TECHNISCHE UNIVERSITÄT BERLIN
FAKULTÄT IV ELEKTROTECHNIK UND INFORMATIK
BACHELORSTUDIENGANG INFORMATIK

WILLIAM BOMBARDELLI DA SILVA

Towards Synchronizing Relations Between Artifacts in the Java Technological Space

Work presented in partial fulfillment
of the requirements for the degree of
Bachelor in Computer Science

Advisor: Prof. Dr.

Berlin
February 2016

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig und eigenhändig sowie ohne unerlaubte fremde Hilfe und ausschließlich unter Verwendung der aufgeführten Quellen und Hilfsmittel angefertigt habe.

Berlin, den

....
Unterschrift

ACKKNOWLEDGMENTS

....

ABSTRACT

...

Keywords: 1. 2.

.

ZUSAMMENFASSUNG

.

Schlagwörter: ..

LIST OF FIGURES

LIST OF ABBREVIATIONS AND ACRONYMS

TGG	Triple graph grammar

CONTENTS

1 INTRODUCTION

The techniques for software development has changed in the course of time since the rise of general-purpose programmable computers and specially in the second half of the 20th century with the rise of digital computers (Ceruzzi, 2003). In the beginning of digital computer programming machine code was used to describe algorithms, but as complexity and size of such algorithms got bigger this technique soon became impracticable, what evoked the need for a more sophisticated way of programming these digital machines. The assembly languages (also known as low-level programming languages) came to solve this problem, but clearly the complexity kept increasing as well as the need for new techniques and technologies for software programming. The popularization of computing, and the increasing application of computers in the practice urged the creation of high-level programming languages (e.g. Cobol, Fortran), which kept evolving mainly in regard to the needs of the software market (Ceruzzi, 2003). More sophisticated languages (e.g. C, Pascal) and new paradigms (e.g. modular and object-oriented programming) also arose in the late 20th century. But the evolution of software development does not seem to stop, evidenced by the lately increasing research on new software engineering techniques such as the Model-driven Engineering.

The newest characteristics of the information system market, like the constant evolution of the software systems, the interoperability between them and the big number of developers working in a common software artifact has required the use of software models; and the research on how to apply systematically and correctly such models in the development processes, what is called Model-driven Engineering (MDE) or Model-driven Software Development (MDSD) (France and Rumpe, 2007). This Bachelor thesis aims therefore to explore one specific realm of Model-driven Engineering research, namely the problem of synchronization of models (or artifacts) in the Java technological space. The goal here is to pick commonly used meta-models in Java systems, describe them and identify their relations, so that they can be synchronized.

TALK ABOUT THE REMAINDER OF THE DOCUMENT.

1.1 Background

According to Czarnecki and Helsen (2006, p. 21): "*Models are system abstractions that allow developers and other stakeholders to effectively address concerns, such as answering a question about the system or effecting a change*". By defining a model as a system abstraction, it becomes clear, that a software system might have several models abstracted from it, each one representing certain aspects of the whole system. These models also have relations between them, in the sense that they all are supposed to describe the actual system consistently by not presenting logical contradictions. Here examples of models are *UML class diagram*, *Use Cases*, or even the source-code itself. The term model and artifacts will be used interchangeably throughout this document.

The constant evolving nature of current large-scale software systems causes their models to be constantly changed (Diskin, 2011). But in order to maintain this whole network of interconnected models consistent, the changes have to be forwarded through the network, i.e. all models have to be synchronized. Suppose one have a *UML Class Diagram*, a series of *UML Sequence Diagrams* and the source-code. If a method has its name changed in the class diagram, all occurrences of this method have to have their name updated in the sequence diagrams and in the source-code. It turns out though, that neither a model synchronization tool comprising the most common meta-models used nowadays in Java information systems is known by us, nor have we found clearly defined relation definitions between them on the literature, even though an expressive effort has been made by the academic community to create solutions for the problem of model synchronization.

1.2 Motivation

Problems of software quality, like security flaws, software not working according to its specification (i.e. bugs), or even performance questions are an issue of interest of both the industrial and the academical community. Model-driven Engineering (MDE) claims to leverage the use of models in order to help bridge the gap between the problem and the implementation domains (France and Rumpe, 2007), and therefore to reduce some problems of quality. This discussion points to the motivation of the use of models, but also highlights the need for synchronization methods, that allow the network of models of a system to be kept consistent. In fact, the synchronization could be done manually by the users, since they can a priori update all models related to the one under changes, but this manual process usually requires much time from expensive workforce and is error-prone. Automated (or at least partially-automated) model synchronization endeavors to reach a higher reliability on the models, as well as lower costs for the software maintainer.

In general, synchronized models enhance the documentation quality, once many of them are used for documentation purposes; ease the act of evolving software, by bridging the gap between problem (abstract) and implementation (concrete) levels; and support the debugging processes, once models are used to consult information about the system under study. On the other hand, if one model is not kept consistent with another, it may lose its validity, once the information it addresses cannot be trusted anymore, and consequently the user cannot rely on it anymore. If the number of inconsistencies between models is large enough, the users run the risk of not being able to use a big part of their set of models, what itself lowers the quality of the software.

Generally, the amount of inconsistencies tend to grow as the size of a program grows. The complexity of the network of models, as well as of the synchronization task increases therefore too, what also gives reason to the application of more robust synchronization techniques. This phenomenon occurs specially in the case of a Java program, so Java is used in this thesis as an example domain.

1.3 Objective

This bachelor thesis aims to (1) present some artifacts from the Java technological space, that might require synchronization, explaining their objectives and some basic elements from them; (2) formalize and explain the relations between these artifacts, so that the synchronization is possible; and finally (3) discuss how synchronization may be accomplished in a representative showcase.

This document presents the documentation of the three objectives mentioned before. Furthermore, the report of the difficulties and experiences found during the work process and an examination of possible future development and challenges of the realm is also a goal.

1.4 Methodology

In order to achieve the goals, the following procedure is taken. In the first moment a collection of common meta-models used in the Java technological space is to be defined, this is done through an state-of-art research, since that some meta-models have already been defined by other authors, plus the creation of our own versions of some meta-models. So for example, in this phase the choice of the applied meta-models (i.e. *UML Class Diagram* and *Java Code*) will be done and their definitions will be written.

Later on, given the defined meta-models, the relations between them can be written. So for example, in this phase the inherent relation between the *UML Class Diagram* meta-model and the *Java Code* meta-model will be written. Analogously, the relations between *Java Code*, *JavaDoc*, *UML Sequence Diagram* and other meta-models of the Java technological space are also to be defined. All of these relations are developed during the work of this thesis.

After having this network of meta-models ready, a showcase using a synchronization method from the current academic literature is applied to illustrate the synchronization between some representative meta-models. We work therefore with the hypothesis, that the meta-models can be found or defined; that some relations between them can be written in some language; and that some of these relations can be synchronized using a tool or technique available in the current literature.

It is out of the scope of this work a creation or implementation of a synchronization algorithm, as well as a theoretic analysis of the problem or the analysis of performance or completeness of the relations.

2 FOUNDATIONS

Before describing the development of this thesis is important to review some important definitions regarded to Model-driven Engineering. Below is a list of necessary basic concept definitions, that will be used throughout this document. Some of these definitions are rather narrower than they could be, but for the scope of this thesis they seem to be suitable.

Technological Space: According to the definition from Kurtev et al. (2002, p. 1): *"A technological space is a working context with a set of associated concepts, body of knowledge, tools, required skills, and possibilities. It is often associated to a given user community with shared know-how, educational support, common literature and even workshop and conference meetings"*. By *Java Technological Space* is meant the set of commonly used models, practices, techniques and technologies in Java software development. For example, object-oriented development, unit tests, code documentation and the *Java Virtual Machine* are items of the Java technological space.

System: *"A system is the primary element of discourse when talking about MDE."* (Favre, 2004a, p. 13). One example of a system, according to this definition, is a Java program, once it can be the primary handled element in a certain software engineering context. Nevertheless, this definition is wide enough to affirm that a UML Class Diagram is a system, since it can be the primary handled element in a certain context. This fact allows an easier-to-understand definition of model.

Model: According to Favre (2004a), a model is a possible role a system can play. A system plays the role of a model, when it represents another system (system under study, or SUS), i.e. By being so, when one refers to the a model M, it is meant, a system that represents (or abstracts) another system S, i.e. $M \mu S$. Seidewitz (2003) affirms that, models can be used (1) to describe a system, in this case the model makes statements about the SUS, an example is an *UML sequence diagram* employed to help understand the behavior of a Java program. But models can also be used (2) to specify a system, in this case it is used in the validation of the system, an example is a *UML class diagram* employed as design specification of a Java system. Further examples of models, according to this definition, are a *relational database diagram* of a database, the documentation of a system in *Java Doc* or even a Java source-code.

Modeling Language: A model is expressed, using a modeling language. *"A modeling language (L) is a set of models"* (Favre, 2004a, p. 13), that contains all the models M_i expressed in that language, i.e. $L \ni M_i$. Examples of modeling languages are the *UML*, the *diagram notation for relational database diagram* or the Java language.

Meta-model: Favre (2004b, p. 14) affirms also: *"A metamodel is a model of a modelling language."*. In other words, a meta-model specifies what can be written using a certain modeling language. One certain model M is conform to a meta-metamodel MM (i.e $M \epsilon MM$), if and only if, $M \in L$ and $MM \mu L$. Thus, examples of meta-models are *UML specification document* (OMG, 2007), the *entity-relationship meta-model* (Chen, 1976) or the Java meta-model (one example is to be found in Heidenreich et al. (2009)). Finally Seidewitz (2003, p. 29) also claims: *"Because a metamodel is a model, we express it in some modeling language"*. One example of a modeling language for meta-models is the *EMF Ecore*[1] (which is the modeling language for meta-models used in this thesis ans will be explained further below).

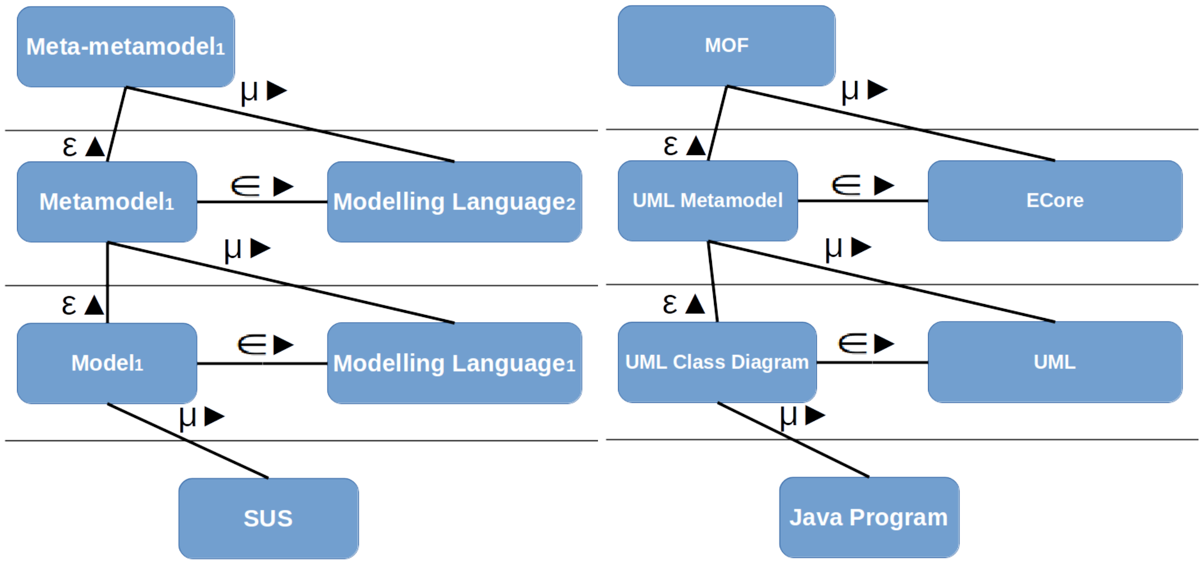
Meta-metamodel: Analogously to the meta-model definition, one can go forth and define meta-metamodel, which is a model that specifies a modelling language for meta-models. An example of meta-metamodel is the *MOF*(Omg, 2015), which the *EMF Ecore* is conform to. It is to note also, that such derivation can be done iteratively in the sense that a $meta^3model$ definition is also possible, although it is not useful for the scope of this thesis. The figure 2.1 illustrate our understanding of the definitions above.

Model Relation: Model relation here is defined abstractly as every relationship or constraint possible to happen between one source model and one target model. For instance, the models *UML class diagram* and Java code have a relation, because once a new class is created in the class diagram, the correspondent class has to be created in the Java code. Moreover, a *UML class diagram* with contracts definitions (pre and post-conditions) have a relation to the *JUnit model*, once that the formers have to be tested correspondingly in the latter.

Model Transformation: Model transformation can be viewed as common data transformation – very common in computer science – with the specificity of dealing with models Czarnecki and Helsen (2006). More specifically, model transformation is defined here as a function $t : M \rightarrow N$, where $t(m) = n$ means that a target model $n \in N$ is created from a source model $m \in M$, M and N being respectively the set of all valid models of the meta-models Φ_M and Φ_N. Practical example: Creation of Java code from *UML class diagram*. Note that, model transformation is by nature deterministic, unidirectional and does not preserve the information of the target model (e.g. comments in the Java code).

[1]https://eclipse.org/modeling/emf

Figure 2.1: On the left is a depiction of the theoretical definitions of system, model, metamodel, meta-metamodel and modeling language. Like stated before, a system is represented by models, which themselves are expressed in languages and are conform to meta-models. A more concrete and practical illustration of the definitions is on the right. This example shows a scenario very close to the implementation made in chapter 4.

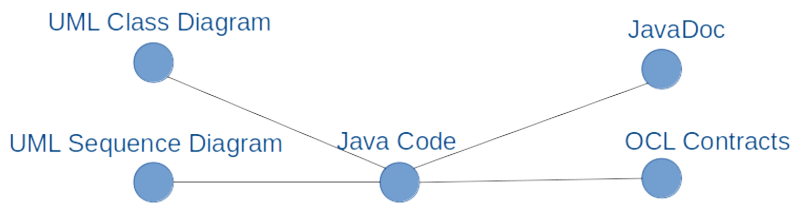


Source: The author

Model Synchronization: The goal of model synchronization is to maintain all relations between the models of a system consistent/correct as updates are performed over them Diskin (2011). More specifically, model synchronization is defined here as a function $s : M \times M \times \Delta_M \times N \times N \times \Delta_M \rightarrow M \times N$, where $s(m_0, m_1, \delta_m, n_0, n_1, \delta_n) = (m_2, n_2)$ means that final synchronized models m_2 and n_2 are created from the initial synchronized models m_0 and n_o and the modified non-synchronized models m_1 and n_1, considering the modifications (respectively δ_m and δ_n) performed over both. Practical example: Modification of a method name (δ_m) in the *UML class diagram*(m_0) has to be forwarded to the Java code(n_0), without losing extra information of it (e.g. comments). Note that, model synchronization is deterministic, bidirectional and preserves the informations of the both models. Other terms for model synchronization are *iterative* or *information preserving bidirectional model transformation*.

Network of Models: A network of models of a system S is an undirected graph $G = (V, E)$, whereas each vertex $v_i \in V$ represents a unique model i abstracting S, and an edge (v_i, v_j) exists if, and only if there is a relation defined between both models i and j. In the figure 2.2 is an example of a network of models, illustrating the possible complexity of such network. More discussion is to find in Mens and Van Gorp (2006).

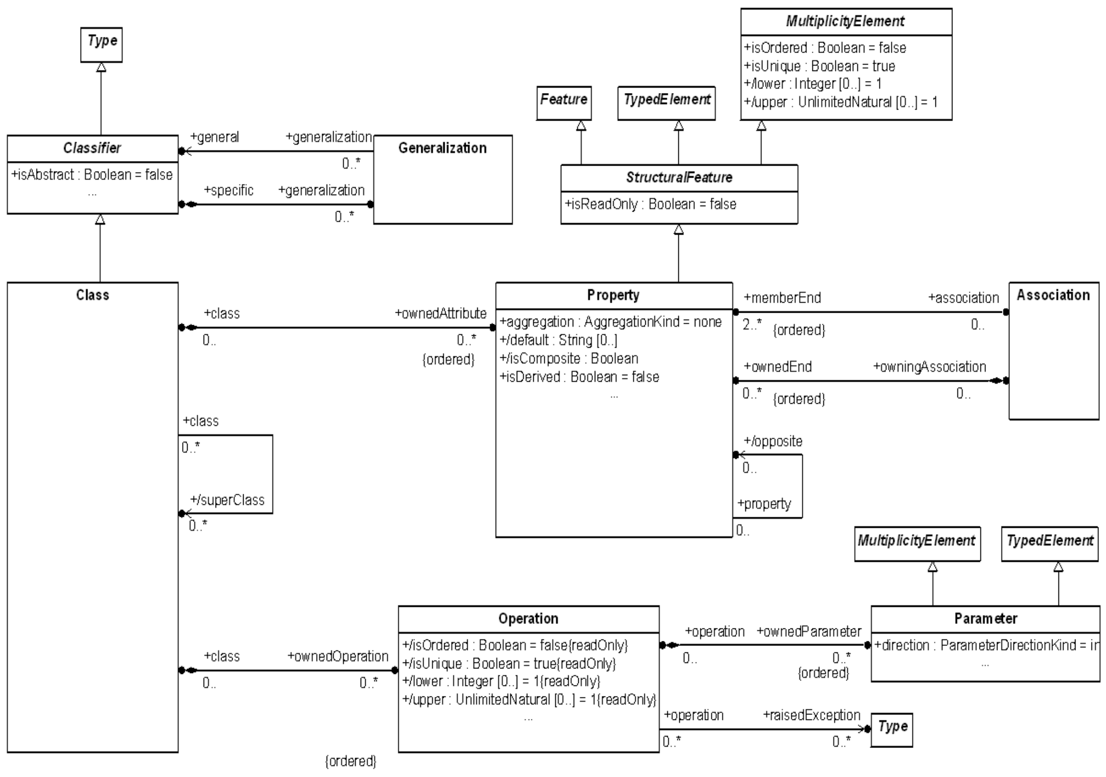
Figure 2.2: An example of a network of models very similar to the one developed in this work.



