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. Frank Trollmann
Reviewer: Prof. Dr. Dr. h.c. Sahin Albayrak
Reviewer: Prof. Dr. habil. Odej Kao

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

ACKNOWLEDGMENTS

....

ABSTRACT

The use of models in software engineering processes has grown in the last few years. And as it grows, grows also the relevance of some problems related to the realm. One of them is the model synchronization problem, that consists basically of keeping all the models of a software system consistent between themselves. In other words, the models of a software tend to be changed over its lifetime, and as it happens, these changes have to be forwarded properly to all the models regarded to this software. For big programs it is clearly inviable to do such synchronization procedure manually, therefore it is desired the creation of automatic methods able to perform synchronization.

This thesis aim to (1) identify models of the Java technological space, (2) define some (or simplified versions) of them formally, (3) identify and formalize some relations between them, creating a network of metamodels supposed to be kept synchronized, and show through a representative showcase how they work, and finally (4) discuss the synchronization of this network of models. The outcomes include the implementation of these relations plus the report about the experience of developing it in this thesis.

Keywords: Model Synchronization. Java Metamodels. Network of Models. Iterative Model Transformation. Model Transformation. Model-driven Engineering. Software Engineering.

Zur Synchronisation von Relationen Zwischen Artefakten im Java Technologischen Raum

ZUSAMMENFASSUNG

Die Anwendung von Modellen in Softwaretechnikprozessen hat in der letzten Jahren gewachsen. Als es zunimmt, nimmt auch die Relevanz eigener auf den Bereich bezogenen Probleme zu. Ein davon ist das Modell-Synchronisationsproblem, das grundsätzlich aus dem konsistenten Bewahren von allen Modellen eines Softwaresystems besteht. Anders ausgedrückt tendieren die Modelle einer Software dazu, im Laufe der Zeit verändert zu werden. Wenn das vorkommt, müssen diese Veränderungen richtig nach allen Modellen weitergeleitet werden. Für größere Programme ist es offensichtlich unmöglich, solches Synchronisationsverfahren manuell zu unternehmen. Daher ist es wünschenswert, die Schaffung von automatischen Verfahren, die in der Lage sind, Synchronisation durchzuführen.

Diese Bachelorarbeit hat als Ziele (1) Modelle des Java technologischen Raums zu identifizieren; (2) einige davon formal zu definieren; (3) einige Relationen zwischen denen zu identifizieren und formalisieren, indem ein Netzwerk von Metamodellen aufgebaut wird, das synchronisiert aufbewahrt werden soll, und ein repräsentatives Beispiel gezeigt wird; und schließlich (4) die Synchronisation dieses Netzwerkes von Modellen zu diskutieren. Die Ergebnisse umfassen sowohl die Implementation dieser Relationen, als auch den Bericht über die Erfahrung der vorliegenden Bachelorarbeit.

Schlagwörter: Modell-Synchronisation, Java Metamodelle, Netzwerk von Modellen, Iterative Modell-Transformation, Modell-Transformation, Modellorientierte Technik, Softwaretechnik.

LIST OF FIGURES

LIST OF ABBREVIATIONS AND ACRONYMS

ATL   Atlas Transformation Language

EMF   Eclipse Modeling Framework

HTML   Hypertext Markup Language

IDE   Integrated Development Environment

LHS   Left-hand Side

MDE   Model-driven Engineering

MDSD Model-driven Software Development

MOF   Meta-Object Facility

RHS   Right-hand Side

SUS   System Under Study

TGG   Triple Graph Grammar

UML   Unified Modeling Language

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 the complexity and the 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, consequently, the research on how to apply them systematically and correctly in the development processes, the so-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 metamodels in Java systems, describe them and identify their relations, so that they can be synchronized.

The remainder of this document is structured in 7 chapters. The first chapter introduces the scope and defines the problem dealt by this thesis. The second chapter presents a literature review comprising the basic definition necessary for the further development of the text. Chapter 3 shows the current stage of scientific research related to the field of this work. Chapter 4 presents the developed metamodels and the relations between, whereas the chapter 5 proposes an algorithm for the synchronization of models. Finally,

chapter 6 and 7 discuss the results and give an overview of the work.

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 metamodels 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 industry and the academic 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 it 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 metamodels used in the Java technological space is to be defined, this is done through an state-of-art research, since that some metamodels have already been defined by other authors, plus the creation of our own versions of some metamodels. So for example, in this phase the choice of the applied metamodels (i.e. *UML Class Diagram* and *Java Code*) is done and their definitions are written.

Later on, given the defined metamodels, the relations between them can be written. So for example, in this phase the inherent relation between the *UML Class Diagram* metamodel and the *Java Code* metamodel is written. Analogously, the relations between *Java Code* and other metamodels 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 metamodels ready, a showcase using a transformation method from the current academic literature is applied to illustrate how the relations between the metamodels work. We work therefore with the hypothesis, that the metamodels can be found or defined; that some relations between them can be written in some language; and that some of these relations can be transformed using a tool or technique available in the current literature.

It is out of the scope of this work the creation of complete metamodel definitions, or relations, or the implementation of a full-working synchronization algorithm. A deep theoretic analysis about the problem or about the performance of the relations is also excluded from this scope.

2 FOUNDATIONS

Before describing the development of this thesis, it 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 it 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). 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.

Metamodel: Favre (2004b, p. 14) affirms also: *"A metamodel is a model of a modelling language."*. In other words, a metamodel 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 metamodels are *UML specification document* (OMG, 2007), the *entity-relationship metamodel* (Chen, 1976) or the Java metamodel (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 metamodels is the *EMF Ecore*[1] (which is the modeling language for metamodels used in this thesis and is explained further below).

Figure 2.1: The summary of the definitions of system, model, metamodel, meta-metamodel and modeling language.
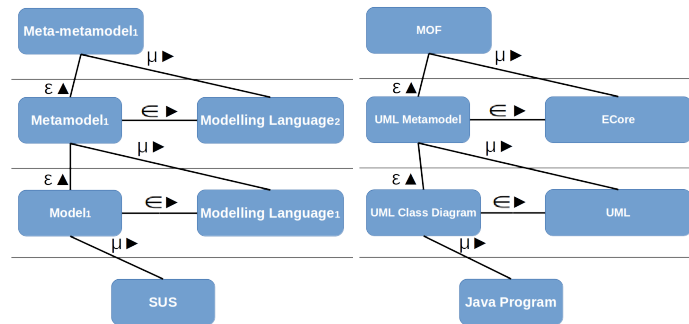
Source: The author

Meta-metamodel: Analogously to the metamodel 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. On the left is a depiction of the theoretical definitions of system,

---

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

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 metamodels. 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.

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 contract 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 $t : M \to N$, 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 metamodels 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).

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 $s : M \times M \times \Delta_M \times N \times N \times \Delta_M \to M \times N$, 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.
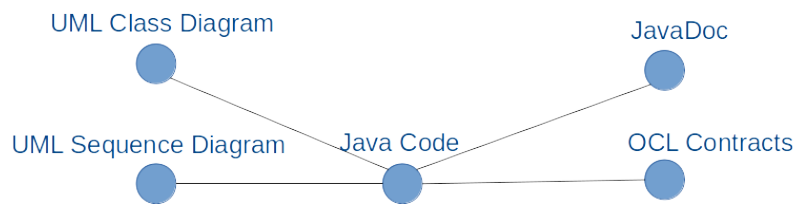
Source: The Author

Figure 2.3: Part of MOF definition, which handles basically classes, properties, operations, associations, and generalization.
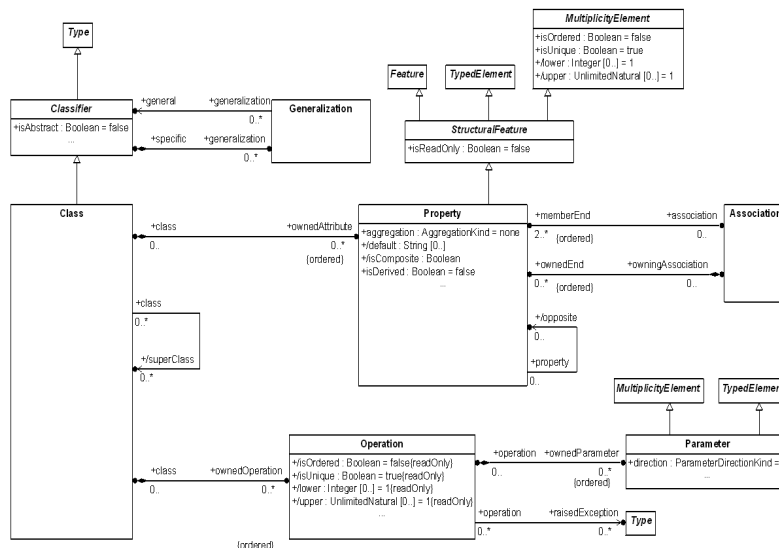
Source: (OMG, 2015, p. 27)

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. 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 metamodels. The Figure 2.3 shows one part of MOF.

