, \text{“B”}]\Big), \\
& \quad \Big((BHPRoomC, (.\mathsf{Room}, \mathsf{room\_id})), [\mathsf{string}, \text{“C”}]\Big), \\
& \quad \Big((TRRoom1, (.\mathsf{Room}, \mathsf{room\_size})), [\mathsf{enum}, (.\mathsf{RoomSize}, \mathsf{LARGE})]\Big), \\
& \quad \Big((TRRoom2, (.\mathsf{Room}, \mathsf{room\_size})), [\mathsf{enum}, (.\mathsf{RoomSize}, \mathsf{MEDIUM})]\Big), \\
& \quad \Big((BHPRoomA, (.\mathsf{Room}, \mathsf{room\_size})), [\mathsf{enum}, (.\mathsf{RoomSize}, \mathsf{SMALL})]\Big), \\
& \quad \Big((BHPRoomB, (.\mathsf{Room}, \mathsf{room\_size})), [\mathsf{enum}, (.\mathsf{RoomSize}, \mathsf{SMALL})]\Big), \\
& \quad \Big((BHPRoomC, (.\mathsf{Room}, \mathsf{room\_size})), [\mathsf{enum}, (.\mathsf{RoomSize}, \mathsf{SMALL})]\Big), \\
& \quad \Big((Tenant1, (.\mathsf{Tenant}, \mathsf{name})), [\mathsf{string}, \text{“B.R. Mankjon”}]\Big), \\
& \quad \Big((Tenant2, (.\mathsf{Tenant}, \mathsf{name})), [\mathsf{string}, \text{“P.J.R. Nam”}]\Big), \\
& \quad \Big((Tenant3, (.\mathsf{Tenant}, \mathsf{name})), [\mathsf{string}, \text{“L. Horn”}]\Big), \\
& \quad \Big((Tenant4, (.\mathsf{Tenant}, \mathsf{name})), [\mathsf{string}, \text{“A.C.C. Turg”}]\Big), \\
& \quad \Big((Tenant5, (.\mathsf{Tenant}, \mathsf{name})), [\mathsf{string}, \text{“M. Silon”}]\Big)\Big\} \\
& \text{DefaultValue} = \{\}
\end{aligned}
$$

Models after step 9

$$
\begin{aligned}
TG_9 = \quad & \text{combine}(TG_8, TG_{DataField}) = \\
& NT = \{\langle\mathsf{House}\rangle, \langle\mathsf{Room}\rangle, \langle\mathsf{Tenant}\rangle, \langle\mathsf{RoomSize}\rangle, \mathsf{string}\} \\
& ET = \big\{\big(\langle\mathsf{House}\rangle, \langle\mathsf{name}\rangle, \mathsf{string}\big), \big(\langle\mathsf{House}\rangle, \langle\mathsf{rooms}\rangle, \langle\mathsf{Room}\rangle\big), \\
& \quad \big(\langle\mathsf{Room}\rangle, \langle\mathsf{room\_id}\rangle, \mathsf{string}\big), \big(\langle\mathsf{Room}\rangle, \langle\mathsf{room\_size}\rangle, \langle\mathsf{RoomSize}\rangle\big), \\
& \quad \big(\langle\mathsf{Tenant}\rangle, \langle\mathsf{name}\rangle, \mathsf{string}\big), \big(\langle\mathsf{RoomSize}\rangle, \langle\mathsf{SMALL}\rangle, \langle\mathsf{RoomSize}\rangle\big), \\
& \quad \big(\langle\mathsf{RoomSize}\rangle, \langle\mathsf{MEDIUM}\rangle, \langle\mathsf{RoomSize}\rangle\big), \big(\langle\mathsf{RoomSize}\rangle, \langle\mathsf{LARGE}\rangle, \langle\mathsf{RoomSize}\rangle\big)\big\} \\
& \sqsubseteq = \big\{\big(\langle\mathsf{House}\rangle, \langle\mathsf{House}\rangle\big), \big(\langle\mathsf{Room}\rangle, \langle\mathsf{Room}\rangle\big), \big(\langle\mathsf{Tenant}\rangle, \langle\mathsf{Tenant}\rangle\big), \\
& \quad \big(\langle\mathsf{RoomSize}\rangle, \langle\mathsf{RoomSize}\rangle\big), \big(\mathsf{string}, \mathsf{string}\big)\big\} \\
& abs = \{\} \\
& \text{mult} = \Big\{\Big(\big(\langle\mathsf{House}\rangle, \langle\mathsf{name}\rangle, \mathsf{string}\big), \big(0..*, 1..1\big)\Big), \\
& \quad \Big(\big(\langle\mathsf{House}\rangle, \langle\mathsf{rooms}\rangle, \langle\mathsf{Room}\rangle\big), \big(0..1, 0..9\big)\Big), \\
& \quad \Big(\big(\langle\mathsf{Room}\rangle, \langle\mathsf{room\_id}\rangle, \mathsf{string}\big), \big(0..*, 1..1\big)\Big), \\
& \quad \Big(\big(\langle\mathsf{Room}\rangle, \langle\mathsf{room\_size}\rangle, \langle\mathsf{RoomSize}\rangle\big), \big(0..*, 1..1\big)\Big), \\
& \quad \Big(\big(\langle\mathsf{Tenant}\rangle, \langle\mathsf{name}\rangle, \mathsf{string}\big), \big(0..*, 1..1\big)\Big), \\
& \quad \Big(\big(\langle\mathsf{RoomSize}\rangle, \langle\mathsf{SMALL}\rangle, \langle\mathsf{RoomSize}\rangle\big), \big(0..1, 0..1\big)\Big), \\
& \quad \Big(\big(\langle\mathsf{RoomSize}\rangle, \langle\mathsf{MEDIUM}\rangle, \langle\mathsf{RoomSize}\rangle\big), \big(0..1, 0..1\big)\Big), \\
& \quad \Big(\big(\langle\mathsf{RoomSize}\rangle, \langle\mathsf{LARGE}\rangle, \langle\mathsf{RoomSize}\rangle\big), \big(0..1, 0..1\big)\Big)\Big\} \\
& contains = \big\{\big(\langle\mathsf{House}\rangle, \langle\mathsf{rooms}\rangle, \langle\mathsf{Room}\rangle\big)\big\}
\end{aligned}
$$

Models after step 9

$IG_9 =$ combine$(TG_8, IG_{DataField}) =$

$N = \{TR, BHP, TRRoom1, TRRoom2, BHPRoomA, BHPRoomB, BHPRoomC,$
$Tenant1, Tenant2, Tenant3, Tenant4, Tenant5, SmallSize, MediumSize,$
$LargeSize,$ "TwoRem", "B.H. Paleis", "1", "2", "A", "B", "C", "B.R. Mankjon",
"P.J.R. Nam", "L. Horn", "A.C.C. Turg", "M. Silon"$\}$

$E = \Big\{ \Big(TR, (\langle\mathsf{House}\rangle, \langle\mathsf{name}\rangle, \mathsf{string}), \text{"TwoRem"}\Big),$
$\Big(BHP, (\langle\mathsf{House}\rangle, \langle\mathsf{name}\rangle, \mathsf{string}), \text{"B.H. Paleis"}\Big),$
$\Big(TR, (\langle\mathsf{House}\rangle, \langle\mathsf{rooms}\rangle, \langle\mathsf{Room}\rangle), TRRoom1\Big),$
$\Big(TR, (\langle\mathsf{House}\rangle, \langle\mathsf{rooms}\rangle, \langle\mathsf{Room}\rangle), TRRoom2\Big),$
$\Big(BHP, (\langle\mathsf{House}\rangle, \langle\mathsf{rooms}\rangle, \langle\mathsf{Room}\rangle), BHPRoomA\Big),$
$\Big(BHP, (\langle\mathsf{House}\rangle, \langle\mathsf{rooms}\rangle, \langle\mathsf{Room}\rangle), BHPRoomB\Big),$
$\Big(BHP, (\langle\mathsf{House}\rangle, \langle\mathsf{rooms}\rangle, \langle\mathsf{Room}\rangle), BHPRoomC\Big),$
$\Big(TRRoom1, (\langle\mathsf{Room}\rangle, \langle\mathsf{room\_id}\rangle, \mathsf{string}), \text{"1"}\Big),$
$\Big(TRRoom2, (\langle\mathsf{Room}\rangle, \langle\mathsf{room\_id}\rangle, \mathsf{string}), \text{"2"}\Big),$
$\Big(BHPRoomA, (\langle\mathsf{Room}\rangle, \langle\mathsf{room\_id}\rangle, \mathsf{string}), \text{"A"}\Big),$
$\Big(BHPRoomB, (\langle\mathsf{Room}\rangle, \langle\mathsf{room\_id}\rangle, \mathsf{string}), \text{"B"}\Big),$
$\Big(BHPRoomC, (\langle\mathsf{Room}\rangle, \langle\mathsf{room\_id}\rangle, \mathsf{string}), \text{"C"}\Big),$
$\Big(TRRoom1, (\langle\mathsf{Room}\rangle, \langle\mathsf{room\_size}\rangle, \langle\mathsf{RoomSize}\rangle), LargeSize\Big),$
$\Big(TRRoom2, (\langle\mathsf{Room}\rangle, \langle\mathsf{room\_size}\rangle, \langle\mathsf{RoomSize}\rangle), MediumSize\Big),$
$\Big(BHPRoomA, (\langle\mathsf{Room}\rangle, \langle\mathsf{room\_size}\rangle, \langle\mathsf{RoomSize}\rangle), SmallSize\Big),$
$\Big(BHPRoomB, (\langle\mathsf{Room}\rangle, \langle\mathsf{room\_size}\rangle, \langle\mathsf{RoomSize}\rangle), SmallSize\Big),$
$\Big(BHPRoomC, (\langle\mathsf{Room}\rangle, \langle\mathsf{room\_size}\rangle, \langle\mathsf{RoomSize}\rangle), SmallSize\Big),$
$\Big(Tenant1, (\langle\mathsf{Tenant}\rangle, \langle\mathsf{name}\rangle, \mathsf{string}), \text{"B.R. Mankjon"}\Big),$
$\Big(Tenant2, (\langle\mathsf{Tenant}\rangle, \langle\mathsf{name}\rangle, \mathsf{string}), \text{"P.J.R. Nam"}\Big),$
$\Big(Tenant3, (\langle\mathsf{Tenant}\rangle, \langle\mathsf{name}\rangle, \mathsf{string}), \text{"L. Horn"}\Big),$
$\Big(Tenant4, (\langle\mathsf{Tenant}\rangle, \langle\mathsf{name}\rangle, \mathsf{string}), \text{"A.C.C. Turg"}\Big),$
$\Big(Tenant5, (\langle\mathsf{Tenant}\rangle, \langle\mathsf{name}\rangle, \mathsf{string}), \text{"M. Silon"}\Big),$
$\Big(SmallSize, (\langle\mathsf{RoomSize}\rangle, \langle\mathsf{SMALL}\rangle, \langle\mathsf{RoomSize}\rangle), SmallSize\Big),$
$\Big(MediumSize, (\langle\mathsf{RoomSize}\rangle, \langle\mathsf{MEDIUM}\rangle, \langle\mathsf{RoomSize}\rangle), MediumSize\Big),$
$\Big(LargeSize, (\langle\mathsf{RoomSize}\rangle, \langle\mathsf{LARGE}\rangle, \langle\mathsf{RoomSize}\rangle), LargeSize\Big)\Big\}$

Models after step 9	
	ident = $\{(TR, TR), (BHP, BHP), (TRRoom1, TRRoom1), (TRRoom2, TRRoom2),$ $(BHPRoomA, BHPRoomA), (BHPRoomB, BHPRoomB),$ $(BHPRoomC, BHPRoomC), (Tenant1, Tenant1), (Tenant2, Tenant2),$ $(Tenant3, Tenant3), (Tenant4, Tenant4), (Tenant5, Tenant5),$ $(SmallSize, SmallSize), (MediumSize, MediumSize),$ $(LargeSize, LargeSize)\}$
	type_n = $\{(TR, \langle\text{House}\rangle), (BHP, \langle\text{House}\rangle), (TRRoom1, \langle\text{Room}\rangle), (TRRoom2, \langle\text{Room}\rangle),$ $(BHPRoomA, \langle\text{Room}\rangle), (BHPRoomB, \langle\text{Room}\rangle), (BHPRoomC, \langle\text{Room}\rangle),$ $(Tenant1, \langle\text{Tenant}\rangle), (Tenant2, \langle\text{Tenant}\rangle), (Tenant3, \langle\text{Tenant}\rangle),$ $(Tenant4, \langle\text{Tenant}\rangle), (Tenant5, \langle\text{Tenant}\rangle), (SmallSize, \langle\text{RoomSize}\rangle),$ $(MediumSize, \langle\text{RoomSize}\rangle), (LargeSize, \langle\text{RoomSize}\rangle), (\text{“TwoRem”}, \text{string}),$ $(\text{“B.H. Paleis”}, \text{string}), (\text{“1”}, \text{string}), (\text{“2”}, \text{string}), (\text{“A”}, \text{string}), (\text{“B”}, \text{string}),$ $(\text{“C”}, \text{string}), (\text{“B.R. Mankjon”}, \text{string}), (\text{“P.J.R. Nam”}, \text{string}),$ $(\text{“A.C.C. Turg”}, \text{string}), (\text{“L. Horn”}, \text{string}), (\text{“M. Silon”}, \text{string})\}$
$f_9(Im_9) =$	$f_8(Im_8) \sqcup f_{DataField}(Im_{DataField})$ (Definition 4.4.27)
$f'_9(IG_9) =$	$f'_8(IG_8) \sqcup f'_{DataField}(IG_{DataField})$ (Definition 4.4.32)

A visual representation of Tm_9 and Im_9 can be found in Figure 6.18. Similarly, a visual representation of TG_9 and IG_9 can be found in Figure 6.19. Please note that because of the definitions of $f_9(Im_9)$ and $f'_9(IG_9)$, we have that $f_9(Im_9) = IG_9$ and $f'_9(IG_9) = Im_9$. Furthermore, $f_9(Im_9)$ and $f'_9(IG_9)$ are valid mapping functions themselves, such that they can be combined with another mapping function in the next step.

Just like the other types, it is possible to introduce fields for the tenant objects. This way, even types that are introduced later on can be enriched with new information. The visualisation shows the different names set for the tenants in both the Ecore model and GROOVE graphs.

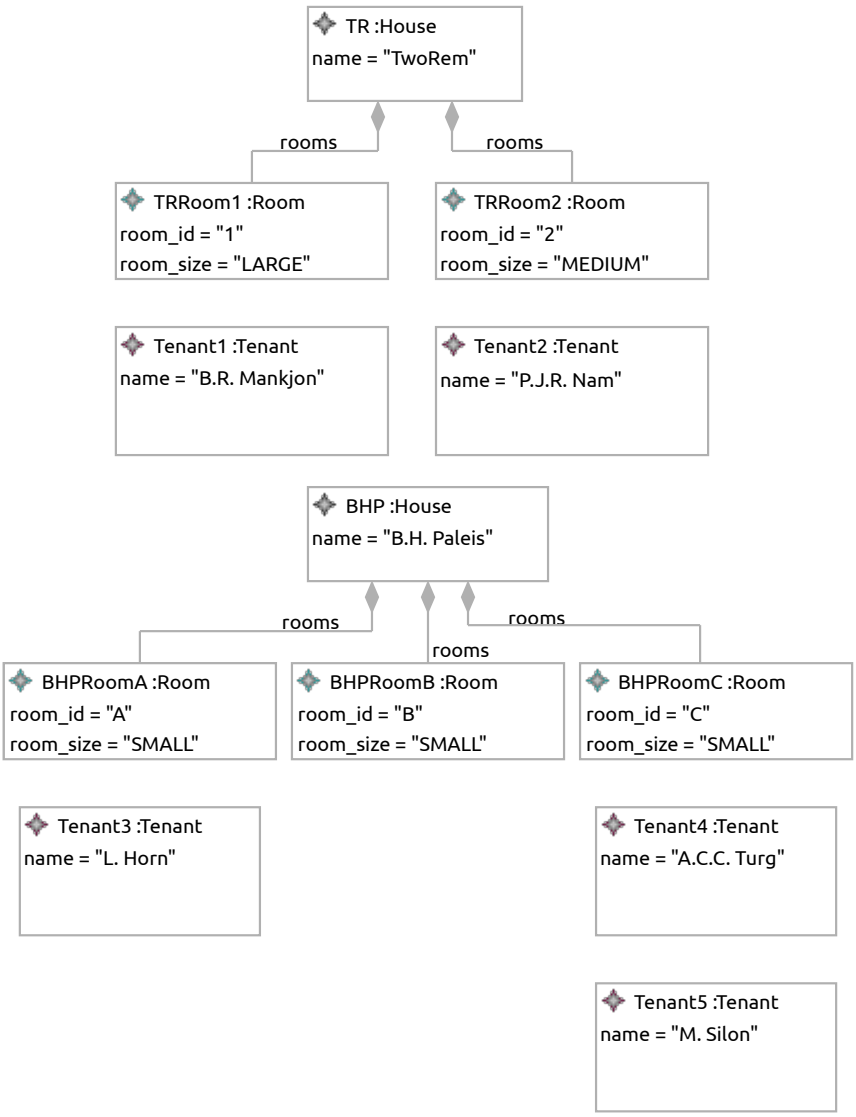
6.2.10 Tenant ages

In the tenth step, yet another data field is introduced. In this step, the .Tenant class is enriched with an age field and its values. As before, Section 5.2.6 is used to introduce the field, while on the instance level, Section 5.3.6 is used to introduce the values.

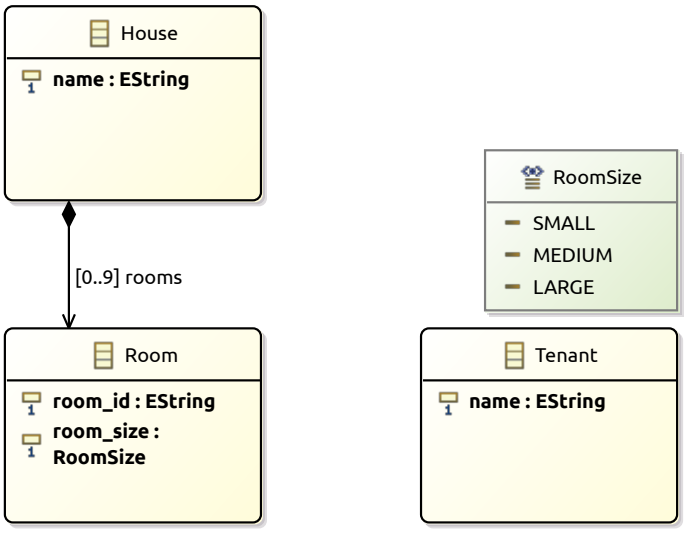
The *classtype* of the new field is .Tenant, as the field will be defined for tenants. The *name* of the new field is age and the $fieldtype$ is int. The set of objects of which the value is set is is equal to all tenant objects, so $objects = \{Tenant1, Tenant2, Tenant3, Tenant4, Tenant5\}$. The function for *obids* returns the existing identifier of each of these objects. The *values* function is defined as follows:

$$values = values = \{(Tenant1, 23), (Tenant2, 24), (Tenant3, 18),$$
$$(Tenant4, 24), (Tenant5, 19)\}$$

The following model is obtained:

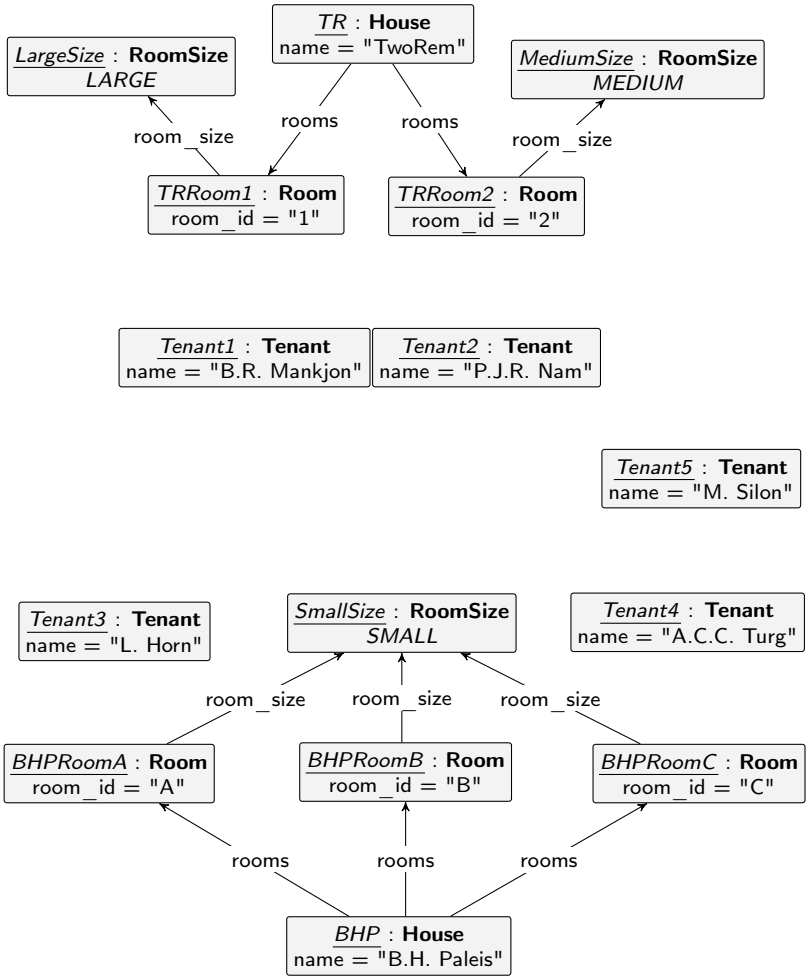


(a) Instance Model Im_9

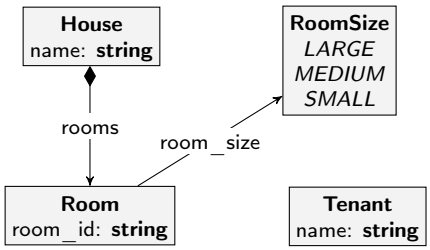


(b) Type Model Tm_9

Figure 6.18: The Ecore model after step 9



(a) Instance Graph IG_9



(b) Type Graph TG_9

Figure 6.19: The GROOVE graphs after step 9

Models after step 10

$$
\begin{aligned}
Tm_{10} = \quad & \text{combine}(Tm_9, Tm_{DataField}) = \\
& Class = \{\text{.House}, \text{.Room}, \text{.Tenant}\} \\
& Enum = \{\text{.RoomSize}\} \\
& UserDataType = \{\} \\
& Field = \{(\text{.House}, \text{name}), (\text{.House}, \text{rooms}), (\text{.Room}, \text{room\_id}), \\
& \qquad (\text{.Room}, \text{room\_size}), (\text{.Tenant}, \text{name}), (\text{.Tenant}, \text{age})\} \\
& \text{FieldSig} = \big\{\big((\text{.House}, \text{name}), (\text{string}, 1..1)\big), \\
& \qquad \big((\text{.House}, \text{rooms}), ([\text{setof}, \text{!.Room}], 0..9)\big), \\
& \qquad \big((\text{.Room}, \text{room\_id}), (\text{string}, 1..1)\big), \\
& \qquad \big((\text{.Room}, \text{room\_size}), (\text{.RoomSize}, 1..1)\big), \\
& \qquad \big((\text{.Tenant}, \text{name}), (\text{string}, 1..1)\big), \\
& \qquad \big((\text{.Tenant}, \text{age}), (\text{int}, 1..1)\big)\big\} \\
& EnumValue = \{(\text{.RoomSize}, \text{SMALL}), (\text{.RoomSize}, \text{MEDIUM}), (\text{.RoomSize}, \text{LARGE})\} \\
& Inh = \{\} \\
& Prop = \big\{\big(\text{containment}, (\text{.House}, \text{rooms})\big)\big\} \\
& Constant = \{\} \\
& \text{ConstType} = \{\}
\end{aligned}
$$

Models after step 10

$Im_{10} =$ combine$(Tm_9, Im_{DataField}) =$

$$Object = \{TR, BHP, TRRoom1, TRRoom2, BHPRoomA, BHPRoomB, BHPRoomC, Tenant1, Tenant2, Tenant3, Tenant4, Tenant5\}$$

$$\text{ObjectClass}(ob) = \{(TR, .\mathsf{House}), (BHP, .\mathsf{House}), (TRRoom1, .\mathsf{Room}), (TRRoom2, .\mathsf{Room}), (BHPRoomA, .\mathsf{Room}), (BHPRoomB, .\mathsf{Room}), (BHPRoomC, .\mathsf{Room}, (Tenant1, .\mathsf{Tenant}), (Tenant2, .\mathsf{Tenant}), (Tenant3, .\mathsf{Tenant}), (Tenant4, .\mathsf{Tenant}), (Tenant5, .\mathsf{Tenant})\}$$

$$\text{ObjectId} = \{(TR, TR), (BHP, BHP), (TRRoom1, TRRoom1), (TRRoom2, TRRoom2), (BHPRoomA, BHPRoomA), (BHPRoomB, BHPRoomB), (BHPRoomC, BHPRoomC), (Tenant1, Tenant1), (Tenant2, Tenant2), (Tenant3, Tenant3), (Tenant4, Tenant4), (Tenant5, Tenant5)\}$$

$$\begin{aligned}\text{FieldValue} = \Big\{ & \Big(\big(TR, (.\mathsf{House}, \mathsf{name}) \big), \big[\mathsf{string}, \text{“TwoRem”} \big] \Big), \\ & \Big(\big(BHP, (.\mathsf{House}, \mathsf{name}) \big), \big[\mathsf{string}, \text{“B.H. Paleis”} \big] \Big), \\ & \Big(\big(TR, (.\mathsf{House}, \mathsf{rooms}) \big), \big[\mathsf{setof}, \langle [\mathsf{obj}, TRRoom1], \\ & \quad [\mathsf{obj}, TRRoom2] \rangle \big] \Big), \\ & \Big(\big(BHP, (.\mathsf{House}, \mathsf{rooms}) \big) \big[\mathsf{setof}, \langle [\mathsf{obj}, BHPRoomA], \\ & \quad [\mathsf{obj}, BHPRoomB], (\mathsf{obj}, BHPRoomC) \rangle \big] \Big), \\ & \Big(\big(TRRoom1, (.\mathsf{Room}, \mathsf{room\_id}) \big), \big[\mathsf{string}, \text{“1”} \big] \Big), \\ & \Big(\big(TRRoom2, (.\mathsf{Room}, \mathsf{room\_id}) \big), \big[\mathsf{string}, \text{“2”} \big] \Big), \\ & \Big(\big(BHPRoomA, (.\mathsf{Room}, \mathsf{room\_id}) \big), \big[\mathsf{string}, \text{“A”} \big] \Big), \\ & \Big(\big(BHPRoomB, (.\mathsf{Room}, \mathsf{room\_id}) \big), \big[\mathsf{string}, \text{“B”} \big] \Big), \\ & \Big(\big(BHPRoomC, (.\mathsf{Room}, \mathsf{room\_id}) \big), \big[\mathsf{string}, \text{“C”} \big] \Big), \\ & \Big(\big(TRRoom1, (.\mathsf{Room}, \mathsf{room\_size}) \big), \big[\mathsf{enum}, (.\mathsf{RoomSize}, \mathsf{LARGE}) \big] \Big), \\ & \Big(\big(TRRoom2, (.\mathsf{Room}, \mathsf{room\_size}) \big), \big[\mathsf{enum}, (.\mathsf{RoomSize}, \mathsf{MEDIUM}) \big] \Big), \\ & \Big(\big(BHPRoomA, (.\mathsf{Room}, \mathsf{room\_size}) \big), \big[\mathsf{enum}, (.\mathsf{RoomSize}, \mathsf{SMALL}) \big] \Big), \\ & \Big(\big(BHPRoomB, (.\mathsf{Room}, \mathsf{room\_size}) \big), \big[\mathsf{enum}, (.\mathsf{RoomSize}, \mathsf{SMALL}) \big] \Big), \\ & \Big(\big(BHPRoomC, (.\mathsf{Room}, \mathsf{room\_size}) \big), \big[\mathsf{enum}, (.\mathsf{RoomSize}, \mathsf{SMALL}) \big] \Big), \\ & \Big(\big(Tenant1, (.\mathsf{Tenant}, \mathsf{name}) \big), \big[\mathsf{string}, \text{“B.R. Mankjon”} \big] \Big), \\ & \Big(\big(Tenant2, (.\mathsf{Tenant}, \mathsf{name}) \big), \big[\mathsf{string}, \text{“P.J.R. Nam”} \big] \Big), \\ & \Big(\big(Tenant3, (.\mathsf{Tenant}, \mathsf{name}) \big), \big[\mathsf{string}, \text{“L. Horn”} \big] \Big), \\ & \Big(\big(Tenant4, (.\mathsf{Tenant}, \mathsf{name}) \big), \big[\mathsf{string}, \text{“A.C.C. Turg”} \big] \Big), \\ & \Big(\big(Tenant5, (.\mathsf{Tenant}, \mathsf{name}) \big), \big[\mathsf{string}, \text{“M. Silon”} \big] \Big), \\ & \Big(\big(Tenant1, (.\mathsf{Tenant}, \mathsf{age}) \big), \big[\mathsf{int}, 23 \big] \Big), \\ & \Big(\big(Tenant2, (.\mathsf{Tenant}, \mathsf{age}) \big), \big[\mathsf{int}, 24 \big] \Big), \\ & \Big(\big(Tenant3, (.\mathsf{Tenant}, \mathsf{age}) \big), \big[\mathsf{int}, 18 \big] \Big), \\ & \Big(\big(Tenant4, (.\mathsf{Tenant}, \mathsf{age}) \big), \big[\mathsf{int}, 24 \big] \Big), \\ & \Big(\big(Tenant5, (.\mathsf{Tenant}, \mathsf{age}) \big), \big[\mathsf{int}, 19 \big] \Big) \Big\}\end{aligned}$$

$$\text{DefaultValue} = \{\}$$

Models after step 10

$$
\begin{aligned}
TG_{10} = \quad & \text{combine}(TG_9, TG_{DataField}) = \\
& NT = \{\langle\mathsf{House}\rangle, \langle\mathsf{Room}\rangle, \langle\mathsf{Tenant}\rangle, \langle\mathsf{RoomSize}\rangle, \mathsf{string}, \mathsf{int}\} \\
& ET = \big\{\big(\langle\mathsf{House}\rangle, \langle\mathsf{name}\rangle, \mathsf{string}\big), \big(\langle\mathsf{House}\rangle, \langle\mathsf{rooms}\rangle, \langle\mathsf{Room}\rangle\big), \\
& \quad \big(\langle\mathsf{Room}\rangle, \langle\mathsf{room\_id}\rangle, \mathsf{string}\big), \big(\langle\mathsf{Room}\rangle, \langle\mathsf{room\_size}\rangle, \langle\mathsf{RoomSize}\rangle\big), \\
& \quad \big(\langle\mathsf{Tenant}\rangle, \langle\mathsf{name}\rangle, \mathsf{string}\big), \big(\langle\mathsf{Tenant}\rangle, \langle\mathsf{age}\rangle, \mathsf{int}\big), \\
& \quad \big(\langle\mathsf{RoomSize}\rangle, \langle\mathsf{SMALL}\rangle, \langle\mathsf{RoomSize}\rangle\big), \big(\langle\mathsf{RoomSize}\rangle, \langle\mathsf{MEDIUM}\rangle, \langle\mathsf{RoomSize}\rangle\big), \\
& \quad \big(\langle\mathsf{RoomSize}\rangle, \langle\mathsf{LARGE}\rangle, \langle\mathsf{RoomSize}\rangle\big)\big\} \\
& \sqsubseteq = \big\{\big(\langle\mathsf{House}\rangle, \langle\mathsf{House}\rangle\big), \big(\langle\mathsf{Room}\rangle, \langle\mathsf{Room}\rangle\big), \big(\langle\mathsf{Tenant}\rangle, \langle\mathsf{Tenant}\rangle\big), \\
& \quad \big(\langle\mathsf{RoomSize}\rangle, \langle\mathsf{RoomSize}\rangle\big), \big(\mathsf{string}, \mathsf{string}\big), \big(\mathsf{int}, \mathsf{int}\big)\big\} \\
& abs = \{\} \\
& \text{mult} = \Big\{\Big(\big(\langle\mathsf{House}\rangle, \langle\mathsf{name}\rangle, \mathsf{string}\big), \big(0..*, 1..1\big)\Big), \\
& \quad \Big(\big(\langle\mathsf{House}\rangle, \langle\mathsf{rooms}\rangle, \langle\mathsf{Room}\rangle\big), \big(0..1, 0..9\big)\Big), \\
& \quad \Big(\big(\langle\mathsf{Room}\rangle, \langle\mathsf{room\_id}\rangle, \mathsf{string}\big), \big(0..*, 1..1\big)\Big), \\
& \quad \Big(\big(\langle\mathsf{Room}\rangle, \langle\mathsf{room\_size}\rangle, \langle\mathsf{RoomSize}\rangle\big), \big(0..*, 1..1\big)\Big), \\
& \quad \Big(\big(\langle\mathsf{Tenant}\rangle, \langle\mathsf{name}\rangle, \mathsf{string}\big), \big(0..*, 1..1\big)\Big), \\
& \quad \Big(\big(\langle\mathsf{Tenant}\rangle, \langle\mathsf{age}\rangle, \mathsf{int}\big), \big(0..*, 0..1\big)\Big), \\
& \quad \Big(\big(\langle\mathsf{RoomSize}\rangle, \langle\mathsf{SMALL}\rangle, \langle\mathsf{RoomSize}\rangle\big), \big(0..1, 0..1\big)\Big), \\
& \quad \Big(\big(\langle\mathsf{RoomSize}\rangle, \langle\mathsf{MEDIUM}\rangle, \langle\mathsf{RoomSize}\rangle\big), \big(0..1, 0..1\big)\Big), \\
& \quad \Big(\big(\langle\mathsf{RoomSize}\rangle, \langle\mathsf{LARGE}\rangle, \langle\mathsf{RoomSize}\rangle\big), \big(0..1, 0..1\big)\Big)\Big\} \\
& contains = \big\{\big(\langle\mathsf{House}\rangle, \langle\mathsf{rooms}\rangle, \langle\mathsf{Room}\rangle\big)\big\}
\end{aligned}
$$