Source: The Author

Meta Object Facility: *"The Meta Object Facility (MOF) provides an open and platform-independent metadata management framework and associated set of metadata services to enable the development and interoperability of model and metadata driven systems"* (Omg, 2015). The MOF describes therefore the MOF modeling language, which is used to model the meta-metamodel utilized in this thesis. Essentially, it inherits much from the UML and deals with the ideas of classes, properties and associations, providing and extensible but simple fashion to define meta-models. The figure 2.3 shows the part of MOF.

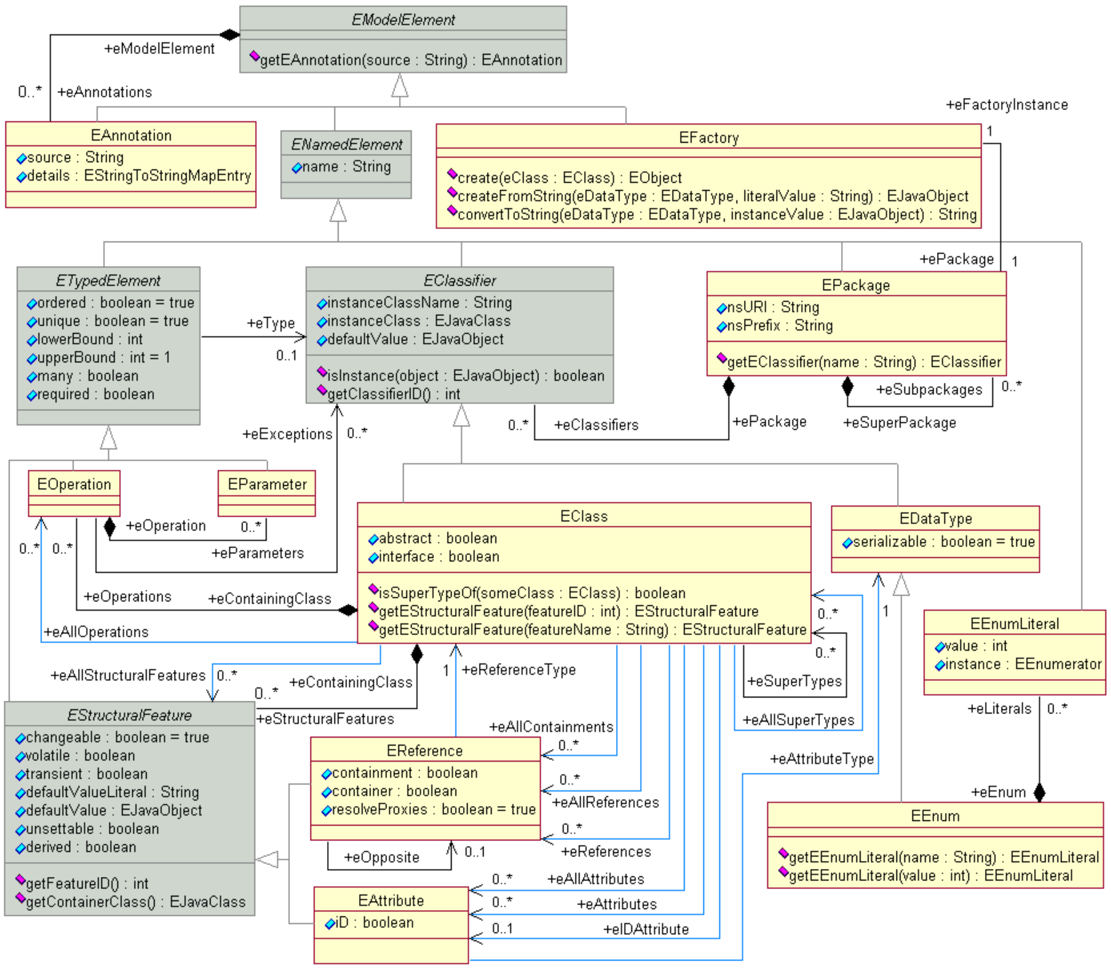
Figure 2.3: Part of MOF definition, which handles basically classes, properties, operations, associations, and generalization.



Source: (Omg, 2015, p. 27)

Ecore: Ecore[2] is the meta-metamodel utilized in this thesis to describe all the applied meta-models (e.g the Java meta-model). Ecore is an initiative of the EMF Project and aims to provide not only a meta-metamodel but a set of tools for criating meta-models, like an Eclipse plug-in generation feature, that enables the model developer to easily test and debug its meta-models. The Ecore meta-metamodel is at least similar to the essential MOF standard, and that is the reasons it is applied here. A proof of such compliance is not know by us though. The figure 2.4 shows the essential part of Ecore.

Figure 2.4: Ecore definition illustrating the use of classes, attributes, operations, references and super types, analogously to the figure 2.3



Source: http://download.eclipse.org/modeling/emf/emf/javadoc/2.9.0/org/eclipse/emf/ecore/package-summary.html. On the December 29th, 2015

[2]https://eclipse.org/modeling/emf

Triple Graph: With the use of a triple graph a relation between a source model S and a target model T are abstracted into a triple (G_s, G_c, G_t) – where G_s is the graph representation of source model elements, G_t is the graph representation of target model elements, and G_c represents the correspondence between the two set of model elements – together with two mappings $s_g : G_c \to G_s$ and $t_g : G_c \to G_t$, which bind the three graphs (Hermann et al., 2011).

In this case, an addition in the triple graph $G = (G_s, G_c, G_t)$, that leads to a new triple graph $H = (H_s, H_c, H_t)$ consists in a triple graph morphism $m : G \to H$, with $m = (m_s, m_c, m_t)$. According to the figure 2.5.

Figure 2.5: The morphism $m : G \to H$ is a triple graph $m = (m_s, m_c, m_t)$.

$$\begin{tikzcd}
G \arrow[d, "m"'] & = (G^S \arrow[d, "m^S"'] & G^C \arrow[l, "s_G"'] \arrow[r, "t_G"] \arrow[d, "m^C"'] & G^T) \arrow[d, "m^T"'] \\
H & = (H^S & H^C \arrow[l, "s_H"] \arrow[r, "t_H"'] & H^T)
\end{tikzcd}$$

Source: (Hermann et al., 2011)

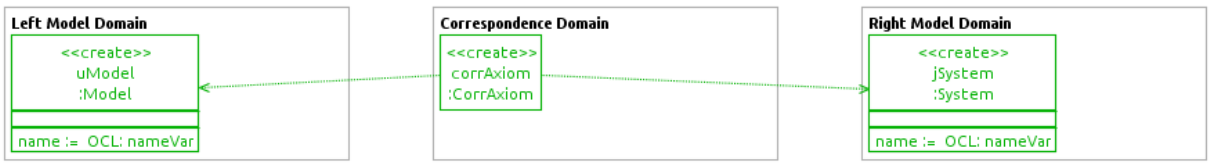
Triple Rule: A triple rule is a triple graph morphism $t_r = (s, c, t) : L \to R$, where L and R are called respectively the left-hand the right-hand sides (resp. LHS and RHS) (Ehrig et al., 2007).

Triple Axiom: A triple axiom is a triple rule $t_a = (s, c, t) : \emptyset \to R$. In order to apply such definitions in the practice, it is common to use attributed graphs and a easier to read diagram scheme depicted in the figure 2.6.

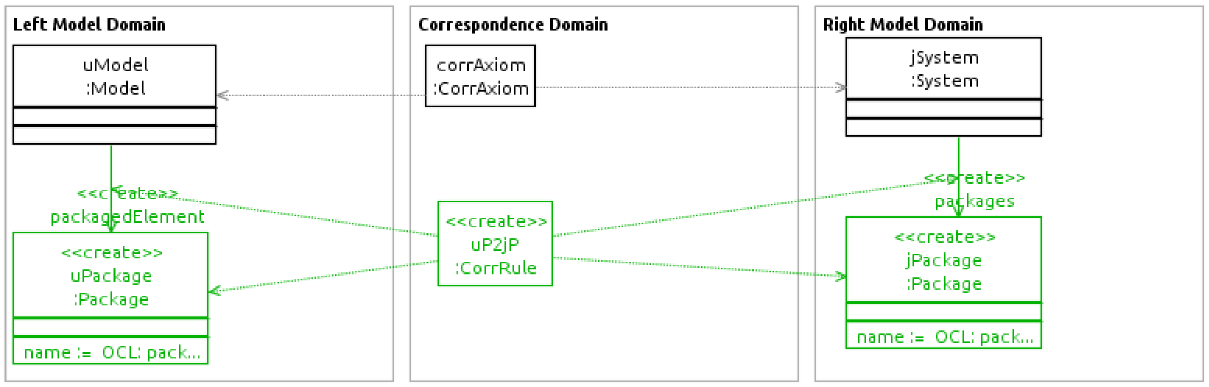
Triple Graph Grammar: A triple graph grammar $TGG = (t_a, T_{rules})$ consists of a triple axiom t_a and a set of triple rules T_{rules} (Giese et al., 2010a, p. 4). While one triple graph can be used as a description of a relation between two meta-models, one TGG describes the language of the these two related models and serves rather as description of consistency. Nevertheless, extra rules can be derived from a TGG, in order to create the operational semantic of a transformation procedure (Giese et al., 2010a). Figure 2.7 summarizes the so far defined terms.

Figure 2.6: In this kind of diagram for triple rules a triple graph is represented by three columns (left model domain, correspondence domain, and right model domain) each one representing respectively the source model elements, the correspondence between source and target and finally the target model elements. A triple rule in turn is represented by a triple graph in black (left-hand side) plus a triple graph in green (right-hand side) (see 2.6b). Because an axiom is a triple rule with empty left-hand side, only green graph occurs in an axiom (see 2.6a).

(a) Triple axiom example for the relation between UML and Java

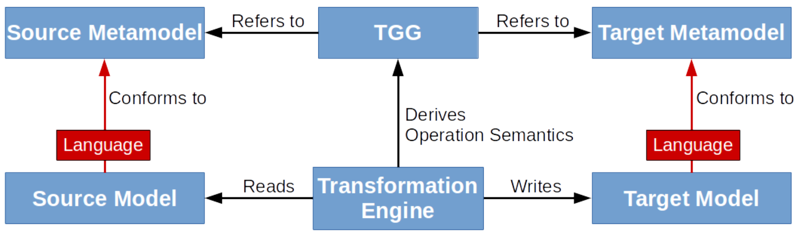


(b) Triple Rule example for the relation between UML package and Java package



Source: The author

Figure 2.7: Illustration of the definitions of model relation, transformation and synchronization as well as triple graph grammars (TGG). Relations between metamodels are coded by triple graphs; additions in the models are coded by triple rules, which are then organized in a TGG. A TGG can be used to derive operational semantic definitions. The concept of modeling language is pictured as red lines.



Source: Adapted from Czarnecki and Helsen (2006, p. 623)

3 STATE OF THE ART

Some endeavors have been made in order to code relations between some meta-models and mainly to develop theoretical results and synchronization methods. Heidenreich et al. present in (2009) and (2010) a Java meta-model using *Ecore*, what influenced considerably the development of our work, although it has not been used by us because of its size and unnecessary comprehensiveness for our needs. Greenyer et al. (2008) comes up with a transformation between *UML activity diagrams* and *CSP diagrams* using *TGG*. Foss et al. (2011) defined the translation between *UML* and *Simulink* using graph grammars. Blouin et al. (2014) reports about the synchronization between some specific meta-models of the automotive standards and influences our work, by using the same modeling language and transformation method as us, namely *EMF* (Steinberg et al., 2008) and *MoTE* (Giese et al., 2010a). Finally Giese et al. (2010b) introduce their approach to the synchronization of two automotive industry meta-models, lightening in the paper the *MoTE* tool and its algorithm for synchronization.

We judge that the *MoTE* transformation tool is the most adequate option for our needs, once literature about it is widely available (see also (Giese and Hildebrandt, 2009) and (Hildebrandt et al., 2012)). Nevertheless there are other attempts to build a model synchronization tool, like the *ATL Eclipse Plug-in* (Jouault et al., 2008), which uses the *Atlas Transformation Language* to code the relations between models; the Medini QVT [1], which claims to implement the *Query/View/Transformation Language* to code the relations; and the FUJABA (Nickel et al., 2000), in which relations are coded using *Triple Graph Grammars*. Hildebrandt et al. (2013) also publicized a survey on synchronization tools based on TGG. Other publications aim to solve specific problems, like the ones in Hermann et al. (2011), Xiong et al. (2007), Giese and Wagner (2006), Ivkovic and Kontogiannis (2004), or Song et al. (2011), where advanced algorithms for bidirectional synchronization have been proposed.

A research road-map for model synchronization found in France and Rumpe (2007) gives an overview on the realm, and together with Mattsson et al. (2009) show an interesting point of view about the challenges. Seidewitz (2003) writes an interesting reflection about what models mean and how to interpret them and in Mens and Van Gorp (2006) a taxonomy for model transformation is proposed, what helps to carry more precise analysis. In Czarnecki and Helsen (2006) a survey was driven and a framework for classification of model transformation approaches was presented. In Diskin et al. (2014) and (2016) a taxonomy for a network of models is presented and in Diskin (2011) a theoretical algebraic basis is proposed.

[1]http://projects.ikv.de/qvt

Additionally, one can judge by the date of publication of these works, that the topic of model synchronization is extremely active and is actually the edge of current academic research, what motivates even more the development of this thesis.

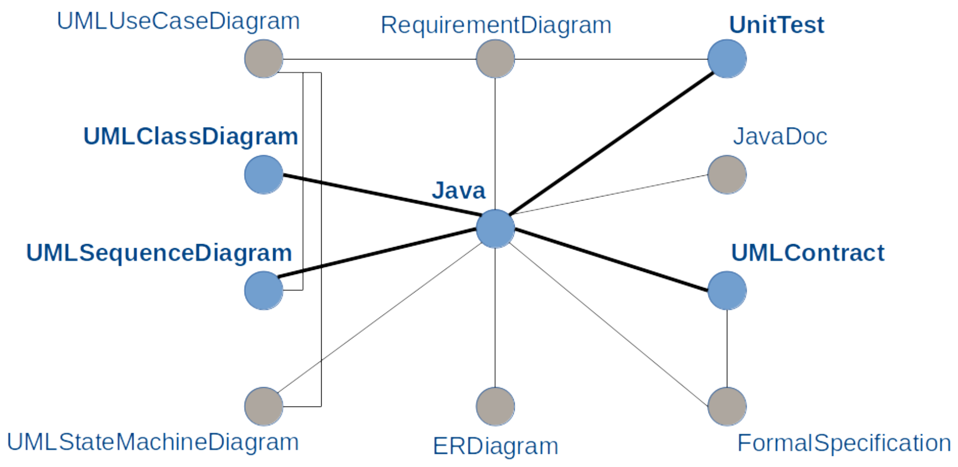
4 META-MODEL RELATIONS IN THE JAVA TECHNOLOGICAL SPACE

With the terms and the theoretical basis clarified, the report of the main development phase of the thesis is shown below. The idea here is first to present the selected network of meta-models, by describing what each model represents and how they relate to each other. Then the developed meta-model definitions and the relations between them are exposed, as well as the justification of the choices made during the work.

4.1 The Network of Meta-models

The choice of which models are used in a certain Java program may vary considerably depending on the context of the development and the software requirements, which themselves can range from high dependability (e.g. airplane software) to continuous evolution (e.g. applications for cellphones), for example. Nevertheless, we selected a few typical models and the relations between them and created a network of meta-models. The figure 4.1 depicts this network. The bold printed vertices and edges represent respectively the meta-models and the relations, that are treated more deeply in this thesis, namely *Java*, *UMLClassDiagram*, *UMLSequenceDiagram*, *UMLContract* and *UnitTest*, whereas the other vertices are by virtue of the scope of this thesis some rather more briefly discussed meta-models, namely *UMLUseCaseDiagram*, *RequirementDiagram*, *JavaDoc*, *UMLStateMachine*, *ERDiagram*, *FormalSpecification*. This does not mean however, that they are not worth further studying.

Figure 4.1: A network of meta-models in the Java technological spaces



Source: The author

The central element of the network is the *Java* meta-model. A Java model (or, Java code) contains both structural and behavioral information about the system, which is represented among other elements through *classes*, *fields*, *methods* and *statements*.

Once a meta-model might be relatively big and comprise a huge number of elements, it is interesting to split it into smaller pieces for a specific set of relations. Take for example the UML, that includes a large number of different concerns (e.g. classifiers, state machines, activities, interaction, etc) and may be split into sub meta-models in order to ease the writing of the relations. For this reason we separate the UML into *UMLClassDiagram*, *UMLSequenceDiagram*, *UMLContract*, *UMLUseCaseDiagram* and *UMLStateMachineDiagram*. The first is constructed around the concepts of *class*, *property*, *operation*, *interface* and *package*. This meta-model is usually used to describe the structure of a object oriented Java program through a class diagram, representing the definition of its classes, field and methods, but leaving out behavioral aspects. The relations between *UMLClassDiagram* and *Java* is then given by an almost direct translation between their elements. A *class* in the former is transformed into a *class* in the latter, a *property* in a *field*, an *operation* in a *method* and so on.