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

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 rep-

---

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

resentation 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 $m = (m_s, m_c, m_t)$.

$$
\begin{array}{ccccc}
G = (G^S & \xleftarrow{s_G} & G^C & \xrightarrow{t_G} & G^T) \\
m\downarrow \quad m^S\downarrow & & m^C\downarrow & & m^T\downarrow \\
H = (H^S & \xleftarrow{s_H} & H^C & \xrightarrow{t_H} & H^T)
\end{array}
$$

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 (respectively LHS and RHS) (Ehrig et al., 2007).

Figure 2.6: An example of two triple rules

(a) Triple axiom example for a relation between *UML* and *Java*
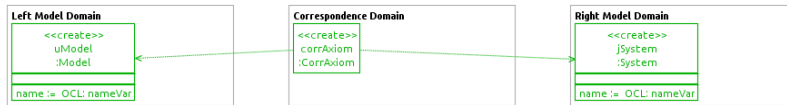
(b) Triple rule example for a relation between *UML* and *Java*
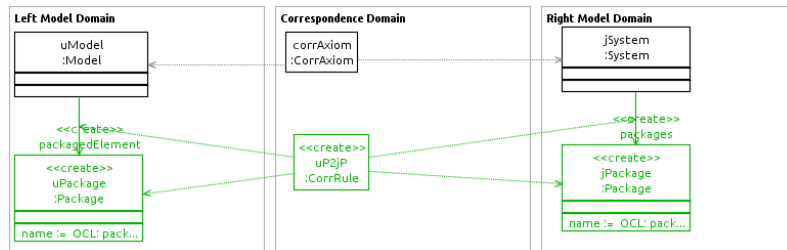
Source: The author

Triple Axiom: A triple axiom is a triple rule $t_a = (s, c, t) : \emptyset \to R$. In order to represent triple graphs, it is common to use attributed graphs together with an easier-to-read

diagram scheme, that comprises 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).

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 metamodels, 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 definitions of model relation, transformation and synchronization as well as triple graph grammars (TGG). The concept of modeling language is pictured as red lines.

Figure 2.7: Illustration of the terms of model relation, transformation and synchronization and triple graph grammars (TGG)
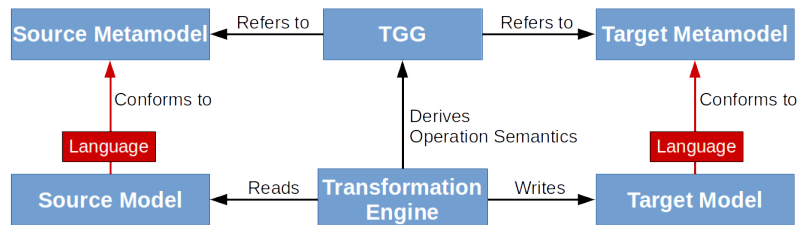
Source: Adapted from Czarnecki and Helsen (2006, p. 623)

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. Heiden-reich et al. present in (2009) and (2010) a Java metamodel using *Ecore*, what influences considerably the development of our work, although it is not directly 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) define the translation between *UML* and *Simulink* using graph grammars. Blouin et al. (2014) report about the synchronization between some specific metamodels of the automotive standards and influence 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 metamodels, 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 TGG's. 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 classi-

---

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

fication 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.

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 METAMODEL 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 metamodels, by describing what each model represents and how they relate to each other. Then the developed metamodel definitions and the relations between them are exposed, as well as the justification of the choices made during the work. Attached to this thesis is the digital version of these metamodels and the definitions of the relations.

4.1 Network of Metamodels

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 select a few typical models plus the relations between them and create a network of metamodels. The Figure 4.1 depicts this network. The bold printed vertices and edges represent respectively the metamodels and the relations, that are treated more deeply in this thesis, namely *Java*, *UMLClassDiagram*, *UMLSequenceDiagram* and *UMLContract*, whereas the other vertices are by virtue of the scope of this thesis some rather more briefly discussed metamodels, namely *UMLUseCaseDiagram*, *RequirementDiagram*, *JavaDoc*, *UnitTest*, *UMLStateMachine*, *ERDiagram*, *FormalSpecification*.

Figure 4.1: A network of metamodels in the Java technological space
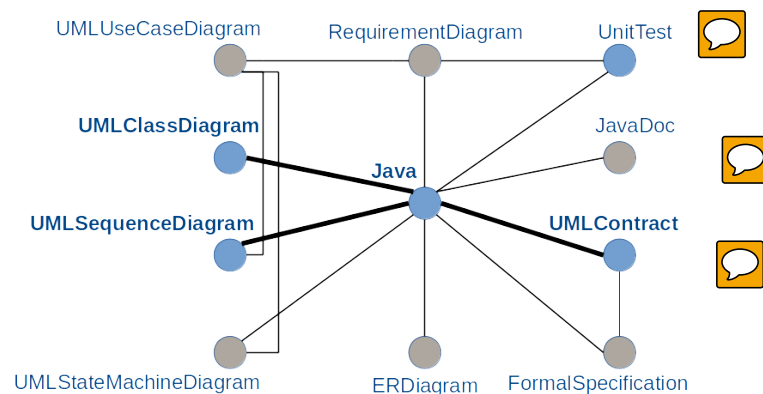
Source: The author

The central element of the network is the *Java* metamodel. 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 metamodel 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 metamodels 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 metamodel is usually used to describe the structure of an object-oriented Java program through a class diagram, representing the definition of its classes, fields 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 metamodel 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 (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 metamodel 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, *UMLUseCasesDiagrams* are used to relate *actors* (basically users of the program) and *use cases* (specifications of behavior), and therefore have a relationship with *RequirementDiagrams*, a very common tool on information system analysis for description of features of a system, and with *UMLSequenceDiagrams* (discussed before) and *UMLStateMachineDiagrams*, two artifacts aiming to describe the behaviors specified by the *use cases* (OMG, 2007, p. 637). *UMLStateMachines* are well known means for modeling functionalities of Java programs, rely on *states* and *transitions* and has relations with the *UMLUseCasesDiagrams*, in the sense that it describes the behavior of a *use case*. For this reason, one may argue, that the behavior expressed by *UMLStateMachines* may also be synchronized with the implementation of methods in the *Java* model.

The linking from *RequirementDiagrams* to *UnitTests* 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 currently 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 Metamodel Definitions

As stated in the last section, the metamodels highlighted in the Figure 4.1 have its metamodels defined here, and the relation between them formalized. The metamodels *UMLClassDiagram*, *UMLSequenceDiagram*, *UMLContract*, and *Java* are explained below with help of a running example.

The modeling language used to write these metamodels is the *EMF Ecore* and the

---

[1] http://www.oracle.com/technetwork/articles/java/index-jsp-135444.html

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 running the transformations. For this reason, the *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 widely available. The *EMF Ecore* language is chosen not only for its extensible documentation and popularity in the community, but also for its ease to use in the *Eclipse IDE*. The metamodels are listed in the sections below.

4.2.1 UML Class Diagram

The metamodel 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, clearly integrates easily with the IDE and seems to be suitable for our needs. Alternatively, we could use the metamodel 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.

The *Model* (on the left top of Figure 4.2) represents the whole model and is the root element of a class diagram, in the sense that every other element is contained by it. *Model* inherits *Package*, and thus may contain *PackageableElements*. Because *Packages* inherit *PackageableElements*, they may be contained by the *Model*, what is the most common case. An example of a *UMLClassDiagram* model (not the metamodel) 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 this scheme other *Packages* as well as a *Classifier* (center on Figure 4.2), because this one inherits *PackageableElement*. The two classifiers handled in the figure are *Class* and *Interface*. The model in the running example contains one *Model* (*Example01*), one *Package* (*main*), three *Classes* (*Person*, *Drive* and *Car*), and one *Interface*, namely *Drivable*.