Models after step 10

$IG_{10} =$ $\text{combine}(TG_9, IG_{DataField}) =$

$N = \{TR, BHP, TRRoom1, TRRoom2, BHPRoomA, BHPRoomB, BHPRoomC,$
$Tenant1, Tenant2, Tenant3, Tenant4, Tenant5, SmallSize, MediumSize,$
$LargeSize$, "TwoRem", "B.H. Paleis", "1", "2", "A", "B", "C", "B.R. Mankjon",
"P.J.R. Nam", "L. Horn", "A.C.C. Turg", "M. Silon", 23, 24, 18, 19$\}$

$E = \Big\{\Big(TR, (\langle\text{House}\rangle, \langle\text{name}\rangle, \text{string}), \text{"TwoRem"}\Big),$
$\Big(BHP, (\langle\text{House}\rangle, \langle\text{name}\rangle, \text{string}), \text{"B.H. Paleis"}\Big),$
$\Big(TR, (\langle\text{House}\rangle, \langle\text{rooms}\rangle, \langle\text{Room}\rangle), TRRoom1\Big),$
$\Big(TR, (\langle\text{House}\rangle, \langle\text{rooms}\rangle, \langle\text{Room}\rangle), TRRoom2\Big),$
$\Big(BHP, (\langle\text{House}\rangle, \langle\text{rooms}\rangle, \langle\text{Room}\rangle), BHPRoomA\Big),$
$\Big(BHP, (\langle\text{House}\rangle, \langle\text{rooms}\rangle, \langle\text{Room}\rangle), BHPRoomB\Big),$
$\Big(BHP, (\langle\text{House}\rangle, \langle\text{rooms}\rangle, \langle\text{Room}\rangle), BHPRoomC\Big),$
$\Big(TRRoom1, (\langle\text{Room}\rangle, \langle\text{room\_id}\rangle, \text{string}), \text{"1"}\Big),$
$\Big(TRRoom2, (\langle\text{Room}\rangle, \langle\text{room\_id}\rangle, \text{string}), \text{"2"}\Big),$
$\Big(BHPRoomA, (\langle\text{Room}\rangle, \langle\text{room\_id}\rangle, \text{string}), \text{"A"}\Big),$
$\Big(BHPRoomB, (\langle\text{Room}\rangle, \langle\text{room\_id}\rangle, \text{string}), \text{"B"}\Big),$
$\Big(BHPRoomC, (\langle\text{Room}\rangle, \langle\text{room\_id}\rangle, \text{string}), \text{"C"}\Big),$
$\Big(TRRoom1, (\langle\text{Room}\rangle, \langle\text{room\_size}\rangle, \langle\text{RoomSize}\rangle), LargeSize\Big),$
$\Big(TRRoom2, (\langle\text{Room}\rangle, \langle\text{room\_size}\rangle, \langle\text{RoomSize}\rangle), MediumSize\Big),$
$\Big(BHPRoomA, (\langle\text{Room}\rangle, \langle\text{room\_size}\rangle, \langle\text{RoomSize}\rangle), SmallSize\Big),$
$\Big(BHPRoomB, (\langle\text{Room}\rangle, \langle\text{room\_size}\rangle, \langle\text{RoomSize}\rangle), SmallSize\Big),$
$\Big(BHPRoomC, (\langle\text{Room}\rangle, \langle\text{room\_size}\rangle, \langle\text{RoomSize}\rangle), SmallSize\Big),$
$\Big(Tenant1, (\langle\text{Tenant}\rangle, \langle\text{name}\rangle, \text{string}), \text{"B.R. Mankjon"}\Big),$
$\Big(Tenant2, (\langle\text{Tenant}\rangle, \langle\text{name}\rangle, \text{string}), \text{"P.J.R. Nam"}\Big),$
$\Big(Tenant3, (\langle\text{Tenant}\rangle, \langle\text{name}\rangle, \text{string}), \text{"L. Horn"}\Big),$
$\Big(Tenant4, (\langle\text{Tenant}\rangle, \langle\text{name}\rangle, \text{string}), \text{"A.C.C. Turg"}\Big),$
$\Big(Tenant5, (\langle\text{Tenant}\rangle, \langle\text{name}\rangle, \text{string}), \text{"M. Silon"}\Big),$
$\Big(Tenant1, (\langle\text{Tenant}\rangle, \langle\text{age}\rangle, \text{int}), 23\Big), \Big(Tenant2, (\langle\text{Tenant}\rangle, \langle\text{age}\rangle, \text{int}), 24\Big),$
$\Big(Tenant3, (\langle\text{Tenant}\rangle, \langle\text{age}\rangle, \text{int}), 18\Big), \Big(Tenant4, (\langle\text{Tenant}\rangle, \langle\text{age}\rangle, \text{int}), 24\Big),$
$\Big(Tenant5, (\langle\text{Tenant}\rangle, \langle\text{age}\rangle, \text{int}), 19\Big),$
$\Big(SmallSize, (\langle\text{RoomSize}\rangle, \langle\text{SMALL}\rangle, \langle\text{RoomSize}\rangle), SmallSize\Big),$
$\Big(MediumSize, (\langle\text{RoomSize}\rangle, \langle\text{MEDIUM}\rangle, \langle\text{RoomSize}\rangle), MediumSize\Big),$
$\Big(LargeSize, (\langle\text{RoomSize}\rangle, \langle\text{LARGE}\rangle, \langle\text{RoomSize}\rangle), LargeSize\Big)\Big\}$

Models after step 10	
	ident $= \{(TR, TR), (BHP, BHP), (TRRoom1, TRRoom1), (TRRoom2, TRRoom2),$ $(BHPRoomA, BHPRoomA), (BHPRoomB, BHPRoomB),$ $(BHPRoomC, BHPRoomC), (Tenant1, Tenant1), (Tenant2, Tenant2),$ $(Tenant3, Tenant3), (Tenant4, Tenant4), (Tenant5, Tenant5),$ $(SmallSize, SmallSize), (MediumSize, MediumSize),$ $(LargeSize, LargeSize)\}$
	type$_n$ $= \{(TR, \langle\mathsf{House}\rangle), (BHP, \langle\mathsf{House}\rangle), (TRRoom1, \langle\mathsf{Room}\rangle), (TRRoom2, \langle\mathsf{Room}\rangle),$ $(BHPRoomA, \langle\mathsf{Room}\rangle), (BHPRoomB, \langle\mathsf{Room}\rangle), (BHPRoomC, \langle\mathsf{Room}\rangle),$ $(Tenant1, \langle\mathsf{Tenant}\rangle), (Tenant2, \langle\mathsf{Tenant}\rangle), (Tenant3, \langle\mathsf{Tenant}\rangle),$ $(Tenant4, \langle\mathsf{Tenant}\rangle), (Tenant5, \langle\mathsf{Tenant}\rangle), (SmallSize, \langle\mathsf{RoomSize}\rangle),$ $(MediumSize, \langle\mathsf{RoomSize}\rangle), (LargeSize, \langle\mathsf{RoomSize}\rangle), (\text{“TwoRem”}, \mathsf{string}),$ $(\text{“B.H. Paleis”}, \mathsf{string}), (\text{“1”}, \mathsf{string}), (\text{“2”}, \mathsf{string}), (\text{“A”}, \mathsf{string}), (\text{“B”}, \mathsf{string}),$ $(\text{“C”}, \mathsf{string}), (\text{“B.R. Mankjon”}, \mathsf{string}), (\text{“P.J.R. Nam”}, \mathsf{string}),$ $(\text{“A.C.C. Turg”}, \mathsf{string}), (\text{“L. Horn”}, \mathsf{string}), (\text{“M. Silon”}, \mathsf{string}), (23, \mathsf{int}),$ $(24, \mathsf{int}), (18, \mathsf{int}), (19, \mathsf{int})\}$
$f_{10}(Im_{10}) =$	$f_9(Im_9) \sqcup f_{DataField}(Im_{DataField})$ (Definition 4.4.27)
$f'_{10}(IG_{10}) =$	$f'_9(IG_9) \sqcup f'_{DataField}(IG_{DataField})$ (Definition 4.4.32)

A visual representation of Tm_{10} and Im_{10} can be found in Figure 6.20. Similarly, a visual representation of TG_{10} and IG_{10} can be found in Figure 6.21. Please note that because of the definitions of $f_{10}(Im_{10})$ and $f'_{10}(IG_{10})$, we have that $f_{10}(Im_{10}) = IG_{10}$ and $f'_{10}(IG_{10}) = Im_{10}$. Furthermore, $f_{10}(Im_{10})$ and $f'_{10}(IG_{10})$ are valid mapping functions themselves, such that they can be combined with another mapping function in the next step.

The introduction of age for the tenant shows that the data field transformation can also be applied with another type than string. The visualisation shows the different ages for the different tenants visually.

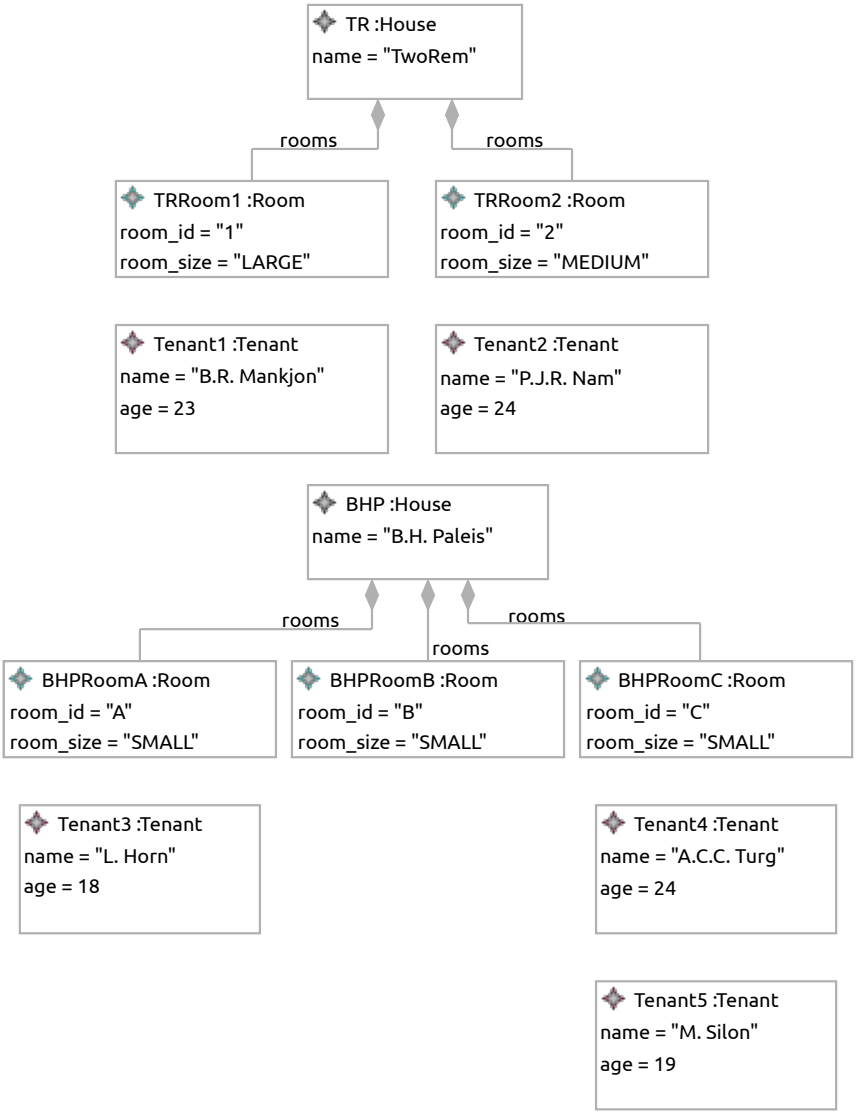
6.2.11 The tenant type enumeration type

In the eleventh step, another enumeration type is introduced. This time, the .TenantType enumeration type used to specify a type for a tenant is introduced. Section 5.2.4 is used to introduce the enumeration type on the type level, while on the instance level, Section 5.3.4 is used to introduce the values of the enumeration.

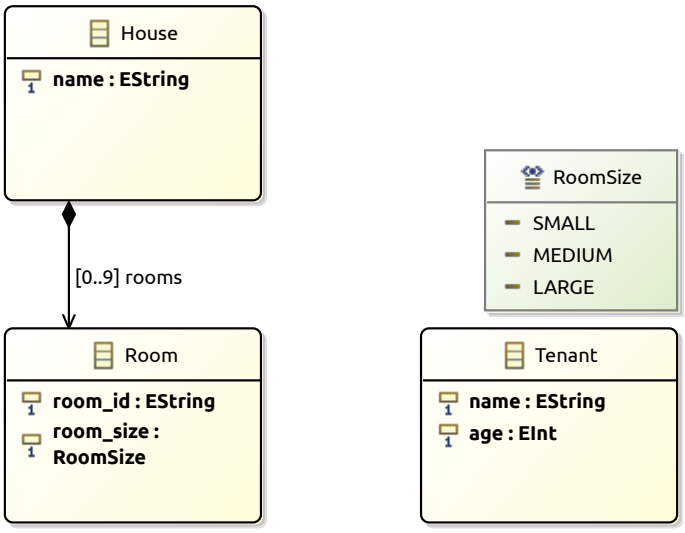
The *name* of the new enumeration type is .TenantType, and it has $values = \{\mathsf{REGULAR}, \mathsf{SUBTENANT}\}$. Please note that there are multiple encodings possible to encode an enumeration type in GROOVE. This time, the encoding using nodes is chosen. The function fob which maps the enumeration values to internal node ids is defined as follows:

$$fob = \{(\mathsf{REGULAR}, \text{RegularType}), (\mathsf{SUBTENANT}, SubtenantType)\}$$

The function fid which maps the enumeration values to explicit node identifiers is equal to the definition of fob, so $fid = fob$. The following model is obtained:

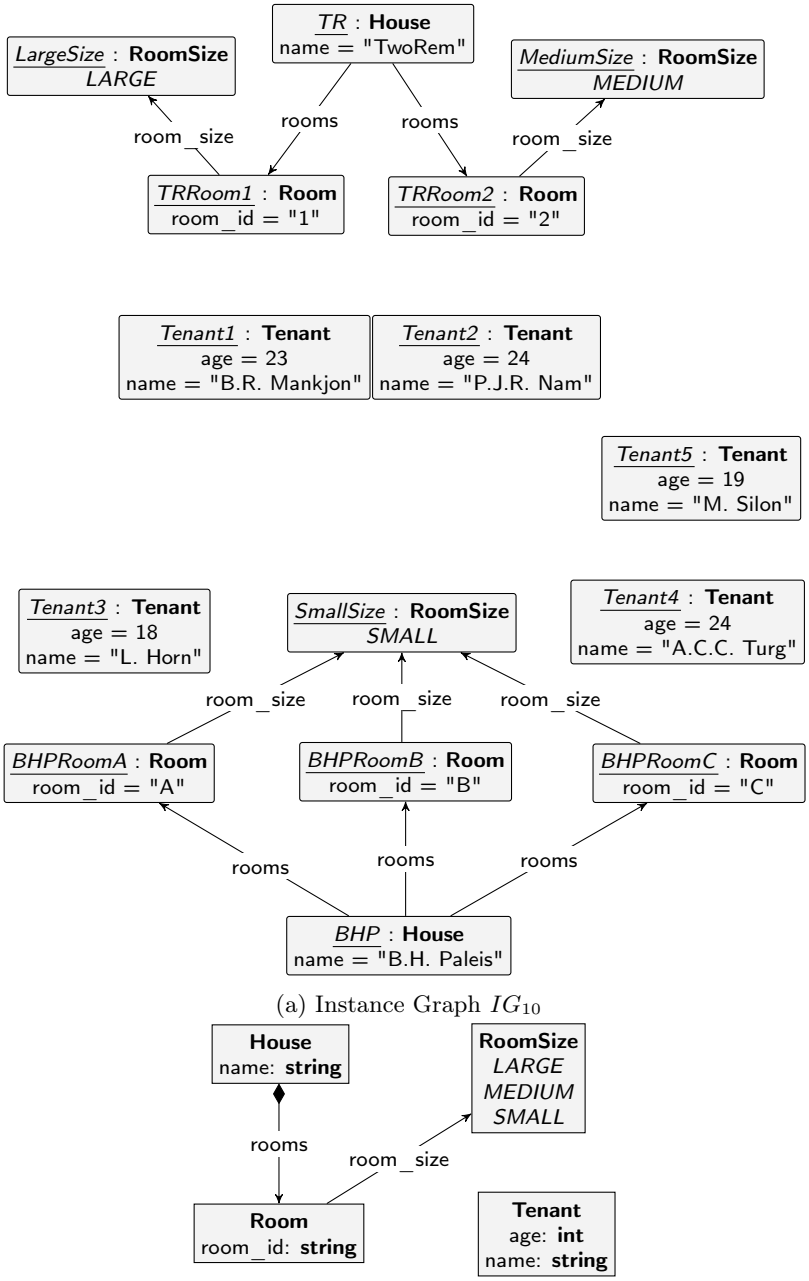


(a) Instance Model Im_{10}



(b) Type Model Tm_{10}

Figure 6.20: The Ecore model after step 10



(a) Instance Graph IG_{10}

(b) Type Graph TG_{10}

Figure 6.21: The GROOVE graphs after step 10

Models after step 11

$Tm_{11} =$ $\text{combine}(Tm_{10}, Tm_{Enum}) =$

$$
\begin{aligned}
Class &= \{\text{.House}, \text{.Room}, \text{.Tenant}\} \\
Enum &= \{\text{.RoomSize}, \text{.TenantType}\} \\
UserDataType &= \{\} \\
Field &= \{(\text{.House}, \text{name}), (\text{.House}, \text{rooms}), (\text{.Room}, \text{room\_id}), \\
&\quad (\text{.Room}, \text{room\_size}), (\text{.Tenant}, \text{name}), (\text{.Tenant}, \text{age})\} \\
\text{FieldSig} &= \big\{\big((\text{.House}, \text{name}), (\text{string}, 1..1)\big), \\
&\quad \big((\text{.House}, \text{rooms}), ([\text{setof}, \text{!.Room}], 0..9)\big), \\
&\quad \big((\text{.Room}, \text{room\_id}), (\text{string}, 1..1)\big), \\
&\quad \big((\text{.Room}, \text{room\_size}), (\text{.RoomSize}, 1..1)\big), \\
&\quad \big((\text{.Tenant}, \text{name}), (\text{string}, 1..1)\big), \\
&\quad \big((\text{.Tenant}, \text{age}), (\text{int}, 1..1)\big)\big\} \\
EnumValue &= \{(\text{.RoomSize}, \text{SMALL}), (\text{.RoomSize}, \text{MEDIUM}), (\text{.RoomSize}, \text{LARGE}) \\
&\quad (\text{.TenantType}, \text{REGULAR}), (\text{.TenantType}, \text{SUBTENANT})\} \\
Inh &= \{\} \\
Prop &= \big\{\big(\text{containment}, (\text{.House}, \text{rooms})\big)\big\} \\
Constant &= \{\} \\
\text{ConstType} &= \{\}
\end{aligned}
$$

Models after step 11

$Im_{11} =$ combine$(Tm_{10}, Im_{Enum}) =$

$$
\begin{aligned}
Object &= \{TR, BHP, TRRoom1, TRRoom2, BHPRoomA, BHPRoomB,\\
&\quad BHPRoomC, Tenant1, Tenant2, Tenant3, Tenant4, Tenant5\}\\
\text{ObjectClass}(ob) &= \{(TR, .\mathsf{House}), (BHP, .\mathsf{House}), (TRRoom1, .\mathsf{Room}),\\
&\quad (TRRoom2, .\mathsf{Room}), (BHPRoomA, .\mathsf{Room}), (BHPRoomB, .\mathsf{Room}),\\
&\quad (BHPRoomC, .\mathsf{Room}, (Tenant1, .\mathsf{Tenant}), (Tenant2, .\mathsf{Tenant}),\\
&\quad (Tenant3, .\mathsf{Tenant}), (Tenant4, .\mathsf{Tenant}), (Tenant5, .\mathsf{Tenant})\}\\
\text{ObjectId} &= \{(TR, TR), (BHP, BHP), (TRRoom1, TRRoom1),\\
&\quad (TRRoom2, TRRoom2), (BHPRoomA, BHPRoomA),\\
&\quad (BHPRoomB, BHPRoomB), (BHPRoomC, BHPRoomC),\\
&\quad (Tenant1, Tenant1), (Tenant2, Tenant2), (Tenant3, Tenant3),\\
&\quad (Tenant4, Tenant4), (Tenant5, Tenant5)\}\\
\text{FieldValue} &= \Big\{\Big((TR, (.\mathsf{House}, \mathsf{name})), [\mathsf{string}, \text{“TwoRem”}]\Big),\\
&\quad \Big((BHP, (.\mathsf{House}, \mathsf{name})), [\mathsf{string}, \text{“B.H. Paleis”}]\Big),\\
&\quad \Big((TR, (.\mathsf{House}, \mathsf{rooms})), [\mathsf{setof}, \langle[\mathsf{obj}, TRRoom1],\\
&\qquad [\mathsf{obj}, TRRoom2]\rangle]\Big),\\
&\quad \Big((BHP, (.\mathsf{House}, \mathsf{rooms}))\,[\mathsf{setof}, \langle[\mathsf{obj}, BHPRoomA],\\
&\qquad [\mathsf{obj}, BHPRoomB], (\mathsf{obj}, BHPRoomC)\rangle]\Big),\\
&\quad \Big((TRRoom1, (.\mathsf{Room}, \mathsf{room\_id})), [\mathsf{string}, \text{“1”}]\Big),\\
&\quad \Big((TRRoom2, (.\mathsf{Room}, \mathsf{room\_id})), [\mathsf{string}, \text{“2”}]\Big),\\
&\quad \Big((BHPRoomA, (.\mathsf{Room}, \mathsf{room\_id})), [\mathsf{string}, \text{“A”}]\Big),\\
&\quad \Big((BHPRoomB, (.\mathsf{Room}, \mathsf{room\_id})), [\mathsf{string}, \text{“B”}]\Big),\\
&\quad \Big((BHPRoomC, (.\mathsf{Room}, \mathsf{room\_id})), [\mathsf{string}, \text{“C”}]\Big),\\
&\quad \Big((TRRoom1, (.\mathsf{Room}, \mathsf{room\_size})), [\mathsf{enum}, (.\mathsf{RoomSize}, \mathsf{LARGE})]\Big),\\
&\quad \Big((TRRoom2, (.\mathsf{Room}, \mathsf{room\_size})), [\mathsf{enum}, (.\mathsf{RoomSize}, \mathsf{MEDIUM})]\Big),\\
&\quad \Big((BHPRoomA, (.\mathsf{Room}, \mathsf{room\_size})), [\mathsf{enum}, (.\mathsf{RoomSize}, \mathsf{SMALL})]\Big),\\
&\quad \Big((BHPRoomB, (.\mathsf{Room}, \mathsf{room\_size})), [\mathsf{enum}, (.\mathsf{RoomSize}, \mathsf{SMALL})]\Big),\\
&\quad \Big((BHPRoomC, (.\mathsf{Room}, \mathsf{room\_size})), [\mathsf{enum}, (.\mathsf{RoomSize}, \mathsf{SMALL})]\Big),\\
&\quad \Big((Tenant1, (.\mathsf{Tenant}, \mathsf{name})), [\mathsf{string}, \text{“B.R. Mankjon”}]\Big),\\
&\quad \Big((Tenant2, (.\mathsf{Tenant}, \mathsf{name})), [\mathsf{string}, \text{“P.J.R. Nam”}]\Big),\\
&\quad \Big((Tenant3, (.\mathsf{Tenant}, \mathsf{name})), [\mathsf{string}, \text{“L. Horn”}]\Big),\\
&\quad \Big((Tenant4, (.\mathsf{Tenant}, \mathsf{name})), [\mathsf{string}, \text{“A.C.C. Turg”}]\Big),\\
&\quad \Big((Tenant5, (.\mathsf{Tenant}, \mathsf{name})), [\mathsf{string}, \text{“M. Silon”}]\Big),\\
&\quad \Big((Tenant1, (.\mathsf{Tenant}, \mathsf{age})), [\mathsf{int}, 23]\Big),\\
&\quad \Big((Tenant2, (.\mathsf{Tenant}, \mathsf{age})), [\mathsf{int}, 24]\Big),\\
&\quad \Big((Tenant3, (.\mathsf{Tenant}, \mathsf{age})), [\mathsf{int}, 18]\Big),\\
&\quad \Big((Tenant4, (.\mathsf{Tenant}, \mathsf{age})), [\mathsf{int}, 24]\Big),\\
&\quad \Big((Tenant5, (.\mathsf{Tenant}, \mathsf{age})), [\mathsf{int}, 19]\Big)\Big\}\\
\text{DefaultValue} &= \{\}
\end{aligned}
$$

Models after step 11

$$
\begin{aligned}
TG_{11} = \quad & \text{combine}(TG_{10}, TG_{EnumNodes}) = \\
& NT = \{\langle \mathsf{House}\rangle, \langle \mathsf{Room}\rangle, \langle \mathsf{Tenant}\rangle, \langle \mathsf{RoomSize}\rangle, \langle \mathsf{TenantType}\rangle, \\
& \quad \langle \mathsf{TenantType}, \mathsf{REGULAR}\rangle, \langle \mathsf{TenantType}, \mathsf{SUBTENANT}\rangle, \mathsf{string}, \mathsf{int}\} \\
& ET = \{(\langle \mathsf{House}\rangle, \langle \mathsf{name}\rangle, \mathsf{string}), (\langle \mathsf{House}\rangle, \langle \mathsf{rooms}\rangle, \langle \mathsf{Room}\rangle), \\
& \quad (\langle \mathsf{Room}\rangle, \langle \mathsf{room\_id}\rangle, \mathsf{string}), (\langle \mathsf{Room}\rangle, \langle \mathsf{room\_size}\rangle, \langle \mathsf{RoomSize}\rangle), \\
& \quad (\langle \mathsf{Tenant}\rangle, \langle \mathsf{name}\rangle, \mathsf{string}), (\langle \mathsf{Tenant}\rangle, \langle \mathsf{age}\rangle, \mathsf{int}), \\
& \quad (\langle \mathsf{RoomSize}\rangle, \langle \mathsf{SMALL}\rangle, \langle \mathsf{RoomSize}\rangle), (\langle \mathsf{RoomSize}\rangle, \langle \mathsf{MEDIUM}\rangle, \langle \mathsf{RoomSize}\rangle), \\
& \quad (\langle \mathsf{RoomSize}\rangle, \langle \mathsf{LARGE}\rangle, \langle \mathsf{RoomSize}\rangle)\} \\
& \sqsubseteq = \{(\langle \mathsf{House}\rangle, \langle \mathsf{House}\rangle), (\langle \mathsf{Room}\rangle, \langle \mathsf{Room}\rangle), (\langle \mathsf{Tenant}\rangle, \langle \mathsf{Tenant}\rangle), \\
& \quad (\langle \mathsf{RoomSize}\rangle, \langle \mathsf{RoomSize}\rangle), (\langle \mathsf{TenantType}\rangle, \langle \mathsf{TenantType}\rangle), \\
& \quad (\langle \mathsf{TenantType}, \mathsf{REGULAR}\rangle, \langle \mathsf{TenantType}, \mathsf{REGULAR}\rangle), \\
& \quad (\langle \mathsf{TenantType}, \mathsf{SUBTENANT}\rangle, \langle \mathsf{TenantType}, \mathsf{SUBTENANT}\rangle), \\
& \quad (\langle \mathsf{TenantType}, \mathsf{REGULAR}\rangle, \langle \mathsf{TenantType}\rangle), \\
& \quad (\langle \mathsf{TenantType}, \mathsf{SUBTENANT}\rangle, \langle \mathsf{TenantType}\rangle), (\mathsf{string}, \mathsf{string}), (\mathsf{int}, \mathsf{int})\} \\
& abs = \{\langle \mathsf{TenantType}\rangle\} \\
& \text{mult} = \Big\{\Big((\langle \mathsf{House}\rangle, \langle \mathsf{name}\rangle, \mathsf{string}), (0..*, 1..1)\Big), \\
& \quad \Big((\langle \mathsf{House}\rangle, \langle \mathsf{rooms}\rangle, \langle \mathsf{Room}\rangle), (0..1, 0..9)\Big), \\
& \quad \Big((\langle \mathsf{Room}\rangle, \langle \mathsf{room\_id}\rangle, \mathsf{string}), (0..*, 1..1)\Big), \\
& \quad \Big((\langle \mathsf{Room}\rangle, \langle \mathsf{room\_size}\rangle, \langle \mathsf{RoomSize}\rangle), (0..*, 1..1)\Big), \\
& \quad \Big((\langle \mathsf{Tenant}\rangle, \langle \mathsf{name}\rangle, \mathsf{string}), (0..*, 1..1)\Big), \\
& \quad \Big((\langle \mathsf{Tenant}\rangle, \langle \mathsf{age}\rangle, \mathsf{int}), (0..*, 0..1)\Big), \\
& \quad \Big((\langle \mathsf{RoomSize}\rangle, \langle \mathsf{SMALL}\rangle, \langle \mathsf{RoomSize}\rangle), (0..1, 0..1)\Big), \\
& \quad \Big((\langle \mathsf{RoomSize}\rangle, \langle \mathsf{MEDIUM}\rangle, \langle \mathsf{RoomSize}\rangle), (0..1, 0..1)\Big), \\
& \quad \Big((\langle \mathsf{RoomSize}\rangle, \langle \mathsf{LARGE}\rangle, \langle \mathsf{RoomSize}\rangle), (0..1, 0..1)\Big)\Big\} \\
& contains = \{(\langle \mathsf{House}\rangle, \langle \mathsf{rooms}\rangle, \langle \mathsf{Room}\rangle)\}
\end{aligned}
$$

Models after step 11

$IG_{11} =$ $\text{combine}(TG_{10}, IG_{EnumNodes}) =$

$N = \{TR, BHP, TRRoom1, TRRoom2, BHPRoomA, BHPRoomB, BHPRoomC,$
$Tenant1, Tenant2, Tenant3, Tenant4, Tenant5, SmallSize, MediumSize,$
$LargeSize, RegularType, SubtenantType,$ "TwoRem", "B.H. Paleis", "1", "2", "A", "B", "C", "B.R. Mankjon", "P.J.R. Nam", "L. Horn", "A.C.C. Turg", "M. Silon", 23, 24, 18, 19$\}$