To represent some behavioral aspects, *UMLSequenceDiagram* is used instead. The elements of this meta-model are usually reproduced with sequence diagrams, where *lifelines* and *messages* provide information about the sequence of event occurrences. In a Java program it may correspond to the sequence of calls inside a specific method. This means, the semantical information of each sequence diagram could be brought to the correspondent method in the Java model.

UMLContract is based on the ideas of design-by-contract, whose main goal is to improve reliability of object oriented software (Meyer, 1992), and where operations have *pre* and *postconditions* as well as *invariants*. Its relation with *Java* is basically, that each constraint of the contracts can (1) be tested as assertions and (2) expressed in terms of annotations in the Java source-code. Moreover, one can have check methods in Java, which serve to verify the constraints of the class and therefore are supposed to be updated as soon as the contracts undergo changes. Related to contracts are also *Formal Specifications*, these taking several different forms, among them the Z Notation (see (Spivey and Abrial, 1992)), which itself also refers to *pre* and *postconditions*.

The *UnitTest* endeavors to enhance the software quality by means of tests. It tests small units of code, by basically verifying the pre and postconditions as well as invariants of each method. Because unit tests for Java programs are usually written in Java, we use the same meta-model for the vertices *Java* and *UnitTests* of our network. Anyways, the relation between both is based on creating test cases in the latter according to the contract annotations (e.g pre and postconditions, invariants, examples, etc) present in the methods of the former.

Moreover, *UML Use Cases Diagrams* are used to relate *Actors* (basically users of the program) and *Use Cases* (specifications of behavior), and therefore have a relationship with *Requirement Diagrams*, a very common tool on information system analysis for description of features of a system, and with *Sequence Diagrams* (discussed before) and *State Machine Diagrams*, two artifacts aiming to describe the behaviors specified by the use cases (OMG, 2007, p. 637). *State Machines* are well known means for modeling functionalities of Java programs, rely on *states* and *transitions* and relates to *Use Cases Diagrams* in the sense that it describes the behavior of a *Use Case*. For this reason, one may argue, that the behavior expressed by *State Machines* may also be synchronized with the implementaion of methods in the *Java source-code*.

The linking from *Requirement Diagrams* to *Unit Tests* has been proposed by Noack (2013) and by Post et al. (2009) and a kind of linking to source-code has been proposed by Antoniol et al. (2002). *JavaDoc* models play an important role as well, as they serve as program documentation for the developers. A transformation from the Java source-code to a *JavaDoc* model is already achievable through the *JavaDoc Tool* [1], which transforms comments from the source-code into HTML documentation.

Finally, there is the *ERDiagram* (Entity-relationship model, see (Chen, 1976)), used to construct data models and is applied specially to describe database schemes through basically *Entities*, that often correspond to *Java classes*, and *Relationships* between them, what may be seen often as *Java attributes*.

4.2 Meta-model Definitions

As stated in the last section, the meta-models highlighted in the figure 4.1 have its meta-models defined here, and the relation between them formalized. The meta-models *UMLClassDiagram*, *UMLSequenceDiagram*, *UMLContract*, and *Java* − which serves also for the vertice *UnitTest* − are explained below with help of a running example.

The modeling language used to write these meta-models is the *EMF Ecore* and the tool used is the special version for model development of the *Eclipse Mars 4.5.1 IDE*[2], which eases the creations of models and their diagrams as well as the generation of plug-ins necessary for

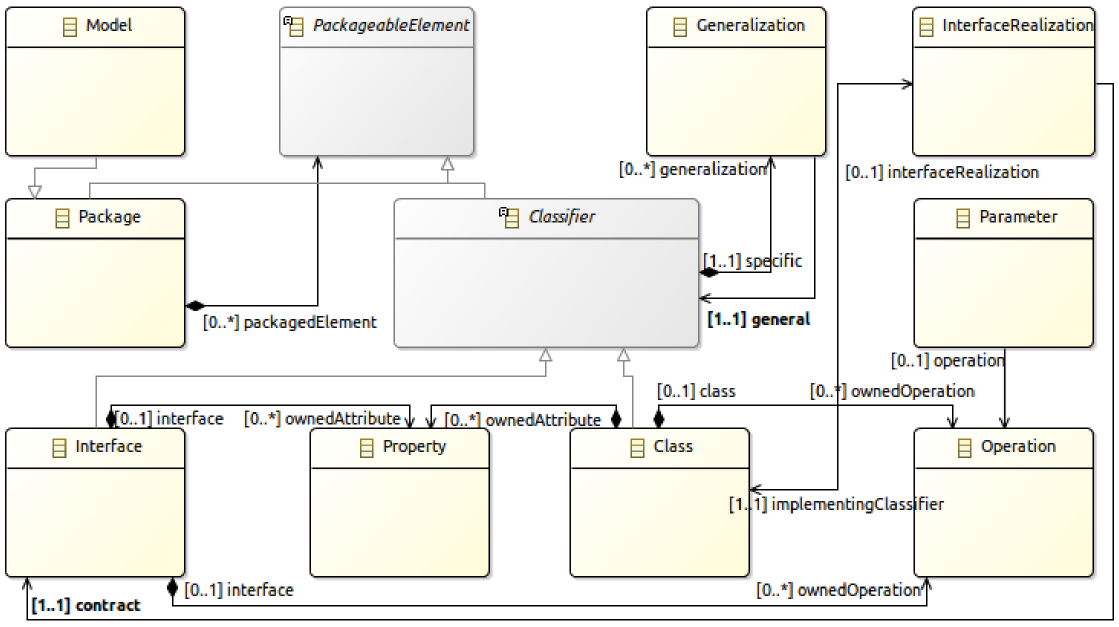
[1]http://www.oracle.com/technetwork/articles/java/index-jsp-135444.html
[2]https://projects.eclipse.org/releases/mars

running the transformations. For this reason *Eclipse IDE* seems to be more suitable than the alternatives *Netbeans IDE*[3] or FUJABA (Nickel et al., 2000), whose support or popularity in the community are not so valuable. The *EMF Ecore* language has been chosen not only for its extensible documentation and popularity in the community, but also for its ease to use in the *Eclipse IDE*. The meta-models are listed in the sections below.

4.2.1 UML Class Diagram

The meta-model utilized for *UMLClassDiagram* (also for *UMLSequenceDiagram*, *UMLContract*) represents the version 2.0 of the UML standard and is provided by the *EMF plug-in*[4] for Eclipse, which clearly integrates easily with the IDE and seems to be suitable for our needs. Alternatively we could use the meta-model provided by the OMG, but then unnecessary work of adaption could late our progress. The figure 4.2 addresses a simplified view of the *UMLClassDiagram*. Elements in blue are *EMF Ecore* abstract elements, whilst elements in yellow are concrete. Some features like operations and some relations between elements were omitted for a better visualization.

Figure 4.2: Simplification of the *UMLClassDiagram* meta-model



Source: Image created by the author using the *Eclipse IDE*. Meta-model from *EMF plug-in*

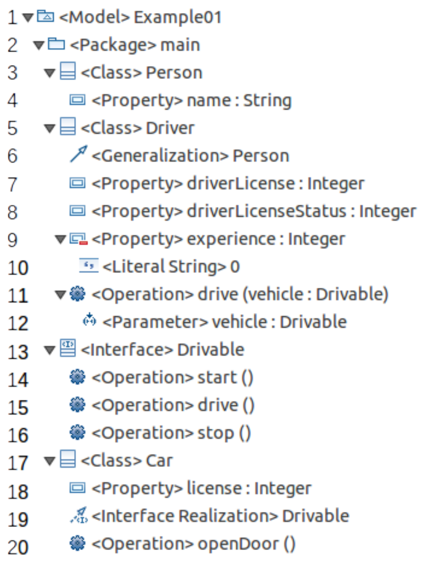
[3]https://netbeans.org
[4]https://eclipse.org/modeling/emf

The *Model* (top, link on figure 4.2) represents the whole model and is the root element of a class diagram, in the sense every other element is contained by it. *Model* inherits *Package*, and thus may contain *Packageable Elements*. Because *Packages* inherit *Packageable Elements*, they may be contained by the *Model*, what is the most common case. An example of a *UML-ClassDiagram* model (not the meta-model) is depicted in the figure 4.3 in two forms: In abstract and in concrete syntax. There a *Model* named *Example01* contains a *Package* named *main*.

A *Package* may contain according to these scheme other *Packages* as well as a *Classifier* (center on the figure 4.2), because this one inherits *Packageable Element*.The two classifiers handled in the figure are *Class* and *Interface*. The model if the running example contains the *Classes Person*, *Driver*, and *Car*; and the *Interface Drivable*.

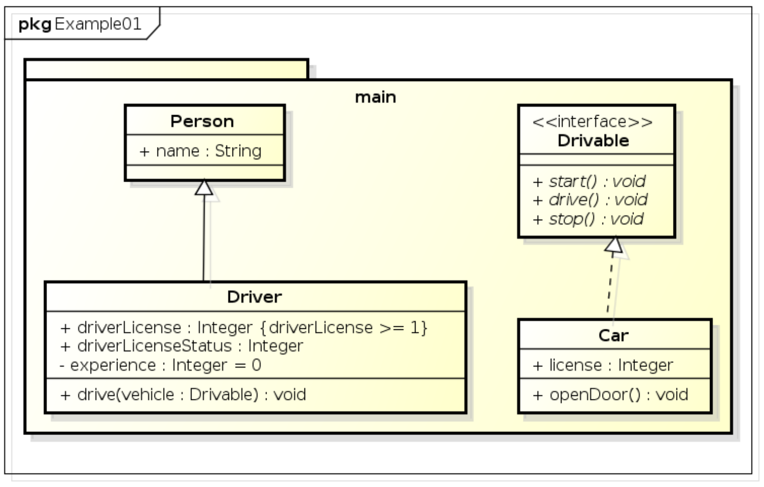
Figure 4.3: An example of a model *UMLClassDiagram* visualized in two different ways, containing one *Model* (*Example01*), one *Package* (*main*), three *Classes* (*Person*, *Drive* and *Car*), and one *Interface*, namely *Drivable*.

(a) Abstract Syntax



```
<Model> Example01
  <Package> main
    <Class> Person
      <Property> name : String
    <Class> Driver
      <Generalization> Person
      <Property> driverLicense : Integer
      <Property> driverLicenseStatus : Integer
      <Property> experience : Integer
        <Literal String> 0
      <Operation> drive (vehicle : Drivable)
        <Parameter> vehicle : Drivable
    <Interface> Drivable
      <Operation> start ()
      <Operation> drive ()
      <Operation> stop ()
    <Class> Car
      <Property> license : Integer
      <Interface Realization> Drivable
      <Operation> openDoor ()
```

(b) Concrete Syntax



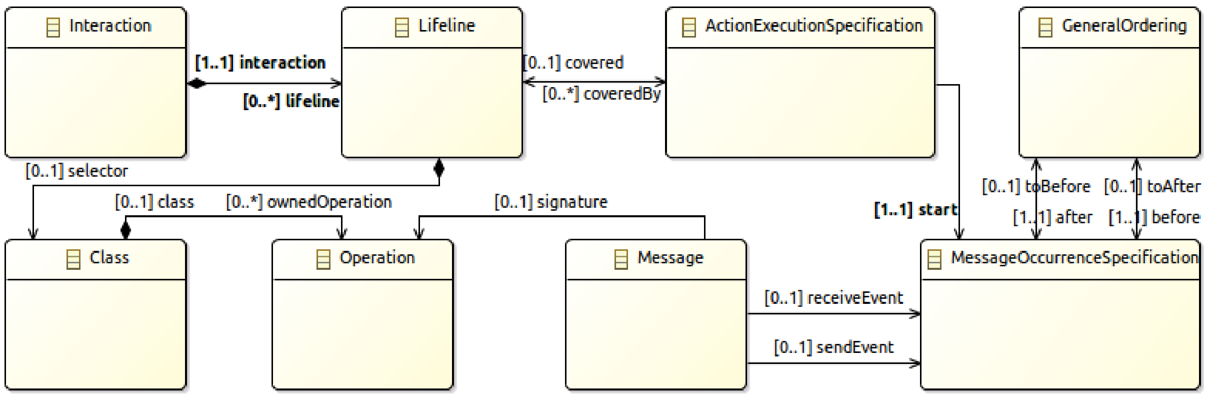
Source: Image created by the author using the *Astah Software*[5].

According to the meta-model, a *Classifier* may have a *Generalization* (i.e. inheritance), illustrated with a straight arrow from *Driver* to *Person* in the example on the figure 4.3, or an *InterfaceRealization* illustrated with a dashed arrow from *Car* to *Drivable*. Moreover, a *Class* is possible to have not only *Properties* (through the aggregation *ownedAttribute*), but also *Operations* (through the *ownedOperation* attribute). This characteristic is analogous to *Interfaces*. In the running example the *Person* has the *Property name*, as well as the *Driver* has the *Property driverLicense* and the *Operation drive(Drivable):void*.

4.2.2 UML Sequence Diagram

The *UMLSequenceDiagram* is essentially based on the elements *Interaction*, *Lifeline*, and *Messages*. A simplified view of the meta-model is given on the figure 4.4. According to OMG (2007, p. 563): *"Interactions [...] are used to get a better grip of an interaction situation"* (OMG, 2007, p. 563), by being so the important aspect of *Sequence Diagram* are the exchange of messages between object (i.e. interaction). Sequence diagrams are quite flexible in regard to its semantics, so developers interpret the exchange of messages in different ways. Nevertheless, they are interpreted in this thesis in a rather simpler manner. An *Interaction* models one scenario, in which an *Operation* of a *Class* is executed, and contains one or more *Lifelines*, that express the life of an instance of a class. A concrete illustration of these elements is to find in the example in the figure 4.5b.

Figure 4.4: Simplification of the *UMLSequenceDiagram* meta-model

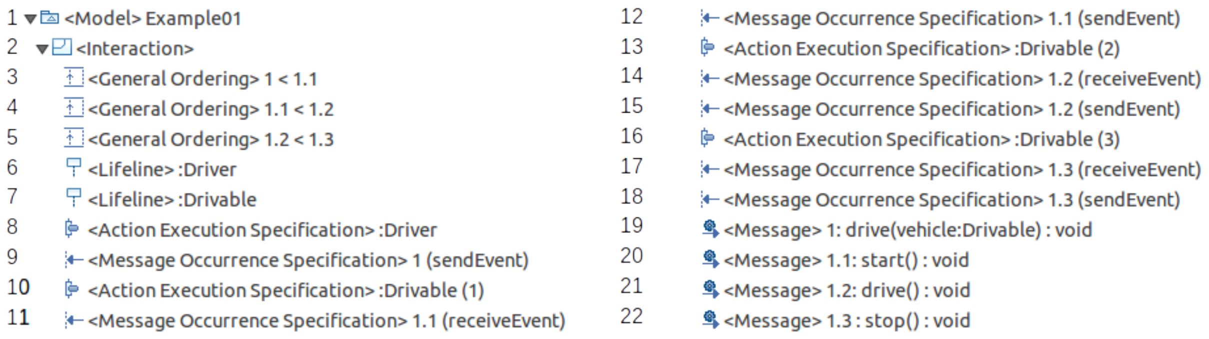


Source: Image created by the author using the *Eclipse IDE*. Meta-model from *EMF plug-in*

One *Lifeline* is connected to a *Class* through the attribute *selector* and is *covered by* one *Action Execution Specification*, which in a sequence diagram is depicted by a rectangle over the *Lifeline* and symbolize the time, during which the respective class is executed. An *Action Execution Specification* has then a *MessageOccurenceSpecification* as *start* point, which itself is related to a *Message*. Finally, each *Message* is linked to two *MessageOccurenceSpecifications* – one *receiveEvent* that lies on the beginning and one *sendEvent* that lies on the end of the *Message*); and has a *signature* of an *Operation*. The comprehension of this interpretation over *Sequence Diagrams* requires the reader to grasp the figure 4.5

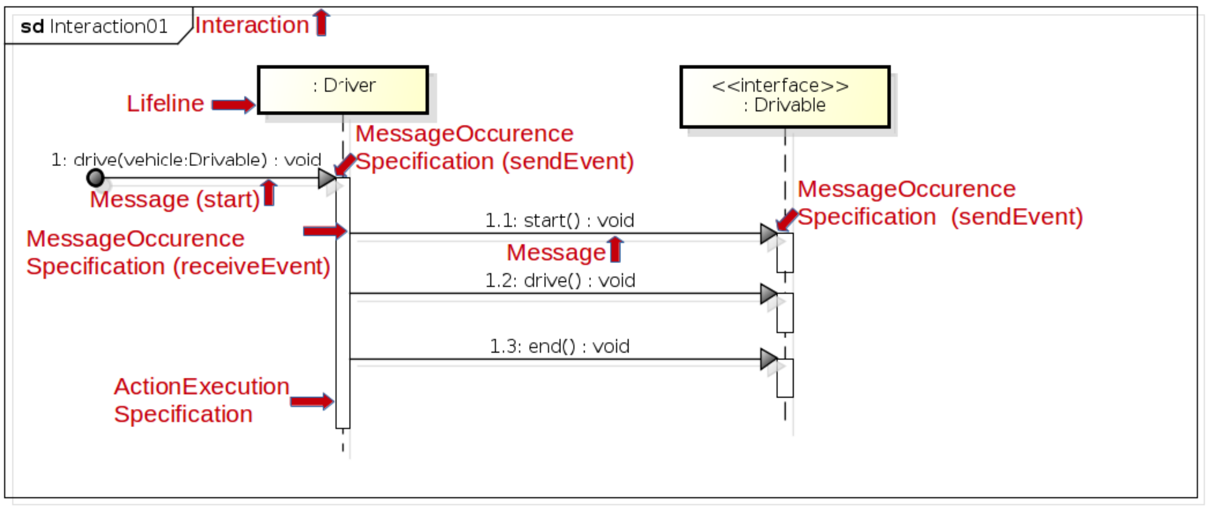
Figure 4.5: An example of a model *UMLSequenceDiagram* containing one *Interaction* (*Interaction01*), two *Lifelines* (*:Driver* and *:Drivable*), and four *Messages*.