---

[2]https://projects.eclipse.org/releases/mars
[3]https://netbeans.org
[4]https://eclipse.org/modeling/emf

Figure 4.2: Simplification of the *UMLClassDiagram* metamodel
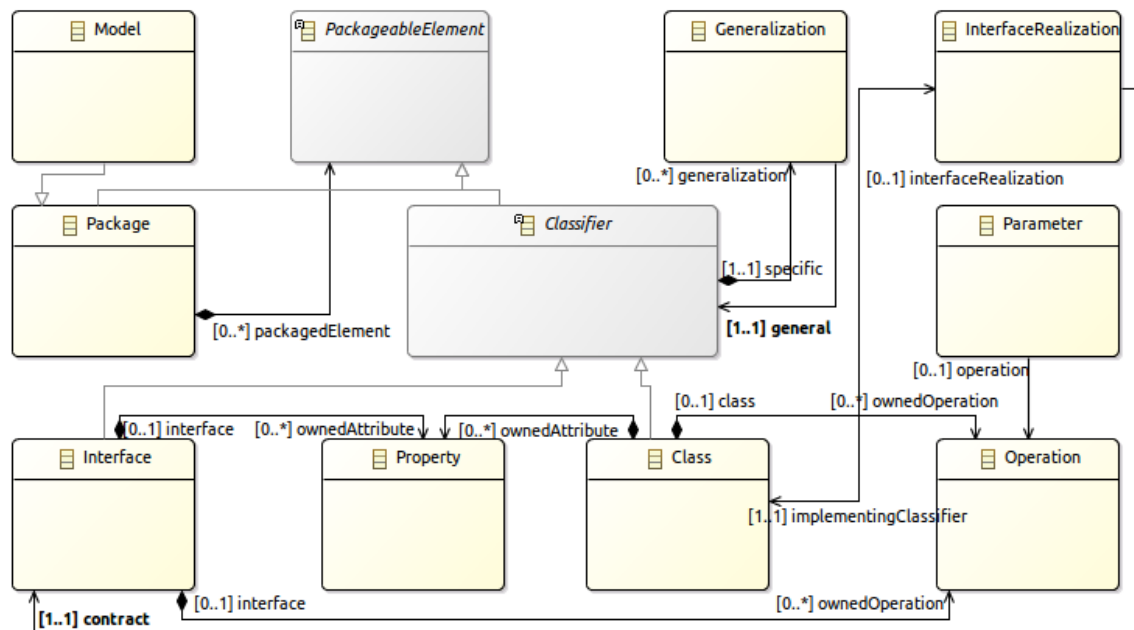
Source: Image created by the author using the *Eclipse IDE*. metamodel from *EMF plug-in*

Figure 4.3: An example of a model *UMLClassDiagram* visualized in two different ways

(a) Abstract Syntax   (b) Concrete Syntax
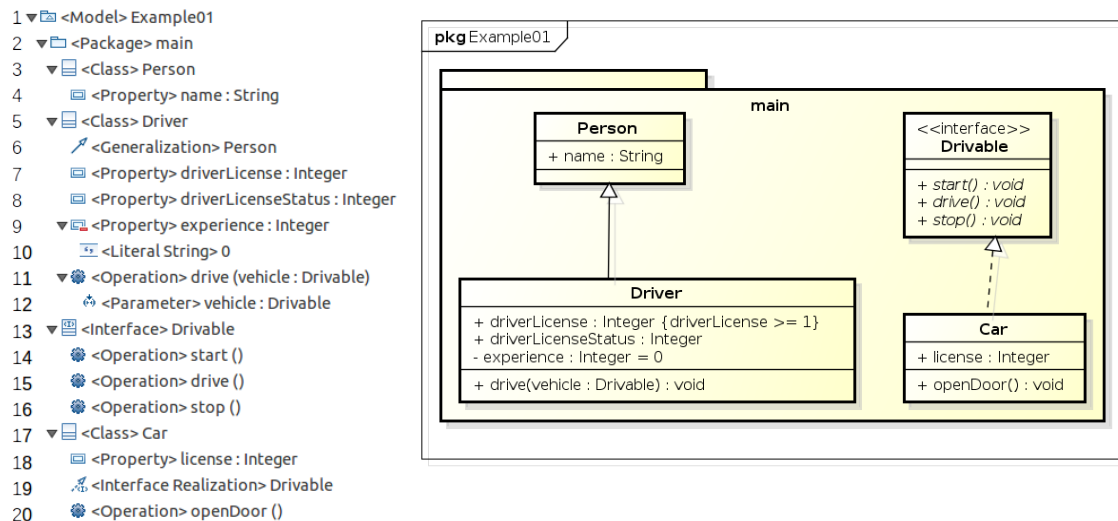
Source: Image created by the author using the *Astah Software*.

      According to the metamodel, a *Classifier* may have a *Generalization* (i.e. inheritance), illustrated with a straight arrow from *Driver* to *Person* in the example on the Figure 4.3b, or an *InterfaceRealization* illustrated with a dashed arrow from *Car* to *Driv-*

able. Moreover, a *Class* is possible to have not only *Properties* (through the aggregation *ownedAttribute*), but also *Operations* (through the *ownedOperation* attribute), which themselves may have *Parameters*. 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 metamodel 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 diagrams are the exchange of messages between objects (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* metamodel

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

One *Lifeline* is connected to a *Class* through the attribute *selector* and is *covered by* one *ActionExecutionSpecification*, 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 *ActionExecutionSpecification* 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 *UMLSequenceDiagrams* requires the reader to
grasp the Figure 4.5 containing one *Interaction* (*Interaction01*), two *Lifelines* (*:Driver*
and *:Drivable*), and four *Messages*.

Figure 4.5: An example of a model *UMLSequenceDiagram* visualized in two different
ways

(a) Abstract Syntax

(b) Concrete Syntax

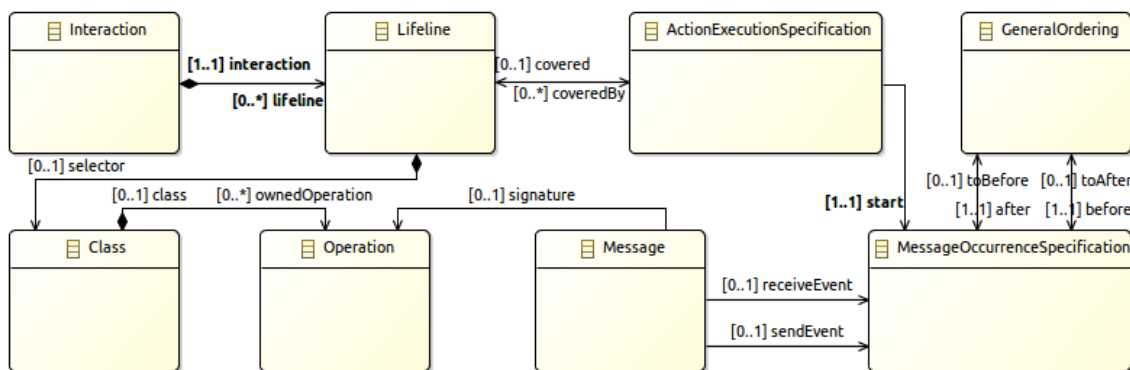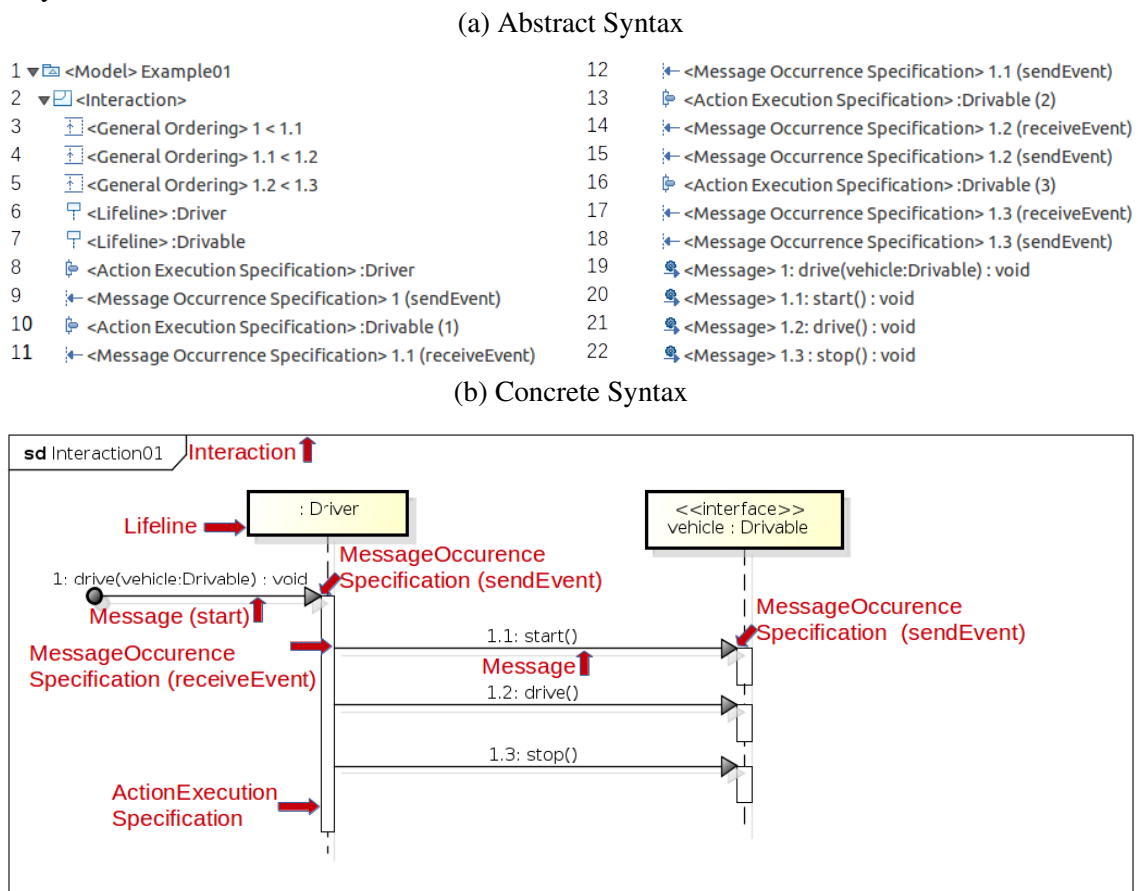Source: Image created by the author using the *Astah Software*.