$E = \Big\{\Big(TR, (\langle\mathsf{House}\rangle, \langle\mathsf{name}\rangle, \mathsf{string}), \text{"TwoRem"}\Big),$
$\Big(BHP, (\langle\mathsf{House}\rangle, \langle\mathsf{name}\rangle, \mathsf{string}), \text{"B.H. Paleis"}\Big),$
$\Big(TR, (\langle\mathsf{House}\rangle, \langle\mathsf{rooms}\rangle, \langle\mathsf{Room}\rangle), TRRoom1\Big),$
$\Big(TR, (\langle\mathsf{House}\rangle, \langle\mathsf{rooms}\rangle, \langle\mathsf{Room}\rangle), TRRoom2\Big),$
$\Big(BHP, (\langle\mathsf{House}\rangle, \langle\mathsf{rooms}\rangle, \langle\mathsf{Room}\rangle), BHPRoomA\Big),$
$\Big(BHP, (\langle\mathsf{House}\rangle, \langle\mathsf{rooms}\rangle, \langle\mathsf{Room}\rangle), BHPRoomB\Big),$
$\Big(BHP, (\langle\mathsf{House}\rangle, \langle\mathsf{rooms}\rangle, \langle\mathsf{Room}\rangle), BHPRoomC\Big),$
$\Big(TRRoom1, (\langle\mathsf{Room}\rangle, \langle\mathsf{room\_id}\rangle, \mathsf{string}), \text{"1"}\Big),$
$\Big(TRRoom2, (\langle\mathsf{Room}\rangle, \langle\mathsf{room\_id}\rangle, \mathsf{string}), \text{"2"}\Big),$
$\Big(BHPRoomA, (\langle\mathsf{Room}\rangle, \langle\mathsf{room\_id}\rangle, \mathsf{string}), \text{"A"}\Big),$
$\Big(BHPRoomB, (\langle\mathsf{Room}\rangle, \langle\mathsf{room\_id}\rangle, \mathsf{string}), \text{"B"}\Big),$
$\Big(BHPRoomC, (\langle\mathsf{Room}\rangle, \langle\mathsf{room\_id}\rangle, \mathsf{string}), \text{"C"}\Big),$
$\Big(TRRoom1, (\langle\mathsf{Room}\rangle, \langle\mathsf{room\_size}\rangle, \langle\mathsf{RoomSize}\rangle), LargeSize\Big),$
$\Big(TRRoom2, (\langle\mathsf{Room}\rangle, \langle\mathsf{room\_size}\rangle, \langle\mathsf{RoomSize}\rangle), MediumSize\Big),$
$\Big(BHPRoomA, (\langle\mathsf{Room}\rangle, \langle\mathsf{room\_size}\rangle, \langle\mathsf{RoomSize}\rangle), SmallSize\Big),$
$\Big(BHPRoomB, (\langle\mathsf{Room}\rangle, \langle\mathsf{room\_size}\rangle, \langle\mathsf{RoomSize}\rangle), SmallSize\Big),$
$\Big(BHPRoomC, (\langle\mathsf{Room}\rangle, \langle\mathsf{room\_size}\rangle, \langle\mathsf{RoomSize}\rangle), SmallSize\Big),$
$\Big(Tenant1, (\langle\mathsf{Tenant}\rangle, \langle\mathsf{name}\rangle, \mathsf{string}), \text{"B.R. Mankjon"}\Big),$
$\Big(Tenant2, (\langle\mathsf{Tenant}\rangle, \langle\mathsf{name}\rangle, \mathsf{string}), \text{"P.J.R. Nam"}\Big),$
$\Big(Tenant3, (\langle\mathsf{Tenant}\rangle, \langle\mathsf{name}\rangle, \mathsf{string}), \text{"L. Horn"}\Big),$
$\Big(Tenant4, (\langle\mathsf{Tenant}\rangle, \langle\mathsf{name}\rangle, \mathsf{string}), \text{"A.C.C. Turg"}\Big),$
$\Big(Tenant5, (\langle\mathsf{Tenant}\rangle, \langle\mathsf{name}\rangle, \mathsf{string}), \text{"M. Silon"}\Big),$
$\Big(Tenant1, (\langle\mathsf{Tenant}\rangle, \langle\mathsf{age}\rangle, \mathsf{int}), 23\Big), \Big(Tenant2, (\langle\mathsf{Tenant}\rangle, \langle\mathsf{age}\rangle, \mathsf{int}), 24\Big),$
$\Big(Tenant3, (\langle\mathsf{Tenant}\rangle, \langle\mathsf{age}\rangle, \mathsf{int}), 18\Big), \Big(Tenant4, (\langle\mathsf{Tenant}\rangle, \langle\mathsf{age}\rangle, \mathsf{int}), 24\Big),$
$\Big(Tenant5, (\langle\mathsf{Tenant}\rangle, \langle\mathsf{age}\rangle, \mathsf{int}), 19\Big),$
$\Big(SmallSize, (\langle\mathsf{RoomSize}\rangle, \langle\mathsf{SMALL}\rangle, \langle\mathsf{RoomSize}\rangle), SmallSize\Big),$
$\Big(MediumSize, (\langle\mathsf{RoomSize}\rangle, \langle\mathsf{MEDIUM}\rangle, \langle\mathsf{RoomSize}\rangle), MediumSize\Big),$
$\Big(LargeSize, (\langle\mathsf{RoomSize}\rangle, \langle\mathsf{LARGE}\rangle, \langle\mathsf{RoomSize}\rangle), LargeSize\Big)\Big\}$

Models after step 11	
	$\text{ident} = \{(TR, TR), (BHP, BHP), (TRRoom1, TRRoom1), (TRRoom2, TRRoom2),$ $(BHPRoomA, BHPRoomA), (BHPRoomB, BHPRoomB),$ $(BHPRoomC, BHPRoomC), (Tenant1, Tenant1), (Tenant2, Tenant2),$ $(Tenant3, Tenant3), (Tenant4, Tenant4), (Tenant5, Tenant5),$ $(SmallSize, SmallSize), (MediumSize, MediumSize),$ $(LargeSize, LargeSize), (RegularType, RegularType),$ $(SubtenantType, SubtenantType)\}$
	$\text{type}_n = \{(TR, \langle\mathsf{House}\rangle), (BHP, \langle\mathsf{House}\rangle), (TRRoom1, \langle\mathsf{Room}\rangle), (TRRoom2, \langle\mathsf{Room}\rangle),$ $(BHPRoomA, \langle\mathsf{Room}\rangle), (BHPRoomB, \langle\mathsf{Room}\rangle), (BHPRoomC, \langle\mathsf{Room}\rangle),$ $(Tenant1, \langle\mathsf{Tenant}\rangle), (Tenant2, \langle\mathsf{Tenant}\rangle), (Tenant3, \langle\mathsf{Tenant}\rangle),$ $(Tenant4, \langle\mathsf{Tenant}\rangle), (Tenant5, \langle\mathsf{Tenant}\rangle), (SmallSize, \langle\mathsf{RoomSize}\rangle),$ $(MediumSize, \langle\mathsf{RoomSize}\rangle), (LargeSize, \langle\mathsf{RoomSize}\rangle),$ $(RegularType, \langle\mathsf{TenantType}, \mathsf{REGULAR}\rangle),$ $(SubtenantType, \langle\mathsf{TenantType}, \mathsf{SUBTENANT}\rangle), (\text{“TwoRem”}, \mathsf{string}),$ $(\text{“B.H. Paleis”}, \mathsf{string}), (\text{“1”}, \mathsf{string}), (\text{“2”}, \mathsf{string}), (\text{“A”}, \mathsf{string}), (\text{“B”}, \mathsf{string}),$ $(\text{“C”}, \mathsf{string}), (\text{“B.R. Mankjon”}, \mathsf{string}), (\text{“P.J.R. Nam”}, \mathsf{string}),$ $(\text{“A.C.C. Turg”}, \mathsf{string}), (\text{“L. Horn”}, \mathsf{string}), (\text{“M. Silon”}, \mathsf{string}), (23, \mathsf{int}),$ $(24, \mathsf{int}), (18, \mathsf{int}), (19, \mathsf{int})\}$
$f_{11}(Im_{11}) =$	$f_{10}(Im_{10}) \sqcup f_{EnumNodes}(Im_{Enum})$ (Definition 4.4.27)
$f'_{11}(IG_{11}) =$	$f'_{10}(IG_{10}) \sqcup f'_{EnumNodes}(IG_{EnumNodes})$ (Definition 4.4.32)

A visual representation of Tm_{11} and Im_{11} can be found in Figure 6.22. Similarly, a visual representation of TG_{11} and IG_{11} can be found in Figure 6.23. Please note that because of the definitions of $f_{11}(Im_{11})$ and $f'_{11}(IG_{11})$, we have that $f_{11}(Im_{11}) = IG_{11}$ and $f'_{11}(IG_{11}) = Im_{11}$. Furthermore, $f_{11}(Im_{11})$ and $f'_{11}(IG_{11})$ are valid mapping functions themselves, such that they can be combined with another mapping function in the next step.

The introduction of the tenant type enumeration type shows how different encodings can be combined within the transformation framework. The previous enumeration for room sizes was encoded using flags, while this enumeration is encoded using nodes types. Both of these encodings are present in the same model with the same transformation function.

6.2.12 Tenant types

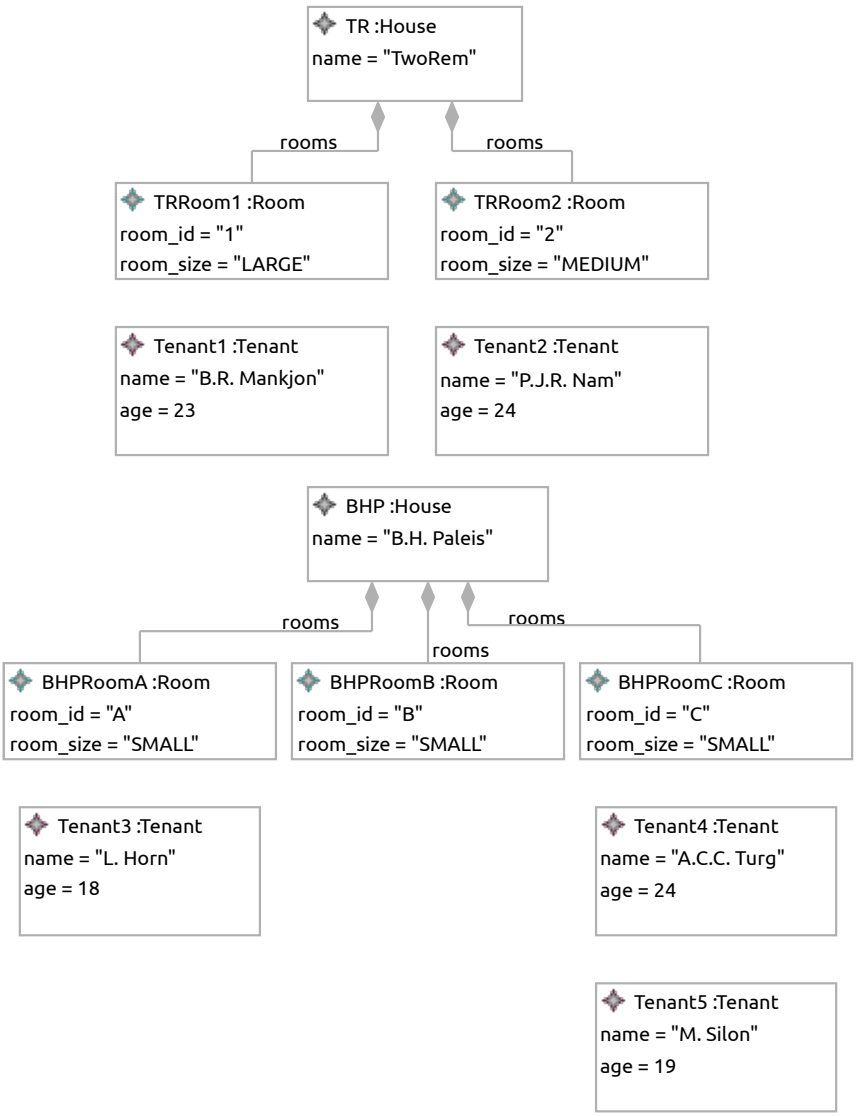
In the twelfth step, a field referencing the new enumeration type is introduced. Section 5.2.7 is used to introduce the enumeration field on the type level, while on the instance level, Section 5.3.7 is used to introduce the values.

The *classtype* of the new field is .Tenant, as the field will be defined for tenants. The *name* of the new field is type and the *enumid* is .TenantType. Furthermore, the set of *enumvalues* is equal to the set of values for the .TenantType enumeration type, so $\mathit{enumvalues} = \{\mathsf{REGULAR}, \mathsf{SUBTENANT}\}$. Then, *enumids* returns for each enumeration value the corresponding node identifier used in the GROOVE graph, while *enumob* lists the corresponding internal node id.

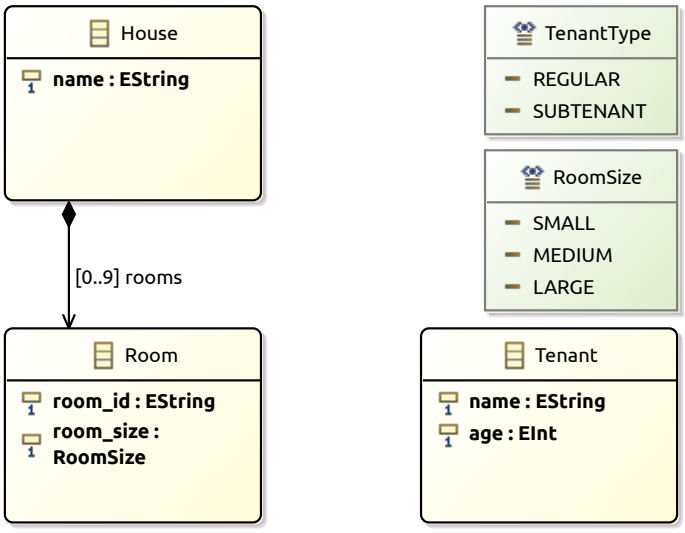
The set of objects of which the value is set is equal to all tenant objects, so $\mathit{objects} = \{Tenant1, Tenant2, Tenant3, Tenant4, Tenant5\}$. The *values* function is defined as follows:

$$\begin{aligned} \mathit{values} = \mathit{values} = \{&(Tenant1, \mathsf{REGULAR}), (Tenant2, \mathsf{REGULAR}), (Tenant3, \mathsf{REGULAR}),\\ &(Tenant4, \mathsf{REGULAR}), (Tenant5, \mathsf{SUBTENANT})\} \end{aligned}$$

The following model is obtained:

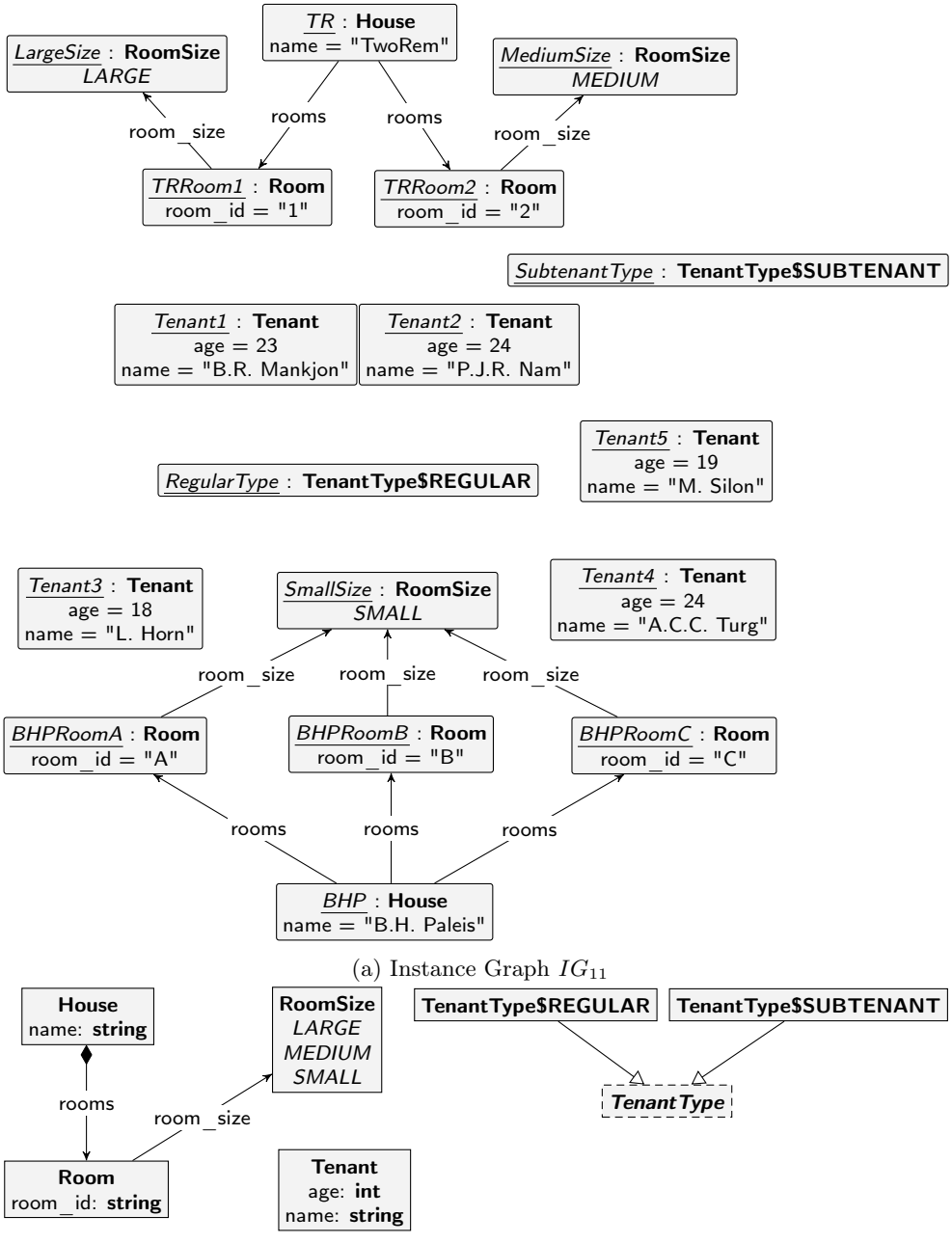


(a) Instance Model Im_{11}



(b) Type Model Tm_{11}

Figure 6.22: The Ecore model after step 11



(a) Instance Graph IG_{11}

(b) Type Graph TG_{11}

Figure 6.23: The GROOVE graphs after step 11

Models after step 12

$Tm_{12} =$ $\text{combine}(Tm_{11}, Tm_{EnumField}) =$

$$Class = \{.\text{House}, .\text{Room}, .\text{Tenant}\}$$

$$Enum = \{.\text{RoomSize}, .\text{TenantType}\}$$

$$UserDataType = \{\}$$

$$Field = \{(.\text{House}, \text{name}), (.\text{House}, \text{rooms}), (.\text{Room}, \text{room\_id}),$$
$$(.\text{Room}, \text{room\_size}), (.\text{Tenant}, \text{name}), (.\text{Tenant}, \text{age}), (.\text{Tenant}, \text{type})\}$$

$$\text{FieldSig} = \big\{\big((.\text{House}, \text{name}), (\text{string}, 1..1)\big),$$
$$\big((.\text{House}, \text{rooms}), ([\text{setof}, !.\text{Room}], 0..9)\big),$$
$$\big((.\text{Room}, \text{room\_id}), (\text{string}, 1..1)\big),$$
$$\big((.\text{Room}, \text{room\_size}), (.\text{RoomSize}, 1..1)\big),$$
$$\big((.\text{Tenant}, \text{name}), (\text{string}, 1..1)\big),$$
$$\big((.\text{Tenant}, \text{age}), (\text{int}, 1..1)\big),$$
$$\big((.\text{Tenant}, \text{type}), (.\text{TenantType}, 1..1)\big)\big\}$$

$$EnumValue = \{(.\text{RoomSize}, \text{SMALL}), (.\text{RoomSize}, \text{MEDIUM}), (.\text{RoomSize}, \text{LARGE})$$
$$(.\text{TenantType}, \text{REGULAR}), (.\text{TenantType}, \text{SUBTENANT})\}$$

$$Inh = \{\}$$

$$Prop = \big\{\big(\text{containment}, (.\text{House}, \text{rooms})\big)\big\}$$

$$Constant = \{\}$$

$$\text{ConstType} = \{\}$$

$Im_{12} =$ $\text{combine}(Tm_{11}, Im_{EnumField}) =$

$$Object = \{TR, BHP, TRRoom1, TRRoom2, BHPRoomA, BHPRoomB,$$
$$BHPRoomC, Tenant1, Tenant2, Tenant3, Tenant4, Tenant5\}$$

$$\text{ObjectClass}(ob) = \{(TR, .\text{House}), (BHP, .\text{House}), (TRRoom1, .\text{Room}),$$
$$(TRRoom2, .\text{Room}), (BHPRoomA, .\text{Room}), (BHPRoomB, .\text{Room}),$$
$$(BHPRoomC, .\text{Room}, (Tenant1, .\text{Tenant}), (Tenant2, .\text{Tenant}),$$
$$(Tenant3, .\text{Tenant}), (Tenant4, .\text{Tenant}), (Tenant5, .\text{Tenant})\}$$

$$\text{ObjectId} = \{(TR, TR), (BHP, BHP), (TRRoom1, TRRoom1),$$
$$(TRRoom2, TRRoom2), (BHPRoomA, BHPRoomA),$$
$$(BHPRoomB, BHPRoomB), (BHPRoomC, BHPRoomC),$$
$$(Tenant1, Tenant1), (Tenant2, Tenant2), (Tenant3, Tenant3),$$
$$(Tenant4, Tenant4), (Tenant5, Tenant5)\}$$

Models after step 12

$$\begin{aligned}
\text{FieldValue} = \Big\{ & \Big(\big(TR, (.\mathsf{House}, \mathsf{name}) \big), \big[\mathsf{string}, \text{“TwoRem”} \big] \Big), \\
& \Big(\big(BHP, (.\mathsf{House}, \mathsf{name}) \big), \big[\mathsf{string}, \text{“B.H. Paleis”} \big] \Big), \\
& \Big(\big(TR, (.\mathsf{House}, \mathsf{rooms}) \big), \big[\mathsf{setof}, \langle [\mathsf{obj}, TRRoom1], \\
& \quad [\mathsf{obj}, TRRoom2] \rangle \big] \Big), \\
& \Big(\big(BHP, (.\mathsf{House}, \mathsf{rooms}) \big) \big[\mathsf{setof}, \langle [\mathsf{obj}, BHPRoomA], \\
& \quad [\mathsf{obj}, BHPRoomB], (\mathsf{obj}, BHPRoomC) \rangle \big] \Big), \\
& \Big(\big(TRRoom1, (.\mathsf{Room}, \mathsf{room_id}) \big), \big[\mathsf{string}, \text{“1”} \big] \Big), \\
& \Big(\big(TRRoom2, (.\mathsf{Room}, \mathsf{room_id}) \big), \big[\mathsf{string}, \text{“2”} \big] \Big), \\
& \Big(\big(BHPRoomA, (.\mathsf{Room}, \mathsf{room_id}) \big), \big[\mathsf{string}, \text{“A”} \big] \Big), \\
& \Big(\big(BHPRoomB, (.\mathsf{Room}, \mathsf{room_id}) \big), \big[\mathsf{string}, \text{“B”} \big] \Big), \\
& \Big(\big(BHPRoomC, (.\mathsf{Room}, \mathsf{room_id}) \big), \big[\mathsf{string}, \text{“C”} \big] \Big), \\
& \Big(\big(TRRoom1, (.\mathsf{Room}, \mathsf{room_size}) \big), \big[\mathsf{enum}, (.\mathsf{RoomSize}, \mathsf{LARGE}) \big] \Big), \\
& \Big(\big(TRRoom2, (.\mathsf{Room}, \mathsf{room_size}) \big), \big[\mathsf{enum}, (.\mathsf{RoomSize}, \mathsf{MEDIUM}) \big] \Big), \\
& \Big(\big(BHPRoomA, (.\mathsf{Room}, \mathsf{room_size}) \big), \big[\mathsf{enum}, (.\mathsf{RoomSize}, \mathsf{SMALL}) \big] \Big), \\
& \Big(\big(BHPRoomB, (.\mathsf{Room}, \mathsf{room_size}) \big), \big[\mathsf{enum}, (.\mathsf{RoomSize}, \mathsf{SMALL}) \big] \Big), \\
& \Big(\big(BHPRoomC, (.\mathsf{Room}, \mathsf{room_size}) \big), \big[\mathsf{enum}, (.\mathsf{RoomSize}, \mathsf{SMALL}) \big] \Big), \\
& \Big(\big(Tenant1, (.\mathsf{Tenant}, \mathsf{name}) \big), \big[\mathsf{string}, \text{“B.R. Mankjon”} \big] \Big), \\
& \Big(\big(Tenant2, (.\mathsf{Tenant}, \mathsf{name}) \big), \big[\mathsf{string}, \text{“P.J.R. Nam”} \big] \Big), \\
& \Big(\big(Tenant3, (.\mathsf{Tenant}, \mathsf{name}) \big), \big[\mathsf{string}, \text{“L. Horn”} \big] \Big), \\
& \Big(\big(Tenant4, (.\mathsf{Tenant}, \mathsf{name}) \big), \big[\mathsf{string}, \text{“A.C.C. Turg”} \big] \Big), \\
& \Big(\big(Tenant5, (.\mathsf{Tenant}, \mathsf{name}) \big), \big[\mathsf{string}, \text{“M. Silon”} \big] \Big), \\
& \Big(\big(Tenant1, (.\mathsf{Tenant}, \mathsf{age}) \big), \big[\mathsf{int}, 23 \big] \Big), \\
& \Big(\big(Tenant2, (.\mathsf{Tenant}, \mathsf{age}) \big), \big[\mathsf{int}, 24 \big] \Big), \\
& \Big(\big(Tenant3, (.\mathsf{Tenant}, \mathsf{age}) \big), \big[\mathsf{int}, 18 \big] \Big), \\
& \Big(\big(Tenant4, (.\mathsf{Tenant}, \mathsf{age}) \big), \big[\mathsf{int}, 24 \big] \Big), \\
& \Big(\big(Tenant5, (.\mathsf{Tenant}, \mathsf{age}) \big), \big[\mathsf{int}, 19 \big] \Big), \\
& \Big(\big(Tenant1, (.\mathsf{Tenant}, \mathsf{type}) \big), \big[\mathsf{enum}, (.\mathsf{TenantType}, \mathsf{REGULAR}) \big] \Big), \\
& \Big(\big(Tenant2, (.\mathsf{Tenant}, \mathsf{type}) \big), \big[\mathsf{enum}, (.\mathsf{TenantType}, \mathsf{REGULAR}) \big] \Big), \\
& \Big(\big(Tenant3, (.\mathsf{Tenant}, \mathsf{type}) \big), \big[\mathsf{enum}, (.\mathsf{TenantType}, \mathsf{REGULAR}) \big] \Big), \\
& \Big(\big(Tenant4, (.\mathsf{Tenant}, \mathsf{type}) \big), \big[\mathsf{enum}, (.\mathsf{TenantType}, \mathsf{REGULAR}) \big] \Big), \\
& \Big(\big(Tenant5, (.\mathsf{Tenant}, \mathsf{type}) \big), \big[\mathsf{enum}, (.\mathsf{TenantType}, \mathsf{SUBTENANT}) \big] \Big) \Big\}
\end{aligned}$$

$$\text{DefaultValue} = \{\}$$

Models after step 12

$TG_{12} = \text{combine}(TG_{11}, TG_{EnumFieldNodes}) =$

$$NT = \{\langle\mathsf{House}\rangle, \langle\mathsf{Room}\rangle, \langle\mathsf{Tenant}\rangle, \langle\mathsf{RoomSize}\rangle, \langle\mathsf{TenantType}\rangle,$$
$$\langle\mathsf{TenantType}, \mathsf{REGULAR}\rangle, \langle\mathsf{TenantType}, \mathsf{SUBTENANT}\rangle, \mathsf{string}, \mathsf{int}\}$$

$$ET = \{\big(\langle\mathsf{House}\rangle, \langle\mathsf{name}\rangle, \mathsf{string}\big), \big(\langle\mathsf{House}\rangle, \langle\mathsf{rooms}\rangle, \langle\mathsf{Room}\rangle\big),$$
$$\big(\langle\mathsf{Room}\rangle, \langle\mathsf{room\_id}\rangle, \mathsf{string}\big), \big(\langle\mathsf{Room}\rangle, \langle\mathsf{room\_size}\rangle, \langle\mathsf{RoomSize}\rangle\big),$$
$$\big(\langle\mathsf{Tenant}\rangle, \langle\mathsf{name}\rangle, \mathsf{string}\big), \big(\langle\mathsf{Tenant}\rangle, \langle\mathsf{age}\rangle, \mathsf{int}\big),$$
$$\big(\langle\mathsf{Tenant}\rangle, \langle\mathsf{type}\rangle, \langle\mathsf{TenantType}\rangle\big), \big(\langle\mathsf{RoomSize}\rangle, \langle\mathsf{SMALL}\rangle, \langle\mathsf{RoomSize}\rangle\big),$$
$$\big(\langle\mathsf{RoomSize}\rangle, \langle\mathsf{MEDIUM}\rangle, \langle\mathsf{RoomSize}\rangle\big),$$
$$\big(\langle\mathsf{RoomSize}\rangle, \langle\mathsf{LARGE}\rangle, \langle\mathsf{RoomSize}\rangle\big)\}$$

$$\sqsubseteq = \{\big(\langle\mathsf{House}\rangle, \langle\mathsf{House}\rangle\big), \big(\langle\mathsf{Room}\rangle, \langle\mathsf{Room}\rangle\big), \big(\langle\mathsf{Tenant}\rangle, \langle\mathsf{Tenant}\rangle\big),$$
$$\big(\langle\mathsf{RoomSize}\rangle, \langle\mathsf{RoomSize}\rangle\big), \big(\langle\mathsf{TenantType}\rangle, \langle\mathsf{TenantType}\rangle\big),$$
$$\big(\langle\mathsf{TenantType}, \mathsf{REGULAR}\rangle, \langle\mathsf{TenantType}, \mathsf{REGULAR}\rangle\big),$$
$$\big(\langle\mathsf{TenantType}, \mathsf{SUBTENANT}\rangle, \langle\mathsf{TenantType}, \mathsf{SUBTENANT}\rangle\big),$$
$$\big(\langle\mathsf{TenantType}, \mathsf{REGULAR}\rangle, \langle\mathsf{TenantType}\rangle\big),$$
$$\big(\langle\mathsf{TenantType}, \mathsf{SUBTENANT}\rangle, \langle\mathsf{TenantType}\rangle\big), \big(\mathsf{string}, \mathsf{string}\big), \big(\mathsf{int}, \mathsf{int}\big)\}$$

$$abs = \{\langle\mathsf{TenantType}\rangle\}$$

$$\text{mult} = \Big\{\Big(\big(\langle\mathsf{House}\rangle, \langle\mathsf{name}\rangle, \mathsf{string}\big), (0..*, 1..1)\Big),$$
$$\Big(\big(\langle\mathsf{House}\rangle, \langle\mathsf{rooms}\rangle, \langle\mathsf{Room}\rangle\big), (0..1, 0..9)\Big),$$
$$\Big(\big(\langle\mathsf{Room}\rangle, \langle\mathsf{room\_id}\rangle, \mathsf{string}\big), (0..*, 1..1)\Big),$$
$$\Big(\big(\langle\mathsf{Room}\rangle, \langle\mathsf{room\_size}\rangle, \langle\mathsf{RoomSize}\rangle\big), (0..*, 1..1)\Big),$$
$$\Big(\big(\langle\mathsf{Tenant}\rangle, \langle\mathsf{name}\rangle, \mathsf{string}\big), (0..*, 1..1)\Big),$$
$$\Big(\big(\langle\mathsf{Tenant}\rangle, \langle\mathsf{age}\rangle, \mathsf{int}\big), (0..*, 0..1)\Big),$$
$$\Big(\big(\langle\mathsf{Tenant}\rangle, \langle\mathsf{type}\rangle, \langle\mathsf{TenantType}\rangle\big), (0..*, 1..1)\Big),$$
$$\Big(\big(\langle\mathsf{RoomSize}\rangle, \langle\mathsf{SMALL}\rangle, \langle\mathsf{RoomSize}\rangle\big), (0..1, 0..1)\Big),$$
$$\Big(\big(\langle\mathsf{RoomSize}\rangle, \langle\mathsf{MEDIUM}\rangle, \langle\mathsf{RoomSize}\rangle\big), (0..1, 0..1)\Big),$$
$$\Big(\big(\langle\mathsf{RoomSize}\rangle, \langle\mathsf{LARGE}\rangle, \langle\mathsf{RoomSize}\rangle\big), (0..1, 0..1)\Big)\Big\}$$

$$contains = \{\big(\langle\mathsf{House}\rangle, \langle\mathsf{rooms}\rangle, \langle\mathsf{Room}\rangle\big)\}$$