(a) Abstract Syntax



```
<Model> Example01
  <Interaction>
    <General Ordering> 1 < 1.1
    <General Ordering> 1.1 < 1.2
    <General Ordering> 1.2 < 1.3
    <Lifeline> :Driver
    <Lifeline> :Drivable
    <Action Execution Specification> :Driver
    <Message Occurrence Specification> 1 (sendEvent)
    <Action Execution Specification> :Drivable (1)
    <Message Occurrence Specification> 1.1 (receiveEvent)
    <Message Occurrence Specification> 1.1 (sendEvent)
    <Action Execution Specification> :Drivable (2)
    <Message Occurrence Specification> 1.2 (receiveEvent)
    <Message Occurrence Specification> 1.2 (sendEvent)
    <Action Execution Specification> :Drivable (3)
    <Message Occurrence Specification> 1.3 (receiveEvent)
    <Message Occurrence Specification> 1.3 (sendEvent)
    <Message> 1: drive(vehicle:Drivable) : void
    <Message> 1.1: start() : void
    <Message> 1.2: drive() : void
    <Message> 1.3 : stop() : void
```

(b) Concrete Syntax


sd Interaction01
Interaction
Lifeline
: Driver
<<interface>>
: Drivable
MessageOccurence Specification (sendEvent)
1: drive(vehicle:Drivable) : void
Message (start)
MessageOccurence Specification (sendEvent)
MessageOccurence Specification (receiveEvent)
1.1: start() : void
Message
1.2: drive() : void
1.3: end() : void
ActionExecution Specification

Source: Image created by the author using the *Astah Software*[6].

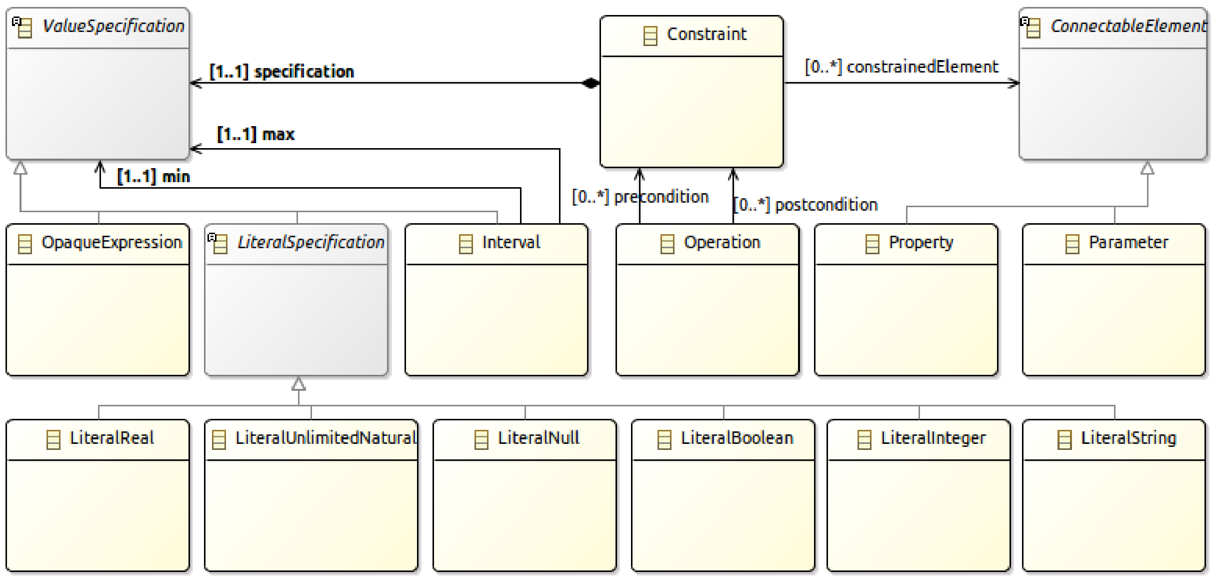
To model the order in which the *Messages* occur, a *GeneralOrdering* establishes an order between two *MessageOccurenceSpecifications*, by signalizing which of them occur *before* or *after* the other. So in the running example there is a *GeneralOdering* instance holding the *MessageOccurenceSpecifications* related to the *Message 1* as happening *before* the one related to the *Message 1.1* (that is held as *after*).

As stated before, developers tend to interpret and utilize sequence diagrams in different fashions. For this thesis a set of assumptions is made in regard to that, among them lifelines represent only classes (excluding thus representation of actors); and only synchronous message are handled.

4.2.3 UML Contract

The *UMLContract* is a slice of the *UML* meta-model, that aims basically to provide constraints to *Operations* and *Properties* of *Classes*. *Operations* may have *pre*, *postconditions*, as well as *Invariants*, which are modeled through the EMF class *Constraint* (on the top of the figure 4.6).

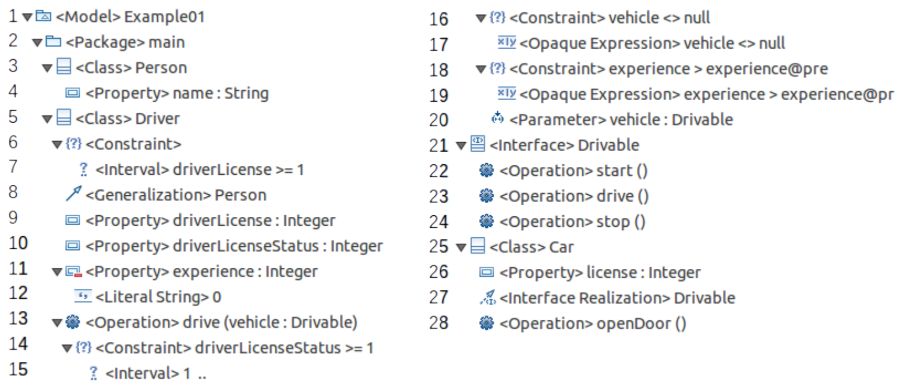
Figure 4.6: Simplification of the *UMLContract* meta-model



Source: Image created by the author using the *Eclipse IDE*. Meta-model from *EMF plug-in*

A *Constraint* may have then *constrainedElements*, which in the scope of this thesis are whether *Properties* or *Parameters* (of operations). In addition, it has also a *ValueSpecification*, defining the constraint itself, that in this thesis may be a *OpaqueExpression* or a *Interval*. The former is rather a free definition of the constraint composed by a String *body*, and the *language*, in which it is written (see lines 16 to 19 in figure 4.7). The latter defines a interval of values, in which the *constrained element* has to lie. Therefore it has one *value specification* for the minimal and one for the maximal value. Here only *intervals* with one *literal integer value specification* for the minimal value will be handled, see an example in the lines 6 to 7 in figure 4.7.

Figure 4.7: The expanded version of the model from the picture 4.3 including elements from *UMLContract*, namely lines 6 to 7 and 14 to 15 (*Constraint* with *Interval*), and lines 16 to 19 (*Constraint* with *OpaqueExpression*). Only abstract syntax is used for this example.



```
1  <Model> Example01
2    <Package> main
3      <Class> Person
4        <Property> name : String
5      <Class> Driver
6        <Constraint>
7          <Interval> driverLicense >= 1
8        <Generalization> Person
9        <Property> driverLicense : Integer
10       <Property> driverLicenseStatus : Integer
11       <Property> experience : Integer
12         <Literal String> 0
13       <Operation> drive (vehicle : Drivable)
14         <Constraint> driverLicenseStatus >= 1
15           <Interval> 1 ..
16         <Constraint> vehicle <> null
17           <Opaque Expression> vehicle <> null
18         <Constraint> experience > experience@pre
19           <Opaque Expression> experience > experience@pr
20         <Parameter> vehicle : Drivable
21   <Interface> Drivable
22     <Operation> start ()
23     <Operation> drive ()
24     <Operation> stop ()
25   <Class> Car
26     <Property> license : Integer
27     <Interface Realization> Drivable
28     <Operation> openDoor ()
```

Source: Image created by the author using the *Eclipse IDE*.

4.2.4 Java

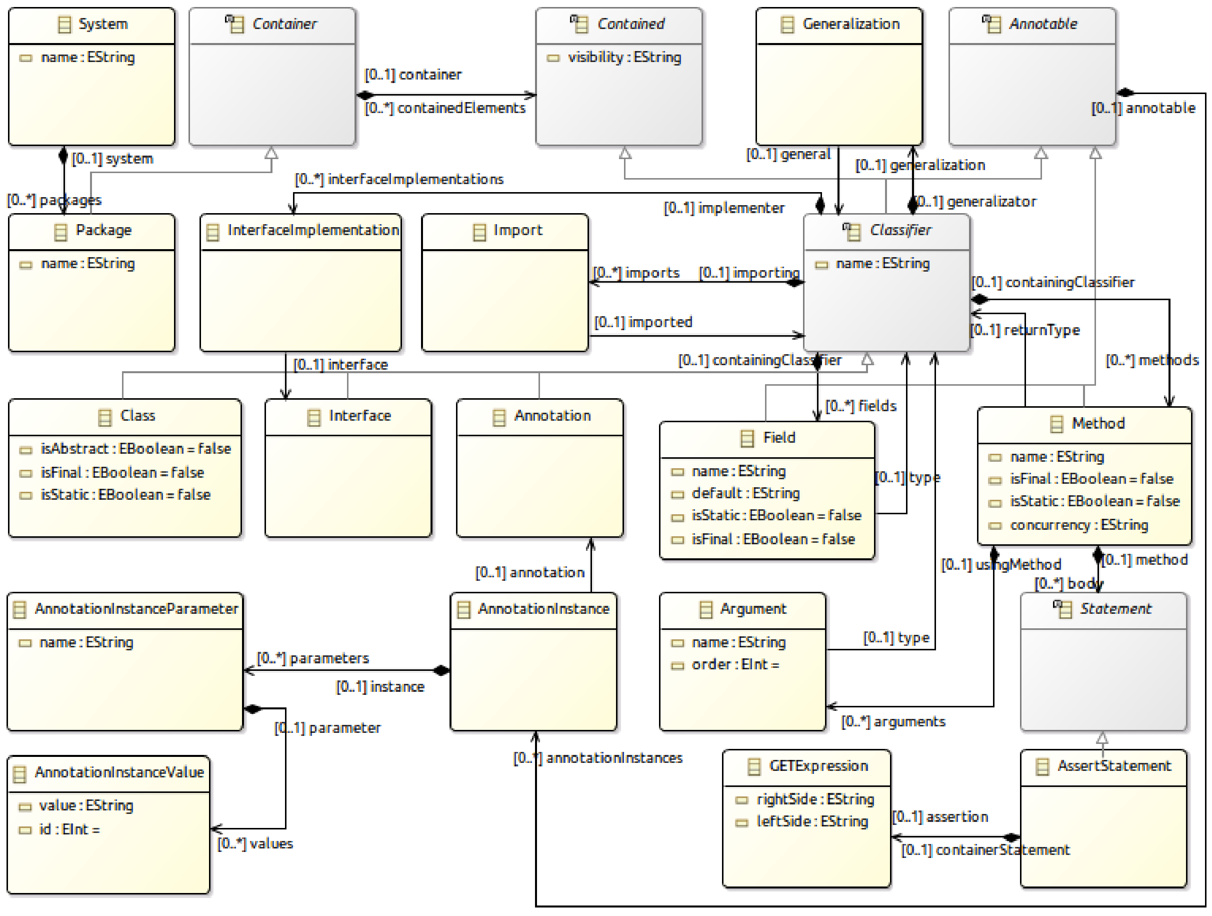
A *Java* model is usually the main artifact of a network of models, in that it comprises much information (both structural and behavioral) about the system under study. It can be visualized either as the digram in the figure 4.9 or as plain-text (figure 4.10). The first option is used here, since it seems to be more suitable when handling the model for synchronization than a plain-text source-code format. Nevertheless, there are techniques[7] to transform one format into another.

A new meta-model for *Java* was designed in regard to the necessities of this work and includes not only structural elements (e.g. *Packages*, *Classes*, *Fields*, etc.), but also some behavioral aspects (e.g. *Statements*), the latter are very shallowly modeled here, but could be further developed in future works. Other possibilities for the meta-model included the meta-model[8] provided by the *Eclipse IDE*, whose simplicity hindered its use; or yet the one found in Heidenreich et al. (2010), but this one happens to be so extensive that could bring unnecessary complexity to this thesis. The figure 4.8 reports the complete *Java* meta-model created. Elements in blue are *EMF Ecore* abstract elements, whilst elements in yellow are concrete.

The root element is the *System* (on the left top of figure 4.8) and represents the whole Java program. It contains *Packages*, that, because of the *Container* inheritance, contains *Classifiers* (through the *Contained* inheritance). The example on the picture 4.9 illustrates concrete types of *Classifiers* in the lines 3, 5 and 48 (*Class*) and 44 (*Interface*).

[7]https://eclipse.org/modeling/m2t
[8]http://www.eclipse.org/modeling/emf/downloads

Figure 4.8: The Java meta-model created. It goes beyond the common classes and attributes scenario, by comprising also annotations and statements.

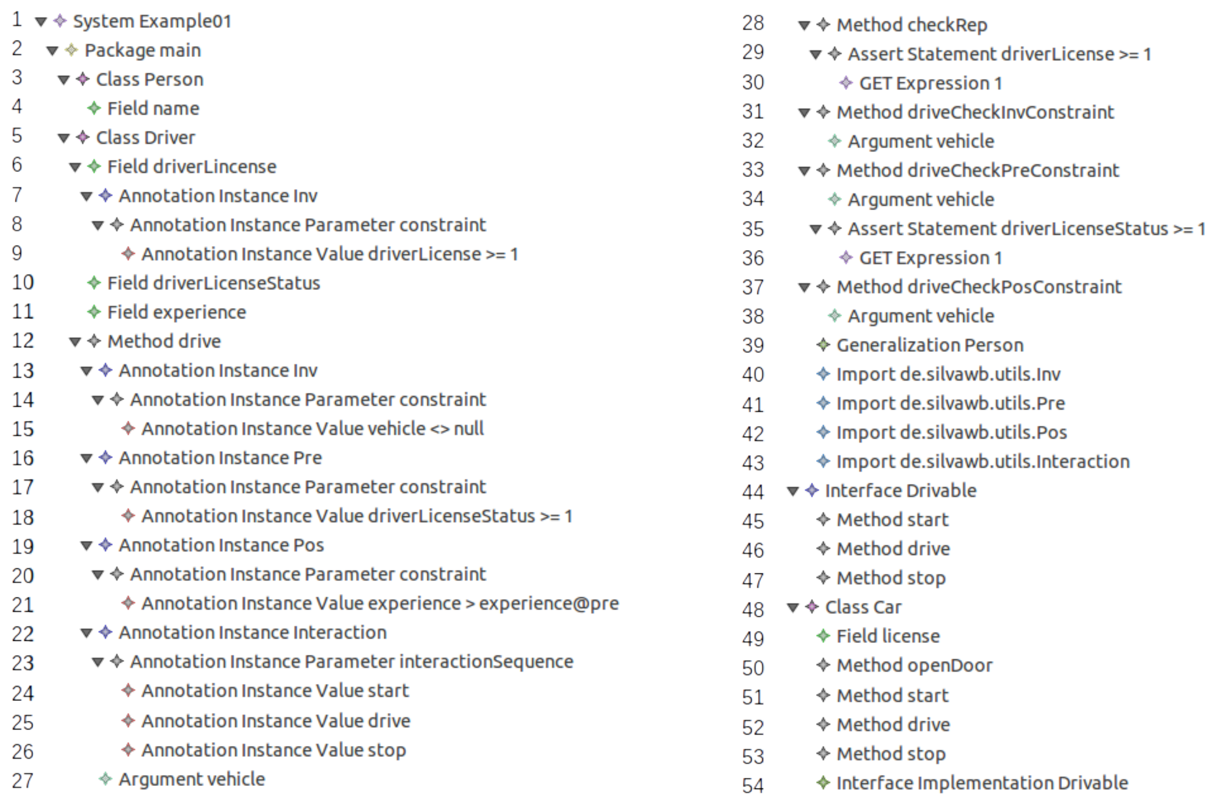


Source: Image created by the author using the *Eclipse IDE*.

One *Classifier* may contain *Fields* (lines 4, 6, 10, 11, 49), *Methods* (lines 12, 28, 31, 33, 35, 37), *Imports* (lines 40 to 43), in addition to *interface implementations* (line 54), that refer to one *Interface*; and to one *Generalization* (line 39), also known as inheritance or extension, which refers to another *Classifier* as its *general*.