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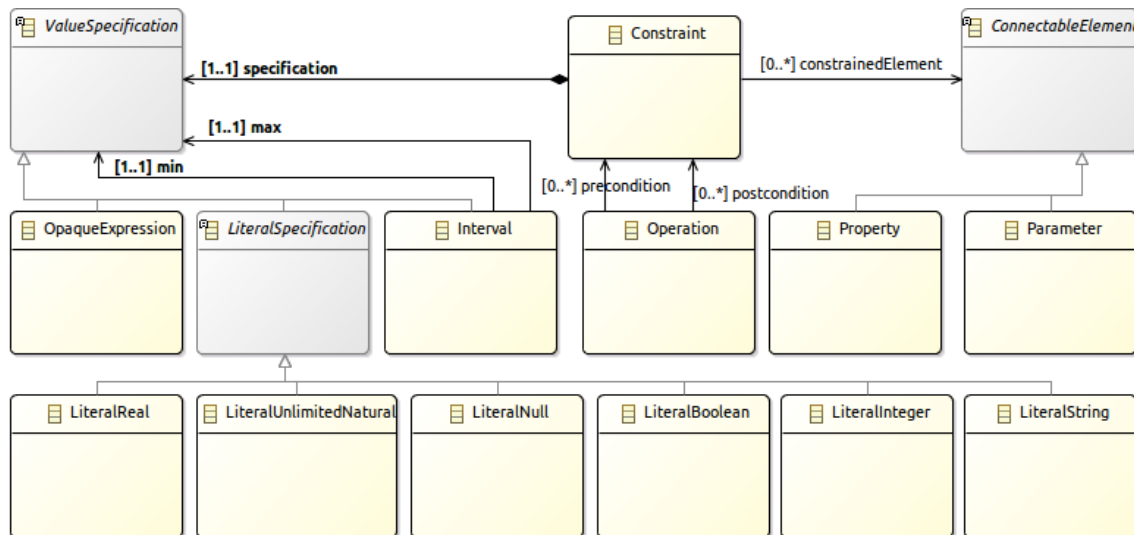
Source: Image created by the author using the *Eclipse IDE*. metamodel from *EMF plug-in*

A *Constraint* may have *constrainedElements*, which in the scope of this thesis are whether *Properties* or *Parameters*. In addition, it has also a *ValueSpecification*, defining the constraint itself, that in this thesis may be a *OpaqueExpression* or an *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 an interval of values, in which the constrained element has to lie. Therefore, it has one *ValueSpecification* for the minimal and one for the maximal value. Here only *Intervals*

with one *LiteralInteger ValueSpecification* for the minimal value will be handled, see an example in the lines 6 to 7 in Figure 4.7, that includes elements from *UMLContract*, namely lines 6 to 7 and 14 to 15 (*Constraint* with *Interval*), and lines 16 to 19 (*Constraint* with *OpaqueExpression*).

Figure 4.7: The expanded version of the model from the picture 4.3 as an example for *UMLContract*. Only abstract syntax is used for this example.

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

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[5] to transform one format into another.

A new metamodel for *Java* is 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 metamodel included the metamodel[6] 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 whole *Java* metamodel

---

[5]https://eclipse.org/modeling/m2t
[6]http://www.eclipse.org/modeling/emf/downloads

created. Elements in blue are *EMF Ecore* abstract elements, whilst elements in yellow are concrete.

Figure 4.8: The Java metamodel created

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

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 Figure 4.9 illustrates concrete types of *Classifiers* in the lines 3, 5 and 48 (*Class*) and 44 (*Interface*).

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 *InterfaceImplementations* (line 54) − that refer to one *Interface* − and 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* picture − also known as parameters − likewise *Statements* (to find on the left bottom of the Figure 4.8). In this imperative perspective of the *Java* metamodel, 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* metamodel, 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 *AssertStatements* 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 Figure 4.8), but again the construction of a more complete metamodel 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 Figure 4.9.

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

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

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

To finish the description of the metamodel, there is the *Annotation*, which is also a kind of *Classifier*. An *AnnotationInstance* is then contained by an *Annotable* element (i.e *Classifier*, *Field*, or *Method*) and may contain *AnnotationInstanceParameters*, which themselves may contain *AnnotationInstanceValues*. The Figure 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* metamodel, 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 *UMLSequenceDiagrams* and *UMLContracts*. 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.

```
1   package main;
2
3   import de.silvawb.utils.*;
4
5   public class Driver extends Person {
6       /*
7        * Fields
8        */
9       @Inv(constraint = "driverLicense >= 1")
10      public Integer driverLicense;
11      public Integer driverLicenseStatus;
12      private Integer experience = 0;
13
14      /*
15       * Methods
16       */
17      public void checkRep(){
18          assert driverLicense >= 1;
19      }
20      public void driveCheckInvConstraint(Drivable vehicle){
21          assert vehicle != null;
22      }
23      public void driveCheckPreConstraint(Drivable vehicle){
24          assert driverLicenseStatus >= 1;
25      }
26      public void driveCheckPosConstraint(Drivable vehicle){
27      }

28
29      @Inv(constraint = "vehicle <> null")
30      @Pre(constraint = "driverLicenseStatus >= 1")
31      @Pos(constraint = "experience > experience@pre")
32      @Interaction(interactionSequence = {
33          "start","drive","stop",
34      })
35      public void drive(Drivable vehicle){
36          checkRep();
37          driveCheckInvConstraint(vehicle);
38          driveCheckPreConstraint(vehicle);
39
40          vehicle.start();
41          vehicle.drive();
42          vehicle.stop();
43
44          checkRep();
45          driveCheckInvConstraint(vehicle);
46          driveCheckPosConstraint(vehicle);
47      }
48  }
```

Source: The Author.

4.3 Metamodel Relations

With all the metamodels 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 because it 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 Chapter 2. Here the most representative relations are presented, as well as some explanations over them. The complete version of the TGG's are attached to this thesis in digital form.