Models after step 12

$IG_{12} =$ $\text{combine}(TG_{11}, IG_{EnumFieldNodes}) =$

$N = \{TR, BHP, TRRoom1, TRRoom2, BHPRoomA, BHPRoomB, BHPRoomC,$
$Tenant1, Tenant2, Tenant3, Tenant4, Tenant5, SmallSize, MediumSize,$
$LargeSize, RegularType, SubtenantType,$ "TwoRem", "B.H. Paleis", "1", "2", "A",
"B", "C", "B.R. Mankjon", "P.J.R. Nam", "L. Horn", "A.C.C. Turg", "M. Silon", 23,
24, 18, 19$\}$

$E = \Big\{\Big(TR, \big(\langle\mathsf{House}\rangle, \langle\mathsf{name}\rangle, \mathsf{string}\big), \text{"TwoRem"}\Big),$
$\Big(BHP, \big(\langle\mathsf{House}\rangle, \langle\mathsf{name}\rangle, \mathsf{string}\big), \text{"B.H. Paleis"}\Big),$
$\Big(TR, \big(\langle\mathsf{House}\rangle, \langle\mathsf{rooms}\rangle, \langle\mathsf{Room}\rangle\big), TRRoom1\Big),$
$\Big(TR, \big(\langle\mathsf{House}\rangle, \langle\mathsf{rooms}\rangle, \langle\mathsf{Room}\rangle\big), TRRoom2\Big),$
$\Big(BHP, \big(\langle\mathsf{House}\rangle, \langle\mathsf{rooms}\rangle, \langle\mathsf{Room}\rangle\big), BHPRoomA\Big),$
$\Big(BHP, \big(\langle\mathsf{House}\rangle, \langle\mathsf{rooms}\rangle, \langle\mathsf{Room}\rangle\big), BHPRoomB\Big),$
$\Big(BHP, \big(\langle\mathsf{House}\rangle, \langle\mathsf{rooms}\rangle, \langle\mathsf{Room}\rangle\big), BHPRoomC\Big),$
$\Big(TRRoom1, \big(\langle\mathsf{Room}\rangle, \langle\mathsf{room\_id}\rangle, \mathsf{string}\big), \text{"1"}\Big),$
$\Big(TRRoom2, \big(\langle\mathsf{Room}\rangle, \langle\mathsf{room\_id}\rangle, \mathsf{string}\big), \text{"2"}\Big),$
$\Big(BHPRoomA, \big(\langle\mathsf{Room}\rangle, \langle\mathsf{room\_id}\rangle, \mathsf{string}\big), \text{"A"}\Big),$
$\Big(BHPRoomB, \big(\langle\mathsf{Room}\rangle, \langle\mathsf{room\_id}\rangle, \mathsf{string}\big), \text{"B"}\Big),$
$\Big(BHPRoomC, \big(\langle\mathsf{Room}\rangle, \langle\mathsf{room\_id}\rangle, \mathsf{string}\big), \text{"C"}\Big),$
$\Big(TRRoom1, \big(\langle\mathsf{Room}\rangle, \langle\mathsf{room\_size}\rangle, \langle\mathsf{RoomSize}\rangle\big), LargeSize\Big),$
$\Big(TRRoom2, \big(\langle\mathsf{Room}\rangle, \langle\mathsf{room\_size}\rangle, \langle\mathsf{RoomSize}\rangle\big), MediumSize\Big),$
$\Big(BHPRoomA, \big(\langle\mathsf{Room}\rangle, \langle\mathsf{room\_size}\rangle, \langle\mathsf{RoomSize}\rangle\big), SmallSize\Big),$
$\Big(BHPRoomB, \big(\langle\mathsf{Room}\rangle, \langle\mathsf{room\_size}\rangle, \langle\mathsf{RoomSize}\rangle\big), SmallSize\Big),$
$\Big(BHPRoomC, \big(\langle\mathsf{Room}\rangle, \langle\mathsf{room\_size}\rangle, \langle\mathsf{RoomSize}\rangle\big), SmallSize\Big),$
$\Big(Tenant1, \big(\langle\mathsf{Tenant}\rangle, \langle\mathsf{name}\rangle, \mathsf{string}\big), \text{"B.R. Mankjon"}\Big),$
$\Big(Tenant2, \big(\langle\mathsf{Tenant}\rangle, \langle\mathsf{name}\rangle, \mathsf{string}\big), \text{"P.J.R. Nam"}\Big),$
$\Big(Tenant3, \big(\langle\mathsf{Tenant}\rangle, \langle\mathsf{name}\rangle, \mathsf{string}\big), \text{"L. Horn"}\Big),$
$\Big(Tenant4, \big(\langle\mathsf{Tenant}\rangle, \langle\mathsf{name}\rangle, \mathsf{string}\big), \text{"A.C.C. Turg"}\Big),$
$\Big(Tenant5, \big(\langle\mathsf{Tenant}\rangle, \langle\mathsf{name}\rangle, \mathsf{string}\big), \text{"M. Silon"}\Big),$
$\Big(Tenant1, \big(\langle\mathsf{Tenant}\rangle, \langle\mathsf{age}\rangle, \mathsf{int}\big), 23\Big), \Big(Tenant2, \big(\langle\mathsf{Tenant}\rangle, \langle\mathsf{age}\rangle, \mathsf{int}\big), 24\Big),$
$\Big(Tenant3, \big(\langle\mathsf{Tenant}\rangle, \langle\mathsf{age}\rangle, \mathsf{int}\big), 18\Big), \Big(Tenant4, \big(\langle\mathsf{Tenant}\rangle, \langle\mathsf{age}\rangle, \mathsf{int}\big), 24\Big),$
$\Big(Tenant5, \big(\langle\mathsf{Tenant}\rangle, \langle\mathsf{age}\rangle, \mathsf{int}\big), 19\Big),$
$\Big(Tenant1, \big(\langle\mathsf{Tenant}\rangle, \langle\mathsf{type}\rangle, \langle\mathsf{TenantType}\rangle\big), RegularType\Big),$
$\Big(Tenant2, \big(\langle\mathsf{Tenant}\rangle, \langle\mathsf{type}\rangle, \langle\mathsf{TenantType}\rangle\big), RegularType\Big),$
$\Big(Tenant3, \big(\langle\mathsf{Tenant}\rangle, \langle\mathsf{type}\rangle, \langle\mathsf{TenantType}\rangle\big), RegularType\Big),$
$\Big(Tenant4, \big(\langle\mathsf{Tenant}\rangle, \langle\mathsf{type}\rangle, \langle\mathsf{TenantType}\rangle\big), RegularType\Big),$
$\Big(Tenant5, \big(\langle\mathsf{Tenant}\rangle, \langle\mathsf{type}\rangle, \langle\mathsf{TenantType}\rangle\big), SubtenantType\Big),$

Models after step 12

$$\Big(SmallSize, \big(\langle \mathsf{RoomSize}\rangle, \langle \mathsf{SMALL}\rangle, \langle \mathsf{RoomSize}\rangle\big), SmallSize\Big),$$
$$\Big(MediumSize, \big(\langle \mathsf{RoomSize}\rangle, \langle \mathsf{MEDIUM}\rangle, \langle \mathsf{RoomSize}\rangle\big), MediumSize\Big),$$
$$\Big(LargeSize, \big(\langle \mathsf{RoomSize}\rangle, \langle \mathsf{LARGE}\rangle, \langle \mathsf{RoomSize}\rangle\big), LargeSize\Big)\Big\}$$

$$\begin{aligned}\text{ident} = \{&(TR, TR), (BHP, BHP), (TRRoom1, TRRoom1), (TRRoom2, TRRoom2),\\ &(BHPRoomA, BHPRoomA), (BHPRoomB, BHPRoomB),\\ &(BHPRoomC, BHPRoomC), (Tenant1, Tenant1), (Tenant2, Tenant2),\\ &(Tenant3, Tenant3), (Tenant4, Tenant4), (Tenant5, Tenant5),\\ &(SmallSize, SmallSize), (MediumSize, MediumSize),\\ &(LargeSize, LargeSize), (RegularType, RegularType),\\ &(SubtenantType, SubtenantType)\}\end{aligned}$$

$$\begin{aligned}\text{type}_n = \{&(TR, \langle \mathsf{House}\rangle), (BHP, \langle \mathsf{House}\rangle), (TRRoom1, \langle \mathsf{Room}\rangle), (TRRoom2, \langle \mathsf{Room}\rangle),\\ &(BHPRoomA, \langle \mathsf{Room}\rangle), (BHPRoomB, \langle \mathsf{Room}\rangle), (BHPRoomC, \langle \mathsf{Room}\rangle),\\ &(Tenant1, \langle \mathsf{Tenant}\rangle), (Tenant2, \langle \mathsf{Tenant}\rangle), (Tenant3, \langle \mathsf{Tenant}\rangle),\\ &(Tenant4, \langle \mathsf{Tenant}\rangle), (Tenant5, \langle \mathsf{Tenant}\rangle), (SmallSize, \langle \mathsf{RoomSize}\rangle),\\ &(MediumSize, \langle \mathsf{RoomSize}\rangle), (LargeSize, \langle \mathsf{RoomSize}\rangle),\\ &(RegularType, \langle \mathsf{TenantType}, \mathsf{REGULAR}\rangle),\\ &(SubtenantType, \langle \mathsf{TenantType}, \mathsf{SUBTENANT}\rangle), (\text{“TwoRem”}, \mathsf{string}),\\ &(\text{“B.H. Paleis”}, \mathsf{string}), (\text{“1”}, \mathsf{string}), (\text{“2”}, \mathsf{string}), (\text{“A”}, \mathsf{string}), (\text{“B”}, \mathsf{string}),\\ &(\text{“C”}, \mathsf{string}), (\text{“B.R. Mankjon”}, \mathsf{string}), (\text{“P.J.R. Nam”}, \mathsf{string}),\\ &(\text{“A.C.C. Turg”}, \mathsf{string}), (\text{“L. Horn”}, \mathsf{string}), (\text{“M. Silon”}, \mathsf{string}), (23, \mathsf{int}),\\ &(24, \mathsf{int}), (18, \mathsf{int}), (19, \mathsf{int})\}\end{aligned}$$

$f_{12}(Im_{12}) = f_{11}(Im_{11}) \sqcup f_{EnumFieldNodes}(Im_{EnumField})$ (Definition 4.4.27)
$f'_{12}(IG_{12}) = f'_{11}(IG_{11}) \sqcup f'_{EnumFieldNodes}(IG_{EnumFieldNodes})$ (Definition 4.4.32)

A visual representation of Tm_{12} and Im_{12} can be found in Figure 6.24. Similarly, a visual representation of TG_{12} and IG_{12} can be found in Figure 6.25. Please note that because of the definitions of $f_{12}(Im_{12})$ and $f'_{12}(IG_{12})$, we have that $f_{12}(Im_{12}) = IG_{12}$ and $f'_{12}(IG_{12}) = Im_{12}$. Furthermore, $f_{12}(Im_{12})$ and $f'_{12}(IG_{12})$ are valid mapping functions themselves, such that they can be combined with another mapping function in the next step.

The previous step showed how two different encodings of an enumeration type were combined within the same model. This step concludes the combination of these encodings by showing that it is possible to reference values from the tenant type enumeration type, even though it is encoded in a different encoding than the room sizes enumeration type.

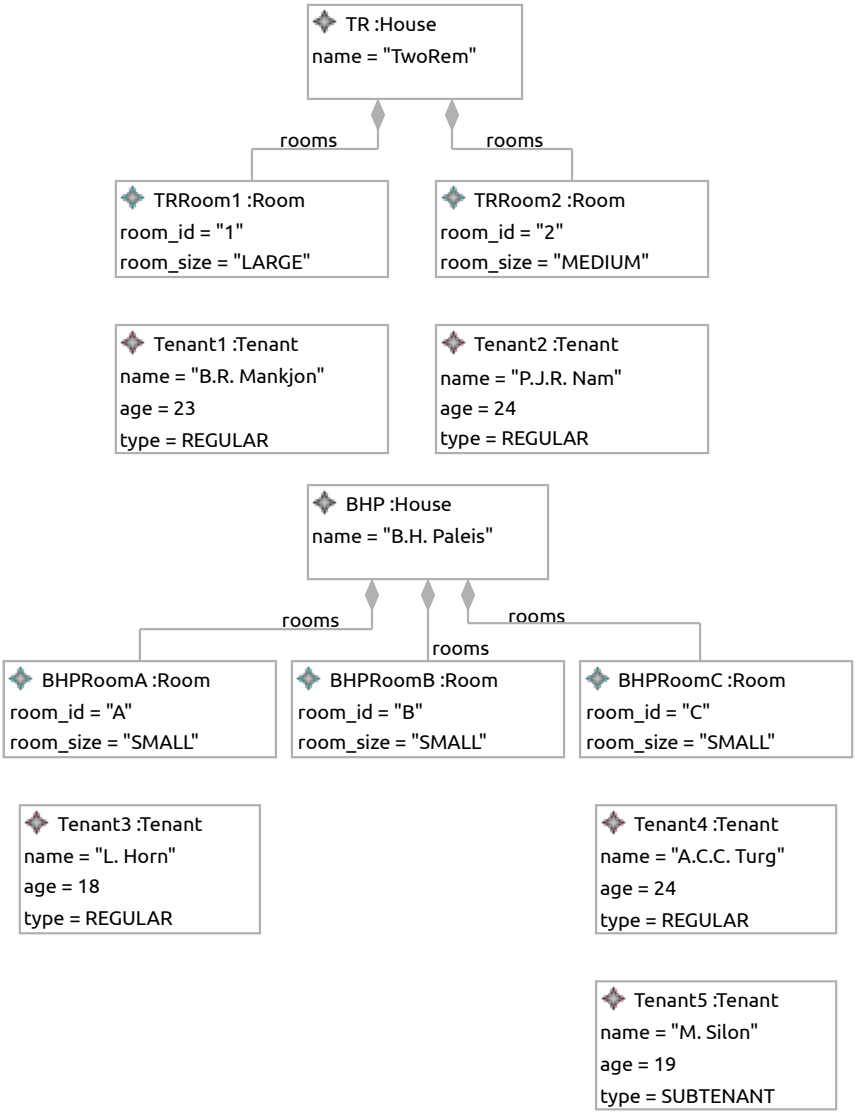
6.2.13 Room & tenant relationship

In the thirteenth step, a relationship between two objects is introduced. This is the first step in which a relation is introduced between existing objects. For this relation, a object of .Room may reference a single .Tenant object. Section 5.2.8 is used to introduce the field on the type level, while on the instance level, Section 5.3.8 is used to introduce the values.

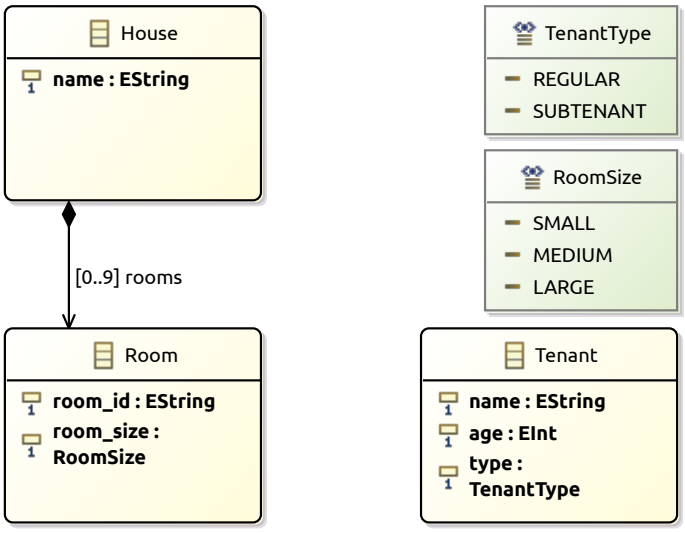
The *classtype* of the new field is .Room, as the field will be defined for rooms. The *name* of the new field is tenant and the *fieldtype* is .Tenant. The set of objects that will receive a value is defined as $valobjects = \{TRRoom1, TRRoom2, BHPRoomA, BHPRoomC\}$, while the set of objects that will not receive a value is defined as $nilobjects = \{BHPRoomB\}$. The union of these sets is equal to all the room objects, as expected. The function for *obids* returns the existing identifier of each of these objects. The *values* function is defined as follows:

$$\begin{aligned}values = \{&(TRRoom1, Tenant1), (TRRoom2, Tenant2), (BHPRoomA, Tenant3),\\ &(BHPRoomC, Tenant4)\}\end{aligned}$$

For the objects referenced by the objects in *valobjects*, the function *obids* returns their existing identifier. The following model is obtained:

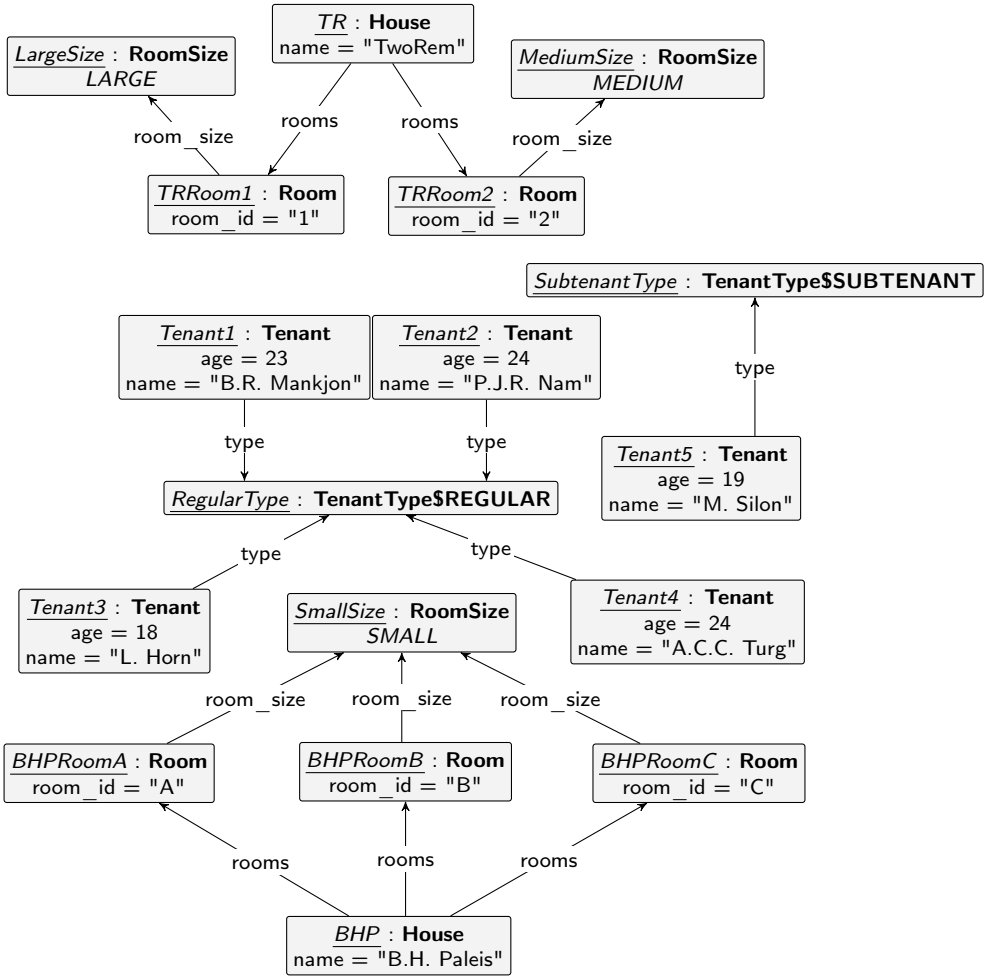


(a) Instance Model Im_{12}

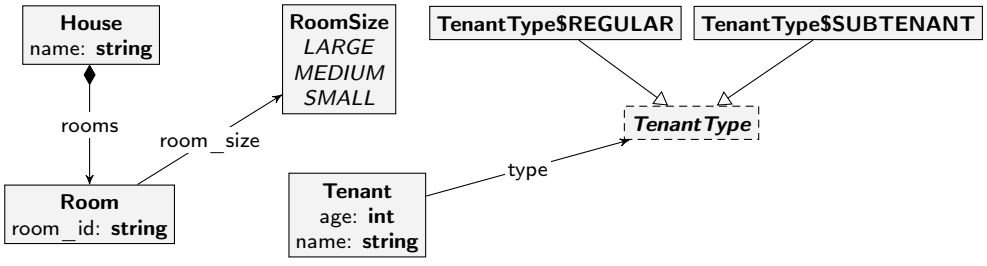


(b) Type Model Tm_{12}

Figure 6.24: The Ecore model after step 12



(a) Instance Graph IG_{12}



(b) Type Graph TG_{12}

Figure 6.25: The GROOVE graphs after step 12

Models after step 13

$Tm_{13} =$ $\text{combine}(Tm_{12}, Tm_{NullableClassField}) =$

$$Class = \{\text{.House}, \text{.Room}, \text{.Tenant}\}$$

$$Enum = \{\text{.RoomSize}, \text{.TenantType}\}$$

$$UserDataType = \{\}$$

$$Field = \{(\text{.House}, \text{name}), (\text{.House}, \text{rooms}), (\text{.Room}, \text{room\_id}),$$
$$(\text{.Room}, \text{room\_size}), (\text{.Room}, \text{tenant}), (\text{.Tenant}, \text{name}), (\text{.Tenant}, \text{age}),$$
$$(\text{.Tenant}, \text{type})\}$$

$$\text{FieldSig} = \big\{\big((\text{.House}, \text{name}), (\text{string}, 1..1)\big),$$
$$\big((\text{.House}, \text{rooms}), ([\text{setof}, \text{!.Room}], 0..9)\big),$$
$$\big((\text{.Room}, \text{room\_id}), (\text{string}, 1..1)\big),$$
$$\big((\text{.Room}, \text{room\_size}), (\text{.RoomSize}, 1..1)\big),$$
$$\big((\text{.Room}, \text{tenant}), (\text{?.Tenant}, 0..1)\big),$$
$$\big((\text{.Tenant}, \text{name}), (\text{string}, 1..1)\big),$$
$$\big((\text{.Tenant}, \text{age}), (\text{int}, 1..1)\big),$$
$$\big((\text{.Tenant}, \text{type}), (\text{.TenantType}, 1..1)\big)\big\}$$

$$EnumValue = \{(\text{.RoomSize}, \text{SMALL}), (\text{.RoomSize}, \text{MEDIUM}), (\text{.RoomSize}, \text{LARGE})$$
$$(\text{.TenantType}, \text{REGULAR}), (\text{.TenantType}, \text{SUBTENANT})\}$$

$$Inh = \{\}$$

$$Prop = \big\{\big(\text{containment}, (\text{.House}, \text{rooms})\big)\big\}$$

$$Constant = \{\}$$

$$\text{ConstType} = \{\}$$

$Im_{13} =$ $\text{combine}(Tm_{12}, Im_{NullableClassField}) =$

$$Object = \{TR, BHP, TRRoom1, TRRoom2, BHPRoomA, BHPRoomB,$$
$$BHPRoomC, Tenant1, Tenant2, Tenant3, Tenant4, Tenant5\}$$

$$\text{ObjectClass}(ob) = \{(TR, \text{.House}), (BHP, \text{.House}), (TRRoom1, \text{.Room}),$$
$$(TRRoom2, \text{.Room}), (BHPRoomA, \text{.Room}), (BHPRoomB, \text{.Room}),$$
$$(BHPRoomC, \text{.Room}, (Tenant1, \text{.Tenant}), (Tenant2, \text{.Tenant}),$$
$$(Tenant3, \text{.Tenant}), (Tenant4, \text{.Tenant}), (Tenant5, \text{.Tenant})\}$$

$$\text{ObjectId} = \{(TR, TR), (BHP, BHP), (TRRoom1, TRRoom1),$$
$$(TRRoom2, TRRoom2), (BHPRoomA, BHPRoomA),$$
$$(BHPRoomB, BHPRoomB), (BHPRoomC, BHPRoomC),$$
$$(Tenant1, Tenant1), (Tenant2, Tenant2), (Tenant3, Tenant3),$$
$$(Tenant4, Tenant4), (Tenant5, Tenant5)\}$$

$$\text{FieldValue} = \Big\{\Big(\big(TR, (\text{.House}, \text{name})\big), \big[\text{string}, \text{“TwoRem”}\big]\Big),$$
$$\Big(\big(BHP, (\text{.House}, \text{name})\big), \big[\text{string}, \text{“B.H. Paleis”}\big]\Big),$$
$$\Big(\big(TR, (\text{.House}, \text{rooms})\big), \big[\text{setof}, \big\langle[\text{obj}, TRRoom1],$$
$$[\text{obj}, TRRoom2]\big\rangle\big]\Big),$$
$$\Big(\big(BHP, (\text{.House}, \text{rooms})\big)\big[\text{setof}, \big\langle[\text{obj}, BHPRoomA],$$
$$[\text{obj}, BHPRoomB], (\text{obj}, BHPRoomC)\big\rangle\big]\Big),$$

Models after step 13

$$\Big((TRRoom1, (.\mathsf{Room}, \mathsf{room\_id})), [\mathsf{string}, \text{“1”}]\Big),$$
$$\Big((TRRoom2, (.\mathsf{Room}, \mathsf{room\_id})), [\mathsf{string}, \text{“2”}]\Big),$$
$$\Big((BHPRoomA, (.\mathsf{Room}, \mathsf{room\_id})), [\mathsf{string}, \text{“A”}]\Big),$$
$$\Big((BHPRoomB, (.\mathsf{Room}, \mathsf{room\_id})), [\mathsf{string}, \text{“B”}]\Big),$$
$$\Big((BHPRoomC, (.\mathsf{Room}, \mathsf{room\_id})), [\mathsf{string}, \text{“C”}]\Big),$$
$$\Big((TRRoom1, (.\mathsf{Room}, \mathsf{room\_size})), [\mathsf{enum}, (.\mathsf{RoomSize}, \mathsf{LARGE})]\Big),$$
$$\Big((TRRoom2, (.\mathsf{Room}, \mathsf{room\_size})), [\mathsf{enum}, (.\mathsf{RoomSize}, \mathsf{MEDIUM})]\Big),$$
$$\Big((BHPRoomA, (.\mathsf{Room}, \mathsf{room\_size})), [\mathsf{enum}, (.\mathsf{RoomSize}, \mathsf{SMALL})]\Big),$$
$$\Big((BHPRoomB, (.\mathsf{Room}, \mathsf{room\_size})), [\mathsf{enum}, (.\mathsf{RoomSize}, \mathsf{SMALL})]\Big),$$
$$\Big((BHPRoomC, (.\mathsf{Room}, \mathsf{room\_size})), [\mathsf{enum}, (.\mathsf{RoomSize}, \mathsf{SMALL})]\Big),$$
$$\Big((TRRoom1, (.\mathsf{Room}, \mathsf{tenant}), [\mathsf{obj}, Tenant1]\Big),$$
$$\Big((TRRoom2, (.\mathsf{Room}, \mathsf{tenant}), [\mathsf{obj}, Tenant2]\Big),$$
$$\Big((BHPRoomA, (.\mathsf{Room}, \mathsf{tenant}), [\mathsf{obj}, Tenant3]\Big),$$
$$\Big((BHPRoomB, (.\mathsf{Room}, \mathsf{tenant}), [\mathsf{nil}]\Big),$$
$$\Big((BHPRoomC, (.\mathsf{Room}, \mathsf{tenant}), [\mathsf{obj}, Tenant4]\Big),$$
$$\Big((Tenant1, (.\mathsf{Tenant}, \mathsf{name})), [\mathsf{string}, \text{“B.R. Mankjon”}]\Big),$$
$$\Big((Tenant2, (.\mathsf{Tenant}, \mathsf{name})), [\mathsf{string}, \text{“P.J.R. Nam”}]\Big),$$
$$\Big((Tenant3, (.\mathsf{Tenant}, \mathsf{name})), [\mathsf{string}, \text{“L. Horn”}]\Big),$$
$$\Big((Tenant4, (.\mathsf{Tenant}, \mathsf{name})), [\mathsf{string}, \text{“A.C.C. Turg”}]\Big),$$
$$\Big((Tenant5, (.\mathsf{Tenant}, \mathsf{name})), [\mathsf{string}, \text{“M. Silon”}]\Big),$$
$$\Big((Tenant1, (.\mathsf{Tenant}, \mathsf{age})), [\mathsf{int}, 23]\Big),$$
$$\Big((Tenant2, (.\mathsf{Tenant}, \mathsf{age})), [\mathsf{int}, 24]\Big),$$
$$\Big((Tenant3, (.\mathsf{Tenant}, \mathsf{age})), [\mathsf{int}, 18]\Big),$$
$$\Big((Tenant4, (.\mathsf{Tenant}, \mathsf{age})), [\mathsf{int}, 24]\Big),$$
$$\Big((Tenant5, (.\mathsf{Tenant}, \mathsf{age})), [\mathsf{int}, 19]\Big),$$
$$\Big((Tenant1, (.\mathsf{Tenant}, \mathsf{type})), [\mathsf{enum}, (.\mathsf{TenantType}, \mathsf{REGULAR})]\Big),$$
$$\Big((Tenant2, (.\mathsf{Tenant}, \mathsf{type})), [\mathsf{enum}, (.\mathsf{TenantType}, \mathsf{REGULAR})]\Big),$$
$$\Big((Tenant3, (.\mathsf{Tenant}, \mathsf{type})), [\mathsf{enum}, (.\mathsf{TenantType}, \mathsf{REGULAR})]\Big),$$
$$\Big((Tenant4, (.\mathsf{Tenant}, \mathsf{type})), [\mathsf{enum}, (.\mathsf{TenantType}, \mathsf{REGULAR})]\Big),$$
$$\Big((Tenant5, (.\mathsf{Tenant}, \mathsf{type})), [\mathsf{enum}, (.\mathsf{TenantType}, \mathsf{SUBTENANT})]\Big)\Big\}$$