A *Method* may have zero or more *Arguments*, also known as parameters; likewise *Statements* (to find on the left bottom of the picture 4.8). In this imperative perspective of the *Java* meta-model, each method has thus an ordered list of *Statements*, representing usually commands that carve the behavior of the program. In a full representation of the *Java* meta-model, they could be of several kinds (e.g. arithmetic expression, logic expression, method call, control structure like *if*), but for the scope of this bachelor thesis only *assert statements* are modeled. An *AssertStatement* basically tests a logical expression. If it does not hold, than an exception is thrown. The only logical expression supported here is the *greater-or-equal expression* (*GETExpression*, left bottom on the picture 4.8), but again the construction of a more complete meta-model shall be possible in a wider scope. An example of a *Method* with *AssertStatement* is to find on the lines 28 to 30 and 33 to 36 of the picture 4.9.

Figure 4.9: The equivalent *Java* version of the *UML* models from the pictures 4.3, 4.7 and 4.5 in abstract syntax.



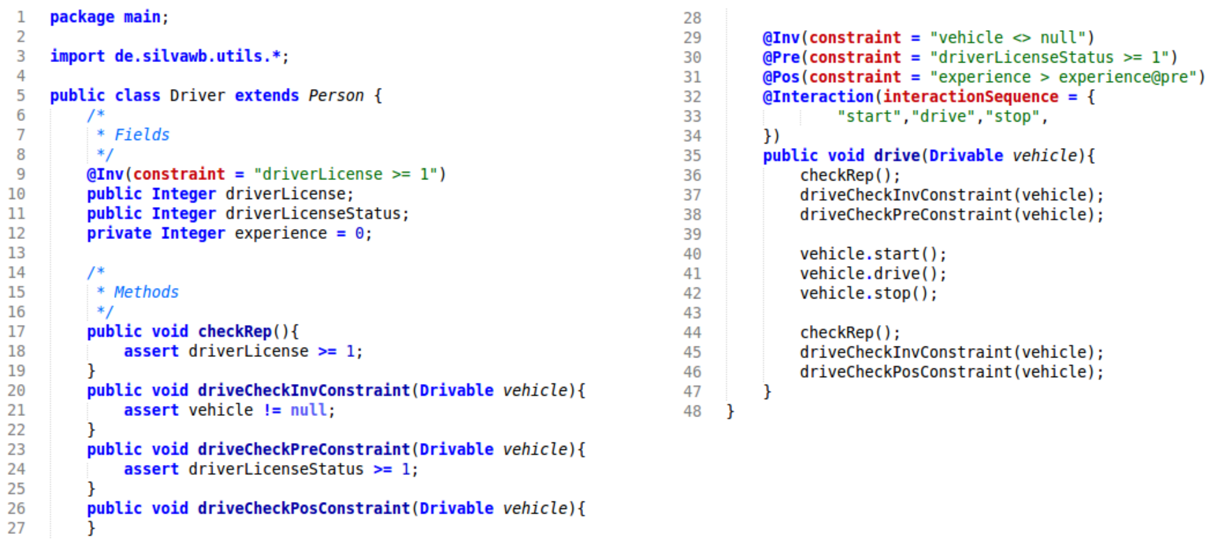
```
1  ▼ ✧ System Example01
2    ▼ ✧ Package main
3      ▼ ✧ Class Person
4          ✧ Field name
5      ▼ ✧ Class Driver
6        ▼ ✧ Field driverLincense
7          ▼ ✧ Annotation Instance Inv
8            ▼ ✧ Annotation Instance Parameter constraint
9                ✧ Annotation Instance Value driverLicense >= 1
10         ✧ Field driverLicenseStatus
11         ✧ Field experience
12       ▼ ✧ Method drive
13         ▼ ✧ Annotation Instance Inv
14           ▼ ✧ Annotation Instance Parameter constraint
15               ✧ Annotation Instance Value vehicle <> null
16         ▼ ✧ Annotation Instance Pre
17           ▼ ✧ Annotation Instance Parameter constraint
18               ✧ Annotation Instance Value driverLicenseStatus >= 1
19         ▼ ✧ Annotation Instance Pos
20           ▼ ✧ Annotation Instance Parameter constraint
21               ✧ Annotation Instance Value experience > experience@pre
22         ▼ ✧ Annotation Instance Interaction
23           ▼ ✧ Annotation Instance Parameter interactionSequence
24               ✧ Annotation Instance Value start
25               ✧ Annotation Instance Value drive
26               ✧ Annotation Instance Value stop
27           ✧ Argument vehicle
28       ▼ ✧ Method checkRep
29         ▼ ✧ Assert Statement driverLicense >= 1
30             ✧ GET Expression 1
31       ▼ ✧ Method driveCheckInvConstraint
32           ✧ Argument vehicle
33       ▼ ✧ Method driveCheckPreConstraint
34           ✧ Argument vehicle
35         ▼ ✧ Assert Statement driverLicenseStatus >= 1
36             ✧ GET Expression 1
37       ▼ ✧ Method driveCheckPosConstraint
38           ✧ Argument vehicle
39         ✧ Generalization Person
40         ✧ Import de.silvawb.utils.Inv
41         ✧ Import de.silvawb.utils.Pre
42         ✧ Import de.silvawb.utils.Pos
43         ✧ Import de.silvawb.utils.Interaction
44     ▼ ✧ Interface Drivable
45         ✧ Method start
46         ✧ Method drive
47         ✧ Method stop
48     ▼ ✧ Class Car
49         ✧ Field license
50         ✧ Method openDoor
51         ✧ Method start
52         ✧ Method drive
53         ✧ Method stop
54         ✧ Interface Implementation Drivable
```

Source: Image created by the author using the *Eclipse IDE*.

To finish the description of the meta-model, there is the *Annotation*, which is also a kind of *Classifier*. An *AnnotationInstance* is than contained by an *Annotable* element (i.e *Classifier*, *Field*, or *Method*) and may contain *AnnotationInstanceParameters*, which themselves may contain *AnnotationInstanceValues*. The picture 4.9 contains examples of annotations on the lines 7 to 9 and 13 to 26.

The figure 4.10 shows the plain-text view over the *class Driver* of the previous example model only with some small differences. In fact this plain-text view has two elements that are not modeled in our *Java* meta-model, namely expansion of logical expressions (line 21) and method calls (lines 36 to 46), but anyway it exposes the ideas behind the use of *annotations* (lines 9 and 29 to 34) and their relation with *UML sequence diagrams* and *UML contracts*. More details are shown in the next chapter.

Figure 4.10: A more comprehensive Java model based on the figure 4.9, but depicted in plain-text form, expressing the Java concrete syntax.



```java
package main;

import de.silvawb.utils.*;

public class Driver extends Person {
    /*
     * Fields
     */
    @Inv(constraint = "driverLicense >= 1")
    public Integer driverLicense;
    public Integer driverLicenseStatus;
    private Integer experience = 0;

    /*
     * Methods
     */
    public void checkRep(){
        assert driverLicense >= 1;
    }
    public void driveCheckInvConstraint(Drivable vehicle){
        assert vehicle != null;
    }
    public void driveCheckPreConstraint(Drivable vehicle){
        assert driverLicenseStatus >= 1;
    }
    public void driveCheckPosConstraint(Drivable vehicle){
    }

    @Inv(constraint = "vehicle <> null")
    @Pre(constraint = "driverLicenseStatus >= 1")
    @Pos(constraint = "experience > experience@pre")
    @Interaction(interactionSequence = {
            "start","drive","stop",
    })
    public void drive(Drivable vehicle){
        checkRep();
        driveCheckInvConstraint(vehicle);
        driveCheckPreConstraint(vehicle);

        vehicle.start();
        vehicle.drive();
        vehicle.stop();

        checkRep();
        driveCheckInvConstraint(vehicle);
        driveCheckPosConstraint(vehicle);
    }
}
```

Source: The Author.

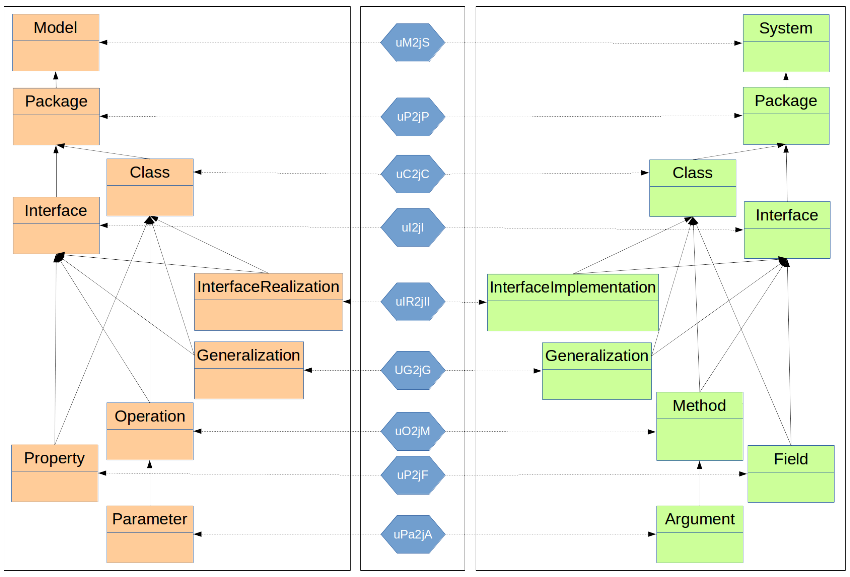
4.3 Relations

With all the meta-models defined, the definition of the relations between their elements can be done. In order to accomplish that, TGG is used to code the relations, because it has been extensively used in current academic research; and is supported by several tools for synchronization. Other options included the *ATL* (Jouault et al., 2008), which does not seem to be enough ripe for our use; or the *Henshin* (Arendt et al., 2010), that is not widely supported for the best synchronization tools (Hildebrandt et al., 2013). A theoretical basis of TGG has been given in the chapter 2. Here the most representative relations are presented as well as some explanations over them.

4.3.1 Relations between *Uml Class Diagram* and *Java*

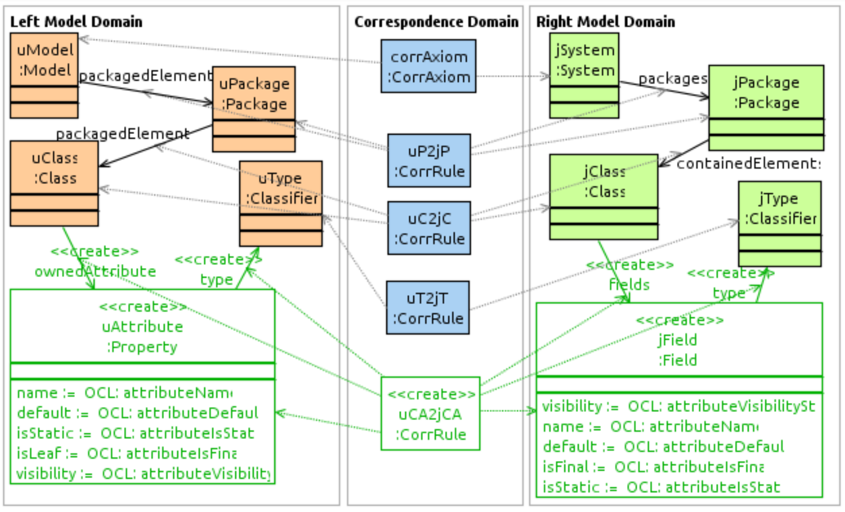
Fig. 4.11 shows the triple type graph for the relations between the *UmlClassDiagram* (left) and the *Java* (right) domains, with the correspondence domain being in the middle. In this graph, elements from the left domain are connected to the elements on the right domain, with whom they have a relation. So the element *Model* in *UmlClassDiagram* has a relation to the element *System* in *Java*; analogously, a *Property* in *UmlClassDiagram* is related to an *Field* in *Java*, namely whenever the former is created, the correspondent latter has to be created according to the former's characteristics (i.e. name, type, the class it belongs, etc.).

Figure 4.11: The triple type graph for *Uml Class Diagram* and *Java*



Source: The author

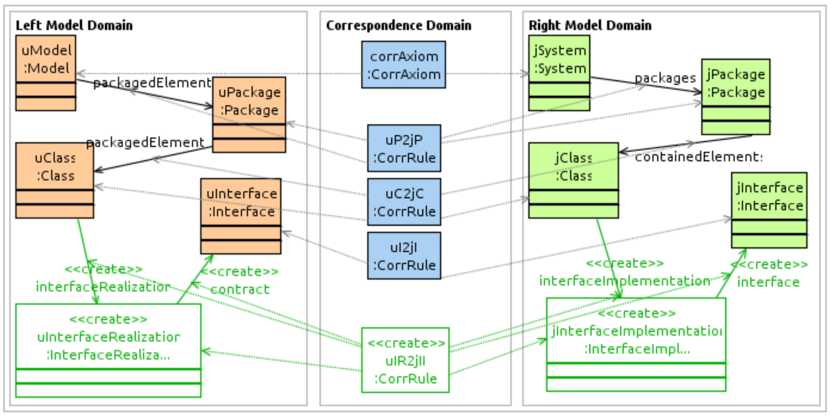
Figure 4.12: The triple rule *uClassAttribute2jClassField*


Left Model Domain
uModel :Model
packagedElement
uPackage :Package
packagedElement
uClass :Class
uType :Classifier
<<create>> ownedAttribute
<<create>> type
<<create>> uAttribute :Property
name := OCL: attributeName
default := OCL: attributeDefaul
isStatic := OCL: attributeIsStat
isLeaf := OCL: attributeIsFina
visibility := OCL: attributeVisibility
Correspondence Domain
corrAxiom :CorrAxiom
uP2jP :CorrRule
uC2jC :CorrRule
uT2jT :CorrRule
<<create>> uCA2jCA :CorrRule
Right Model Domain
jSystem :System
packages
jPackage :Package
containedElements
jClass :Class
jType :Classifier
<<create>> fields
<<create>> type
<<create>> jField :Field
visibility := OCL: attributeVisibilitySt
name := OCL: attributeName
default := OCL: attributeDefaul
isFinal := OCL: attributeIsFina
isStatic := OCL: attributeIsStat

Source: Image created by the author using the *Eclipse IDE*.

The triple rules describing the relation between *Model* and *System*, as well as *Package*(UML) and *Package*(Java) is available on fig. 2.6b. The triple rule encoding the relation between an *Attribute* of a *Class* in *UmlClassDiagram* and a *Field* of a *Class* in *Java* is shown on fig. 4.12. In essence, this rule formalize the fact that for every *Class Attribute* (white filling in the image) in *UmlClassDiagram* a correspondent *Class Field* in *Java* is supposed to exist with corresponding values for the respective meta-attributes. These are in the *UML Property name*, *default*, *isStatic*, *isLeaf*, and *visibility*, which correspond in order to *name*, *default*, *isStatic*, *isFinal*, and *visibility* in the *Java Field*.

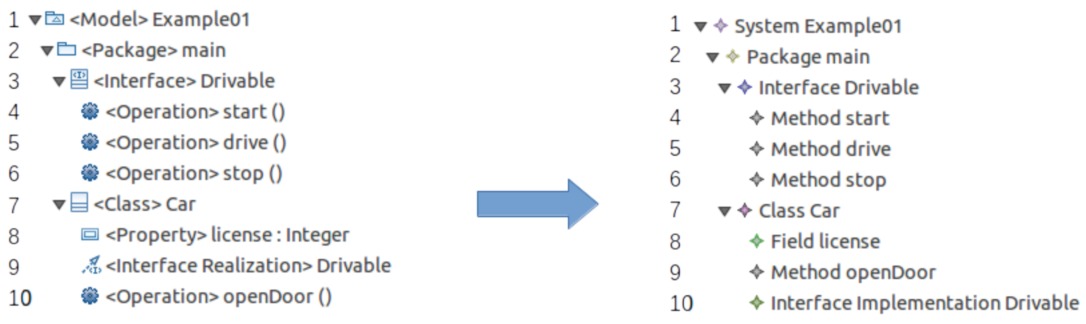
Figure 4.13: The triple rule *uInterfaceRealization2jInterfaceImplementation*



Source: Image created by the author using the *Eclipse IDE*.

Analogously, the figure 4.13 shows the triple rule $L_sL_cL_t \to R_sR_cR_t$ for *UML InterfaceRealization* and *Java InterfaceImplementation*. In this case, an *InterfaceRealizations* have none meta-attributes to be described, besides the associations with *Class* and *Interface*. This rule could be as as follows: Given a state S_i with a triple graph $L_sL_cL_t$ (LHS) containing all the color-filled in fig. 4.13, the creation of a *UML InterfaceRealization* ($uInterfaceRealization \in R_s$) implies the creation of a *Java InterfaceImplementation* ($uInterfaceImplementation \in R_t$) connected by an element of the correspondence domain ($uIR2jII \in R_c$) and vice-versa. The problem with this transformation is that it does not encode the fact that the implementing class in Java (*jClass*) should contain the m methods defined by the interface (*uInterface*). A possible solution would be the creation of a new rule including an object (*uMethod*) contained by (*uInterface*) in L_s linked with the respective *jMethod* contained in *jClass* in R_t (RHS). This new rule should then be evaluated m times by the transformation engine (for the creation of the m methods in *jClass*), but in fact it can be executed only once in the operational semantic scheme proposed by Giese et al. (2010a, p. 9). In the same scheme, the creation of such rule would entail a critical pair, given that two different rules have the object *uInterfaceRealization* in the R_s. A definite solution is not know by us, therefore this synchronization task ends up being left to the developer.