4.3.1 Relations between *UMLClassDiagram* and *Java*

Figure 4.11 shows the triple type graph for the relations between the *UMLClass-Diagram* (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 *UML-ClassDiagram* 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 *UMLClassDiagram* and *Java*
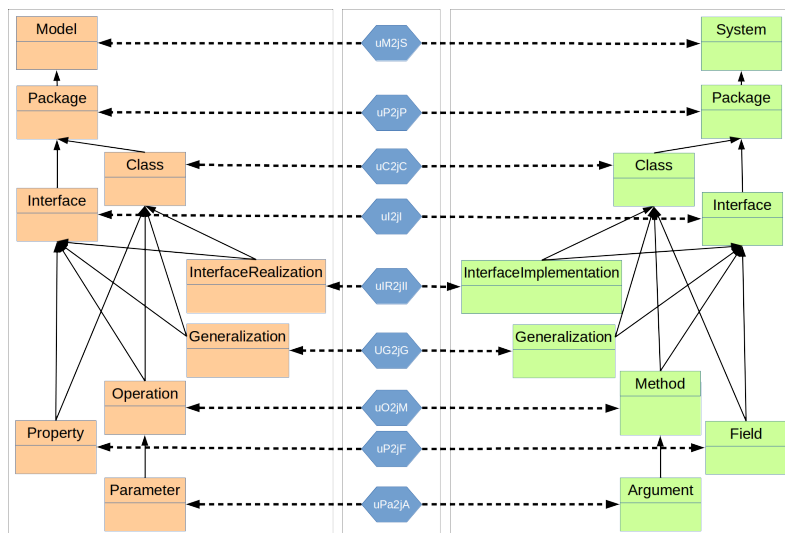
Source: The author

The triple rules describing the relation between *Model* and *System*, as well as *Package* (UML) and *Package* (Java) is available on Figure 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 the Figure 4.12. In essence, this rule formalizes the fact that for every *Class Attribute* (white filling object 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 *name*, *default*, *isStatic*, *isLeaf*, and *visibility* in the *UML Property*, which correspond respectively to *name*, *default*, *isStatic*, *isFinal*, and *visibility* in the *Java Field*.

Figure 4.12: The triple rule *uClassAttribute2jClassField*
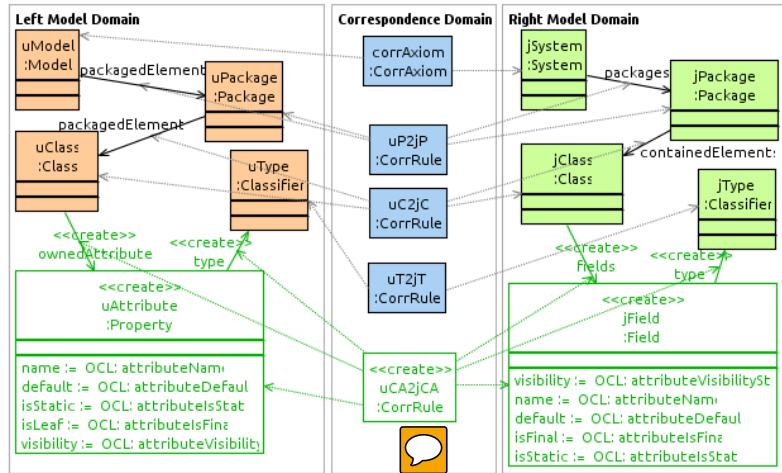
Source: Image created by the author using the *Eclipse IDE*.

Figure 4.13: The triple rule *uInterfaceRealization2jInterfaceImplementation*
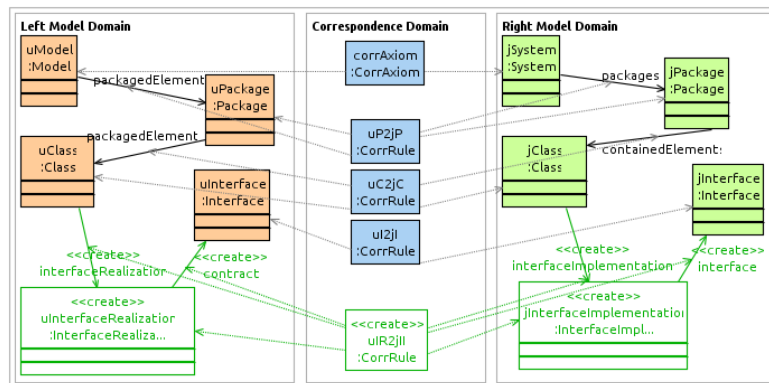
Source: Image created by the author using the *Eclipse IDE*.

Analogously, the figure 4.13 shows the triple rule $L_sL_cL_t \rightarrow R_sR_cR_t$ 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 read as follows: Given a state S_i with a triple graph $L_sL_cL_t$ (LHS) containing all the color-filled elements of Figure 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.

4.3.2 Relations between *UMLSequenceDiagram* and *Java*

Figure 4.14 shows the triple type graph for the relations between the *UMLSequenceDiagram* (left) and the *Java* (right) domains, with the correspondence domain being in the middle. A *Lifeline* and its respective *ActionExecutionSpecification* are related to an *AnnotationInstance* and its respective *AnnotationInstanceParameter*. So, every *Lifeline* regarding to one specific *Method* is related to one *AnnotationInstance* over this method, responsible for representing in the Java model the sequence of method calls modeled by the sequence diagram. An example of such annotation may be seen in the line 32 of the Figure 4.10. The rule responsible for this relation is depicted in Figure 4.15. Briefly, each *Lifeline* (*uLifeline*), together with related elements *uAExecSpecification*, *uMOSSpec* and *uMessage*, is connected to an *AnnotationInstance* (*jAnnInstance*) and a *AnnotationInstanceParameter* (*jAnnInstParam*) belonging to an already existing method (*jMethod*) and referencing an already existing *Annotation* (*InteractionSequenceAnnotation* with name *Interaction*).

Figure 4.14: The triple type graph for *UMLSequenceDiagram* and *Java*
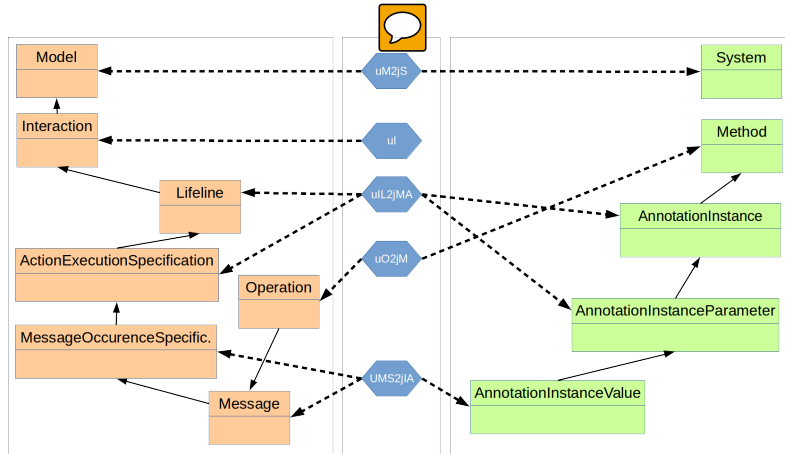
Source: The author

Furthermore, for each *MessageOccurenceSpecification* and its respective *Message* (i.e. a method call in the sequence diagram) exists one *AnnotationInstanceValue* containing the name of the invoked method in the Java model (see line 33 of Figure 4.10). In our implementation this is done by two rules. The first rule is in Figure 4.16 and is responsible for the first *MessageOccurenceSpecification* in the *Lifeline* (e.g. the *Message 1.1: start(): void* in Figure 4.5b). The second rule is in Figure 4.17 and is responsible for

further *MessageOccurenceSpecifications* in the *Lifeline* (e.g. the *Message 1.2: drive():
void* or *1.3: stop(): void* in Figure 4.5b). The relation is split, because in the first rule a
GeneralOrdering (*uOrdering*) links the *start message* (*uStartMessage*) of *uLifeline* to the
first *Message* leaving *uLifeline*, whereas in the second rule, *uOrdering* links two ordinary
Messages leaving *uLifeline*, namely *uMsgBefore* and *uMsgAfter*.

Figure 4.15: The triple rule *uLifeline2jMethodAnnotation*
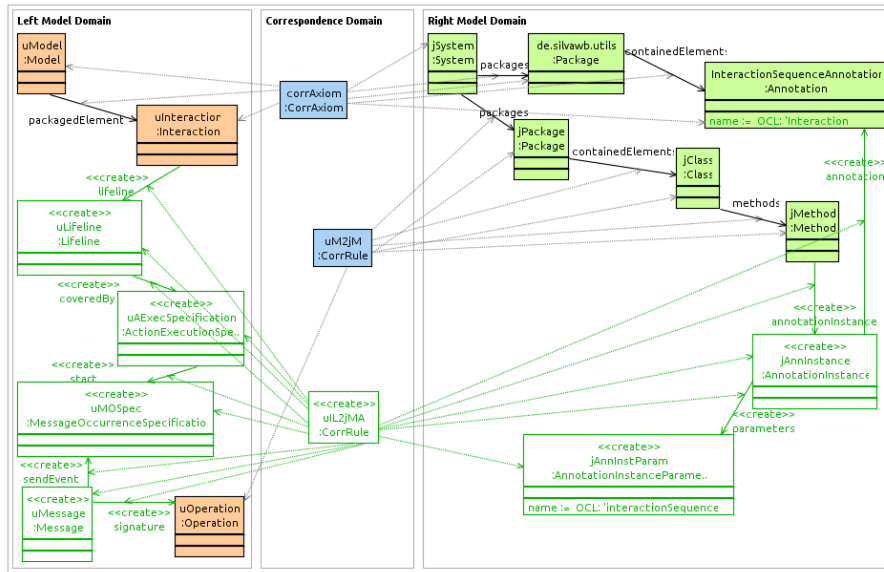
Source: Image created by the author using the *Eclipse IDE*.