$\mathrm{DefaultValue} = \{\}$

Models after step 13

$$
\begin{aligned}
TG_{13} = \quad & \text{combine}(TG_{12}, TG_{NullableClassField}) = \\
& NT = \{\langle \mathsf{House}\rangle, \langle \mathsf{Room}\rangle, \langle \mathsf{Tenant}\rangle, \langle \mathsf{RoomSize}\rangle, \langle \mathsf{TenantType}\rangle, \\
& \quad \langle \mathsf{TenantType}, \mathsf{REGULAR}\rangle, \langle \mathsf{TenantType}, \mathsf{SUBTENANT}\rangle, \mathsf{string}, \mathsf{int}\} \\
& ET = \{(\langle \mathsf{House}\rangle, \langle \mathsf{name}\rangle, \mathsf{string}), (\langle \mathsf{House}\rangle, \langle \mathsf{rooms}\rangle, \langle \mathsf{Room}\rangle), \\
& \quad (\langle \mathsf{Room}\rangle, \langle \mathsf{room\_id}\rangle, \mathsf{string}), (\langle \mathsf{Room}\rangle, \langle \mathsf{room\_size}\rangle, \langle \mathsf{RoomSize}\rangle), \\
& \quad (\langle \mathsf{Room}\rangle, \langle \mathsf{tenant}\rangle, \langle \mathsf{Tenant}\rangle), (\langle \mathsf{Tenant}\rangle, \langle \mathsf{name}\rangle, \mathsf{string}), \\
& \quad (\langle \mathsf{Tenant}\rangle, \langle \mathsf{age}\rangle, \mathsf{int}), (\langle \mathsf{Tenant}\rangle, \langle \mathsf{type}\rangle, \langle \mathsf{TenantType}\rangle), \\
& \quad (\langle \mathsf{RoomSize}\rangle, \langle \mathsf{SMALL}\rangle, \langle \mathsf{RoomSize}\rangle), \\
& \quad (\langle \mathsf{RoomSize}\rangle, \langle \mathsf{MEDIUM}\rangle, \langle \mathsf{RoomSize}\rangle), \\
& \quad (\langle \mathsf{RoomSize}\rangle, \langle \mathsf{LARGE}\rangle, \langle \mathsf{RoomSize}\rangle)\} \\
& \sqsubseteq = \{(\langle \mathsf{House}\rangle, \langle \mathsf{House}\rangle), (\langle \mathsf{Room}\rangle, \langle \mathsf{Room}\rangle), (\langle \mathsf{Tenant}\rangle, \langle \mathsf{Tenant}\rangle), \\
& \quad (\langle \mathsf{RoomSize}\rangle, \langle \mathsf{RoomSize}\rangle), (\langle \mathsf{TenantType}\rangle, \langle \mathsf{TenantType}\rangle), \\
& \quad (\langle \mathsf{TenantType}, \mathsf{REGULAR}\rangle, \langle \mathsf{TenantType}, \mathsf{REGULAR}\rangle), \\
& \quad (\langle \mathsf{TenantType}, \mathsf{SUBTENANT}\rangle, \langle \mathsf{TenantType}, \mathsf{SUBTENANT}\rangle), \\
& \quad (\langle \mathsf{TenantType}, \mathsf{REGULAR}\rangle, \langle \mathsf{TenantType}\rangle), \\
& \quad (\langle \mathsf{TenantType}, \mathsf{SUBTENANT}\rangle, \langle \mathsf{TenantType}\rangle), (\mathsf{string}, \mathsf{string}), (\mathsf{int}, \mathsf{int})\} \\
& abs = \{\langle \mathsf{TenantType}\rangle\} \\
& \text{mult} = \Big\{\Big((\langle \mathsf{House}\rangle, \langle \mathsf{name}\rangle, \mathsf{string}), (0..*, 1..1)\Big), \\
& \quad \Big((\langle \mathsf{House}\rangle, \langle \mathsf{rooms}\rangle, \langle \mathsf{Room}\rangle), (0..1, 0..9)\Big), \\
& \quad \Big((\langle \mathsf{Room}\rangle, \langle \mathsf{room\_id}\rangle, \mathsf{string}), (0..*, 1..1)\Big), \\
& \quad \Big((\langle \mathsf{Room}\rangle, \langle \mathsf{room\_size}\rangle, \langle \mathsf{RoomSize}\rangle), (0..*, 1..1)\Big), \\
& \quad \Big((\langle \mathsf{Room}\rangle, \langle \mathsf{tenant}\rangle, \langle \mathsf{Tenant}\rangle), (0..*, 0..1)\Big), \\
& \quad \Big((\langle \mathsf{Tenant}\rangle, \langle \mathsf{name}\rangle, \mathsf{string}), (0..*, 1..1)\Big), \\
& \quad \Big((\langle \mathsf{Tenant}\rangle, \langle \mathsf{age}\rangle, \mathsf{int}), (0..*, 0..1)\Big), \\
& \quad \Big((\langle \mathsf{Tenant}\rangle, \langle \mathsf{type}\rangle, \langle \mathsf{TenantType}\rangle), (0..*, 1..1)\Big), \\
& \quad \Big((\langle \mathsf{RoomSize}\rangle, \langle \mathsf{SMALL}\rangle, \langle \mathsf{RoomSize}\rangle), (0..1, 0..1)\Big), \\
& \quad \Big((\langle \mathsf{RoomSize}\rangle, \langle \mathsf{MEDIUM}\rangle, \langle \mathsf{RoomSize}\rangle), (0..1, 0..1)\Big), \\
& \quad \Big((\langle \mathsf{RoomSize}\rangle, \langle \mathsf{LARGE}\rangle, \langle \mathsf{RoomSize}\rangle), (0..1, 0..1)\Big)\Big\} \\
& contains = \{(\langle \mathsf{House}\rangle, \langle \mathsf{rooms}\rangle, \langle \mathsf{Room}\rangle)\}
\end{aligned}
$$

Models after step 13

$IG_{13} =$ $\text{combine}(TG_{12}, IG_{NullableClassField}) =$

$N = \{TR, BHP, TRRoom1, TRRoom2, BHPRoomA, BHPRoomB, BHPRoomC, Tenant1, Tenant2, Tenant3, Tenant4, Tenant5, SmallSize, MediumSize, LargeSize, RegularType, SubtenantType,$ "TwoRem", "B.H. Paleis", "1", "2", "A", "B", "C", "B.R. Mankjon", "P.J.R. Nam", "L. Horn", "A.C.C. Turg", "M. Silon", 23, 24, 18, 19$\}$

$E = \Big\{ \Big(TR, \big(\langle\mathsf{House}\rangle, \langle\mathsf{name}\rangle, \mathsf{string}\big), \text{"TwoRem"}\Big),$

$\Big(BHP, \big(\langle\mathsf{House}\rangle, \langle\mathsf{name}\rangle, \mathsf{string}\big), \text{"B.H. Paleis"}\Big),$

$\Big(TR, \big(\langle\mathsf{House}\rangle, \langle\mathsf{rooms}\rangle, \langle\mathsf{Room}\rangle\big), TRRoom1\Big),$

$\Big(TR, \big(\langle\mathsf{House}\rangle, \langle\mathsf{rooms}\rangle, \langle\mathsf{Room}\rangle\big), TRRoom2\Big),$

$\Big(BHP, \big(\langle\mathsf{House}\rangle, \langle\mathsf{rooms}\rangle, \langle\mathsf{Room}\rangle\big), BHPRoomA\Big),$

$\Big(BHP, \big(\langle\mathsf{House}\rangle, \langle\mathsf{rooms}\rangle, \langle\mathsf{Room}\rangle\big), BHPRoomB\Big),$

$\Big(BHP, \big(\langle\mathsf{House}\rangle, \langle\mathsf{rooms}\rangle, \langle\mathsf{Room}\rangle\big), BHPRoomC\Big),$

$\Big(TRRoom1, \big(\langle\mathsf{Room}\rangle, \langle\mathsf{room\_id}\rangle, \mathsf{string}\big), \text{"1"}\Big),$

$\Big(TRRoom2, \big(\langle\mathsf{Room}\rangle, \langle\mathsf{room\_id}\rangle, \mathsf{string}\big), \text{"2"}\Big),$

$\Big(BHPRoomA, \big(\langle\mathsf{Room}\rangle, \langle\mathsf{room\_id}\rangle, \mathsf{string}\big), \text{"A"}\Big),$

$\Big(BHPRoomB, \big(\langle\mathsf{Room}\rangle, \langle\mathsf{room\_id}\rangle, \mathsf{string}\big), \text{"B"}\Big),$

$\Big(BHPRoomC, \big(\langle\mathsf{Room}\rangle, \langle\mathsf{room\_id}\rangle, \mathsf{string}\big), \text{"C"}\Big),$

$\Big(TRRoom1, \big(\langle\mathsf{Room}\rangle, \langle\mathsf{tenant}\rangle, \langle\mathsf{Tenant}\rangle\big), Tenant1\Big),$

$\Big(TRRoom2, \big(\langle\mathsf{Room}\rangle, \langle\mathsf{tenant}\rangle, \langle\mathsf{Tenant}\rangle\big), Tenant2\Big),$

$\Big(BHPRoomA, \big(\langle\mathsf{Room}\rangle, \langle\mathsf{tenant}\rangle, \langle\mathsf{Tenant}\rangle\big), Tenant3\Big),$

$\Big(BHPRoomC, \big(\langle\mathsf{Room}\rangle, \langle\mathsf{tenant}\rangle, \langle\mathsf{Tenant}\rangle\big), Tenant4\Big),$

$\Big(TRRoom1, \big(\langle\mathsf{Room}\rangle, \langle\mathsf{room\_size}\rangle, \langle\mathsf{RoomSize}\rangle\big), LargeSize\Big),$

$\Big(TRRoom2, \big(\langle\mathsf{Room}\rangle, \langle\mathsf{room\_size}\rangle, \langle\mathsf{RoomSize}\rangle\big), MediumSize\Big),$

$\Big(BHPRoomA, \big(\langle\mathsf{Room}\rangle, \langle\mathsf{room\_size}\rangle, \langle\mathsf{RoomSize}\rangle\big), SmallSize\Big),$

$\Big(BHPRoomB, \big(\langle\mathsf{Room}\rangle, \langle\mathsf{room\_size}\rangle, \langle\mathsf{RoomSize}\rangle\big), SmallSize\Big),$

$\Big(BHPRoomC, \big(\langle\mathsf{Room}\rangle, \langle\mathsf{room\_size}\rangle, \langle\mathsf{RoomSize}\rangle\big), SmallSize\Big),$

$\Big(Tenant1, \big(\langle\mathsf{Tenant}\rangle, \langle\mathsf{name}\rangle, \mathsf{string}\big), \text{"B.R. Mankjon"}\Big),$

$\Big(Tenant2, \big(\langle\mathsf{Tenant}\rangle, \langle\mathsf{name}\rangle, \mathsf{string}\big), \text{"P.J.R. Nam"}\Big),$

$\Big(Tenant3, \big(\langle\mathsf{Tenant}\rangle, \langle\mathsf{name}\rangle, \mathsf{string}\big), \text{"L. Horn"}\Big),$

$\Big(Tenant4, \big(\langle\mathsf{Tenant}\rangle, \langle\mathsf{name}\rangle, \mathsf{string}\big), \text{"A.C.C. Turg"}\Big),$

$\Big(Tenant5, \big(\langle\mathsf{Tenant}\rangle, \langle\mathsf{name}\rangle, \mathsf{string}\big), \text{"M. Silon"}\Big),$

$\Big(Tenant1, \big(\langle\mathsf{Tenant}\rangle, \langle\mathsf{age}\rangle, \mathsf{int}\big), 23\Big), \Big(Tenant2, \big(\langle\mathsf{Tenant}\rangle, \langle\mathsf{age}\rangle, \mathsf{int}\big), 24\Big),$

$\Big(Tenant3, \big(\langle\mathsf{Tenant}\rangle, \langle\mathsf{age}\rangle, \mathsf{int}\big), 18\Big), \Big(Tenant4, \big(\langle\mathsf{Tenant}\rangle, \langle\mathsf{age}\rangle, \mathsf{int}\big), 24\Big),$

$\Big(Tenant5, \big(\langle\mathsf{Tenant}\rangle, \langle\mathsf{age}\rangle, \mathsf{int}\big), 19\Big),$

Models after step 13

$$\Big(Tenant1, \big(\langle\mathsf{Tenant}\rangle, \langle\mathsf{type}\rangle, \langle\mathsf{TenantType}\rangle\big), RegularType\Big),$$
$$\Big(Tenant2, \big(\langle\mathsf{Tenant}\rangle, \langle\mathsf{type}\rangle, \langle\mathsf{TenantType}\rangle\big), RegularType\Big),$$
$$\Big(Tenant3, \big(\langle\mathsf{Tenant}\rangle, \langle\mathsf{type}\rangle, \langle\mathsf{TenantType}\rangle\big), RegularType\Big),$$
$$\Big(Tenant4, \big(\langle\mathsf{Tenant}\rangle, \langle\mathsf{type}\rangle, \langle\mathsf{TenantType}\rangle\big), RegularType\Big),$$
$$\Big(Tenant5, \big(\langle\mathsf{Tenant}\rangle, \langle\mathsf{type}\rangle, \langle\mathsf{TenantType}\rangle\big), SubtenantType\Big),$$
$$\Big(SmallSize, \big(\langle\mathsf{RoomSize}\rangle, \langle\mathsf{SMALL}\rangle, \langle\mathsf{RoomSize}\rangle\big), SmallSize\Big),$$
$$\Big(MediumSize, \big(\langle\mathsf{RoomSize}\rangle, \langle\mathsf{MEDIUM}\rangle, \langle\mathsf{RoomSize}\rangle\big), MediumSize\Big),$$
$$\Big(LargeSize, \big(\langle\mathsf{RoomSize}\rangle, \langle\mathsf{LARGE}\rangle, \langle\mathsf{RoomSize}\rangle\big), LargeSize\Big)\Big\}$$

$$\text{ident} = \{(TR, TR), (BHP, BHP), (TRRoom1, TRRoom1), (TRRoom2, TRRoom2),$$
$$(BHPRoomA, BHPRoomA), (BHPRoomB, BHPRoomB),$$
$$(BHPRoomC, BHPRoomC), (Tenant1, Tenant1), (Tenant2, Tenant2),$$
$$(Tenant3, Tenant3), (Tenant4, Tenant4), (Tenant5, Tenant5),$$
$$(SmallSize, SmallSize), (MediumSize, MediumSize),$$
$$(LargeSize, LargeSize), (RegularType, RegularType),$$
$$(SubtenantType, SubtenantType)\}$$

$$\text{type}_n = \{(TR, \langle\mathsf{House}\rangle), (BHP, \langle\mathsf{House}\rangle), (TRRoom1, \langle\mathsf{Room}\rangle), (TRRoom2, \langle\mathsf{Room}\rangle),$$
$$(BHPRoomA, \langle\mathsf{Room}\rangle), (BHPRoomB, \langle\mathsf{Room}\rangle), (BHPRoomC, \langle\mathsf{Room}\rangle),$$
$$(Tenant1, \langle\mathsf{Tenant}\rangle), (Tenant2, \langle\mathsf{Tenant}\rangle), (Tenant3, \langle\mathsf{Tenant}\rangle),$$
$$(Tenant4, \langle\mathsf{Tenant}\rangle), (Tenant5, \langle\mathsf{Tenant}\rangle), (SmallSize, \langle\mathsf{RoomSize}\rangle),$$
$$(MediumSize, \langle\mathsf{RoomSize}\rangle), (LargeSize, \langle\mathsf{RoomSize}\rangle),$$
$$(RegularType, \langle\mathsf{TenantType}, \mathsf{REGULAR}\rangle),$$
$$(SubtenantType, \langle\mathsf{TenantType}, \mathsf{SUBTENANT}\rangle), (\text{“TwoRem”}, \mathsf{string}),$$
$$(\text{“B.H. Paleis”}, \mathsf{string}), (\text{“1”}, \mathsf{string}), (\text{“2”}, \mathsf{string}), (\text{“A”}, \mathsf{string}), (\text{“B”}, \mathsf{string}),$$
$$(\text{“C”}, \mathsf{string}), (\text{“B.R. Mankjon”}, \mathsf{string}), (\text{“P.J.R. Nam”}, \mathsf{string}),$$
$$(\text{“A.C.C. Turg”}, \mathsf{string}), (\text{“L. Horn”}, \mathsf{string}), (\text{“M. Silon”}, \mathsf{string}), (23, \mathsf{int}),$$
$$(24, \mathsf{int}), (18, \mathsf{int}), (19, \mathsf{int})\}$$

$f_{13}(Im_{13}) = f_{12}(Im_{12}) \sqcup f_{NullableClassField}(Im_{NullableClassField})$ (Definition 4.4.27)
$f'_{13}(IG_{13}) = f'_{12}(IG_{12}) \sqcup f'_{NullableClassField}(IG_{NullableClassField})$ (Definition 4.4.32)

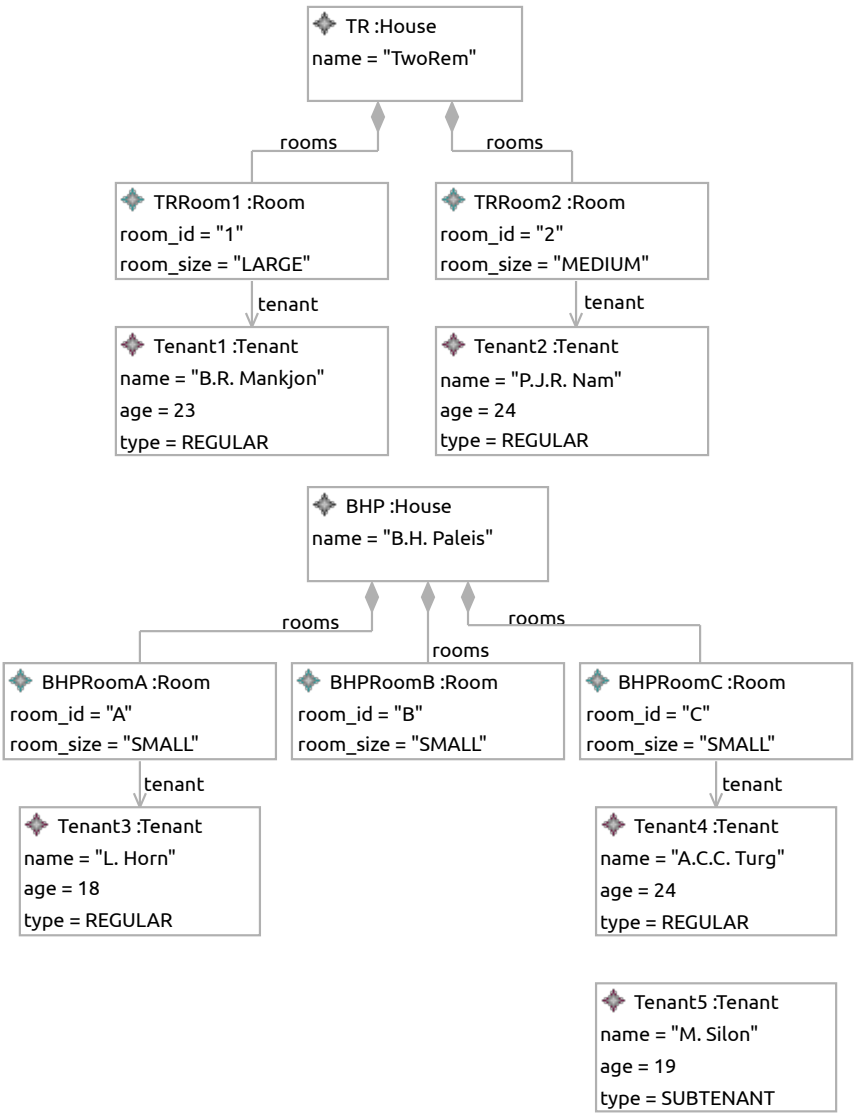
A visual representation of Tm_{13} and Im_{13} can be found in Figure 6.26. Similarly, a visual representation of TG_{13} and IG_{13} can be found in Figure 6.27. Please note that because of the definitions of $f_{13}(Im_{13})$ and $f'_{13}(IG_{13})$, we have that $f_{13}(Im_{13}) = IG_{13}$ and $f'_{13}(IG_{13}) = Im_{13}$. Furthermore, $f_{13}(Im_{13})$ and $f'_{13}(IG_{13})$ are valid mapping functions themselves, such that they can be combined with another mapping function in the next step.

The visualisation shows the references between the rooms and the tenants. This transformation shows how existing objects can be referenced by a new field, while also showing that values can be null. For example, $BHPRoomB$ has no tenant, which can be seen by the absence of a *tenant* relation in the visualisation.

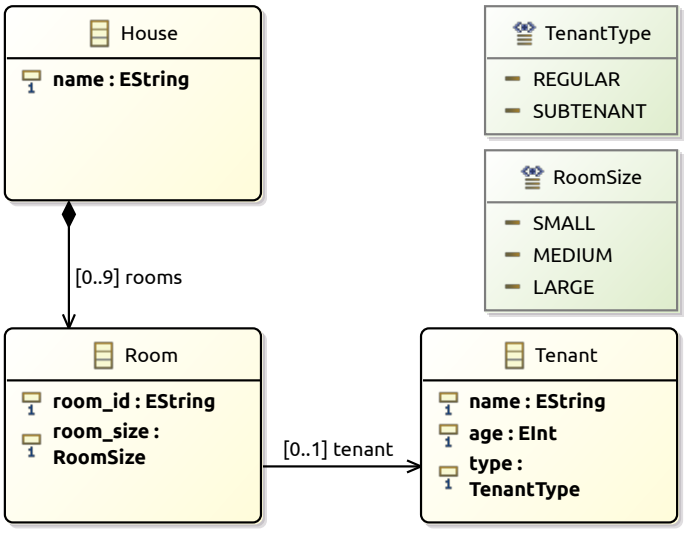
6.2.14 Tenant & subtenant relationship

In the fourteenth step, another relationship between two objects is introduced. For this relation, a object of .Tenant may reference a single .Tenant object. Section 5.2.8 is used to introduce the field on the type level, while on the instance level, Section 5.3.8 is used to introduce the values.

The *classtype* of the new field is .Tenant, as the field will be defined for tenants. The *name* of the new field is subtenant and the $fieldtype$ is equal to the *classtype*, .Tenant. The set of objects that will receive a value is defined as $valobjects = \{Tenant4\}$, while the set of objects that will not receive a value is defined as $nilobjects = \{Tenant1, Tenant2, Tenant3, Tenant5\}$. The union of these sets is equal to all the tenant objects, as expected. The function for *obids* returns the existing identifier of each of these

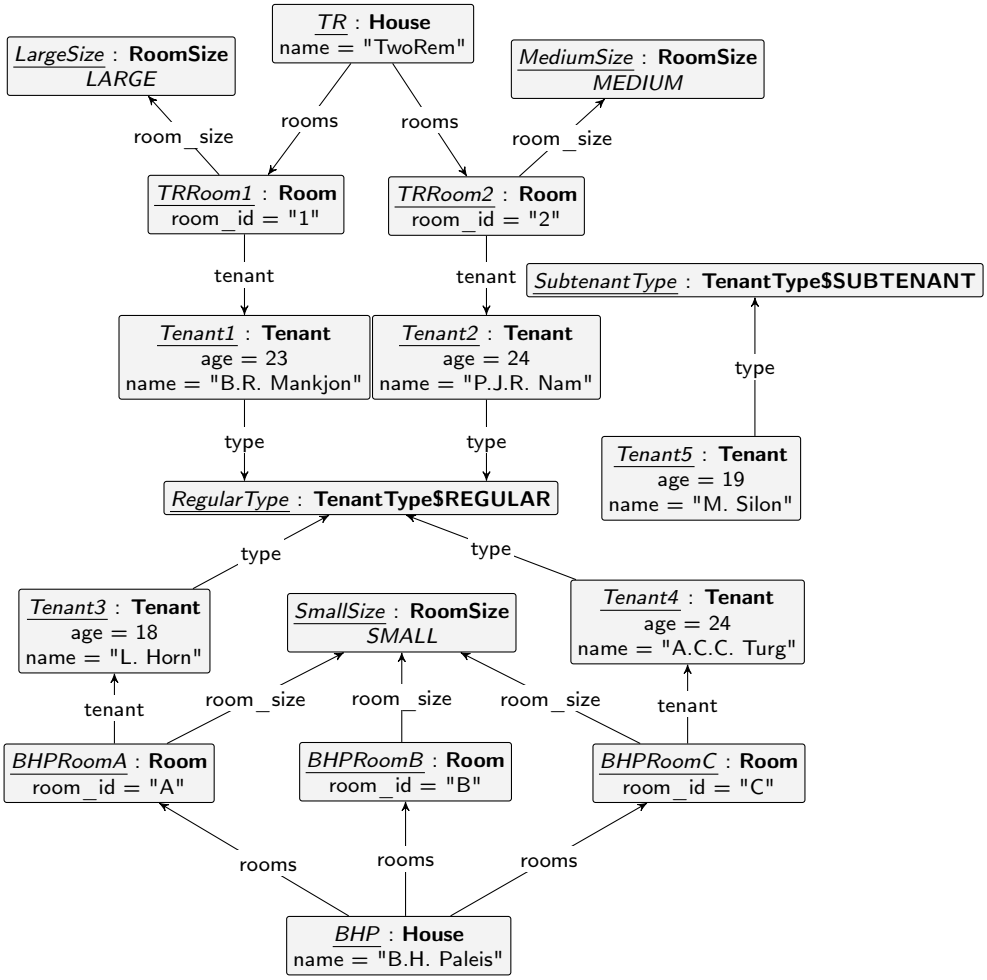


(a) Instance Model Im_{13}

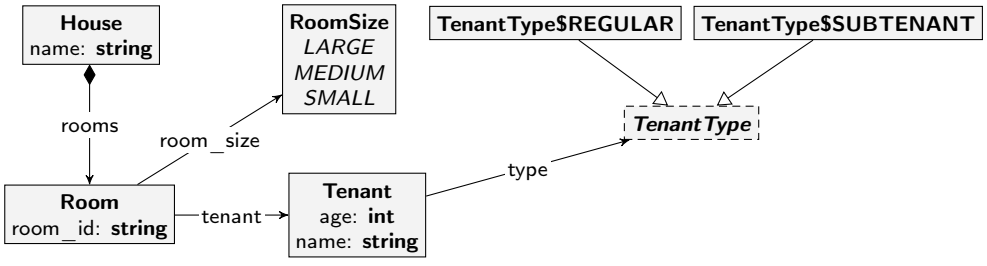


(b) Type Model Tm_{13}

Figure 6.26: The Ecore model after step 13



(a) Instance Graph IG_{13}



(b) Type Graph TG_{13}

Figure 6.27: The GROOVE graphs after step 13

objects. The *values* function is defined as follows:

$$values = \{(Tenant4, Tenant5)$$

For the objects referenced by the objects in *valobjects*, the function *obids* returns their existing identifier. The following model is obtained:

Models after step 14

$Tm_{14} = \text{combine}(Tm_{13}, Tm_{NullableClassField}) =$

$Class = \{.\mathsf{House}, .\mathsf{Room}, .\mathsf{Tenant}\}$

$Enum = \{.\mathsf{RoomSize}, .\mathsf{TenantType}\}$

$UserDataType = \{\}$

$Field = \{(.\mathsf{House}, \mathsf{name}), (.\mathsf{House}, \mathsf{rooms}), (.\mathsf{Room}, \mathsf{room\_id}),$
$(.\mathsf{Room}, \mathsf{room\_size}), (.\mathsf{Room}, \mathsf{tenant}), (.\mathsf{Tenant}, \mathsf{name}), (.\mathsf{Tenant}, \mathsf{age}),$
$(.\mathsf{Tenant}, \mathsf{type}), (.\mathsf{Tenant}, \mathsf{subtenant})\}$

$\text{FieldSig} = \{\big((.\mathsf{House}, \mathsf{name}), (\mathsf{string}, 1..1)\big),$
$\big((.\mathsf{House}, \mathsf{rooms}), ([\mathsf{setof}, !.\mathsf{Room}], 0..9)\big),$
$\big((.\mathsf{Room}, \mathsf{room\_id}), (\mathsf{string}, 1..1)\big),$
$\big((.\mathsf{Room}, \mathsf{room\_size}), (.\mathsf{RoomSize}, 1..1)\big),$
$\big((.\mathsf{Room}, \mathsf{tenant}), (?.\mathsf{Tenant}, 0..1)\big),$
$\big((.\mathsf{Tenant}, \mathsf{name}), (\mathsf{string}, 1..1)\big),$
$\big((.\mathsf{Tenant}, \mathsf{age}), (\mathsf{int}, 1..1)\big),$
$\big((.\mathsf{Tenant}, \mathsf{type}), (.\mathsf{TenantType}, 1..1)\big),$
$\big((.\mathsf{Tenant}, \mathsf{subtenant}), (?.\mathsf{Tenant}, 0..1)\big)\}$

$EnumValue = \{(.\mathsf{RoomSize}, \mathsf{SMALL}), (.\mathsf{RoomSize}, \mathsf{MEDIUM}), (.\mathsf{RoomSize}, \mathsf{LARGE})$
$(.\mathsf{TenantType}, \mathsf{REGULAR}), (.\mathsf{TenantType}, \mathsf{SUBTENANT})\}$

$Inh = \{\}$

$Prop = \{\big(\mathsf{containment}, (.\mathsf{House}, \mathsf{rooms})\big)\}$

$Constant = \{\}$

$\text{ConstType} = \{\}$

Models after step 14

$$
\begin{aligned}
Im_{14} = \quad & \text{combine}(Tm_{13}, Im_{NullableClassField}) = \\
& Object = \{TR, BHP, TRRoom1, TRRoom2, BHPRoomA, BHPRoomB, \\
& \quad BHPRoomC, Tenant1, Tenant2, Tenant3, Tenant4, Tenant5\} \\
& \text{ObjectClass}(ob) = \{(TR, .\mathsf{House}), (BHP, .\mathsf{House}), (TRRoom1, .\mathsf{Room}), \\
& \quad (TRRoom2, .\mathsf{Room}), (BHPRoomA, .\mathsf{Room}), (BHPRoomB, .\mathsf{Room}), \\
& \quad (BHPRoomC, .\mathsf{Room}, (Tenant1, .\mathsf{Tenant}), (Tenant2, .\mathsf{Tenant}), \\
& \quad (Tenant3, .\mathsf{Tenant}), (Tenant4, .\mathsf{Tenant}), (Tenant5, .\mathsf{Tenant})\} \\
& \text{ObjectId} = \{(TR, TR), (BHP, BHP), (TRRoom1, TRRoom1), \\
& \quad (TRRoom2, TRRoom2), (BHPRoomA, BHPRoomA), \\
& \quad (BHPRoomB, BHPRoomB), (BHPRoomC, BHPRoomC), \\
& \quad (Tenant1, Tenant1), (Tenant2, Tenant2), (Tenant3, Tenant3), \\
& \quad (Tenant4, Tenant4), (Tenant5, Tenant5)\} \\
& \text{FieldValue} = \Big\{\Big(\big(TR, (.\mathsf{House}, \mathsf{name})\big), \big[\mathsf{string}, \text{“TwoRem”}\big]\Big), \\
& \quad \Big(\big(BHP, (.\mathsf{House}, \mathsf{name})\big), \big[\mathsf{string}, \text{“B.H. Paleis”}\big]\Big), \\
& \quad \Big(\big(TR, (.\mathsf{House}, \mathsf{rooms})\big), \big[\mathsf{setof}, \big\langle[\mathsf{obj}, TRRoom1], \\
& \qquad [\mathsf{obj}, TRRoom2]\big\rangle\big]\Big), \\
& \quad \Big(\big(BHP, (.\mathsf{House}, \mathsf{rooms})\big)\big[\mathsf{setof}, \big\langle[\mathsf{obj}, BHPRoomA], \\
& \qquad [\mathsf{obj}, BHPRoomB], (\mathsf{obj}, BHPRoomC)\big\rangle\big]\Big), \\
& \quad \Big(\big(TRRoom1, (.\mathsf{Room}, \mathsf{room\_id})\big), \big[\mathsf{string}, \text{“1”}\big]\Big), \\
& \quad \Big(\big(TRRoom2, (.\mathsf{Room}, \mathsf{room\_id})\big), \big[\mathsf{string}, \text{“2”}\big]\Big),
\end{aligned}
$$