Figure 4.14: Example of forward synchronization from a *UMLClassDiagram* model (left) and a *Java* (right) model



```
<Model> Example01
  <Package> main
    <Interface> Drivable
      <Operation> start ()
      <Operation> drive ()
      <Operation> stop ()
    <Class> Car
      <Property> license : Integer
      <Interface Realization> Drivable
      <Operation> openDoor ()
```

```
System Example01
  Package main
    Interface Drivable
      Method start
      Method drive
      Method stop
    Class Car
      Field license
      Method openDoor
      Interface Implementation Drivable
```

Source: Image created by the author using the *Eclipse IDE*.

Fig. 4.14 shows the result of a forward transformation from *UMLClassDiagram* to *Java* executed by the *MoTE transformation tool* based on the rules presented in this section plus some simpler rules not shown here (i.e *uModel2jSystem*, *uPackage2jPackage*, *uClass2jClass*, emphuCOperation2jCOperation and *uIOperation2jIOperation*).

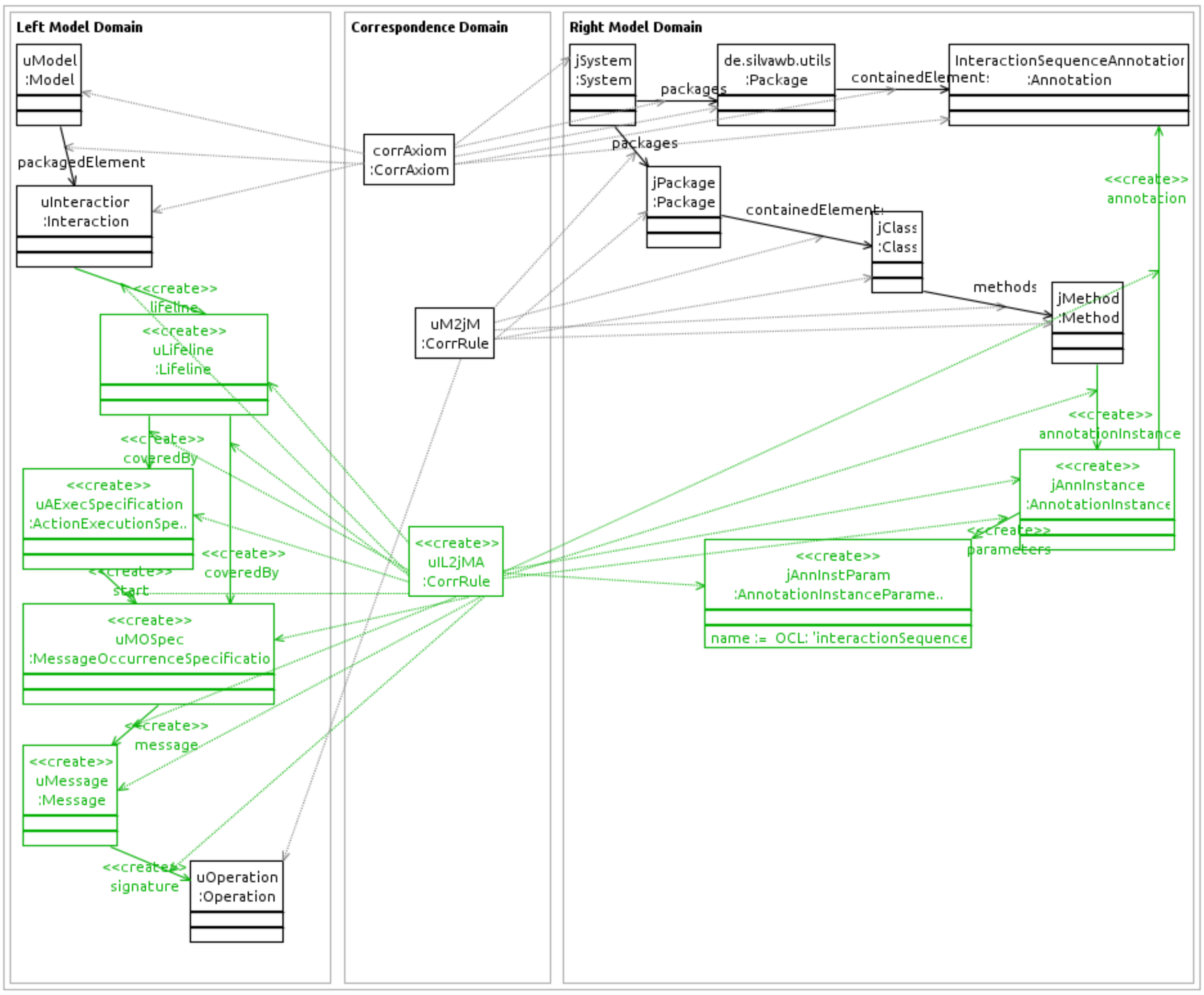
4.3.2 UmlInteraction2Java

FIGURE 4.15 to 4.17

4.3.3 UmlContracts2Java

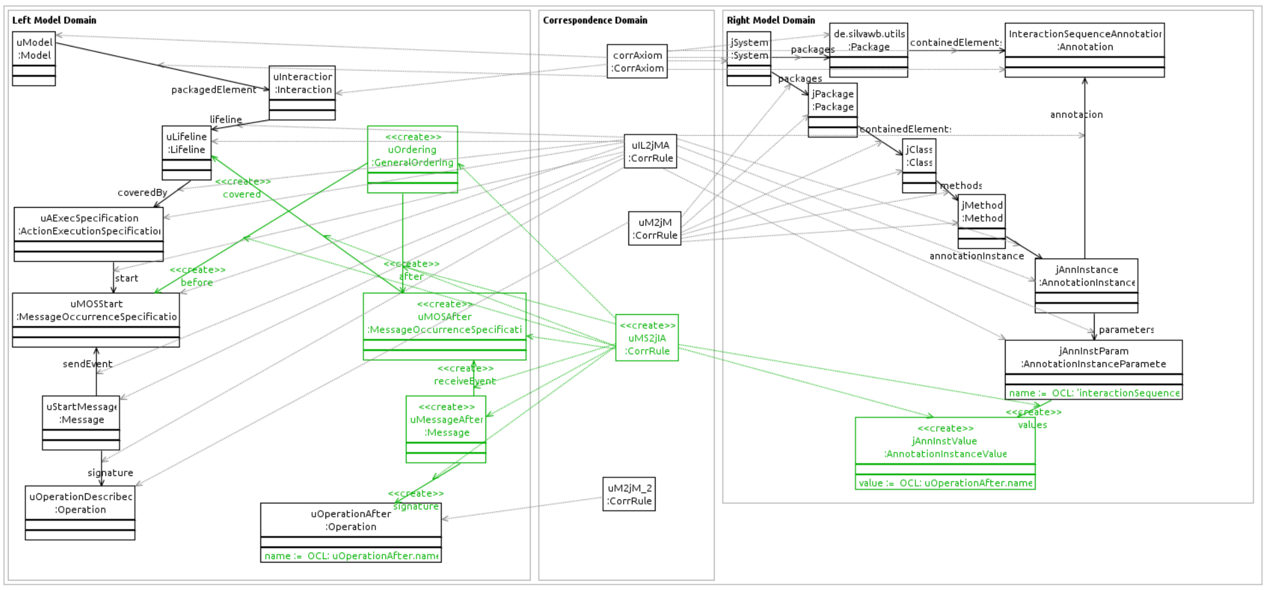
FIGURE 4.18 to 4.22

Figure 4.15


Left Model Domain
Correspondence Domain
Right Model Domain
uModel :Model
packagedElement
uInteraction :Interaction
<<create>> lifeline
<<create>> uLifeline :Lifeline
<<create>> coveredBy
<<create>> uAExecSpecification :ActionExecutionSpe..
<<create>> coveredBy
<<create>> start
<<create>> uMOSpec :MessageOccurrenceSpecificatio
<<create>> message
<<create>> uMessage :Message
<<create>> signature
uOperation :Operation
corrAxiom :CorrAxiom
uM2jM :CorrRule
<<create>> uIL2jMA :CorrRule
jSystem :System
packages
de.silvawb.utils :Package
containedElements
InteractionSequenceAnnotation :Annotation
packages
jPackage :Package
containedElements
jClass :Class
methods
jMethod :Method
<<create>> annotation
<<create>> annotationInstance
<<create>> jAnnInstance :AnnotationInstance
<<create>> parameters
<<create>> jAnnInstParam :AnnotationInstanceParame..
name := OCL: 'interactionSequence

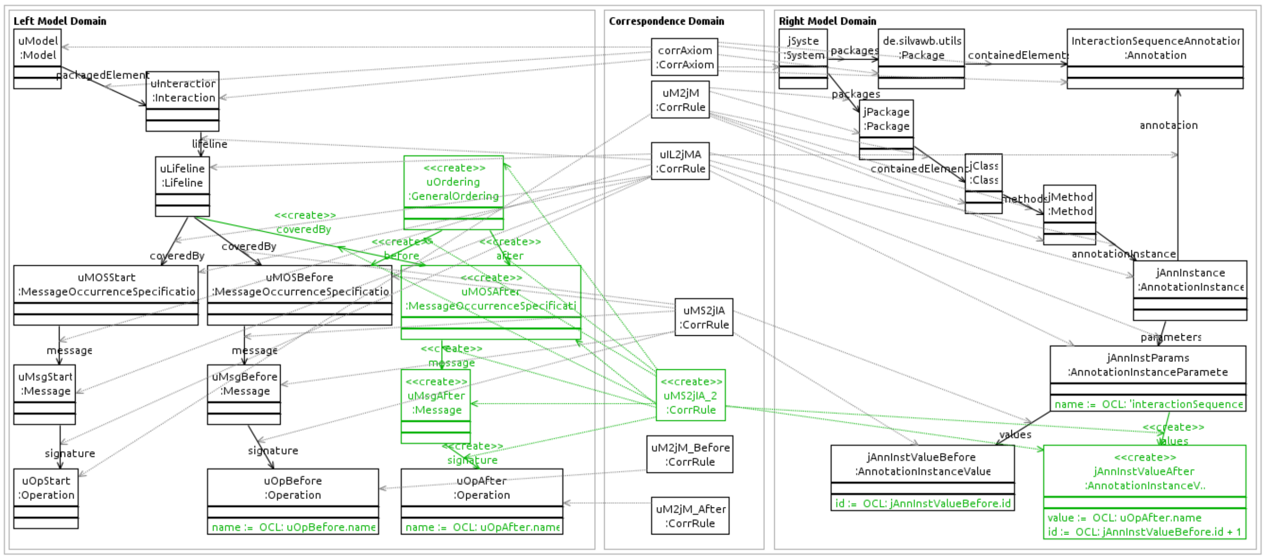
Source: Image created by the author using the *Eclipse IDE*.

Figure 4.16


Left Model Domain
Correspondence Domain
Right Model Domain
uModel :Model
uInteraction :Interaction
packagedElement
lifeline
uLifeline :Lifeline
<<create>> uOrdering :GeneralOrdering
<<create>> covered
coveredBy
uAExecSpecification :ActionExecutionSpecification
start
<<create>> before
<<create>> after
uMOSStart :MessageOccurrenceSpecification
<<create>> uMOSAfter :MessageOccurrenceSpecification
sendEvent
<<create>> receiveEvent
uStartMessage :Message
<<create>> uMessageAfter :Message
signature
<<create>> signature
uOperationDescribed :Operation
uOperationAfter :Operation
name := OCL: uOperationAfter.name
corrAxiom :CorrAxiom
uIL2jMA :CorrRule
uM2jM :CorrRule
<<create>> uMS2jIA :CorrRule
uM2jM_2 :CorrRule
jSystem :System
packages
de.silvawb.utils :Package
containedElements
InteractionSequenceAnnotation :Annotation
jPackage :Package
jClass :Class
methods
jMethod :Method
annotation
annotationInstance
jAnnInstance :AnnotationInstance
parameters
jAnnInstParam :AnnotationInstanceParameter
name := OCL: 'interactionSequence
<<create>> values
<<create>> jAnnInstValue :AnnotationInstanceValue
value := OCL: uOperationAfter.name

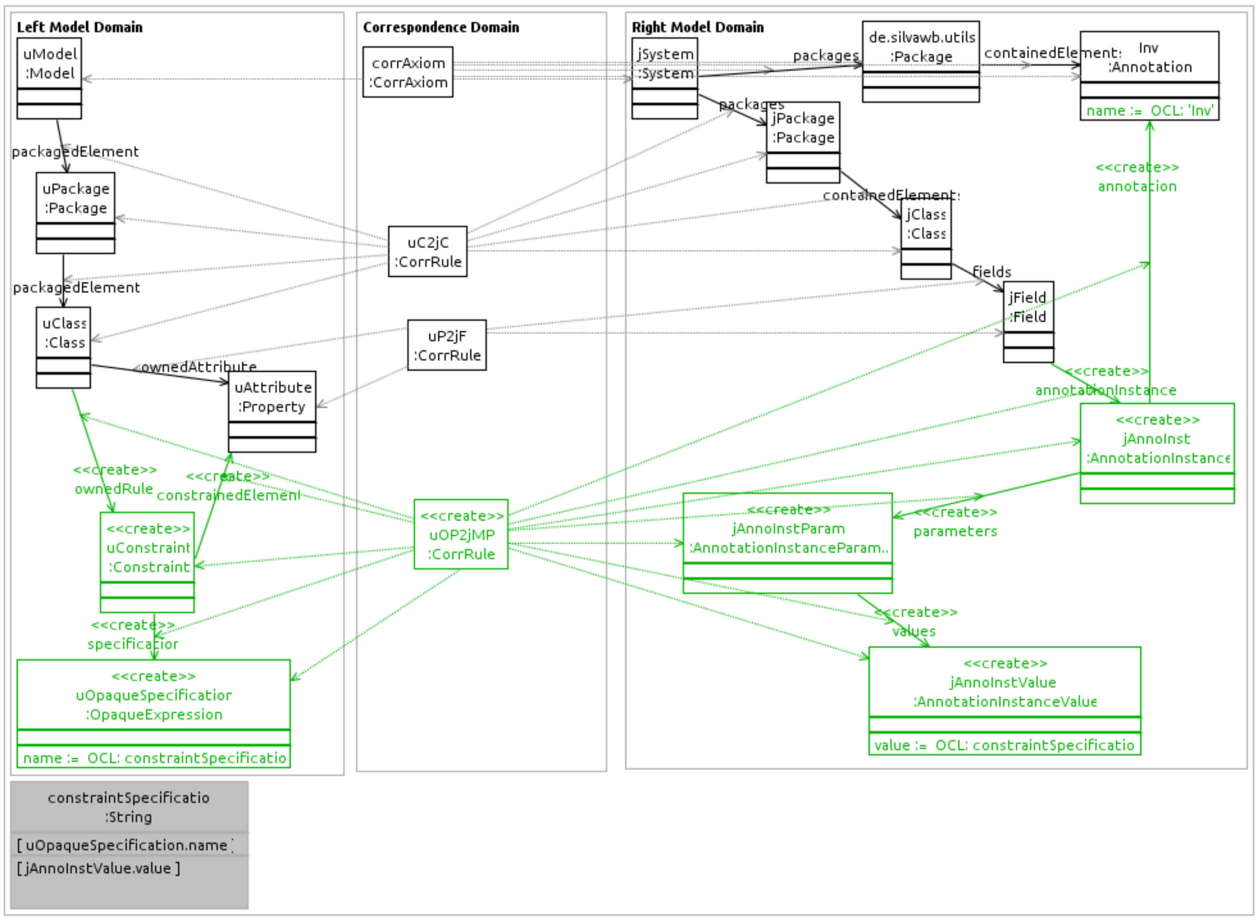
Source: Image created by the author using the *Eclipse IDE*.