The rule *uMessageSequence2jInteractionAnnotation* encodes the fact, that every
Message (*uMOSAfter*) coming right after the initial *Message* (*uStartMessage*) of *uLifeline*
has a correspondent *AnnotationInstanceValue* (*jAnnInstValue*) containing the same *name*
as *uOparationAfter*, and being child of a *AnnotationInstanceParameter* (*jAnnInstParam*)
with name *interactionSequence*.

The occurrence order of the *Messages* is handled by the rule *uMessageSequence2jInte-
ractionAnnotation_2* through the attribute *id* in the *AnnotationInstanceValue* objects. More
specifically, $jAnnInstValueAfter.id = jAnnInstValueBefore.id + 1$. This field is
used in the Java model to indicate the order of the respective objects.

Figure 4.16: The triple rule *uMessageSequence2jInteractionAnnotation*
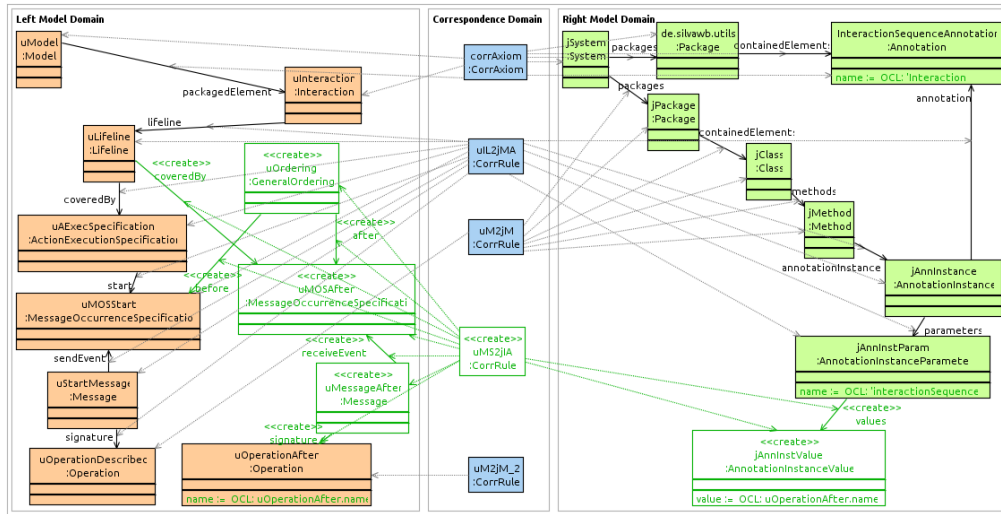
Source: Image created by the author using the *Eclipse IDE*.

Figure 4.17: The triple rule *uMessageSequence2jInteractionAnnotation_2*
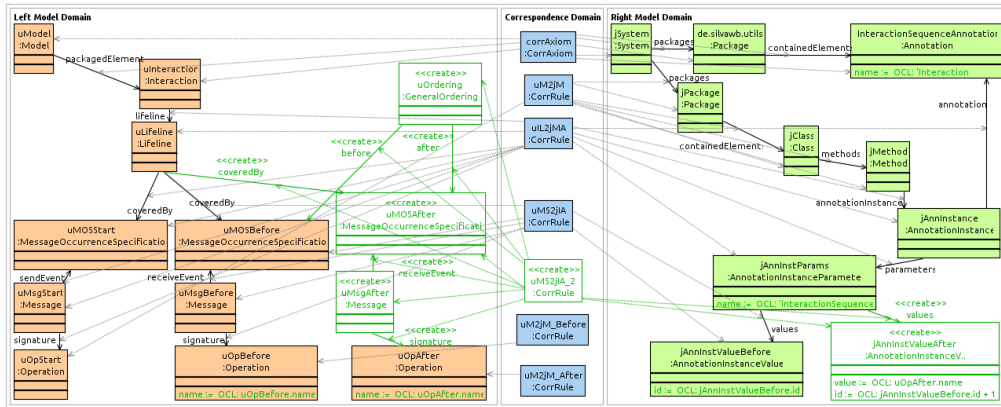
Source: Image created by the author using the *Eclipse IDE*.

4.3.3 Relations between *UMLContract* and *Java*

Figure 4.18 shows the triple type graph for the relations between the *UMLContract* (left) and the *Java* (right) domains, with the correspondence domain being in the middle. Here a *Constraint* may refer to whether a *Property*, an *Operation* or a *Parameter* and may contain (1) an *OpaqueExpression* or (2) an *Interval*. In the first case both the *Constraint*

and *OpaqueExpression* objects are connected to an *AnnotationInstance*, an *AnnotationInstanceParameter*, and an *AnnotationInstanceValue*. Examples of *OpaqueConstraints* are shown in the lines 29 and 31 of the Figure 4.10. The triple rule for the case of a *Opaque-Constraint* over a *Property* is presented in Figure 4.19. This rule, basically, formalizes that every *OpaqueExpression* (*uOpaqueSpecification*) specifying a *Constraint* (*uConstraint*), which itself constraints a *Property* (*uAttribute*), has an *AnnotationInstanceValue* (containing the same specification as *uOpaqueSpecification*), an *AnnotationInstanceParameter*, and an *AnnotationInstance*, which itself belongs to the *Field jField* related to the respective constrained *uAttribute* and references the already existing *Annotation Inv* with name *Inv*.

Figure 4.18: The triple type graph for *UMLContract* and *Java*
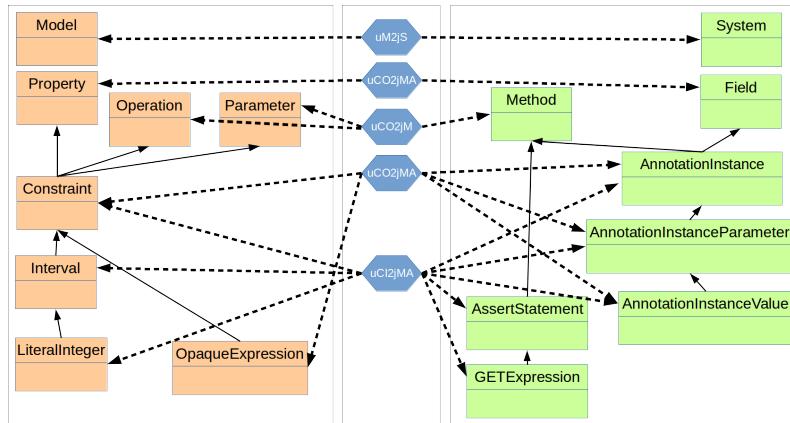
Source: The author

The triple rule for the case of an *IntervalConstraint* over a *Parameter* is presented in Figure 4.20. The difference to the previous figure, is that now on the source domain, an *Interval* (*uInterval*) is used to specify *uConstraint*, which is a *precondition* to *uOperation* and has a *LiteralInteger* (*uLiteralInt*) as minimal value. Moreover, the creation of these elements create not only *Annotation* elements in the Java model, instead it creates also *AssertStatements* in the form of *GETExpressions* belonging to a previously created *check Method*.

In our implementation, every *Method* of the Java model has three *check Methods*, namely *checkPreConstraint*, *checkPosConstraint*, and *checkInvConstraint*. By being so, the rule *uOPPreInt2jMAPreAssert* is able to transform *UML interval pre-conditions* into *Java GETExpression*, by using the attribute *uConstrained.name* as left side and the

uLiteralInt.value as right side of the target expression *jGETExpression*.