Models after step 14

$$\Big((BHPRoomA, (.\mathsf{Room}, \mathsf{room\_id})), [\mathsf{string}, \text{“A”}]\Big),$$
$$\Big((BHPRoomB, (.\mathsf{Room}, \mathsf{room\_id})), [\mathsf{string}, \text{“B”}]\Big),$$
$$\Big((BHPRoomC, (.\mathsf{Room}, \mathsf{room\_id})), [\mathsf{string}, \text{“C”}]\Big),$$
$$\Big((TRRoom1, (.\mathsf{Room}, \mathsf{room\_size})), [\mathsf{enum}, (.\mathsf{RoomSize}, \mathsf{LARGE})]\Big),$$
$$\Big((TRRoom2, (.\mathsf{Room}, \mathsf{room\_size})), [\mathsf{enum}, (.\mathsf{RoomSize}, \mathsf{MEDIUM})]\Big),$$
$$\Big((BHPRoomA, (.\mathsf{Room}, \mathsf{room\_size})), [\mathsf{enum}, (.\mathsf{RoomSize}, \mathsf{SMALL})]\Big),$$
$$\Big((BHPRoomB, (.\mathsf{Room}, \mathsf{room\_size})), [\mathsf{enum}, (.\mathsf{RoomSize}, \mathsf{SMALL})]\Big),$$
$$\Big((BHPRoomC, (.\mathsf{Room}, \mathsf{room\_size})), [\mathsf{enum}, (.\mathsf{RoomSize}, \mathsf{SMALL})]\Big),$$
$$\Big((TRRoom1, (.\mathsf{Room}, \mathsf{tenant}), [\mathsf{obj}, Tenant1]\Big),$$
$$\Big((TRRoom2, (.\mathsf{Room}, \mathsf{tenant}), [\mathsf{obj}, Tenant2]\Big),$$
$$\Big((BHPRoomA, (.\mathsf{Room}, \mathsf{tenant}), [\mathsf{obj}, Tenant3]\Big),$$
$$\Big((BHPRoomB, (.\mathsf{Room}, \mathsf{tenant}), [\mathsf{nil}]\Big),$$
$$\Big((BHPRoomC, (.\mathsf{Room}, \mathsf{tenant}), [\mathsf{obj}, Tenant4]\Big),$$
$$\Big((Tenant1, (.\mathsf{Tenant}, \mathsf{name})), [\mathsf{string}, \text{“B.R. Mankjon”}]\Big),$$
$$\Big((Tenant2, (.\mathsf{Tenant}, \mathsf{name})), [\mathsf{string}, \text{“P.J.R. Nam”}]\Big),$$
$$\Big((Tenant3, (.\mathsf{Tenant}, \mathsf{name})), [\mathsf{string}, \text{“L. Horn”}]\Big),$$
$$\Big((Tenant4, (.\mathsf{Tenant}, \mathsf{name})), [\mathsf{string}, \text{“A.C.C. Turg”}]\Big),$$
$$\Big((Tenant5, (.\mathsf{Tenant}, \mathsf{name})), [\mathsf{string}, \text{“M. Silon”}]\Big),$$
$$\Big((Tenant1, (.\mathsf{Tenant}, \mathsf{age})), [\mathsf{int}, 23]\Big),$$
$$\Big((Tenant2, (.\mathsf{Tenant}, \mathsf{age})), [\mathsf{int}, 24]\Big),$$
$$\Big((Tenant3, (.\mathsf{Tenant}, \mathsf{age})), [\mathsf{int}, 18]\Big),$$
$$\Big((Tenant4, (.\mathsf{Tenant}, \mathsf{age})), [\mathsf{int}, 24]\Big),$$
$$\Big((Tenant5, (.\mathsf{Tenant}, \mathsf{age})), [\mathsf{int}, 19]\Big),$$
$$\Big((Tenant1, (.\mathsf{Tenant}, \mathsf{type})), [\mathsf{enum}, (.\mathsf{TenantType}, \mathsf{REGULAR})]\Big),$$
$$\Big((Tenant2, (.\mathsf{Tenant}, \mathsf{type})), [\mathsf{enum}, (.\mathsf{TenantType}, \mathsf{REGULAR})]\Big),$$
$$\Big((Tenant3, (.\mathsf{Tenant}, \mathsf{type})), [\mathsf{enum}, (.\mathsf{TenantType}, \mathsf{REGULAR})]\Big),$$
$$\Big((Tenant4, (.\mathsf{Tenant}, \mathsf{type})), [\mathsf{enum}, (.\mathsf{TenantType}, \mathsf{REGULAR})]\Big),$$
$$\Big((Tenant5, (.\mathsf{Tenant}, \mathsf{type})), [\mathsf{enum}, (.\mathsf{TenantType}, \mathsf{SUBTENANT})]\Big),$$
$$\Big((Tenant1, (.\mathsf{Tenant}, \mathsf{subtenant})), [\mathsf{nil}]\Big),$$
$$\Big((Tenant2, (.\mathsf{Tenant}, \mathsf{subtenant})), [\mathsf{nil}]\Big),$$
$$\Big((Tenant3, (.\mathsf{Tenant}, \mathsf{subtenant})), [\mathsf{nil}]\Big),$$
$$\Big((Tenant4, (.\mathsf{Tenant}, \mathsf{subtenant})), [\mathsf{obj}, Tenant3]\Big),$$
$$\Big((Tenant5, (.\mathsf{Tenant}, \mathsf{subtenant})), [\mathsf{nil}]\Big)\Big\}$$

$\mathrm{DefaultValue} = \{\}$

Models after step 14

$$
\begin{aligned}
TG_{14} = \quad & \text{combine}(TG_{13}, TG_{NullableClassField}) = \\
& NT = \{\langle\mathsf{House}\rangle, \langle\mathsf{Room}\rangle, \langle\mathsf{Tenant}\rangle, \langle\mathsf{RoomSize}\rangle, \langle\mathsf{TenantType}\rangle, \\
& \quad \langle\mathsf{TenantType}, \mathsf{REGULAR}\rangle, \langle\mathsf{TenantType}, \mathsf{SUBTENANT}\rangle, \mathsf{string}, \mathsf{int}\} \\
& ET = \big\{\big(\langle\mathsf{House}\rangle, \langle\mathsf{name}\rangle, \mathsf{string}\big), \big(\langle\mathsf{House}\rangle, \langle\mathsf{rooms}\rangle, \langle\mathsf{Room}\rangle\big), \\
& \quad \big(\langle\mathsf{Room}\rangle, \langle\mathsf{room\_id}\rangle, \mathsf{string}\big), \big(\langle\mathsf{Room}\rangle, \langle\mathsf{room\_size}\rangle, \langle\mathsf{RoomSize}\rangle\big), \\
& \quad \big(\langle\mathsf{Room}\rangle, \langle\mathsf{tenant}\rangle, \langle\mathsf{Tenant}\rangle\big), \big(\langle\mathsf{Tenant}\rangle, \langle\mathsf{name}\rangle, \mathsf{string}\big), \\
& \quad \big(\langle\mathsf{Tenant}\rangle, \langle\mathsf{age}\rangle, \mathsf{int}\big), \big(\langle\mathsf{Tenant}\rangle, \langle\mathsf{type}\rangle, \langle\mathsf{TenantType}\rangle\big), \\
& \quad \big(\langle\mathsf{Tenant}\rangle, \langle\mathsf{subtenant}\rangle, \langle\mathsf{Tenant}\rangle\big), \big(\langle\mathsf{RoomSize}\rangle, \langle\mathsf{SMALL}\rangle, \langle\mathsf{RoomSize}\rangle\big), \\
& \quad \big(\langle\mathsf{RoomSize}\rangle, \langle\mathsf{MEDIUM}\rangle, \langle\mathsf{RoomSize}\rangle\big), \\
& \quad \big(\langle\mathsf{RoomSize}\rangle, \langle\mathsf{LARGE}\rangle, \langle\mathsf{RoomSize}\rangle\big)\big\} \\
& \sqsubseteq = \big\{\big(\langle\mathsf{House}\rangle, \langle\mathsf{House}\rangle\big), \big(\langle\mathsf{Room}\rangle, \langle\mathsf{Room}\rangle\big), \big(\langle\mathsf{Tenant}\rangle, \langle\mathsf{Tenant}\rangle\big), \\
& \quad \big(\langle\mathsf{RoomSize}\rangle, \langle\mathsf{RoomSize}\rangle\big), \big(\langle\mathsf{TenantType}\rangle, \langle\mathsf{TenantType}\rangle\big), \\
& \quad \big(\langle\mathsf{TenantType}, \mathsf{REGULAR}\rangle, \langle\mathsf{TenantType}, \mathsf{REGULAR}\rangle\big), \\
& \quad \big(\langle\mathsf{TenantType}, \mathsf{SUBTENANT}\rangle, \langle\mathsf{TenantType}, \mathsf{SUBTENANT}\rangle\big), \\
& \quad \big(\langle\mathsf{TenantType}, \mathsf{REGULAR}\rangle, \langle\mathsf{TenantType}\rangle\big), \\
& \quad \big(\langle\mathsf{TenantType}, \mathsf{SUBTENANT}\rangle, \langle\mathsf{TenantType}\rangle\big), \big(\mathsf{string}, \mathsf{string}\big), \big(\mathsf{int}, \mathsf{int}\big)\big\} \\
& abs = \{\langle\mathsf{TenantType}\rangle\} \\
& \text{mult} = \Big\{\Big(\big(\langle\mathsf{House}\rangle, \langle\mathsf{name}\rangle, \mathsf{string}\big), (0..*, 1..1)\Big), \\
& \quad \Big(\big(\langle\mathsf{House}\rangle, \langle\mathsf{rooms}\rangle, \langle\mathsf{Room}\rangle\big), (0..1, 0..9)\Big), \\
& \quad \Big(\big(\langle\mathsf{Room}\rangle, \langle\mathsf{room\_id}\rangle, \mathsf{string}\big), (0..*, 1..1)\Big), \\
& \quad \Big(\big(\langle\mathsf{Room}\rangle, \langle\mathsf{room\_size}\rangle, \langle\mathsf{RoomSize}\rangle\big), (0..*, 1..1)\Big), \\
& \quad \Big(\big(\langle\mathsf{Room}\rangle, \langle\mathsf{tenant}\rangle, \langle\mathsf{Tenant}\rangle\big), (0..*, 0..1)\Big), \\
& \quad \Big(\big(\langle\mathsf{Tenant}\rangle, \langle\mathsf{name}\rangle, \mathsf{string}\big), (0..*, 1..1)\Big), \\
& \quad \Big(\big(\langle\mathsf{Tenant}\rangle, \langle\mathsf{age}\rangle, \mathsf{int}\big), (0..*, 0..1)\Big), \\
& \quad \Big(\big(\langle\mathsf{Tenant}\rangle, \langle\mathsf{type}\rangle, \langle\mathsf{TenantType}\rangle\big), (0..*, 1..1)\Big), \\
& \quad \Big(\big(\langle\mathsf{Tenant}\rangle, \langle\mathsf{subtenant}\rangle, \langle\mathsf{Tenant}\rangle\big), (0..*, 0..1)\Big), \\
& \quad \Big(\big(\langle\mathsf{RoomSize}\rangle, \langle\mathsf{SMALL}\rangle, \langle\mathsf{RoomSize}\rangle\big), (0..1, 0..1)\Big), \\
& \quad \Big(\big(\langle\mathsf{RoomSize}\rangle, \langle\mathsf{MEDIUM}\rangle, \langle\mathsf{RoomSize}\rangle\big), (0..1, 0..1)\Big), \\
& \quad \Big(\big(\langle\mathsf{RoomSize}\rangle, \langle\mathsf{LARGE}\rangle, \langle\mathsf{RoomSize}\rangle\big), (0..1, 0..1)\Big)\Big\} \\
& contains = \big\{\big(\langle\mathsf{House}\rangle, \langle\mathsf{rooms}\rangle, \langle\mathsf{Room}\rangle\big)\big\}
\end{aligned}
$$

Models after step 14

$IG_{14} =$ $\text{combine}(TG_{13}, IG_{NullableClassField}) =$

$N = \{TR, BHP, TRRoom1, TRRoom2, BHPRoomA, BHPRoomB, BHPRoomC, Tenant1, Tenant2, Tenant3, Tenant4, Tenant5, SmallSize, MediumSize, LargeSize, RegularType, SubtenantType,$ "TwoRem", "B.H. Paleis", "1", "2", "A", "B", "C", "B.R. Mankjon", "P.J.R. Nam", "L. Horn", "A.C.C. Turg", "M. Silon", 23, 24, 18, 19$\}$

$E = \Big\{ \Big(TR, \big(\langle\mathsf{House}\rangle, \langle\mathsf{name}\rangle, \mathsf{string}\big), \text{"TwoRem"}\Big),$

$\Big(BHP, \big(\langle\mathsf{House}\rangle, \langle\mathsf{name}\rangle, \mathsf{string}\big), \text{"B.H. Paleis"}\Big),$

$\Big(TR, \big(\langle\mathsf{House}\rangle, \langle\mathsf{rooms}\rangle, \langle\mathsf{Room}\rangle\big), TRRoom1\Big),$

$\Big(TR, \big(\langle\mathsf{House}\rangle, \langle\mathsf{rooms}\rangle, \langle\mathsf{Room}\rangle\big), TRRoom2\Big),$

$\Big(BHP, \big(\langle\mathsf{House}\rangle, \langle\mathsf{rooms}\rangle, \langle\mathsf{Room}\rangle\big), BHPRoomA\Big),$

$\Big(BHP, \big(\langle\mathsf{House}\rangle, \langle\mathsf{rooms}\rangle, \langle\mathsf{Room}\rangle\big), BHPRoomB\Big),$

$\Big(BHP, \big(\langle\mathsf{House}\rangle, \langle\mathsf{rooms}\rangle, \langle\mathsf{Room}\rangle\big), BHPRoomC\Big),$

$\Big(TRRoom1, \big(\langle\mathsf{Room}\rangle, \langle\mathsf{room\_id}\rangle, \mathsf{string}\big), \text{"1"}\Big),$

$\Big(TRRoom2, \big(\langle\mathsf{Room}\rangle, \langle\mathsf{room\_id}\rangle, \mathsf{string}\big), \text{"2"}\Big),$

$\Big(BHPRoomA, \big(\langle\mathsf{Room}\rangle, \langle\mathsf{room\_id}\rangle, \mathsf{string}\big), \text{"A"}\Big),$

$\Big(BHPRoomB, \big(\langle\mathsf{Room}\rangle, \langle\mathsf{room\_id}\rangle, \mathsf{string}\big), \text{"B"}\Big),$

$\Big(BHPRoomC, \big(\langle\mathsf{Room}\rangle, \langle\mathsf{room\_id}\rangle, \mathsf{string}\big), \text{"C"}\Big),$

$\Big(TRRoom1, \big(\langle\mathsf{Room}\rangle, \langle\mathsf{tenant}\rangle, \langle\mathsf{Tenant}\rangle\big), Tenant1\Big),$

$\Big(TRRoom2, \big(\langle\mathsf{Room}\rangle, \langle\mathsf{tenant}\rangle, \langle\mathsf{Tenant}\rangle\big), Tenant2\Big),$

$\Big(BHPRoomA, \big(\langle\mathsf{Room}\rangle, \langle\mathsf{tenant}\rangle, \langle\mathsf{Tenant}\rangle\big), Tenant3\Big),$

$\Big(BHPRoomC, \big(\langle\mathsf{Room}\rangle, \langle\mathsf{tenant}\rangle, \langle\mathsf{Tenant}\rangle\big), Tenant4\Big),$

$\Big(TRRoom1, \big(\langle\mathsf{Room}\rangle, \langle\mathsf{room\_size}\rangle, \langle\mathsf{RoomSize}\rangle\big), LargeSize\Big),$

$\Big(TRRoom2, \big(\langle\mathsf{Room}\rangle, \langle\mathsf{room\_size}\rangle, \langle\mathsf{RoomSize}\rangle\big), MediumSize\Big),$

$\Big(BHPRoomA, \big(\langle\mathsf{Room}\rangle, \langle\mathsf{room\_size}\rangle, \langle\mathsf{RoomSize}\rangle\big), SmallSize\Big),$

$\Big(BHPRoomB, \big(\langle\mathsf{Room}\rangle, \langle\mathsf{room\_size}\rangle, \langle\mathsf{RoomSize}\rangle\big), SmallSize\Big),$

$\Big(BHPRoomC, \big(\langle\mathsf{Room}\rangle, \langle\mathsf{room\_size}\rangle, \langle\mathsf{RoomSize}\rangle\big), SmallSize\Big),$

$\Big(Tenant1, \big(\langle\mathsf{Tenant}\rangle, \langle\mathsf{name}\rangle, \mathsf{string}\big), \text{"B.R. Mankjon"}\Big),$

$\Big(Tenant2, \big(\langle\mathsf{Tenant}\rangle, \langle\mathsf{name}\rangle, \mathsf{string}\big), \text{"P.J.R. Nam"}\Big),$

$\Big(Tenant3, \big(\langle\mathsf{Tenant}\rangle, \langle\mathsf{name}\rangle, \mathsf{string}\big), \text{"L. Horn"}\Big),$

$\Big(Tenant4, \big(\langle\mathsf{Tenant}\rangle, \langle\mathsf{name}\rangle, \mathsf{string}\big), \text{"A.C.C. Turg"}\Big),$

$\Big(Tenant5, \big(\langle\mathsf{Tenant}\rangle, \langle\mathsf{name}\rangle, \mathsf{string}\big), \text{"M. Silon"}\Big),$

$\Big(Tenant1, \big(\langle\mathsf{Tenant}\rangle, \langle\mathsf{age}\rangle, \mathsf{int}\big), 23\Big), \Big(Tenant2, \big(\langle\mathsf{Tenant}\rangle, \langle\mathsf{age}\rangle, \mathsf{int}\big), 24\Big),$

$\Big(Tenant3, \big(\langle\mathsf{Tenant}\rangle, \langle\mathsf{age}\rangle, \mathsf{int}\big), 18\Big), \Big(Tenant4, \big(\langle\mathsf{Tenant}\rangle, \langle\mathsf{age}\rangle, \mathsf{int}\big), 24\Big),$

$\Big(Tenant5, \big(\langle\mathsf{Tenant}\rangle, \langle\mathsf{age}\rangle, \mathsf{int}\big), 19\Big),$

Models after step 14

$$\Big(Tenant1, \big(\langle\mathsf{Tenant}\rangle, \langle\mathsf{type}\rangle, \langle\mathsf{TenantType}\rangle\big), RegularType\Big),$$
$$\Big(Tenant2, \big(\langle\mathsf{Tenant}\rangle, \langle\mathsf{type}\rangle, \langle\mathsf{TenantType}\rangle\big), RegularType\Big),$$
$$\Big(Tenant3, \big(\langle\mathsf{Tenant}\rangle, \langle\mathsf{type}\rangle, \langle\mathsf{TenantType}\rangle\big), RegularType\Big),$$
$$\Big(Tenant4, \big(\langle\mathsf{Tenant}\rangle, \langle\mathsf{type}\rangle, \langle\mathsf{TenantType}\rangle\big), RegularType\Big),$$
$$\Big(Tenant5, \big(\langle\mathsf{Tenant}\rangle, \langle\mathsf{type}\rangle, \langle\mathsf{TenantType}\rangle\big), SubtenantType\Big),$$
$$\Big(Tenant4, \big(\langle\mathsf{Tenant}\rangle, \langle\mathsf{subtenant}\rangle, \langle\mathsf{Tenant}\rangle\big), Tenant5\Big),$$
$$\Big(SmallSize, \big(\langle\mathsf{RoomSize}\rangle, \langle\mathsf{SMALL}\rangle, \langle\mathsf{RoomSize}\rangle\big), SmallSize\Big),$$
$$\Big(MediumSize, \big(\langle\mathsf{RoomSize}\rangle, \langle\mathsf{MEDIUM}\rangle, \langle\mathsf{RoomSize}\rangle\big), MediumSize\Big),$$
$$\Big(LargeSize, \big(\langle\mathsf{RoomSize}\rangle, \langle\mathsf{LARGE}\rangle, \langle\mathsf{RoomSize}\rangle\big), LargeSize\Big)\Big\}$$

$$\text{ident} = \{(TR, TR), (BHP, BHP), (TRRoom1, TRRoom1), (TRRoom2, TRRoom2),$$
$$(BHPRoomA, BHPRoomA), (BHPRoomB, BHPRoomB),$$
$$(BHPRoomC, BHPRoomC), (Tenant1, Tenant1), (Tenant2, Tenant2),$$
$$(Tenant3, Tenant3), (Tenant4, Tenant4), (Tenant5, Tenant5),$$
$$(SmallSize, SmallSize), (MediumSize, MediumSize),$$
$$(LargeSize, LargeSize), (RegularType, RegularType),$$
$$(SubtenantType, SubtenantType)\}$$

$$\text{type}_n = \{(TR, \langle\mathsf{House}\rangle), (BHP, \langle\mathsf{House}\rangle), (TRRoom1, \langle\mathsf{Room}\rangle), (TRRoom2, \langle\mathsf{Room}\rangle),$$
$$(BHPRoomA, \langle\mathsf{Room}\rangle), (BHPRoomB, \langle\mathsf{Room}\rangle), (BHPRoomC, \langle\mathsf{Room}\rangle),$$
$$(Tenant1, \langle\mathsf{Tenant}\rangle), (Tenant2, \langle\mathsf{Tenant}\rangle), (Tenant3, \langle\mathsf{Tenant}\rangle),$$
$$(Tenant4, \langle\mathsf{Tenant}\rangle), (Tenant5, \langle\mathsf{Tenant}\rangle), (SmallSize, \langle\mathsf{RoomSize}\rangle),$$
$$(MediumSize, \langle\mathsf{RoomSize}\rangle), (LargeSize, \langle\mathsf{RoomSize}\rangle),$$
$$(RegularType, \langle\mathsf{TenantType}, \mathsf{REGULAR}\rangle),$$
$$(SubtenantType, \langle\mathsf{TenantType}, \mathsf{SUBTENANT}\rangle), (\text{“TwoRem”}, \mathsf{string}),$$
$$(\text{“B.H. Paleis”}, \mathsf{string}), (\text{“1”}, \mathsf{string}), (\text{“2”}, \mathsf{string}), (\text{“A”}, \mathsf{string}), (\text{“B”}, \mathsf{string}),$$
$$(\text{“C”}, \mathsf{string}), (\text{“B.R. Mankjon”}, \mathsf{string}), (\text{“P.J.R. Nam”}, \mathsf{string}),$$
$$(\text{“A.C.C. Turg”}, \mathsf{string}), (\text{“L. Horn”}, \mathsf{string}), (\text{“M. Silon”}, \mathsf{string}), (23, \mathsf{int}),$$
$$(24, \mathsf{int}), (18, \mathsf{int}), (19, \mathsf{int})\}$$

$f_{14}(Im_{14}) = f_{13}(Im_{13}) \sqcup f_{NullableClassField}(Im_{NullableClassField})$ (Definition 4.4.27)
$f'_{14}(IG_{14}) = f'_{13}(IG_{13}) \sqcup f'_{NullableClassField}(IG_{NullableClassField})$ (Definition 4.4.32)

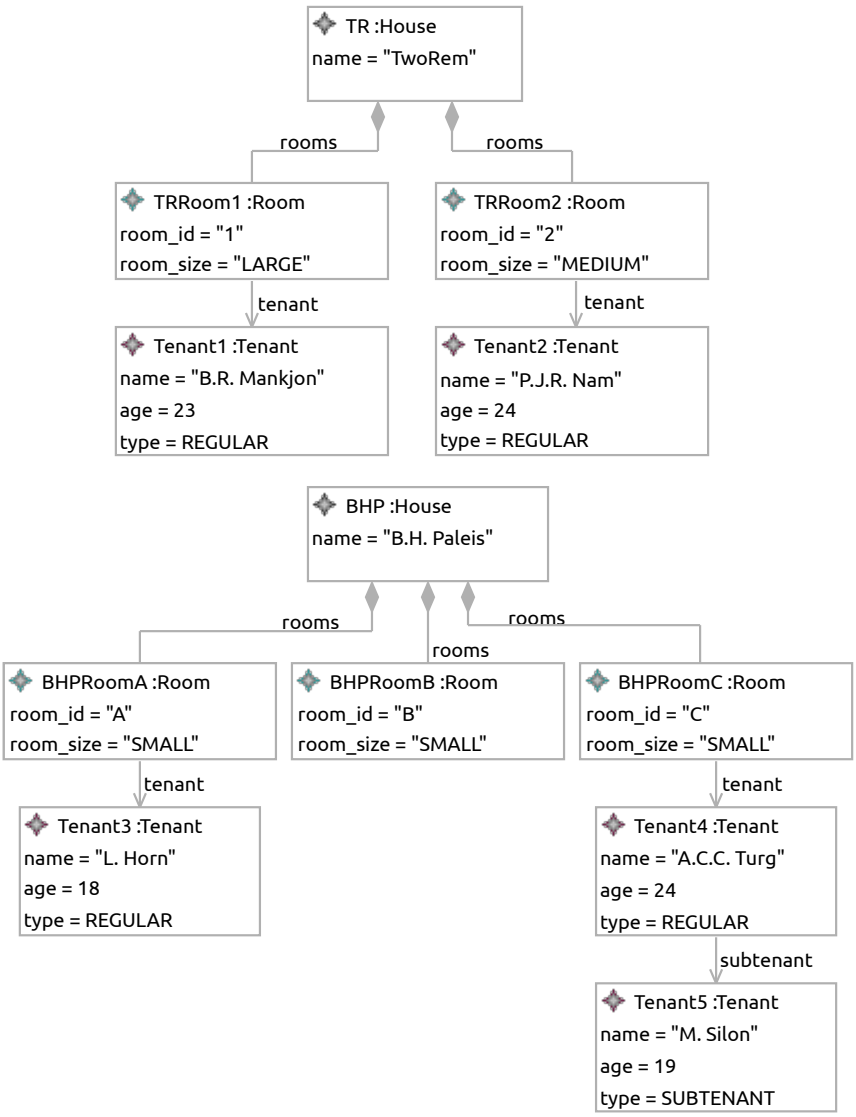
A visual representation of Tm_{14} and Im_{14} can be found in Figure 6.28. Similarly, a visual representation of TG_{14} and IG_{14} can be found in Figure 6.29. Please note that because of the definitions of $f_{14}(Im_{14})$ and $f'_{14}(IG_{14})$, we have that $f_{14}(Im_{14}) = IG_{14}$ and $f'_{14}(IG_{14}) = Im_{14}$. Furthermore, $f_{14}(Im_{14})$ and $f'_{14}(IG_{14})$ are valid mapping functions themselves, such that they can be combined with another mapping function in the next step.

The visualisation shows once more a new field for referencing existing objects. The most important aspect of this field is that it can reference objects from the same type as its source type. This property is useful for making self relations, but also for creating relations between objects of the same type, as can be seen in the visualisation of $Tenant4$, which has $Tenant5$ as a subtenant.

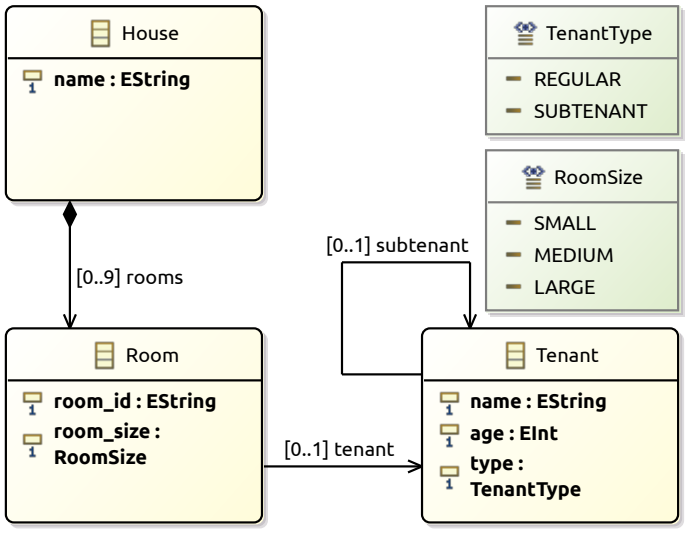
6.2.15 Living rooms

In the final step, a last data field is introduced. In this step, the .House class is enriched with an living_room field and its values. As before, Section 5.2.6 is used to introduce the field, while on the instance level, Section 5.3.6 is used to introduce the values.

The *classtype* of the new field is .House, as the field will be defined for houses. The *name* of the new field is living_room and the $fieldtype$ is bool. The set of objects of which the value is set is is equal to all house objects, so $objects = \{TR, BHP\}$. The function for *obids* returns the existing identifier of each

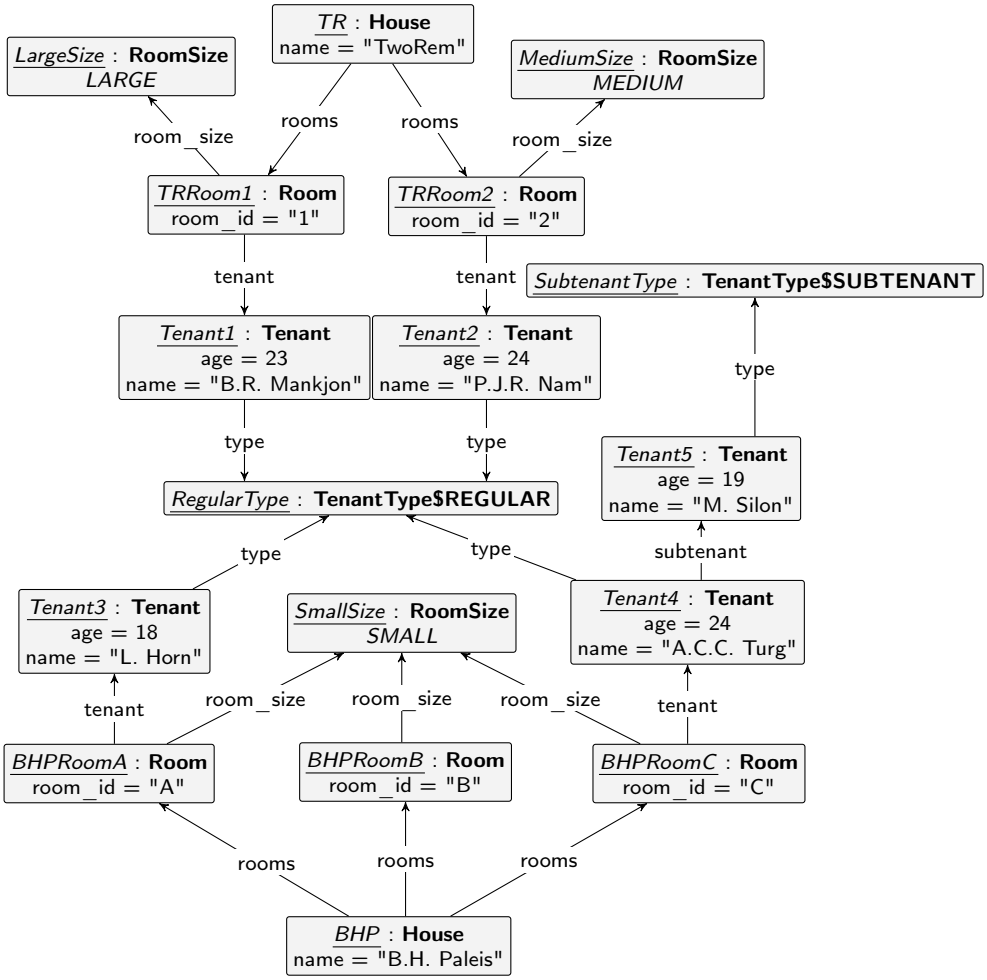


(a) Instance Model Im_{14}

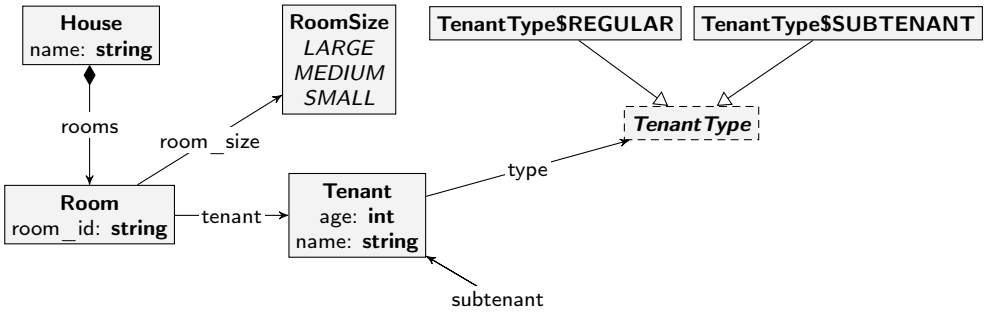


(b) Type Model Tm_{14}

Figure 6.28: The Ecore model after step 14



(a) Instance Graph IG_{14}



(b) Type Graph TG_{14}

Figure 6.29: The GROOVE graphs after step 14

of these objects. The *values* function is defined as follows:

$$values = \{(TR, \text{false}), (BHP, \text{true})\}$$

The following model is obtained:

Models after step 15

$$
\begin{aligned}
Tm_{15} = \quad & \text{combine}(Tm_{14}, Tm_{DataField}) = \\
& Class = \{.\text{House}, .\text{Room}, .\text{Tenant}\} \\
& Enum = \{.\text{RoomSize}, .\text{TenantType}\} \\
& UserDataType = \{\} \\
& Field = \{(.\text{House}, \text{name}), (.\text{House}, \text{rooms}), (.\text{Room}, \text{room\_id}), \\
& \quad (.\text{Room}, \text{living\_room}), (.\text{Room}, \text{room\_size}), (.\text{Room}, \text{tenant}), \\
& \quad (.\text{Tenant}, \text{name}), (.\text{Tenant}, \text{age}), (.\text{Tenant}, \text{type}), \\
& \quad (.\text{Tenant}, \text{subtenant})\} \\
& \text{FieldSig} = \big\{\big((.\text{House}, \text{name}), (\text{string}, 1..1)\big), \\
& \quad \big((.\text{House}, \text{rooms}), ([\text{setof}, !.\text{Room}], 0..9)\big), \\
& \quad \big((.\text{Room}, \text{room\_id}), (\text{string}, 1..1)\big), \\
& \quad \big((.\text{Room}, \text{room\_size}), (.\text{RoomSize}, 1..1)\big), \\
& \quad \big((.\text{Room}, \text{living\_room}), (\text{bool}, 1..1)\big), \\
& \quad \big((.\text{Room}, \text{tenant}), (?.\text{Tenant}, 0..1)\big), \\
& \quad \big((.\text{Tenant}, \text{name}), (\text{string}, 1..1)\big), \\
& \quad \big((.\text{Tenant}, \text{age}), (\text{int}, 1..1)\big), \\
& \quad \big((.\text{Tenant}, \text{type}), (.\text{TenantType}, 1..1)\big), \\
& \quad \big((.\text{Tenant}, \text{subtenant}), (?.\text{Tenant}, 0..1)\big)\big\} \\
& EnumValue = \{(.\text{RoomSize}, \text{SMALL}), (.\text{RoomSize}, \text{MEDIUM}), (.\text{RoomSize}, \text{LARGE}) \\
& \quad (.\text{TenantType}, \text{REGULAR}), (.\text{TenantType}, \text{SUBTENANT})\} \\
& Inh = \{\} \\
& Prop = \big\{\big(\text{containment}, (.\text{House}, \text{rooms})\big)\big\} \\
& Constant = \{\} \\
& \text{ConstType} = \{\}
\end{aligned}
$$