Figure 4.17


Left Model Domain
Correspondence Domain
Right Model Domain
uModel :Model
packagedElement
uInteraction :Interaction
lifeline
uLifeline :Lifeline
<<create>> uOrdering :GeneralOrdering
<<create>> coveredBy
coveredBy
coveredBy
<<create>> before
<<create>> after
uMOSStart :MessageOccurrenceSpecification
uMOSBefore :MessageOccurrenceSpecification
<<create>> uMOSAfter :MessageOccurrenceSpecification
message
message
<<create>> message
uMsgStart :Message
uMsgBefore :Message
<<create>> uMsgAfter :Message
signature
signature
<<create>> signature
uOpStart :Operation
uOpBefore :Operation
name := OCL: uOpBefore.name
uOpAfter :Operation
name := OCL: uOpAfter.name
corrAxiom :CorrAxiom
uM2jM :CorrRule
uIL2jMA :CorrRule
uMS2jIA :CorrRule
<<create>> uMS2jIA_2 :CorrRule
uM2jM_Before :CorrRule
uM2jM_After :CorrRule
jSystem :System
packages
de.silvawb.utils :Package
containedElements
InteractionSequenceAnnotation :Annotation
jPackage :Package
annotation
jClass :Class
methods
jMethod :Method
annotationInstance
jAnnInstance :AnnotationInstance
parameters
jAnnInstParams :AnnotationInstanceParameter
name := OCL: 'interactionSequence
values
<<create>> values
jAnnInstValueBefore :AnnotationInstanceValue
id := OCL: jAnnInstValueBefore.id
<<create>> jAnnInstValueAfter :AnnotationInstanceV..
value := OCL: uOpAfter.name
id := OCL: jAnnInstValueBefore.id + 1

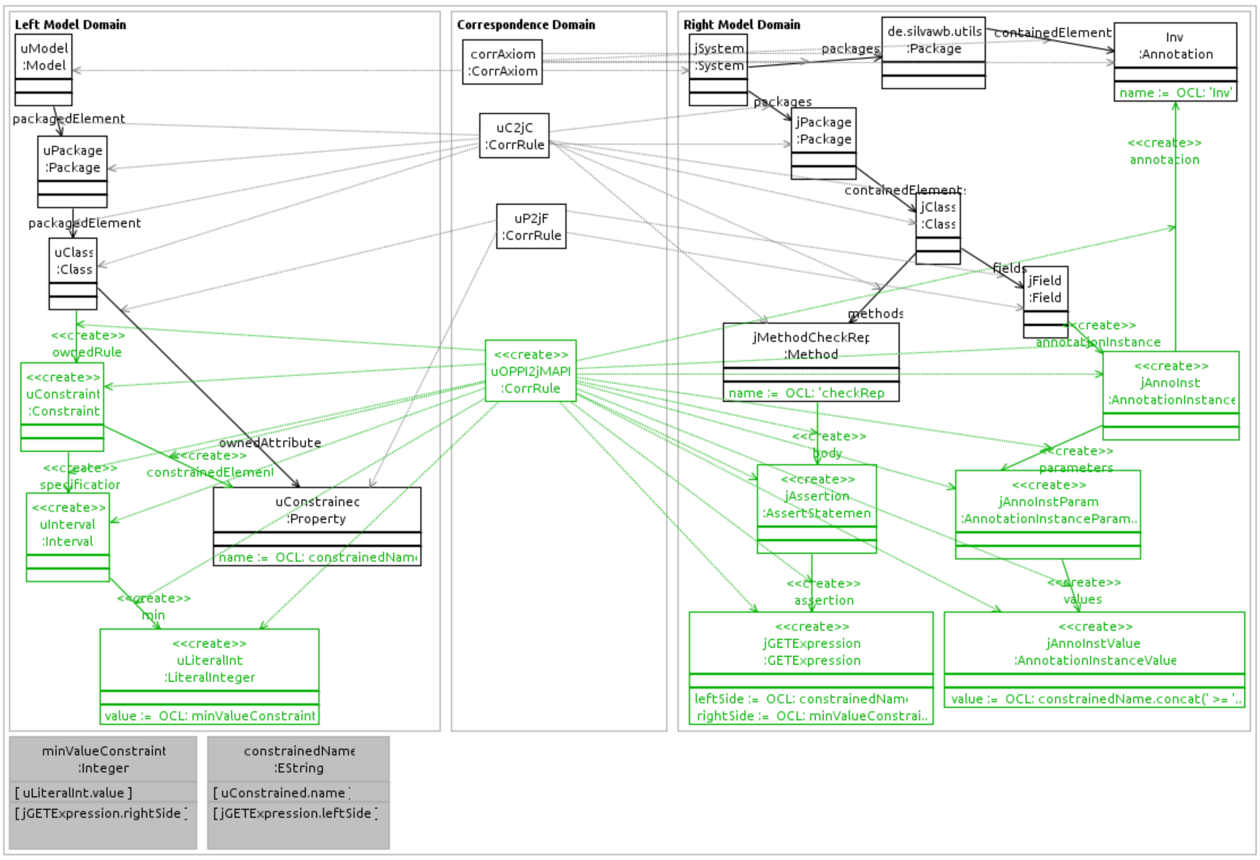
Source: Image created by the author using the *Eclipse IDE*.

Figure 4.18


Left Model Domain
Correspondence Domain
Right Model Domain
uModel :Model
corrAxiom :CorrAxiom
jSystem :System
de.silvawb.utils :Package
Inv :Annotation
name := OCL: 'Inv'
packages
containedElements
packagedElement
uPackage :Package
jPackage :Package
uC2jC :CorrRule
jClass :Class
uClass :Class
fields
jField :Field
uP2jF :CorrRule
ownedAttribute
uAttribute :Property
<<create>> annotation
<<create>> annotationInstance
<<create>> jAnnoInst :AnnotationInstance
<<create>> ownedRule
<<create>> constrainedElement
<<create>> uOP2jMP :CorrRule
<<create>> jAnnoInstParam :AnnotationInstanceParam
<<create>> parameters
<<create>> uConstraint :Constraint
<<create>> specification
<<create>> values
<<create>> uOpaqueSpecification :OpaqueExpression
name := OCL: constraintSpecification
<<create>> jAnnoInstValue :AnnotationInstanceValue
value := OCL: constraintSpecification
constraintSpecification :String
[uOpaqueSpecification.name]
[jAnnoInstValue.value]

Source: Image created by the author using the *Eclipse IDE*.

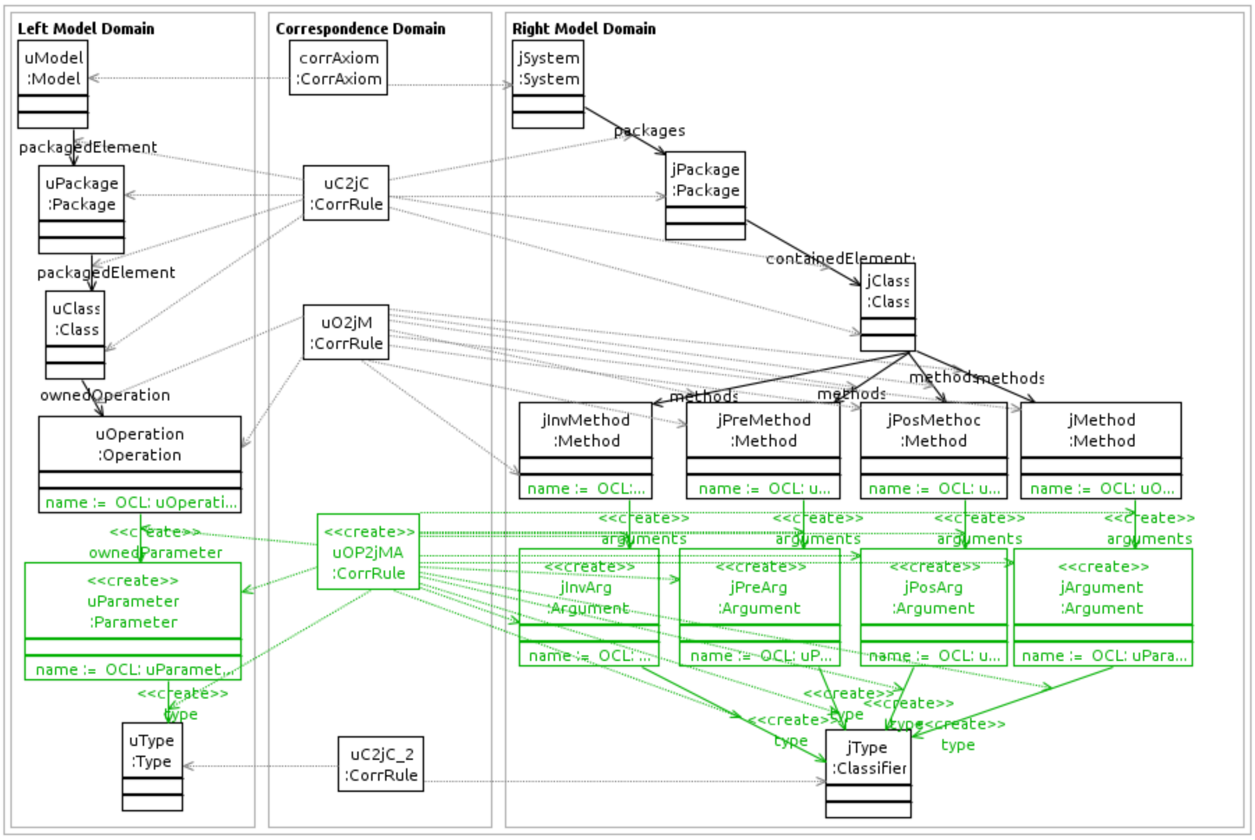
Figure 4.19



Left Model Domain
uModel :Model
packagedElement
uPackage :Package
packagedElement
uClass :Class
<<create>> ownedRule
<<create>> uConstraint :Constraint
<<create>> specification
<<create>> uInterval :Interval
ownedAttribute
<<create>> constrainedElement
uConstrained :Property
name := OCL: constrainedName
<<create>> min
<<create>> uLiteralInt :LiteralInteger
value := OCL: minValueConstraint
Correspondence Domain
corrAxiom :CorrAxiom
uC2jC :CorrRule
uP2jF :CorrRule
<<create>> uOPPI2jMAPI :CorrRule
Right Model Domain
jSystem :System
packages
de.silvawb.utils :Package
containedElement
Inv :Annotation
name := OCL: 'Inv'
packages
jPackage :Package
containedElement
jClass :Class
fields
jField :Field
methods
jMethodCheckRep :Method
name := OCL: 'checkRep
<<create>> annotation
<<create>> annotationInstance
<<create>> jAnnoInst :AnnotationInstance
<<create>> body
<<create>> jAssertion :AssertStatement
<<create>> parameters
<<create>> jAnnoInstParam :AnnotationInstanceParam..
<<create>> assertion
<<create>> jGETExpression :GETExpression
leftSide := OCL: constrainedName
rightSide := OCL: minValueConstrai..
<<create>> values
<<create>> jAnnoInstValue :AnnotationInstanceValue
value := OCL: constrainedName.concat(' >= '..
minValueConstraint :Integer
[uLiteralInt.value]
[jGETExpression.rightSide]
constrainedName :EString
[uConstrained.name]
[jGETExpression.leftSide]

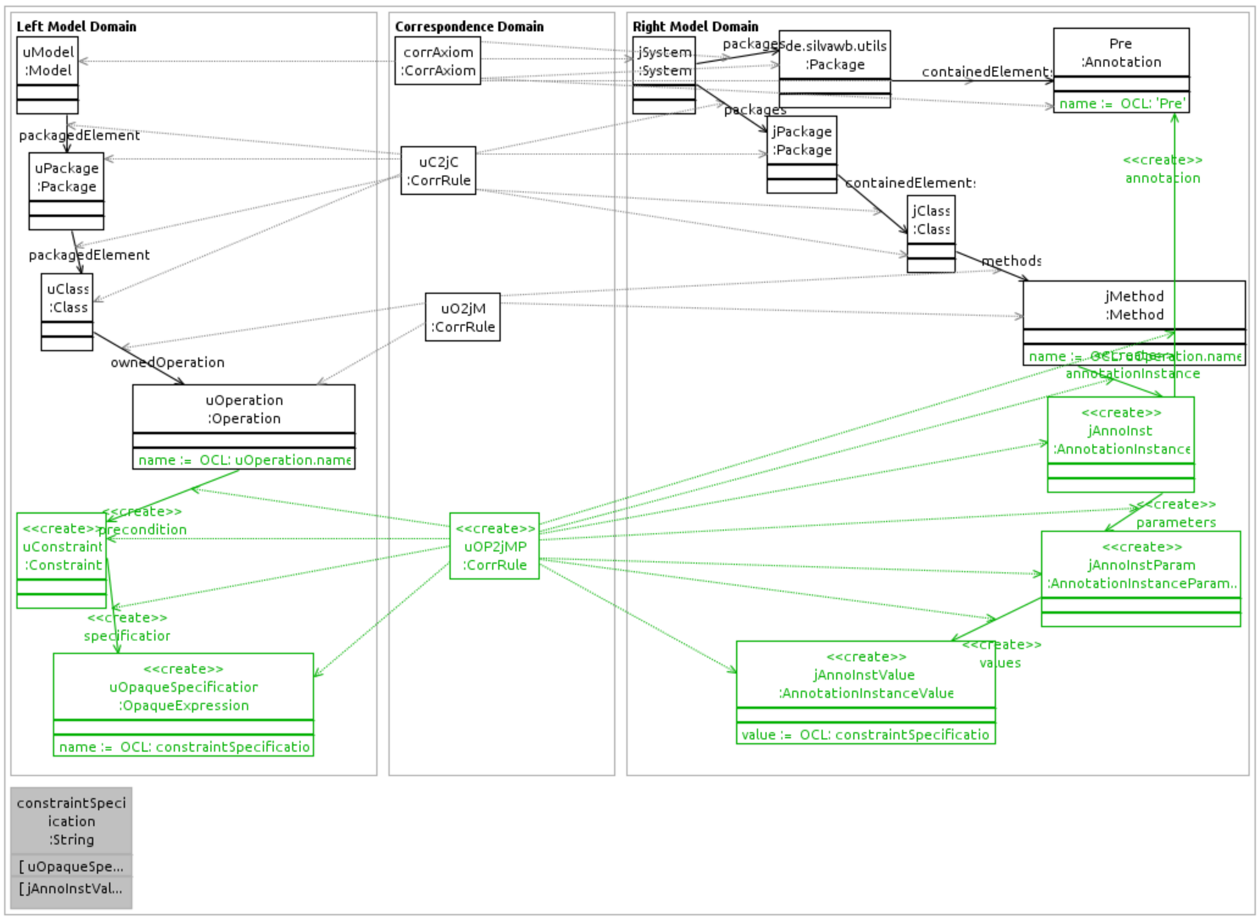
Source: Image created by the author using the *Eclipse IDE*.

Figure 4.20



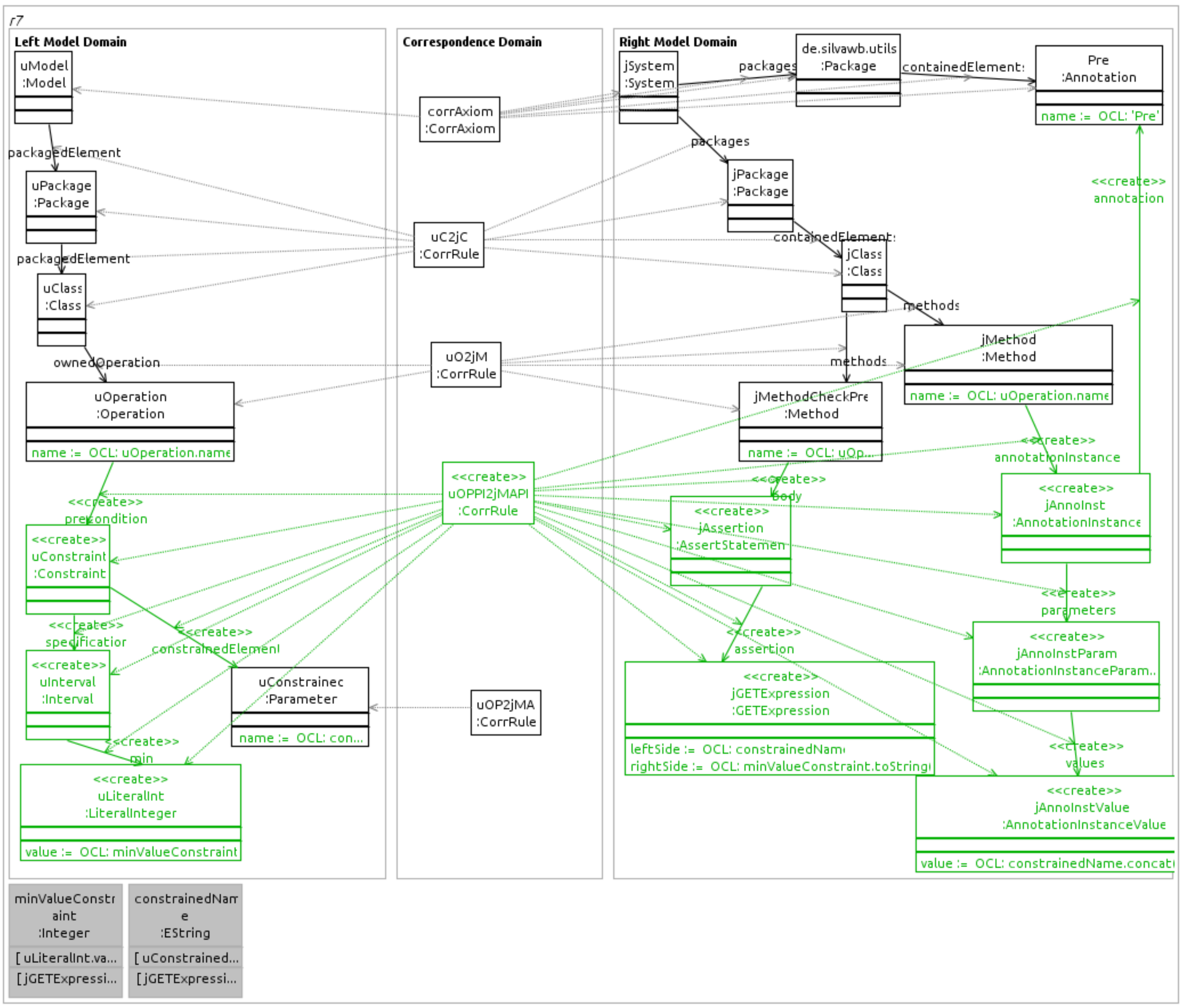
Source: Image created by the author using the *Eclipse IDE*.

Figure 4.21



Source: Image created by the author using the *Eclipse IDE*.

Figure 4.22



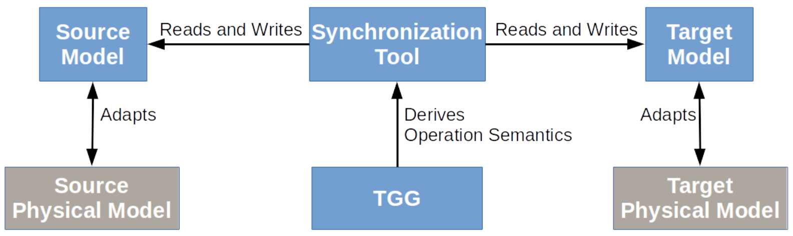
Source: Image created by the author using the *Eclipse IDE*.