Figure 4.19: The triple rule *uCInv2jCInv*
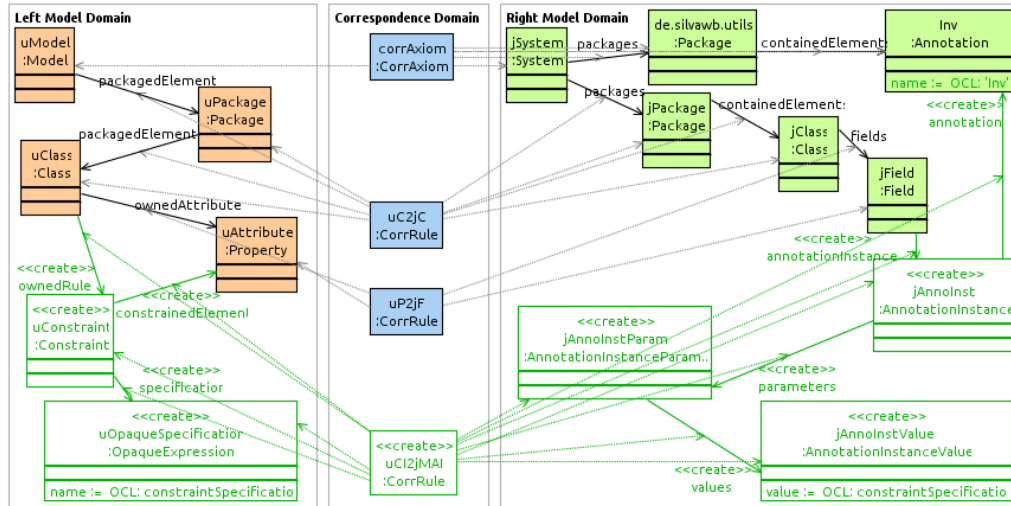
Source: Image created by the author using the *Eclipse IDE*.

Figure 4.20: The triple rule *uOPPreInt2jMAPreAssert*
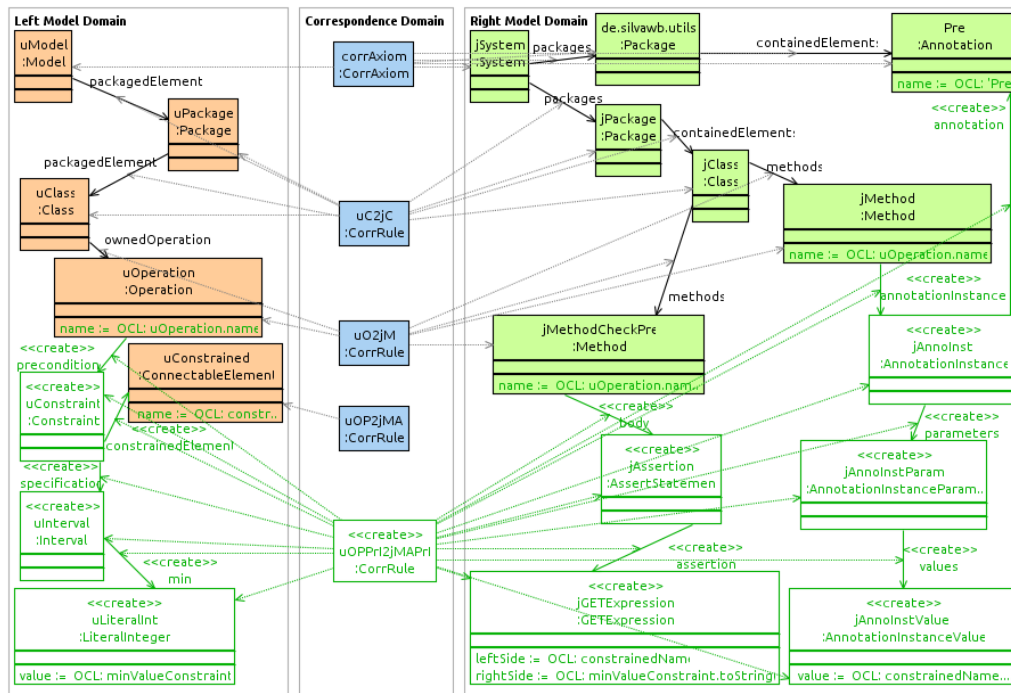
Source: Image created by the author using the *Eclipse IDE*.

4.4 Evaluation

The three TGG's developed in this thesis have been tested separately through some example scenarios of forward transformation. This means, synchronization is not applied in this evaluation, instead only forward transformations using the three developed TGG's (*umlClassDiagram2java*, *umlSequenceDiagram2java*, *umlContracts2java*) are executed for some example cases, some of them are shown in this section. The Figure 4.21 shows the result of a forward transformation from *UMLClassDiagram* to *Java* executed by the *MoTE transformation tool* based on the *umlClassDiagram2java* TGG. This TGG includes the rules presented above (*uClassAttribute2jClassField*, *uInterfaceRealization2jInterfaceImplementation*) plus the following rules, all comprised in digital form, attached to this thesis: *uModel2jSystem* (Axiom), *uPackage2jPackage*, *uClass2jClass*, *uClassOperation2jClassMethod*, *uInterface2jInterface*, *uInterfaceAttribute2jInterfaceField*, *uInterfaceOperation2jInterfaceMethod* and *uCGeneralization- 2jCGeneralization*. In the example, *uModel2jSystem* is applied in line 1; *uPackage2jPackage* in line 2; *uInterface2jInterface* in line 3; *uInterfaceOperation2jInterfaceMethod* in line 4, 5 and 6; *uClass2jClass* in line 7; *uClassAttribute2jClassField* in line 8; *uInterfaceRealization2jInterface- Implementation* in line 9; and *uClassOperation2jClassMethod* in line 10.

Figure 4.21: Example of forward synchronization from a *UMLClassDiagram* model (left) and a *Java* (right) model
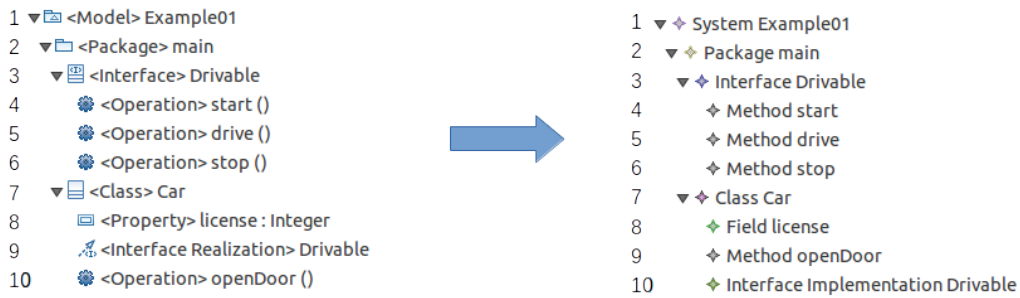
Source: Image created by the author using the *Eclipse IDE*.

The problem with the *umlClassDiagram2java* TGG is that it does not encode the fact that the implementing class *Car* should contain the 3 methods defined by the interface *Drivable* in Java. A possible solution would be the creation of a new rule including an object (*uMethod*) contained by (*uInterface*) in L_s (LHS) 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 cre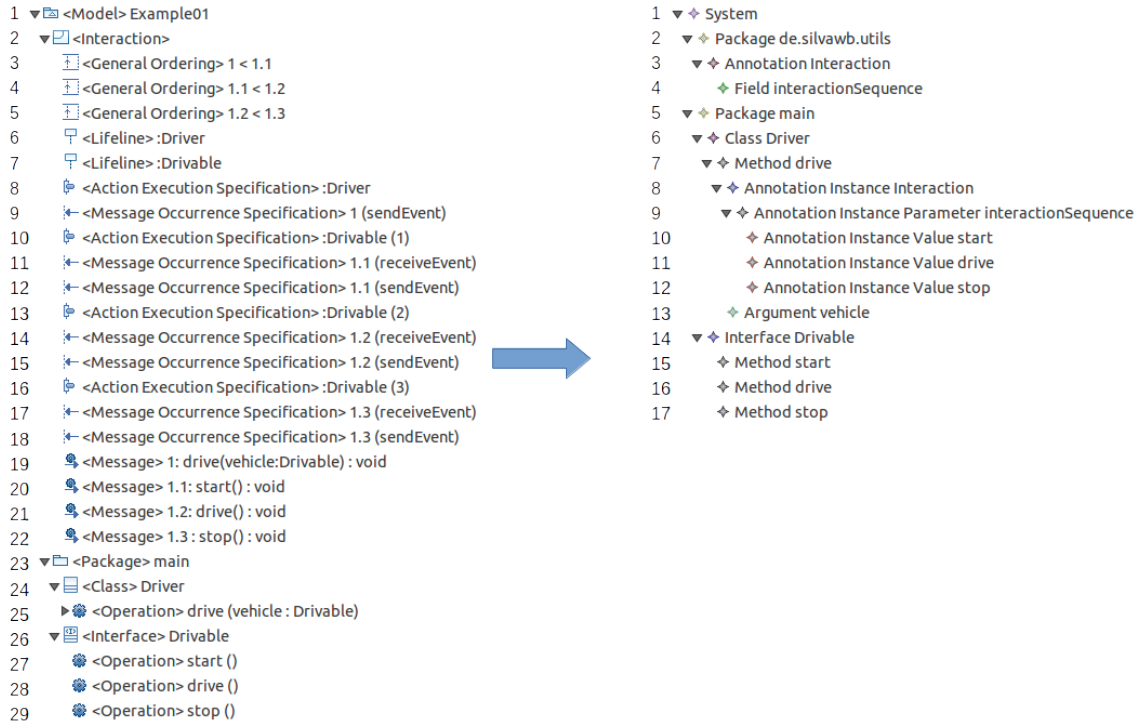ation of such rule would entail a critical pair, given that two different rules have the object *uInterfaceRealization* in the R_s. A definite solution is not know by us, therefore this synchronization task ends up being left to the developer. Moreover, *umlClassDiagram2java* is not complete, as it (1) does not comprise a rule for the *Parameter* element of the UML metamodel − By being so, a *Method* would not have its *Parameters* synchronized − and (2) nested *Packages* are not supported.