Models after step 15

$$
\begin{aligned}
Im_{15} = \quad & \text{combine}(Tm_{14}, Im_{DataField}) = \\
& Object = \{TR, BHP, TRRoom1, TRRoom2, BHPRoomA, BHPRoomB, \\
& \quad BHPRoomC, Tenant1, Tenant2, Tenant3, Tenant4, Tenant5\} \\
& \text{ObjectClass}(ob) = \{(TR, .\mathsf{House}), (BHP, .\mathsf{House}), (TRRoom1, .\mathsf{Room}), \\
& \quad (TRRoom2, .\mathsf{Room}), (BHPRoomA, .\mathsf{Room}), (BHPRoomB, .\mathsf{Room}), \\
& \quad (BHPRoomC, .\mathsf{Room}, (Tenant1, .\mathsf{Tenant}), (Tenant2, .\mathsf{Tenant}), \\
& \quad (Tenant3, .\mathsf{Tenant}), (Tenant4, .\mathsf{Tenant}), (Tenant5, .\mathsf{Tenant})\} \\
& \text{ObjectId} = \{(TR, TR), (BHP, BHP), (TRRoom1, TRRoom1), \\
& \quad (TRRoom2, TRRoom2), (BHPRoomA, BHPRoomA), \\
& \quad (BHPRoomB, BHPRoomB), (BHPRoomC, BHPRoomC), \\
& \quad (Tenant1, Tenant1), (Tenant2, Tenant2), (Tenant3, Tenant3), \\
& \quad (Tenant4, Tenant4), (Tenant5, Tenant5)\} \\
& \text{FieldValue} = \Big\{\Big(\big((TR, (.\mathsf{House}, \mathsf{name})\big), \big[\mathsf{string}, \text{“TwoRem”}\big]\Big), \\
& \quad \Big(\big((BHP, (.\mathsf{House}, \mathsf{name})\big), \big[\mathsf{string}, \text{“B.H. Paleis”}\big]\Big), \\
& \quad \Big(\big((TR, (.\mathsf{House}, \mathsf{rooms})\big), \big[\mathsf{setof}, \big\langle[\mathsf{obj}, TRRoom1], \\
& \qquad [\mathsf{obj}, TRRoom2]\big\rangle\big]\Big), \\
& \quad \Big(\big((BHP, (.\mathsf{House}, \mathsf{rooms})\big)\big[\mathsf{setof}, \big\langle[\mathsf{obj}, BHPRoomA], \\
& \qquad [\mathsf{obj}, BHPRoomB], (\mathsf{obj}, BHPRoomC)\big\rangle\big]\Big), \\
& \quad \Big(\big((TR, (.\mathsf{House}, \mathsf{living\_room}))\big), \big[\mathsf{boolean}, \text{false}\big]\Big), \\
& \quad \Big(\big((BHP, (.\mathsf{House}, \mathsf{living\_room}))\big), \big[\mathsf{boolean}, \text{true}\big]\Big), \\
& \quad \Big(\big((TRRoom1, (.\mathsf{Room}, \mathsf{room\_id})\big), \big[\mathsf{string}, \text{“1”}\big]\Big), \\
& \quad \Big(\big((TRRoom2, (.\mathsf{Room}, \mathsf{room\_id})\big), \big[\mathsf{string}, \text{“2”}\big]\Big),
\end{aligned}
$$

Models after step 15

$$\Big((BHPRoomA, (.\mathsf{Room}, \mathsf{room\_id})), [\mathsf{string}, \text{“A”}]\Big),$$
$$\Big((BHPRoomB, (.\mathsf{Room}, \mathsf{room\_id})), [\mathsf{string}, \text{“B”}]\Big),$$
$$\Big((BHPRoomC, (.\mathsf{Room}, \mathsf{room\_id})), [\mathsf{string}, \text{“C”}]\Big),$$
$$\Big((TRRoom1, (.\mathsf{Room}, \mathsf{room\_size})), [\mathsf{enum}, (.\mathsf{RoomSize}, \mathsf{LARGE})]\Big),$$
$$\Big((TRRoom2, (.\mathsf{Room}, \mathsf{room\_size})), [\mathsf{enum}, (.\mathsf{RoomSize}, \mathsf{MEDIUM})]\Big),$$
$$\Big((BHPRoomA, (.\mathsf{Room}, \mathsf{room\_size})), [\mathsf{enum}, (.\mathsf{RoomSize}, \mathsf{SMALL})]\Big),$$
$$\Big((BHPRoomB, (.\mathsf{Room}, \mathsf{room\_size})), [\mathsf{enum}, (.\mathsf{RoomSize}, \mathsf{SMALL})]\Big),$$
$$\Big((BHPRoomC, (.\mathsf{Room}, \mathsf{room\_size})), [\mathsf{enum}, (.\mathsf{RoomSize}, \mathsf{SMALL})]\Big),$$
$$\Big((TRRoom1, (.\mathsf{Room}, \mathsf{tenant}), [\mathsf{obj}, Tenant1]\Big),$$
$$\Big((TRRoom2, (.\mathsf{Room}, \mathsf{tenant}), [\mathsf{obj}, Tenant2]\Big),$$
$$\Big((BHPRoomA, (.\mathsf{Room}, \mathsf{tenant}), [\mathsf{obj}, Tenant3]\Big),$$
$$\Big((BHPRoomB, (.\mathsf{Room}, \mathsf{tenant}), [\mathsf{nil}]\Big),$$
$$\Big((BHPRoomC, (.\mathsf{Room}, \mathsf{tenant}), [\mathsf{obj}, Tenant4]\Big),$$
$$\Big((Tenant1, (.\mathsf{Tenant}, \mathsf{name})), [\mathsf{string}, \text{“B.R. Mankjon”}]\Big),$$
$$\Big((Tenant2, (.\mathsf{Tenant}, \mathsf{name})), [\mathsf{string}, \text{“P.J.R. Nam”}]\Big),$$
$$\Big((Tenant3, (.\mathsf{Tenant}, \mathsf{name})), [\mathsf{string}, \text{“L. Horn”}]\Big),$$
$$\Big((Tenant4, (.\mathsf{Tenant}, \mathsf{name})), [\mathsf{string}, \text{“A.C.C. Turg”}]\Big),$$
$$\Big((Tenant5, (.\mathsf{Tenant}, \mathsf{name})), [\mathsf{string}, \text{“M. Silon”}]\Big),$$
$$\Big((Tenant1, (.\mathsf{Tenant}, \mathsf{age})), [\mathsf{int}, 23]\Big),$$
$$\Big((Tenant2, (.\mathsf{Tenant}, \mathsf{age})), [\mathsf{int}, 24]\Big),$$
$$\Big((Tenant3, (.\mathsf{Tenant}, \mathsf{age})), [\mathsf{int}, 18]\Big),$$
$$\Big((Tenant4, (.\mathsf{Tenant}, \mathsf{age})), [\mathsf{int}, 24]\Big),$$
$$\Big((Tenant5, (.\mathsf{Tenant}, \mathsf{age})), [\mathsf{int}, 19]\Big),$$
$$\Big((Tenant1, (.\mathsf{Tenant}, \mathsf{type})), [\mathsf{enum}, (.\mathsf{TenantType}, \mathsf{REGULAR})]\Big),$$
$$\Big((Tenant2, (.\mathsf{Tenant}, \mathsf{type})), [\mathsf{enum}, (.\mathsf{TenantType}, \mathsf{REGULAR})]\Big),$$
$$\Big((Tenant3, (.\mathsf{Tenant}, \mathsf{type})), [\mathsf{enum}, (.\mathsf{TenantType}, \mathsf{REGULAR})]\Big),$$
$$\Big((Tenant4, (.\mathsf{Tenant}, \mathsf{type})), [\mathsf{enum}, (.\mathsf{TenantType}, \mathsf{REGULAR})]\Big),$$
$$\Big((Tenant5, (.\mathsf{Tenant}, \mathsf{type})), [\mathsf{enum}, (.\mathsf{TenantType}, \mathsf{SUBTENANT})]\Big),$$
$$\Big((Tenant1, (.\mathsf{Tenant}, \mathsf{subtenant})), [\mathsf{nil}]\Big),$$
$$\Big((Tenant2, (.\mathsf{Tenant}, \mathsf{subtenant})), [\mathsf{nil}]\Big),$$
$$\Big((Tenant3, (.\mathsf{Tenant}, \mathsf{subtenant})), [\mathsf{nil}]\Big),$$
$$\Big((Tenant4, (.\mathsf{Tenant}, \mathsf{subtenant})), [\mathsf{obj}, Tenant3]\Big),$$
$$\Big((Tenant5, (.\mathsf{Tenant}, \mathsf{subtenant})), [\mathsf{nil}]\Big)\Big\}$$

$\mathrm{DefaultValue} = \{\}$

Models after step 15

$$
\begin{aligned}
TG_{15} = \quad & \mathrm{combine}(TG_{14}, TG_{DataField}) = \\
& NT = \{\langle \mathsf{House}\rangle, \langle \mathsf{Room}\rangle, \langle \mathsf{Tenant}\rangle, \langle \mathsf{RoomSize}\rangle, \langle \mathsf{TenantType}\rangle, \\
& \quad \langle \mathsf{TenantType}, \mathsf{REGULAR}\rangle, \langle \mathsf{TenantType}, \mathsf{SUBTENANT}\rangle, \mathsf{string}, \mathsf{int}, \mathsf{bool}\} \\
& ET = \big\{\big(\langle \mathsf{House}\rangle, \langle \mathsf{name}\rangle, \mathsf{string}\big), \big(\langle \mathsf{House}\rangle, \langle \mathsf{rooms}\rangle, \langle \mathsf{Room}\rangle\big), \\
& \quad \big(\langle \mathsf{House}\rangle, \langle \mathsf{living\_room}\rangle, \mathsf{bool}\big), \big(\langle \mathsf{Room}\rangle, \langle \mathsf{room\_id}\rangle, \mathsf{string}\big), \\
& \quad \big(\langle \mathsf{Room}\rangle, \langle \mathsf{room\_size}\rangle, \langle \mathsf{RoomSize}\rangle\big), \big(\langle \mathsf{Room}\rangle, \langle \mathsf{tenant}\rangle, \langle \mathsf{Tenant}\rangle\big), \\
& \quad \big(\langle \mathsf{Tenant}\rangle, \langle \mathsf{name}\rangle, \mathsf{string}\big), \big(\langle \mathsf{Tenant}\rangle, \langle \mathsf{age}\rangle, \mathsf{int}\big), \\
& \quad \big(\langle \mathsf{Tenant}\rangle, \langle \mathsf{type}\rangle, \langle \mathsf{TenantType}\rangle\big), \big(\langle \mathsf{Tenant}\rangle, \langle \mathsf{subtenant}\rangle, \langle \mathsf{Tenant}\rangle\big), \\
& \quad \big(\langle \mathsf{RoomSize}\rangle, \langle \mathsf{SMALL}\rangle, \langle \mathsf{RoomSize}\rangle\big), \\
& \quad \big(\langle \mathsf{RoomSize}\rangle, \langle \mathsf{MEDIUM}\rangle, \langle \mathsf{RoomSize}\rangle\big), \\
& \quad \big(\langle \mathsf{RoomSize}\rangle, \langle \mathsf{LARGE}\rangle, \langle \mathsf{RoomSize}\rangle\big)\big\} \\
& \sqsubseteq = \big\{\big(\langle \mathsf{House}\rangle, \langle \mathsf{House}\rangle\big), \big(\langle \mathsf{Room}\rangle, \langle \mathsf{Room}\rangle\big), \big(\langle \mathsf{Tenant}\rangle, \langle \mathsf{Tenant}\rangle\big), \\
& \quad \big(\langle \mathsf{RoomSize}\rangle, \langle \mathsf{RoomSize}\rangle\big), \big(\langle \mathsf{TenantType}\rangle, \langle \mathsf{TenantType}\rangle\big), \\
& \quad \big(\langle \mathsf{TenantType}, \mathsf{REGULAR}\rangle, \langle \mathsf{TenantType}, \mathsf{REGULAR}\rangle\big), \\
& \quad \big(\langle \mathsf{TenantType}, \mathsf{SUBTENANT}\rangle, \langle \mathsf{TenantType}, \mathsf{SUBTENANT}\rangle\big), \\
& \quad \big(\langle \mathsf{TenantType}, \mathsf{REGULAR}\rangle, \langle \mathsf{TenantType}\rangle\big), \\
& \quad \big(\langle \mathsf{TenantType}, \mathsf{SUBTENANT}\rangle, \langle \mathsf{TenantType}\rangle\big), \big(\mathsf{string}, \mathsf{string}\big), \big(\mathsf{int}, \mathsf{int}\big), \\
& \quad \big(\mathsf{bool}, \mathsf{bool}\big)\big\} \\
& abs = \{\langle \mathsf{TenantType}\rangle\} \\
& \mathrm{mult} = \Big\{\Big(\big(\langle \mathsf{House}\rangle, \langle \mathsf{name}\rangle, \mathsf{string}\big), (0..*, 1..1)\Big), \\
& \quad \Big(\big(\langle \mathsf{House}\rangle, \langle \mathsf{rooms}\rangle, \langle \mathsf{Room}\rangle\big), (0..1, 0..9)\Big), \\
& \quad \Big(\big(\langle \mathsf{House}\rangle, \langle \mathsf{living\_room}\rangle, \mathsf{bool}\big), (0..*, 1..1)\Big), \\
& \quad \Big(\big(\langle \mathsf{Room}\rangle, \langle \mathsf{room\_id}\rangle, \mathsf{string}\big), (0..*, 1..1)\Big), \\
& \quad \Big(\big(\langle \mathsf{Room}\rangle, \langle \mathsf{room\_size}\rangle, \langle \mathsf{RoomSize}\rangle\big), (0..*, 1..1)\Big), \\
& \quad \Big(\big(\langle \mathsf{Room}\rangle, \langle \mathsf{tenant}\rangle, \langle \mathsf{Tenant}\rangle\big), (0..*, 0..1)\Big), \\
& \quad \Big(\big(\langle \mathsf{Tenant}\rangle, \langle \mathsf{name}\rangle, \mathsf{string}\big), (0..*, 1..1)\Big), \\
& \quad \Big(\big(\langle \mathsf{Tenant}\rangle, \langle \mathsf{age}\rangle, \mathsf{int}\big), (0..*, 0..1)\Big), \\
& \quad \Big(\big(\langle \mathsf{Tenant}\rangle, \langle \mathsf{type}\rangle, \langle \mathsf{TenantType}\rangle\big), (0..*, 1..1)\Big), \\
& \quad \Big(\big(\langle \mathsf{Tenant}\rangle, \langle \mathsf{subtenant}\rangle, \langle \mathsf{Tenant}\rangle\big), (0..*, 0..1)\Big), \\
& \quad \Big(\big(\langle \mathsf{RoomSize}\rangle, \langle \mathsf{SMALL}\rangle, \langle \mathsf{RoomSize}\rangle\big), (0..1, 0..1)\Big), \\
& \quad \Big(\big(\langle \mathsf{RoomSize}\rangle, \langle \mathsf{MEDIUM}\rangle, \langle \mathsf{RoomSize}\rangle\big), (0..1, 0..1)\Big), \\
& \quad \Big(\big(\langle \mathsf{RoomSize}\rangle, \langle \mathsf{LARGE}\rangle, \langle \mathsf{RoomSize}\rangle\big), (0..1, 0..1)\Big)\Big\} \\
& contains = \big\{\big(\langle \mathsf{House}\rangle, \langle \mathsf{rooms}\rangle, \langle \mathsf{Room}\rangle\big)\big\}
\end{aligned}
$$

Models after step 15

$IG_{15} =$ $\text{combine}(TG_{14}, IG_{DataField}) =$

$N = \{TR, BHP, TRRoom1, TRRoom2, BHPRoomA, BHPRoomB, BHPRoomC,$
$Tenant1, Tenant2, Tenant3, Tenant4, Tenant5, SmallSize, MediumSize,$
$LargeSize, RegularType, SubtenantType,$ "TwoRem", "B.H. Paleis", "1", "2", "A",
"B", "C", "B.R. Mankjon", "P.J.R. Nam", "L. Horn", "A.C.C. Turg", "M. Silon", 23,
24, 18, 19, false, true$\}$

$E = \Big\{\Big(TR, \big(\langle\mathsf{House}\rangle, \langle\mathsf{name}\rangle, \mathsf{string}\big), \text{"TwoRem"}\Big),$
$\Big(BHP, \big(\langle\mathsf{House}\rangle, \langle\mathsf{name}\rangle, \mathsf{string}\big), \text{"B.H. Paleis"}\Big),$
$\Big(TR, \big(\langle\mathsf{House}\rangle, \langle\mathsf{living\_room}\rangle, \mathsf{bool}\big), \text{false}\Big),$
$\Big(BHP, \big(\langle\mathsf{House}\rangle, \langle\mathsf{living\_room}\rangle, \mathsf{bool}\big), \text{true}\Big),$
$\Big(TR, \big(\langle\mathsf{House}\rangle, \langle\mathsf{rooms}\rangle, \langle\mathsf{Room}\rangle\big), TRRoom1\Big),$
$\Big(TR, \big(\langle\mathsf{House}\rangle, \langle\mathsf{rooms}\rangle, \langle\mathsf{Room}\rangle\big), TRRoom2\Big),$
$\Big(BHP, \big(\langle\mathsf{House}\rangle, \langle\mathsf{rooms}\rangle, \langle\mathsf{Room}\rangle\big), BHPRoomA\Big),$
$\Big(BHP, \big(\langle\mathsf{House}\rangle, \langle\mathsf{rooms}\rangle, \langle\mathsf{Room}\rangle\big), BHPRoomB\Big),$
$\Big(BHP, \big(\langle\mathsf{House}\rangle, \langle\mathsf{rooms}\rangle, \langle\mathsf{Room}\rangle\big), BHPRoomC\Big),$
$\Big(TRRoom1, \big(\langle\mathsf{Room}\rangle, \langle\mathsf{room\_id}\rangle, \mathsf{string}\big), \text{"1"}\Big),$
$\Big(TRRoom2, \big(\langle\mathsf{Room}\rangle, \langle\mathsf{room\_id}\rangle, \mathsf{string}\big), \text{"2"}\Big),$
$\Big(BHPRoomA, \big(\langle\mathsf{Room}\rangle, \langle\mathsf{room\_id}\rangle, \mathsf{string}\big), \text{"A"}\Big),$
$\Big(BHPRoomB, \big(\langle\mathsf{Room}\rangle, \langle\mathsf{room\_id}\rangle, \mathsf{string}\big), \text{"B"}\Big),$
$\Big(BHPRoomC, \big(\langle\mathsf{Room}\rangle, \langle\mathsf{room\_id}\rangle, \mathsf{string}\big), \text{"C"}\Big),$
$\Big(TRRoom1, \big(\langle\mathsf{Room}\rangle, \langle\mathsf{tenant}\rangle, \langle\mathsf{Tenant}\rangle\big), Tenant1\Big),$
$\Big(TRRoom2, \big(\langle\mathsf{Room}\rangle, \langle\mathsf{tenant}\rangle, \langle\mathsf{Tenant}\rangle\big), Tenant2\Big),$
$\Big(BHPRoomA, \big(\langle\mathsf{Room}\rangle, \langle\mathsf{tenant}\rangle, \langle\mathsf{Tenant}\rangle\big), Tenant3\Big),$
$\Big(BHPRoomC, \big(\langle\mathsf{Room}\rangle, \langle\mathsf{tenant}\rangle, \langle\mathsf{Tenant}\rangle\big), Tenant4\Big),$
$\Big(TRRoom1, \big(\langle\mathsf{Room}\rangle, \langle\mathsf{room\_size}\rangle, \langle\mathsf{RoomSize}\rangle\big), LargeSize\Big),$
$\Big(TRRoom2, \big(\langle\mathsf{Room}\rangle, \langle\mathsf{room\_size}\rangle, \langle\mathsf{RoomSize}\rangle\big), MediumSize\Big),$
$\Big(BHPRoomA, \big(\langle\mathsf{Room}\rangle, \langle\mathsf{room\_size}\rangle, \langle\mathsf{RoomSize}\rangle\big), SmallSize\Big),$
$\Big(BHPRoomB, \big(\langle\mathsf{Room}\rangle, \langle\mathsf{room\_size}\rangle, \langle\mathsf{RoomSize}\rangle\big), SmallSize\Big),$
$\Big(BHPRoomC, \big(\langle\mathsf{Room}\rangle, \langle\mathsf{room\_size}\rangle, \langle\mathsf{RoomSize}\rangle\big), SmallSize\Big),$
$\Big(Tenant1, \big(\langle\mathsf{Tenant}\rangle, \langle\mathsf{name}\rangle, \mathsf{string}\big), \text{"B.R. Mankjon"}\Big),$
$\Big(Tenant2, \big(\langle\mathsf{Tenant}\rangle, \langle\mathsf{name}\rangle, \mathsf{string}\big), \text{"P.J.R. Nam"}\Big),$
$\Big(Tenant3, \big(\langle\mathsf{Tenant}\rangle, \langle\mathsf{name}\rangle, \mathsf{string}\big), \text{"L. Horn"}\Big),$
$\Big(Tenant4, \big(\langle\mathsf{Tenant}\rangle, \langle\mathsf{name}\rangle, \mathsf{string}\big), \text{"A.C.C. Turg"}\Big),$
$\Big(Tenant5, \big(\langle\mathsf{Tenant}\rangle, \langle\mathsf{name}\rangle, \mathsf{string}\big), \text{"M. Silon"}\Big),$
$\Big(Tenant1, \big(\langle\mathsf{Tenant}\rangle, \langle\mathsf{age}\rangle, \mathsf{int}\big), 23\Big), \Big(Tenant2, \big(\langle\mathsf{Tenant}\rangle, \langle\mathsf{age}\rangle, \mathsf{int}\big), 24\Big),$
$\Big(Tenant3, \big(\langle\mathsf{Tenant}\rangle, \langle\mathsf{age}\rangle, \mathsf{int}\big), 18\Big), \Big(Tenant4, \big(\langle\mathsf{Tenant}\rangle, \langle\mathsf{age}\rangle, \mathsf{int}\big), 24\Big),$

Models after step 15

$$\Big(Tenant5, \big(\langle \mathsf{Tenant}\rangle, \langle \mathsf{age}\rangle, \mathsf{int}\big), 19\Big),$$
$$\Big(Tenant1, \big(\langle \mathsf{Tenant}\rangle, \langle \mathsf{type}\rangle, \langle \mathsf{TenantType}\rangle\big), RegularType\Big),$$
$$\Big(Tenant2, \big(\langle \mathsf{Tenant}\rangle, \langle \mathsf{type}\rangle, \langle \mathsf{TenantType}\rangle\big), RegularType\Big),$$
$$\Big(Tenant3, \big(\langle \mathsf{Tenant}\rangle, \langle \mathsf{type}\rangle, \langle \mathsf{TenantType}\rangle\big), RegularType\Big),$$
$$\Big(Tenant4, \big(\langle \mathsf{Tenant}\rangle, \langle \mathsf{type}\rangle, \langle \mathsf{TenantType}\rangle\big), RegularType\Big),$$
$$\Big(Tenant5, \big(\langle \mathsf{Tenant}\rangle, \langle \mathsf{type}\rangle, \langle \mathsf{TenantType}\rangle\big), SubtenantType\Big),$$
$$\Big(Tenant4, \big(\langle \mathsf{Tenant}\rangle, \langle \mathsf{subtenant}\rangle, \langle \mathsf{Tenant}\rangle\big), Tenant5\Big),$$
$$\Big(SmallSize, \big(\langle \mathsf{RoomSize}\rangle, \langle \mathsf{SMALL}\rangle, \langle \mathsf{RoomSize}\rangle\big), SmallSize\Big),$$
$$\Big(MediumSize, \big(\langle \mathsf{RoomSize}\rangle, \langle \mathsf{MEDIUM}\rangle, \langle \mathsf{RoomSize}\rangle\big), MediumSize\Big),$$
$$\Big(LargeSize, \big(\langle \mathsf{RoomSize}\rangle, \langle \mathsf{LARGE}\rangle, \langle \mathsf{RoomSize}\rangle\big), LargeSize\Big)\Big\}$$

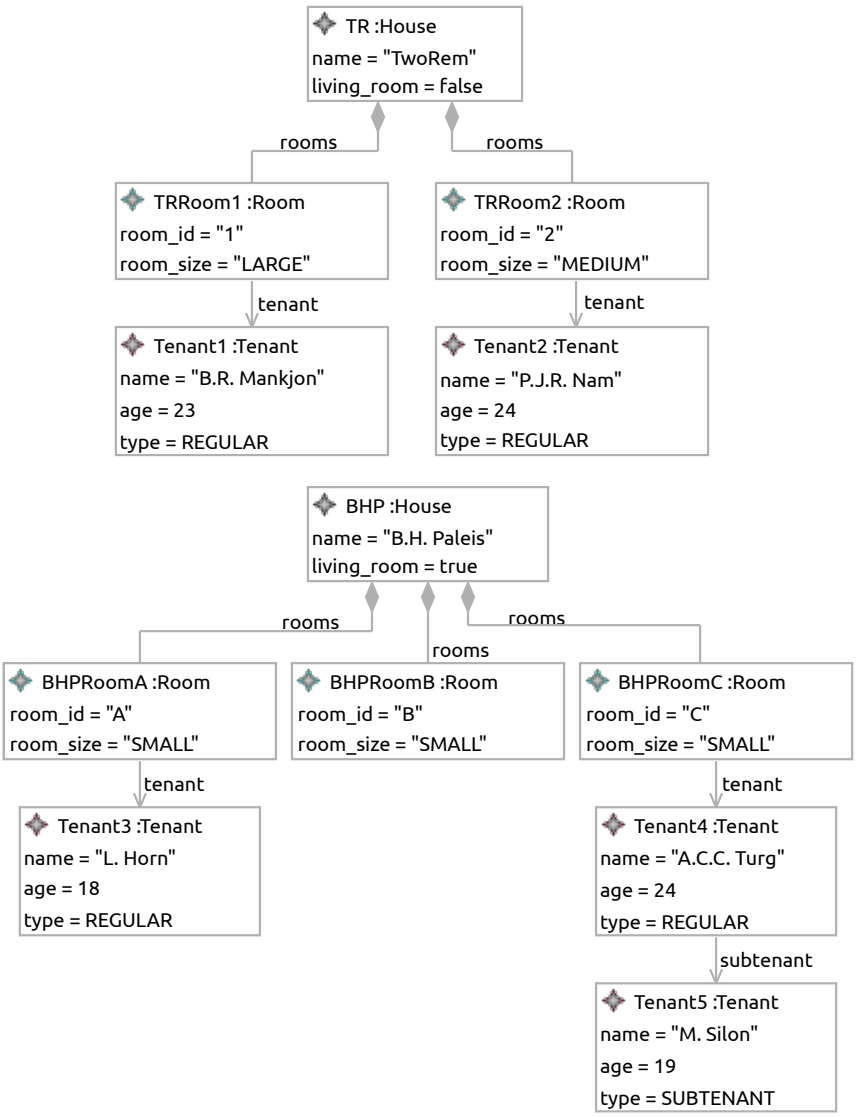
$$\begin{aligned}\text{ident} = \{&(TR, TR), (BHP, BHP), (TRRoom1, TRRoom1), (TRRoom2, TRRoom2),\\ &(BHPRoomA, BHPRoomA), (BHPRoomB, BHPRoomB),\\ &(BHPRoomC, BHPRoomC), (Tenant1, Tenant1), (Tenant2, Tenant2),\\ &(Tenant3, Tenant3), (Tenant4, Tenant4), (Tenant5, Tenant5),\\ &(SmallSize, SmallSize), (MediumSize, MediumSize),\\ &(LargeSize, LargeSize), (RegularType, RegularType),\\ &(SubtenantType, SubtenantType)\}\end{aligned}$$

$$\begin{aligned}\text{type}_n = \{&(TR, \langle \mathsf{House}\rangle), (BHP, \langle \mathsf{House}\rangle), (TRRoom1, \langle \mathsf{Room}\rangle), (TRRoom2, \langle \mathsf{Room}\rangle),\\ &(BHPRoomA, \langle \mathsf{Room}\rangle), (BHPRoomB, \langle \mathsf{Room}\rangle), (BHPRoomC, \langle \mathsf{Room}\rangle),\\ &(Tenant1, \langle \mathsf{Tenant}\rangle), (Tenant2, \langle \mathsf{Tenant}\rangle), (Tenant3, \langle \mathsf{Tenant}\rangle),\\ &(Tenant4, \langle \mathsf{Tenant}\rangle), (Tenant5, \langle \mathsf{Tenant}\rangle), (SmallSize, \langle \mathsf{RoomSize}\rangle),\\ &(MediumSize, \langle \mathsf{RoomSize}\rangle), (LargeSize, \langle \mathsf{RoomSize}\rangle),\\ &(RegularType, \langle \mathsf{TenantType}, \mathsf{REGULAR}\rangle),\\ &(SubtenantType, \langle \mathsf{TenantType}, \mathsf{SUBTENANT}\rangle), (\text{“TwoRem”}, \mathsf{string}),\\ &(\text{“B.H. Paleis”}, \mathsf{string}), (\text{“1”}, \mathsf{string}), (\text{“2”}, \mathsf{string}), (\text{“A”}, \mathsf{string}), (\text{“B”}, \mathsf{string}),\\ &(\text{“C”}, \mathsf{string}), (\text{“B.R. Mankjon”}, \mathsf{string}), (\text{“P.J.R. Nam”}, \mathsf{string}),\\ &(\text{“A.C.C. Turg”}, \mathsf{string}), (\text{“L. Horn”}, \mathsf{string}), (\text{“M. Silon”}, \mathsf{string}), (23, \mathsf{int}),\\ &(24, \mathsf{int}), (18, \mathsf{int}), (19, \mathsf{int}), (\text{false}, \mathsf{bool}), (\text{true}, \mathsf{bool})\}\end{aligned}$$

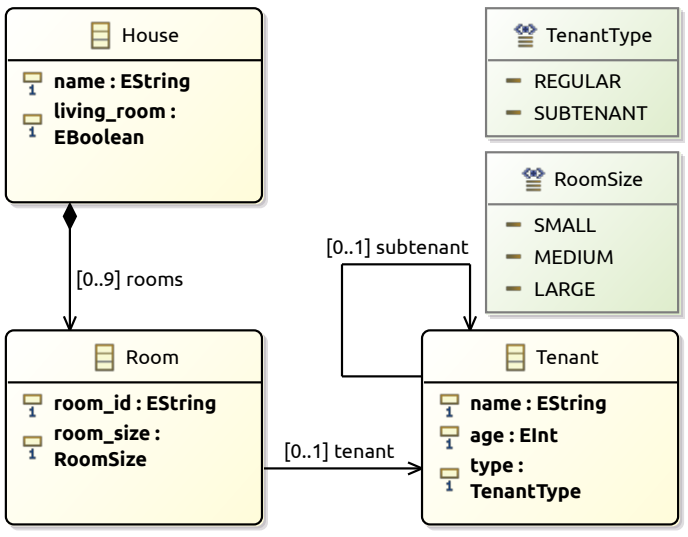
$f_{15}(Im_{15}) = \quad f_{14}(Im_{14}) \sqcup f_{DataField}(Im_{DataField})$ (Definition 4.4.27)
$f'_{15}(IG_{15}) = \quad f'_{14}(IG_{14}) \sqcup f'_{DataField}(IG_{DataField})$ (Definition 4.4.32)

A visual representation of Tm_{15} and Im_{15} can be found in Figure 6.30. Similarly, a visual representation of TG_{15} and IG_{15} can be found in Figure 6.31. Please note that because of the definitions of $f_{15}(Im_{15})$ and $f'_{15}(IG_{15})$, we have that $f_{15}(Im_{15}) = IG_{15}$ and $f'_{15}(IG_{15}) = Im_{15}$. Furthermore, $f_{15}(Im_{15})$ and $f'_{15}(IG_{15})$ are valid mapping functions themselves, such that they can be combined with another mapping function in the next step.