5 SYNCHRONIZATION OF MODELS IN THE JAVA TECHNOLOGICAL SPACE

In the last chapter the relations between some meta-models of the Java technological space were presented in terms of triple graphs, organized as triple rules and consequently in triple graph grammars. Each TGG was shown in a different subsection (namely 4.3.1, 4.3.2 and 4.3.3) and represents a different edge of the network of models built (see fig. 4.1). By being so, each edge corresponds to a different synchronization problem and can be sold, generally speaking, independently from the others. most of the research in the realm deals with such situation. Therefore there are several approaches attempting to solve model synchronization between two models using TGG.

Between them is the *FUJABA* (Nickel et al., 2000), a standalone application that uses TGG to code the relations. Or the *CoWolf*, an extensible framework based on *Eclipse Plug-ins* supporting a pair of meta-models according to Getir et al., but that uses *Henshin* (Arendt et al., 2010) to encode the relations. Finally there is the *MoTE* transformation tool, a series of *Eclipse Plug-ins* for creating TGG's, both graphically and textually; and for transforming models based on these TGG's. We take *MoTE* for the most adequate option for our needs, mainly because of extensive literature about it ((Giese and Hildebrandt, 2009) and (Hildebrandt et al., 2012)) and the easy integration with other technologies, like the *EMF Ecore* or other useful *Eclipse Plug-ins*.

Figure 5.1: Synchronization scheme of the *MoTE* tool



Source: Adapted from (Giese et al., 2010b).

The figure 5.1 summarizes how the *MoTE* tool works. Basically, the input TGG is used to derive the operational semantics of the transformation between the source and target models, which are read and written by the synchronization tool. Only these first components (highlighted in the picture) are treated in this thesis. Note, however, that the final practical application of the synchronization is only possible through the adaptation of the logical models (presented so far in form of abstract syntax, e.g 4.9) to physical models (presented so far in form of concrete syntax, e.g 4.10).

As stated before, there exists one synchronization scheme similar to fig. 5.1 for each edge of our network of models. But as the whole network have to be maintained, it raises the problem of synchronizing not only two models separately, but instead a whole set of models. In this scenario, each modification on a specific model has to be propagated through the net. For that, a theoretical analysis of the problem, followed by an algorithm is presented below.

5.1 Synchronizing the Network of Models

As stated in the chapter 2, a *network of models* is a graph $G = (V, E)$, with V being the models and E the edges linking each pair of related models. When one of these vertices $v \in V$ undergoes changes, all its direct neighbors $N(v)$ have to be updated (synchronized) accordingly. As they possibly undergo changes in the process, their neighbors have to be synchronized too, as well as the next neighbors and so on. The preoccupation here is to describe an algorithm to propagate such modifications, and analyze some properties of this algorithm.

A synchronization is then defined as a function $sync : S \times S \times \Delta_S \times T \rightarrow T \times \Delta_T$, where $sync(s_0, s_1, \delta_s, t_0) = (t_1, \delta_t)$ means that, given a source model s_0 synchronized with the target model t_0; a new updated source model s_1; and δ_s representing the modifications over s_0 that produced s_1; a new model t_1 synchronized with s_1 is produced together with the modifications δ_t necessary for such process. Here two important assumptions were made:

- **Supposition 1:** Only one model can be modified at a time, this means only one vertex can be modified at a time in the whole network – two models are not allowed to be modified simultaneously. This does not seems to be a very problematic restriction, once that in a practical scenario the models could be maintained centralized and observed for changes. Whenever it happens, the synchronization algorithm is ran.
- **Supposition 2:** A synchronization execution has a direction: Whether forward or backward, but not both at the same time. In the first case the source model (that underwent user changes) updates the target model, but does not have side effects, meaning that the synchronization does not provoke extra modification besides the user's ones. To put in other words, in one step the synchronization effects does not ripple back to the source, instead it goes only further.
- **Corollary 1:** One single execution of $sync(s_0, s_1, \delta_s, t_0)$ is enough to synchronize s_1 with

t_0.

The goal of this paper is not the definition of the $sync$ algorithm, but the definition of an algorithm for the synchronization of a whole network of models.

This synchronization is performed through a function $netsync : (V, E) \times V \times V \times \Delta_S \to (V, E)$, where $netsync((V_0, E_0), s_0, s_1, \delta_s) = (V_1, E_1)$ means that for the synchronized network (V_0, E_0) containing the vertex $s_0 \in V_0$, whose model underwent δ_s modifications, resulting in s_1, a new network of synchronized models (V_1, E_1) is delivered. The $netsync$ algorithm is defined in the listing 5.1.

```
1  algorithm netsync ((V0,E0), s0, s1, ds)
2    (Vi,Ei) := (V0 \ s0 U s1, E0) //New net with first modification
3      for each ni := N(s0) do
4        (ni_new,dn) := sync(s0, s1, ds, ni) //Update neighbor
5        //If modified the neighbor
6        if (dn not empty) then
7          //then update net starting from it
8          (Vi,Ei) := neysync((Vi,Ei), ni, ni_new, dn)
9        endif
10     endfor
11     return (Vi,Ei)
12 end
```

Listing 5.1: neysync imperative algorithm definition

Firstly the initial network is updated with the new vertex $s1$, then every neighbor ni of s is synchronized according to the modifications ds. If it causes modifications on ni, then the network is recursively further synchronized starting from ni. Extra assumptions were made by this algorithm:

- **Supposition 3:** The input network of models $(V0, E0)$ is finite.
- **Supposition 4:** The input network of models $(V0, E0)$ has no cycles.

This implies the following properties:

- **Theorem 1:** The algorithm always terminates. Because supposition 2 is made, it suffices only one call to $sinc$ on line 4, representing the synchronization of $s0$ and ni – i.e the edge $(s0, ni)$), to synchronize both models (see corollary 1). Furthermore, every edge synchronization is followed by maximal one recursive call on line 8. As the set of edges is finite (Supposition 3), so is the amount of recursive calls (line 8) and so is the number of loops iterating over any vertex neighbors (lines 3 to 10). So is guaranteed the termination of the algorithm.

- **Theorem 2:** The time complexity of $netsync((V0, E0), s0, s1, ds)$ is in the worst case $O(\Delta(V0, E0)|E0|)$, where a $sync$ call is taken as elementary operation.
- **Theorem 3:** The algorithm is deterministic...

DISCUSS CYCLE; TGG for Multiple models

6 CONCLUSION AND DISCUSSION

A network of meta-models of the Java technological space was developed, comprising a set of common meta-models used in Java software and the respective relations between them, as well as synchronization between the models was showcased, demonstrating as final result that the construction of such network and the employment of synchronization has worked and moreover seems to be promising. Despite the outcomes are not complete and not ripe enough to be put in practice, the ideas and the insights reported plus the summary of the state-of-art are valuable for current research and thus contributing.

In the first phase some meta-models were developed (e.g Java), and in the second phase the relations were coded (e.g. umlClass2java). In regard to these both steps the main legacy are the novel relations between some elements of some meta-models (e.g. the relation between UML contracts and Java assertions). The third phase serve to demonstrate briefly the application of synchronization in the practice. Most of the initially set up goals were achieved, even though the relations could rather be more extensive, likewise a more comprehensive synchronization demonstration would be desired.

Some difficulties were found along the work, but they absolutely did not obstructed the success of the final result. The first one the lack of openly available meta-models in the literature or by the vendors. For instance *Oracle* does not publicize any standard meta-model for Java, nor are them easily to find in the literature. One may find alternative versions in the source code of IDE's, but it still requires some cost. Moreover, they are sometimes because their format incompatible with the employed tool, what also hinders the progress of someone's work. The result of this thesis may be a partial solution for that, although meta-models of other technological spaces still lack a similar work. Another complication are the lack of documentation of some tools – in special the *MoTE* –, that makes both the flow of the development and eventual debug tasks sometimes troublesome. *MoTE* might have publications about it and also a good reputation in the community, but a extensive broad documentation of the plug-in for the *Eclipse IDE* is needed. At last, but not least is the performance problem of such tools. Both the *EMF* and *MoTE* need to generate Java code in order to run the synchronization procedure, and with big models this process happens to cost a considerable computation time.

Some points become therefore remarkable for future work. Firstly, an easy to find and accessible tutorial or instructions for the theoretic and practical basis of triple graph grammars is valuable in order to make the use of models synchronization popular among engineers or software developers, who sometimes are not very used to the area and thus express a big rejection to apply such technique in their projects.

Secondly, the work of this thesis can be naturally continued and expanded, by completing the identification of relation between the meta-model or by creating meta-models that satisfy completeness. Not to mention, such relations could be expressed in other languages (e.g. *ATL*) and the same work extended to other technological spaces (e.g. COBOL, C#).

Furthermore, the implementation of a *Eclipse* plug-in should not be a big problem, since the *MoTE* already generates a plug-in for the execution of its algorithm. So the task in this case would be enhance it with a user-friendly interface plus new functionalities and naturally a slightly more practical approach (e.g. handling of actual Java code instead of quasi-theoretic scenarios like Java expressed in XML files).

And lately, a relatively big issue is the use of TGG's for non-MOF-compliant meta-models – or meta-models that are not naturally seen as MOF-compliant, e.g. the complete Java model. Because under certain conditions it could be interesting to see such models from another point of view than the MOF. A clear example is source-code, which can be treated in an easier way with abstract syntax trees. Angyal et al. (2008) suggests a method to treat this case, but anyways it still remains an open problem and certainly a future challenge.

REFERENCES

László Angyal, László Lengyel, and Hassan Charaf. Novel techniques for model-code synchronization. *Electronic Communications of the EASST*, 8, 2008.

Giuliano Antoniol, Gerardo Canfora, Gerardo Casazza, Andrea De Lucia, and Ettore Merlo. Recovering traceability links between code and documentation. *Software Engineering, IEEE Transactions on*, 28(10):970–983, 2002.

Thorsten Arendt, Enrico Biermann, Stefan Jurack, Christian Krause, and Gabriele Taentzer. Henshin: advanced concepts and tools for in-place emf model transformations. In *Model Driven Engineering Languages and Systems*, pages 121–135. Springer, 2010.

Dominique Blouin, Alain Plantec, Pierre Dissaux, Frank Singhoff, and Jean-Philippe Diguet. Synchronization of models of rich languages with triple graph grammars: An experience report. In *Theory and Practice of Model Transformations*, pages 106–121. Springer, 2014.

Paul E Ceruzzi. *A history of modern computing*. MIT press, 2003.

Peter Pin-Shan Chen. The entity-relationship model—toward a unified view of data. *ACM Transactions on Database Systems (TODS)*, 1(1):9–36, 1976.

Krzysztof Czarnecki and Simon Helsen. Feature-based survey of model transformation approaches. *IBM Systems Journal*, 45(3):621–645, 2006.

Zinovy Diskin. Model synchronization: Mappings, tiles, and categories. In *Generative and Transformational Techniques in Software Engineering III*, pages 92–165. Springer, 2011.

Zinovy Diskin, Arif Wider, Hamid Gholizadeh, and Krzysztof Czarnecki. Towards a rational taxonomy for increasingly symmetric model synchronization. In *Theory and Practice of Model Transformations*, pages 57–73. Springer, 2014.

Zinovy Diskin, Hamid Gholizadeh, Arif Wider, and Krzysztof Czarnecki. A three-dimensional taxonomy for bidirectional model synchronization. *Journal of Systems and Software*, 111: 298–322, 2016.

Hartmut Ehrig, Karsten Ehrig, Claudia Ermel, Frank Hermann, and Gabriele Taentzer. Information preserving bidirectional model transformations. In *Fundamental Approaches to Software Engineering*, pages 72–86. Springer, 2007.

Jean-Marie Favre. Foundations of model (driven)(reverse) engineering: Models. In *Proceedings of the International Seminar on Language Engineering for Model-Driven Software Development, Dagstuhl Seminar 04101*, 2004a.

Jean-Marie Favre. Foundations of meta-pyramids: languages vs. metamodels. In *Episode II. Story of Thotus the Baboon, Procs. Dagstuhl Seminar*, volume 4101. Citeseer, 2004b.

Luciana Foss, S Costa, Nicolas Bisi, Lisane Brisolara, and Flávio Wagner. From uml to simulink: a graph grammar specification. In *14th Brazilian Symposium on Formal Methods: Short Papers*, pages 37–42, 2011.

Robert France and Bernhard Rumpe. Model-driven development of complex software: A research roadmap. In *2007 Future of Software Engineering*, pages 37–54. IEEE Computer Society, 2007.

Sinem Getir, Lars Grunske, Christian Karl Bernasko, Verena Käfer, and Tim Sanwald. Cowolf-a generic framework for multi-view co-evolution and evaluation of models-tool paper.

Holger Giese and Stefan Hildebrandt. *Efficient model synchronization of large-scale models*. Number 28. Universitätsverlag Potsdam, 2009.

Holger Giese and Robert Wagner. Incremental model synchronization with triple graph grammars. In *Model Driven Engineering Languages and Systems*, pages 543–557. Springer, 2006.

Holger Giese, Stephan Hildebrandt, and Leen Lambers. Toward bridging the gap between formal semantics and implementation of triple graph grammars. In *Model-Driven Engineering, Verification, and Validation (MoDeVVa), 2010 Workshop on*, pages 19–24. IEEE, 2010a.

Holger Giese, Stephan Hildebrandt, and Stefan Neumann. Model synchronization at work: keeping sysml and autosar models consistent. In *Graph transformations and model-driven engineering*, pages 555–579. Springer, 2010b.

Joel Greenyer, Ekkart Kindler, Jan Rieke, and Oleg Travkin. Tggs for transforming uml to csp: Contribution to the agtive 2007 graph transformation tools contest. Technical report, Department of Computer Science, University of Paderborn Paderborn, Germany, 2008.

Florian Heidenreich, Jendrik Johannes, Mirko Seifert, and Christian Wende. *Jamopp: The java model parser and printer*. Techn. Univ., Fakultät Informatik, 2009.

Florian Heidenreich, Jendrik Johannes, Mirko Seifert, and Christian Wende. Closing the gap between modelling and java. In *Software Language Engineering*, pages 374–383. Springer, 2010.

Frank Hermann, Hartmut Ehrig, Fernando Orejas, Krzysztof Czarnecki, Zinovy Diskin, and Yingfei Xiong. Correctness of model synchronization based on triple graph grammars. In *Model Driven Engineering Languages and Systems*, pages 668–682. Springer, 2011.

Stephan Hildebrandt, Leen Lambers, and Holger Giese. The mdelab tool framework for the development of correct model transformations with triple graph grammars. In *Proceedings of the First Workshop on the Analysis of Model Transformations*, pages 33–34. ACM, 2012.

Stephan Hildebrandt, Leen Lambers, Holger Giese, Jan Rieke, Joel Greenyer, Wilhelm Schäfer, Marius Lauder, Anthony Anjorin, and Andy Schürr. A survey of triple graph grammar tools. *Electronic Communications of the EASST*, 57, 2013.

Igor Ivkovic and Kostas Kontogiannis. Tracing evolution changes of software artifacts through model synchronization. In *Software Maintenance, 2004. Proceedings. 20th IEEE International Conference on*, pages 252–261. IEEE, 2004.

Frédéric Jouault, Freddy Allilaire, Jean Bézivin, and Ivan Kurtev. Atl: A model transformation tool. *Science of computer programming*, 72(1):31–39, 2008.

Ivan Kurtev, Jean Bézivin, and Mehmet Akşit. Technological spaces: An initial appraisal. 2002.

Anders Mattsson, Björn Lundell, Brian Lings, and Brian Fitzgerald. Linking model-driven development and software architecture: a case study. *Software Engineering, IEEE Transactions on*, 35(1):83–93, 2009.

Tom Mens and Pieter Van Gorp. A taxonomy of model transformation. *Electronic Notes in Theoretical Computer Science*, 152:125–142, 2006.

Bertrand Meyer. Applying'design by contract'. *Computer*, 25(10):40–51, 1992.

Ulrich Nickel, Jörg Niere, and Albert Zündorf. The fujaba environment. In *Proceedings of the 22nd international conference on Software engineering*, pages 742–745. ACM, 2000.

T Noack. Automatic linking of test cases and requirements. In *5th International Conference on Advances in System Testing and Validation Lifecycle, Venice, Italy*, 2013.

Omg. Omg meta object facility (mof) core specification version 2.5. *Final Adopted Specification (June 2015)*, 2015.

OMG OMG. Unified modeling language (omg uml). *Superstructure*, 2007.

Hendrik Post, Carsten Sinz, Florian Merz, Thomas Gorges, and Thomas Kropf. Linking functional requirements and software verification. In *Requirements Engineering Conference, 2009. RE'09. 17th IEEE International*, pages 295–302. IEEE, 2009.

Ed Seidewitz. What models mean. *IEEE software*, (5):26–32, 2003.

Hui Song, Gang Huang, Franck Chauvel, Wei Zhang, Yanchun Sun, Weizhong Shao, and Hong Mei. Instant and incremental qvt transformation for runtime models. In *Model Driven Engineering Languages and Systems*, pages 273–288. Springer, 2011.

J Michael Spivey and JR Abrial. *The Z notation*. Prentice Hall Hemel Hempstead, 1992.

Dave Steinberg, Frank Budinsky, Ed Merks, and Marcelo Paternostro. *EMF: eclipse modeling framework*. Pearson Education, 2008.

Yingfei Xiong, Dongxi Liu, Zhenjiang Hu, Haiyan Zhao, Masato Takeichi, and Hong Mei. Towards automatic model synchronization from model transformations. In *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, pages 164–173. ACM, 2007.