The final step is just for completeness and shows the addition of a data field for the last time. This time, it is shown that fields can still be added to already existing types with (containment) relations, by adding a boolean attribute to the rooms. The final model and graphs can be seen in the visualisations.

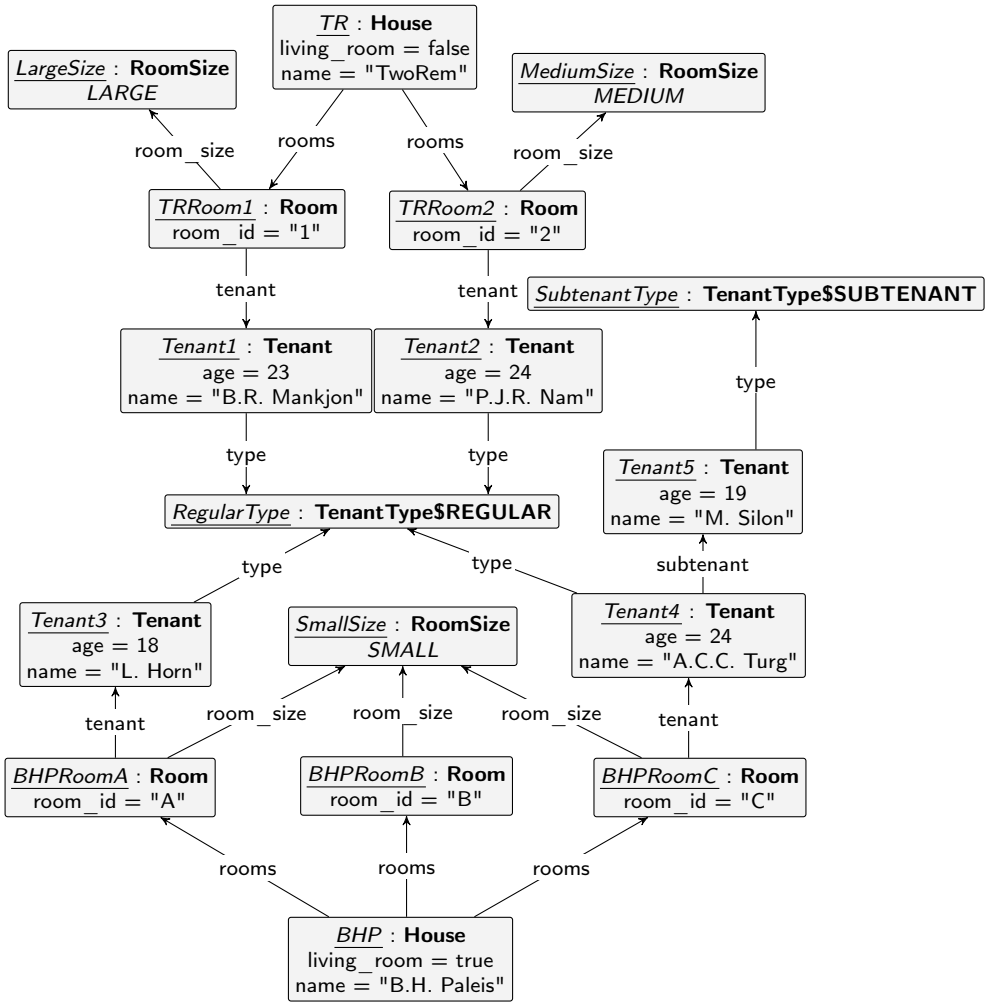


(a) Instance Model Im_{15}

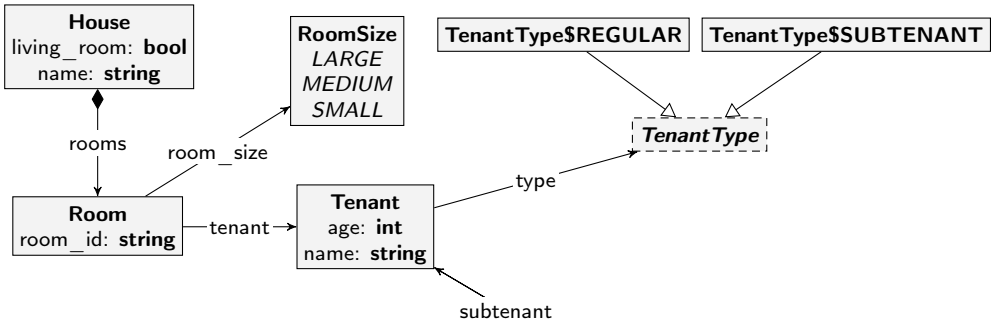


(b) Type Model Tm_{15}

Figure 6.30: The Ecore model after the final step



(a) Instance Graph IG_{15}



(b) Type Graph TG_{15}

Figure 6.31: The GROOVE graphs after the final step

Chapter 7

Conclusion

In Chapter 1 of this thesis, the need for verification in complex software projects was discussed. Modern approaches to software verification use models to verify a piece of software. However, not all models are suited for this task. Moreover, creating multiple models in different modelling languages for the same software is time-consuming and expensive. The use of model transformations to automatically transform models between different modelling languages was presented as a solution. However, in order for model transformations to be of use in the context of software verification, a formal foundation for the model transformations is required. This thesis has shown a formal foundation for model transformations between EMF/Ecore and GROOVE, which can be of use in the field of software verification.

As discussed in Section 2.1, EMF/Ecore is a popular modelling language for modelling software. However, its models are not well-suited for verification. As discussed in Section 2.2, GROOVE is created especially for software verification but uses a different modelling language than EMF/Ecore. Therefore, model transformations should be used to convert between the two models automatically. A formalisation of the modelling languages themselves was provided in Chapter 3. These formalisations are the foundation of the formalisation of the model transformations presented as part of the transformation framework.

The transformation framework presented in Chapter 4 provides the main result of this thesis. The transformation framework includes a formalisation of transformation functions between Ecore models and GROOVE graphs, which allows the user to reason about these transformation functions formally. Furthermore, the transformation framework has presented a structured way to compose these transformation functions iteratively while maintaining the correctness. The composability of these transformation functions is essential, as it allows the user of the framework to build significant model transformations without loss of correctness. The correctness property is relevant in the field of software verification, as it allows for verification of Ecore models within GROOVE, without the loss of confidence that the results might be incorrect due to transformation errors.

In order to further validate the transformation framework, Chapter 5 has presented a small library of transformations that can be used within the transformation framework. The transformations from the library presented in this chapter are all small transformations with a few elements, that can be added to a larger model using the framework. In this way, the transformations can be used to compose significant transformations. The library of transformations allows a user to build specific models without the need to define transformation functions for each created model.

Finally, Chapter 6 has shown an application of the transformation framework and the library of transformations in a practical example. Throughout the chapter, a single model is built from scratch, showing each of the steps taken to build the model using the transformation framework. Each step combines a transformation from the library of transformations with the existing transformation function, making the transformation function larger and allowing more complex models to be transformed.

In this final chapter, the work presented by the thesis will be concluded. First, the advantages and limitations of the work will be discussed. Then the work will be evaluated and the research questions introduced in Section 1.4 will be answered. Finally, some proposals for future work are discussed.

7.1 Advantages & Limitations

Although the approach taken by this thesis has some distinct advantages over other possible methods for proving the correctness of model transformations, there are also some limitations of the work that need to be discussed. These advantages and limitations are further discussed in this section before the work is evaluated.

As explained earlier, the transformation framework presented in Chapter 4 is considered the main result of this thesis. This framework is compelling, in that it allows to compose model transformations, which allows for creating possibly infinitely large model transformations between Ecore and GROOVE. In order to have composable model transformations, the concept of combining models and graphs is used. Within this work, it has been chosen to maintain the correctness of the transformation at each step. This correctness means that only valid or consistent models and graphs are used within each step. The use of correct models and graphs is favourable because it makes proving the correctness of the combination much more straightforward. The correctness properties of the individual models and graphs can be used to prove the correctness of the combinations.

Maintaining correctness in each step of the composition is a definite advantage to maintaining the proof of correctness. However, it also presents limitations for the transformation steps. Because of the required correctness properties, each transformation step has to be valid itself. For some transformations, this means quite significant transformation steps. For example, when introducing a new field on the type level, it is required to introduce the value for this field for all related objects on the instance level within one step. The introduction of all values at once is the only way that correctness is maintained. However, these are already quite large proofs, and they become increasingly complicated when adding inheritance. If a field is added to a supertype, a value for the field must be introduced for all instances of the supertype and its subtypes. Introducing values in this way means that an even more significant transformation step is needed to achieve such a composition. In practice, it is possible to use more substantial transformations, but it means that the complexity of proving each transformation step is increased.

The consequences of this limitation are directly visible from the library of transformations, Chapter 5. The transformations that introduce new fields are already quite complex, especially in their proofs. Furthermore, these transformations cannot be used on extended types because of the limitation above. Separate transformations need to be proven to allow the addition of fields to an extended type.

Because of the limitations of the transformation framework and the limited amount of time available for this thesis, the library of transformations is quite small and incomplete. Only a selected set of transformations is presented here, which does not even cover all concepts of Ecore. On the side of Ecore, the following concepts still need to be covered:

- The introduction of fields on types that are extended by other types, as discussed above.
- The introduction of fields typed by different container types still needs to be covered. Only one transformation shows the use of a setof-type in conjunction with a containment property (Section 5.2.9 and Section 5.3.9), but the other containers also need to be covered. Also, container types containing attributes still need to be covered.
- The concept of multiple inheritance, which is supported by Ecore and GROOVE, but not used in any transformation. Only one transformation with 'single' inheritance is shown as part of Section 5.2.3 and Section 5.3.3.
- Most of the different model properties (Definition 3.2.10), defaultValue, identity, keyset, opposite and readonly to be precise, are not yet covered by any of the transformations. The model properties that are covered, abstract and containment are not yet covered in their full potential.
- The introduction of constants and their corresponding values is not covered yet. Since they are only used in conjunction with defaultValue properties, it makes sense to cover them at the same time.

Covering all concepts of Ecore with one or more encodings in GROOVE would mean that all concepts of GROOVE are also covered. However, this way of achieving coverage means the addition of a lot more transformations, which could be its own research. Therefore, this is considered future work as described in Section 7.3.

A different limitation of this thesis, in general, is the focus on syntactical correctness only. No effort has been made to prove the correctness of the semantics of a model under transformation. This correctness property has been excluded on purpose, as EMF/Ecore is a quite general modelling framework in which a lot of different software models can be expressed. Therefore, it is difficult to prove something about the semantics on an abstract level. A consequence of this decision is that curious encodings are possible, which are still syntactically correct. For example, one might create an encoding that multiplies all integer values within a model with a certain number x, when transformed into a GROOVE graph. Then, a different transformations function can be used to convert back to a model, dividing all integer values with the same number x. Although this is syntactically correct, it could have enormous implications for the use within software verification, as the values of the model have changed. If this is not taken into account beforehand, the results of the software verification could still be questionable.

Although the work presented by this thesis has some limitations, the work is still considered a useful contribution. It is believed (although not proven) that it is possible to work around the limitations of the transformation framework, possibly with more substantial transformations. Moreover, the semantics of a transformation could be addressed in future research and does not invalidate the work presented here. Therefore, the work presented in this thesis should be used as a foundation, rather than a piece of work that is ready to use.

7.2 Evaluation

This section will evaluate the research carried out in this thesis and answer the main research question presented in Section 1.4. In order to answer the main research question, the subquestions need to be evaluated and answered. The answers to the subquestions are the following:

1. "What is a suitable formalisation of Ecore models and what Ecore models are valid within this formalisation?"

   For this thesis, a suitable formalisation of Ecore was presented in Section 3.2. The formalisation is considered suitable since it can express almost all concepts of Ecore. Therefore, it is also able to express all the models used throughout this thesis, which are type models and instance models.

   The set of valid models within this formalisation depend on the model level. For type models, the set of consistent models is constrained by Definition 3.2.11. For instance models, the set of valid models is constrained by Definition 3.2.19. These definitions answer the question which models are valid within the formalisation, and the examples shown throughout this thesis show the existence of these models.

   The answer to this question is validated using the existing theory available on Ecore. Also, each part of the formalisation can be traced back to elements within the Ecore metamodel. Together, the answer to the first question is considered validated.

2. "What is a suitable formalisation of GROOVE grammars and what GROOVE grammars are valid within this formalisation?"

   Just like the previous question, a suitable formalisation was presented within this thesis. The suitable formalisation for GROOVE graphs can be found in Section 3.3. The formalisation is considered suitable since it can express the relevant GROOVE graphs used throughout this thesis, which are type models and instance models.

   The set of valid grammars within the formalisation depend on the set of graphs defined for a GROOVE grammar. In order to answer the main research question, it is only relevant to know when type graphs and instance graphs are valid. For type graphs, the set of consistent graphs is constrained by Definition 3.3.5. For instance graphs, the set of valid graphs is constrained by Definition 3.3.10. These definitions answer the question of which graphs are valid within the formalisation, and the examples shown throughout this thesis show the existence of these graphs. Since the main research question only needs to show the validity of the presented graph types, the answer to the question is sufficient.

   Once more, the answer to this question is validated using the existing theory available. Existing theory on GROOVE and graph theory has been used to validate the formalisation, and therefore the answer to this question is considered validated.

3. "What is a suitable formalisation for the model transformations between Ecore and GROOVE?"

   A formalisation for model transformations between Ecore and GROOVE was given as part of the presented transformation framework in Chapter 4. Definition 4.3.25, Definition 4.3.30, Definition 4.4.26 and Definition 4.4.31 are the relevant definitions that specify the properties of (composable) transformation functions between different models and graphs.

   The formalisation is considered suitable since it gives rise to a significant set of possible transformations, which allow for building large complex models. The existence and correctness of this set are validated using the library of transformations in Chapter 5, which defines and proves a small set of possible transformations within this formalisation. These transformations show the existence of model transformations between Ecore and GROOVE that fit within the presented formalisation. Furthermore, the transformations can map between the formalisations of Ecore and GROOVE by using these formalisations as input and output, giving confidence in the fact that multiple transformation functions can cover all elements of Ecore and GROOVE. Therefore, the formalisation of the model transformations themselves can be considered suitable.

4. "What model transformations are correct within the formalisation?"

Definition 4.3.25, Definition 4.3.30, Definition 4.4.26 and Definition 4.4.31 constrain the transformation functions that are considered syntactically valid within the formalisation. Furthermore, the library of transformations in Chapter 5 has shown some examples of transformations that are proven to be valid with respect to these definitions, and each of these transformations is also reversible. Therefore, multiple transformations have been presented which are each syntactically correct within the formalisation. Furthermore, the formalisation allows us the combine these transformation functions (as presented in Chapter 4), while maintaining the correctness of the functions. This has also been proven as part of this thesis.

The answer to this question is validated by validating all the correctness proofs. For this thesis, all the proofs are validated using the Isabelle theorem prover, as presented by Section 2.3. Therefore, it is validated that the proofs are correct, meaning that by following the definitions, it is clear which model transformations are syntactically valid within the formalisation. Therefore, the question is considered answered.

5. "How can correct model transformations between Ecore and GROOVE be composed?"

   As explained earlier in Section 1.5, answering this question consists of two parts. First Chapter 4 has shown on a formal level how model transformations are combined. This formalisation has been done for transformations at both levels. Secondly, it has been shown how to compose these transformations in practice, which can be found in Chapter 6.

   Chapter 4 describes the composability of model transformations by explaining the necessary definitions to combine models, graphs and transformation functions. Using Theorem 4.3.29, Theorem 4.3.36, Theorem 4.3.36 and Theorem 4.4.38 it is possible to show that these definitions indeed give rise to a new transformation function, which is then proven to be syntactically correct.

   Chapter 6 shows how to apply these definitions by composing a large transformation function out of the small transformation functions presented in the library of transformations (see Chapter 5). This example shows how the definitions from Chapter 4 can be applied in practice, answering the practical part of the question.

   As explained before, all proofs within this thesis have been validated using the Isabelle theorem prover, as presented by Section 2.3. That means that the proofs presented to prove the composition of transformation functions are correct. The correctness of these proofs validates that the method of composing transformation functions is syntactically correct. Furthermore, the example of Chapter 6 shows that it is possible to apply these definitions in practice, which validates that there is actual use for the composability definitions. Together this shows that all parts of the answer are validated, so the question is considered validated.

With the answers to the subquestions given, it is possible to answer the main research question:

"What is a suitable formalisation for composable model transformations between Ecore and GROOVE that gives rise to correct model transformations between Ecore and GROOVE?"

In order to give a formalisation for the model transformations, formalisations for Ecore and GROOVE were needed. Subquestions 1 and 2 have shown suitable transformations for both languages. The formalisation for composable model transformations is given as part of Chapter 4. Subquestions 3 explains that the formalisation is suitable, whereas subquestion 5 shows that the formalisation indeed allows for composable model transformations. The correctness of these transformations is proven, which is explained in subquestions 4 and 5. Finally, Chapter 5 and Chapter 6 have shown that there indeed exist such model transformations in practice. The existence of these model transformations is also explained in subquestions 3 and 5. Altogether, these answer the main research question, as the thesis has shown that there exists a suitable formalisation for composable model transformations between Ecore and GROOVE. It has also shown that this formalisation gives rise to syntactically correct model transformations between Ecore and GROOVE.

7.3 Future work

Although this thesis has provided a complete framework for composing transformation functions between Ecore and GROOVE, much work remains to be done to make this solution viable in practice. In future work, limitations of this work could be addressed, and additional work can be performed in order to make implementations of the work possible. Possible improvements and future work are discussed in this section.

7.3.1 Improvements to the transformation framework

This thesis has provided a transformation framework for composable model transformations between Ecore and GROOVE. As discussed in Section 7.1, the choices made within this transformation framework have advantages and limitations. An important limitation is the requirement to maintain correctness at each step of composing a transformation function. As a consequence, the small transformations presented in the library of transformations can become quite substantial, and are more difficult to prove.

One could investigate the possibility of making the requirement of correctness less strict. This could be done by allowing some partial compositions, that need to be applied in a specific order, to achieve once more a correct transformation. These partial compositions would make the individual transformation steps smaller, and thus easier to prove. However, it will probably introduce some new requirements to maintain the general correctness of the transformation, which should be researched further.

7.3.2 Complete the library of transformations

This thesis has provided a non-exhaustive library of transformations that can be applied within the transformation framework. As explained in Section 7.1, several concepts of Ecore are not yet covered. Future research might focus on defining a complete library of transformations, that can compose every possible transformation function. Such research would consist of adding transformations for missing concepts and possibly a proof that all valid models and graphs can indeed be built.

In order to achieve completeness of the library, at least the following transformations should be provided:

- For all transformations that introduce fields, there should be counterparts that can introduce these fields on types that are extended by other types. In order to create these transformations, the set of shared objects should contain not only all instances of the supertype but also all instances of all subtypes. Furthermore, for all these objects, a value needs to be introduced for the newly introduced field. Please note that this should be done for both the introduction of fields on abstract classes and the introduction of fields on regular classes. When done carefully, the proof created for the instance level could be reused for both of these transformations. Furthermore, the proofs created for regular classes might be able to replace the existing transformations for introducing fields.
- New transformations need to be introduced that cover all the possible container types. This means that transformations should be created that introduce a field typed by a bagof, setof, seqof or ordof container type, containing any other possible type. Because of the limitations of the transformation framework, it will not be possible to prove this for all contained types at once, so multiple transformations will be needed here.
- A transformation should be created that allows for introducing multiple inheritance. Multiple inheritance can be introduced by either creating a transformation that introduces a new type that extends from a set of existing types, or by creating a transformation that introduces a set of new types from which one existing subtype does inherit. The new types should be created in one transformation, to be able to proof that the inheritance relation remains valid.
- For all transformations introducing some field, an additional transformation would need to be created that introduces a defaultValue property on the corresponding field. This transformation would also need to introduce a new constant and its value.
- A transformation should be created that introduces a new class type with an identity property. Because of the limitations of the transformation framework, all the attributes that are part of the identity property need to be introduced as well. Furthermore, all the instances of the new class type need to be created within the same transformation, and all values for all the attributes should be defined. Such a transformation would be substantial, and would undoubtedly benefit from the possibility to do partial compositions, as proposed by Section 7.3.1.
- Multiple transformations should be created which introduce a new field with an keyset property. Because of the limitations of the transformation framework, all the attributes that are part of the keyset property need to be introduced as well. Furthermore, a value for the field and all related attributes needs to be introduced at the instance level, within one transformation step. Once more, such a transformation would be substantial, and would undoubtedly benefit from the possibility to do partial compositions, as proposed by Section 7.3.1.
- Multiple transformations should be created which introduce a new field with an opposite property. Introducing the opposite property can be done between existing types and instances, it only has to be made sure that transformations exist for all possible types for a field with an opposite property. Extra care should be given to introducing a new field that has a opposite property and a containment

property on one of the fields. In this case, it is needed to introduce the instances contained by the containment relation as well.

- For all the transformations introducing some field, an additional transformation would need to be created that introduces a readonly property on the corresponding field. Since a read-only property is only used for the semantics of the model and not anywhere in the syntax, this should be quite trivial.
- Transformations need to be created for some combinations of properties, especially containment and identity, to ensure that models combining these properties can be created within the transformation framework.

Although it is believed that introducing all these transformations would eventually result in a complete library of transformations, no proof for this claim exists. Therefore, this should be taken into account when performing this work.

7.3.3 Add more encodings

This thesis has shown that elements in Ecore can have multiple encodings in GROOVE. An example of this is given in Section 5.2.4 and Section 5.3.4. Future work might focus on defining more encodings for different elements of Ecore. Having the choice between multiple encodings for the same element when transforming an Ecore model to GROOVE might be beneficial for the verification of specific properties. Different encodings might have their advantages and disadvantages within the field of verification, and therefore a choice between encodings might make more use cases possible.

Each of these encodings should be proven correct within the transformation framework. Moreover, as shown in Section 5.2.7 and Section 5.3.7, it might be the case that introducing new encodings might also need specific transformations for related elements that use the encoding. Therefore, providing more encodings could be a quite substantial amount of work.

Besides introducing more encodings, such research would also include research on the possible encodings for an element. Furthermore, the practical use of each of these encodings could be investigated.

7.3.4 Implementation

The framework presented in this thesis only represents a mathematical foundation. In order to apply this framework in practice, it needs to be implemented as part of EMF, GROOVE or as a standalone tool. Such implementation would allow a user to automatically transform Ecore models into GROOVE graphs and vice versa while having the framework as a formal foundation to show the syntactical correctness of the transformations.

The implementation of this framework is not a trivial task in itself. Such an implementation would need to figure out how to decompose models and graphs into their separate components in order to build a transformation. This decomposition might give rise to more fundamental research, as the current framework does not describe the decomposition of models and graphs. Furthermore, an approach to the decomposition of models and graphs might be an ambiguous process, meaning that a model or graph can be decomposed in more than one way. Research on how to handle this ambiguity also needs to be carried out when applicable. Furthermore, an implementation would need to deal with multiple possible encodings when composing the transformation function, as it is possible to transform a model using multiple correct graph encodings.

References

Scientific publications

[2] J. Bang-Jensen and G. Z. Gutin. *Digraphs theory, algorithms and applications*. Springer, 2010. ISBN: 978-1-84800-998-1. DOI: 10.1007/978-1-84800-998-1.

[3] E. Biermann, C. Ermel, and G. Taentzer. "Formal foundation of consistent EMF model transformations by algebraic graph transformation". In: *Software & Systems Modeling* 11.2 (2011), pp. 227–250. DOI: 10.1007/s10270-011-0199-7.

[4] H. Bruintjes. *Bridging GROOVE to the world using a conceptual language model*. Nov. 2012. URL: http://purl.utwente.nl/essays/62759.

[6] C. Ermel, F. Hermann, J. Gall, and D. Binanzer. "Visual Modeling and Analysis of EMF Model Transformations Based on Triple Graph Grammars". In: *Electronic Communications of the EASST* 54 (Oct. 2012). DOI: 10.14279/tuj.eceasst.54.771.

[10] F. Hermann, H. Ehrig, U. Golas, and F. Orejas. "Formal analysis of model transformations based on triple graph grammars". In: *Mathematical Structures in Computer Science* 24.4 (2014). DOI: 10.1017/S0960129512000370.

[13] E. Kindler and R. Wagner. *Triple Graph Grammars: Concepts, Extensions, Implementations, and Application Scenarios*. Tech. rep. tr-ri-07-284. Software Engineering Group, Department of Computer Science, University of Paderborn, June 2007. URL: https://www.hni.uni-paderborn.de/pub/7067.

[14] A. Kleppe and A. Rensink. "On a Graph-Based Semantics for UML Class and Object Diagrams". In: *Graph Transformation and Visual Modelling Techniques*. Ed. by C. Ermel, J. De Lara, and R. Heckel. Electronic Communications of the EASST 69160R. Proceedings of the Seventh International Workshop on Graph Transformation and Visual Modeling Techniques (GT-VMT 2008). EASST, 2008. DOI: 10.14279/tuj.eceasst.10.153.

[16] L. Noschinski. "Graph Theory". In: *Archive of Formal Proofs* (Apr. 2013). http://isa-afp.org/entries/Graph_Theory.html, Formal proof development. ISSN: 2150-914x.

[17] O. Semeráth, Á. Barta, Á. Horváth, Z. Szatmári, and D. Varró. "Formal validation of domain-specific languages with derived features and well-formedness constraints". In: *Software & Systems Modeling* 16.2 (May 2017), pp. 357–392. ISSN: 1619-1374. DOI: 10.1007/s10270-015-0485-x.

[21] F. Wiedijk. "Comparing Mathematical Provers". In: *Mathematical Knowledge Management*. Ed. by A. Asperti, B. Buchberger, and J. H. Davenport. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 188–202. ISBN: 978-3-540-36469-6. DOI: 10.1007/3-540-36469-2_15.

Documentation & Specification

[5] *Package org.eclipse.emf.ecore*. Jan. 2015. URL: http://download.eclipse.org/modeling/emf/emf/javadoc/2.9.0/org/eclipse/emf/ecore/package-summary.html (visited on 12/02/2019).

[18] *Specification of the Object Constraint Language version 2.4*. Feb. 2014. URL: http://www.omg.org/spec/OCL/2.4 (visited on 12/03/2019).

[20] M. Wenzel, C. Ballarin, S. Berghofer, J. Blanchette, T. Bourke, L. Bulwahn, A. Chaieb, L. Dixon, F. Haftmann, B. Huffman, L. Hupel, G. Klein, A. Krauss, O. Kunčar, A. Lochbihler, T. Nipkow, L. Noschinski, D. v. Oheimb, L. Paulson, S. Skalberg, C. Sternagel, and D. Traytel. *The Isabelle/Isar Reference Manual*. Aug. 2018.

Other references

[1] V. Bacvanski and P. Graff. "Mastering Eclipse Modeling Framework". In: *EclipseCon*. The Eclipse Foundation, Feb. 2005. URL: https://www.eclipsecon.org/2005/presentations/EclipseCon2005_Tutorial28.pdf (visited on 12/02/2019).

[7] R. Gronback. *Eclipse Modeling Framework*. URL: https://www.eclipse.org/modeling/emf/ (visited on 12/02/2019).

[8] *GROOVE*. URL: https://groove.ewi.utwente.nl/ (visited on 12/02/2019).

[9] C. Guindon. *EMF-IncQuery*. Mar. 2016. URL: https://projects.eclipse.org/projects/modeling.incquery (visited on 12/03/2019).

[11] *Isabelle Community Wiki*. URL: https://isabelle.in.tum.de/community/Main_Page (visited on 12/02/2019).

[12] *Isabelle2019*. June 2019. URL: https://isabelle.in.tum.de/ (visited on 12/02/2019).

[15] *Newest 'isabelle' Questions on Stack Overflow*. URL: https://stackoverflow.com/questions/tagged/isabelle (visited on 12/02/2019).

[19] *The 3-Clause BSD License*. Open Source Initiative, July 22, 1999. URL: https://opensource.org/licenses/BSD-3-Clause (visited on 12/02/2019).

Appendix A

Example Isabelle Theory

This appendix contains an export of the Multiplicity theory used to formalise Definition 3.1.1 within Isabelle. It is used in Section 2.3 to explain the concepts of Isabelle theories.

theory *Multiplicity*
  imports *Main*
begin

A.1 Linear order of natural numbers including unbounded

datatype $\mathcal{M}$ = *Star* | *Nr nat*

notation
  Star ((⋆) *1000*) **and**
  Nr ((**-**) [*1000*] *1000*)

instantiation $\mathcal{M}$ :: *linorder*
begin

fun *less-eq-*$\mathcal{M}$ :: $\mathcal{M} \Rightarrow \mathcal{M} \Rightarrow$ *bool* **where**
less-eq-$\mathcal{M}$ - ⋆ = *True* |
less-eq-$\mathcal{M}$ (**a**) (**b**) = ($a \leq b$) |
less-eq-$\mathcal{M}$ - - = *False*

fun *less-*$\mathcal{M}$ :: $\mathcal{M} \Rightarrow \mathcal{M} \Rightarrow$ *bool* **where**
less-$\mathcal{M}$ (**-**) ⋆ = *True* |
less-$\mathcal{M}$ (**a**) (**b**) = ($a < b$) |
less-$\mathcal{M}$ - - = *False*

instance proof
  fix $x\ y\ z$ :: $\mathcal{M}$
  show $(x < y) = (x \leq y \wedge \neg\ y \leq x)$
  proof (*induction x arbitrary*: y)
    case *Star*
    then show *?case* **by** *simp-all*
  next
    case (*Nr x*)
    then show *?case* **by** (*cases y*) *auto*
  qed

  show $x \leq x$ **by** (*induction x*) *simp-all*
  then show $x \leq y \Longrightarrow y \leq x \Longrightarrow x = y$
  proof (*induction x arbitrary*: y)
    case *Star*
    then show *?case* **by** (*cases y*) *simp-all*
  next
    case (*Nr x*)
    then show *?case* **by** (*cases y*) *simp-all*
  qed

  show $x \leq y \Longrightarrow y \leq z \Longrightarrow x \leq z$
  proof (*induction x arbitrary*: $y\ z$)
    case *Star*

```
    then show ?case by (cases y) simp-all
  next
    case (Nr x)
    then show ?case
    proof (induction y arbitrary: z)
      case Star
      then show ?case by (cases z) simp-all
    next
      case (Nr x)
      then show ?case by (cases z) simp-all
    qed
  qed

  show x ≤ y ∨ y ≤ x
  proof (induction x arbitrary: y)
    case Star
    then show ?case by simp
  next
    case (Nr x)
    then show ?case by (cases y) auto
  qed
qed

end
```

A.2 Definition of multiplicity

```
type-synonym multiplicity = 𝓜 × 𝓜

definition lower :: multiplicity ⇒ 𝓜 where
  lower m ≡ fst m

declare lower-def[simp add]

definition upper :: multiplicity ⇒ 𝓜 where
  upper m ≡ snd m

declare upper-def[simp add]

locale multiplicity = fixes mult :: multiplicity
  assumes lower-bound-valid[simp]: lower mult ≠ ⋆
  assumes upper-bound-valid: upper mult ≠ 0
  assumes properly-bounded[simp]: lower mult ≤ upper mult

context multiplicity
begin

lemma upper-bound-valid-alt[simp]: upper mult ≥ 1
  using less-𝓜.elims not-less upper-bound-valid by fastforce

end

abbreviation multiplicity-notation :: 𝓜 ⇒ 𝓜 ⇒ multiplicity ((-/..-) [52, 52] 51) where
  l..u ≡ (l,u)

definition within-multiplicity :: nat ⇒ multiplicity ⇒ bool (infixl in 50) where
  n in m ≡ lower m ≤ n ∧ n ≤ upper m

theorem mult-zero-unbounded-valid[simp]: n in 0..⋆
  unfolding within-multiplicity-def
  by simp

theorem mult-single-value-bound[simp]: n in m..m ⟹ n = m
  unfolding within-multiplicity-def
  by auto

end
```