Figure 4.22 shows the result of a forward transformation from *UMLSequenceDiagram* to *Java* based on the *umlSequenceDiagram2java* TGG, which comprises the rules shown above (*uLifeline2jMethodAnnotation*, *uMessageSequence2jInteractionAnnotation*, *uMessageSequen- ce2jInteractionAnnotation_2*) plus an axiom *umlInteraction2javaAxiom* responsible for linking an *Interaction* object in UML to an *Annotation* object in Java. In the example, the axiom is used in the lines 1 and 2 of the left and 1, 2, 3 and 4 of the right model; *uLifeline2jMethodAnnotation* is applied for the lines 6, 8, 9 and 19 of the left model, and 8 and 9 of the right model; *uMessageSequence2jInteractionAnnotation* is applied for the lines 3, 11, 20 of the left, and 10 of the right model; finally, *uMessage-Sequence2jInteractionAnnotation_2* is applied once for the lines 4, 14 and 21 of the left, and 11 of the right model; and once more for the lines 5, 17 and 22 of the left, and 12 of the right model. The rest of the elements, like the *Package main*, the *Class Driver* and the *Operations* are handled by extra rules based on the *umlClassDiagram2java* TGG. This is made, in order to build this richer scenario. Observe that, in a real world synchronization situation these extra rules would not be necessary, once the respective elements would have been created by the *umlClassDiagram2java*.

This TGG makes several assumptions that narrow the possibilities of sequence diagrams. In general, only simple synchronous messages are supported, so create, delete or reply messages are not supported; actor, entity or other types of lifelines cannot occur; furthermore, it is supposed, that further features like *InteractionUse* or *CombinedFragment* (see (OMG, 2015, p. 621 and p. 630)) also do not happen in the sequence diagrams. In a broader view, one could say, that the rule is too simple, once it captures basically only the order of the message occurrences and transforms it into annotations in the Java model (and vice-versa). Indeed, it exposes the problem of TGG's to deal with non MOF-appropriate models, since that more powerful tools to handle method calls inside a Java method would be desired to build a more comprehensive transformation from sequence

Figure 4.22: Example of forward synchronization from a *UMLSequenceDiagram* model (left) and a *Java* (right) model

```
1  ▼🗂 <Model> Example01                                    1  ▼✦ System
2    ▼🖾 <Interaction>                                      2    ▼✦ Package de.silvawb.utils
3        ⊡ <General Ordering> 1 < 1.1                       3      ▼✦ Annotation Interaction
4        ⊡ <General Ordering> 1.1 < 1.2                     4          ✦ Field interactionSequence
5        ⊡ <General Ordering> 1.2 < 1.3                     5      ▼✦ Package main
6        🖵 <Lifeline> :Driver                              6        ▼✦ Class Driver
7        🖵 <Lifeline> :Drivable                            7          ▼✦ Method drive
8        🔓 <Action Execution Specification> :Driver        8            ▼✦ Annotation Instance Interaction
9        ⊩ <Message Occurrence Specification> 1 (sendEvent) 9              ▼✦ Annotation Instance Parameter interactionSequence
10       🔓 <Action Execution Specification> :Drivable (1)  10                 ✦ Annotation Instance Value start
11       ⊩ <Message Occurrence Specification> 1.1 (receiveEvent) 11             ✦ Annotation Instance Value drive
12       ⊩ <Message Occurrence Specification> 1.1 (sendEvent) 12                 ✦ Annotation Instance Value stop
13       🔓 <Action Execution Specification> :Drivable (2)  13               ✦ Argument vehicle
14       ⊩ <Message Occurrence Specification> 1.2 (receiveEvent) 14      ▼✦ Interface Drivable
15       ⊩ <Message Occurrence Specification> 1.2 (sendEvent) 15          ✦ Method start
16       🔓 <Action Execution Specification> :Drivable (3)  16          ✦ Method drive
17       ⊩ <Message Occurrence Specification> 1.3 (receiveEvent) 17       ✦ Method stop
18       ⊩ <Message Occurrence Specification> 1.3 (sendEvent)
19       🔩 <Message> 1: drive(vehicle:Drivable) : void
20       🔩 <Message> 1.1: start() : void
21       🔩 <Message> 1.2: drive() : void
22       🔩 <Message> 1.3 : stop() : void
23   ▼🗂 <Package> main
24     ▼🖳 <Class> Driver
25       ▶⚙ <Operation> drive (vehicle : Drivable)
26     ▼🖳 <Interface> Drivable
27       ⚙ <Operation> start ()
28       ⚙ <Operation> drive ()
29       ⚙ <Operation> stop ()
```
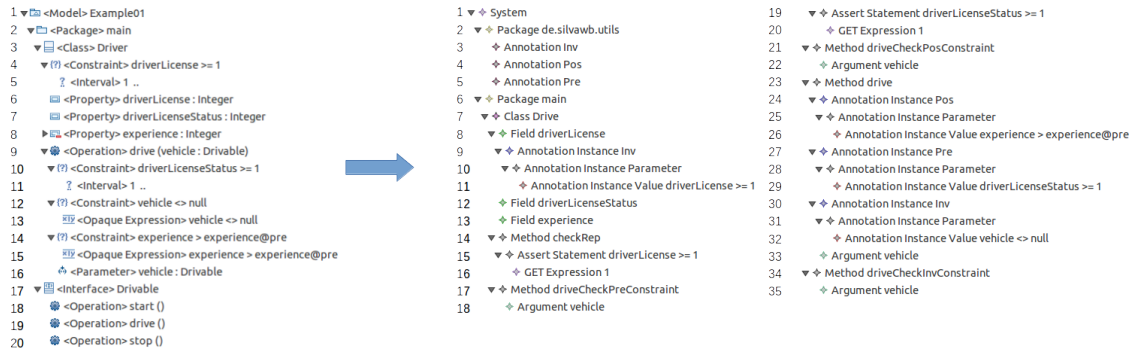
Source: Image created by the author using the *Eclipse IDE*.

diagrams to Java, for instance. Nevertheless, for the simplified scenario of this thesis the implemented TGG seems to work properly. A transformation rule that encodes the order of elements in the left and right domain were not known by us to exist in the current literature, so in this sense this feature is novel and contributive.

Figure 4.23 shows the result of a forward transformation from *UMLContract* to *Java* using the *umlContract2java* TGG. Beyond the two rules exposed in the previous section (*uCInv2- jCInv*, *uOPPreInt2jMAPreAssert*), this TGG comprises basically one Axiom; more three rules that encode the relation between *UML Opaque Constraints* and *Java Annotations* (*uOPPre2jM- APre*, *uOPPos2jMAPos*, *uOInv2jMAInv*, respectively for *pre*, *post* and *invariant* constraints); and more two rules that encode the relation between *UML Interval Constraints* and *Java Annotations* plus *Assertions* (*uOPPosInt2jMAPosAssert*, *uCInvInt2jCInvAssert*, respectively for *post* and *invariant constraints*). In the example the axiom is used on the line 1 of the left, and 1 to 5 of the right model and creates constraint *Annotations*; *uCInvInt2jCInvAssert* is used on the lines 4 and 5 of the left, and 9 to 11, as well as 15 to 16 of the right model; *uOPPreInt2jMAPreAssert* is used on the lines 10 and 11 of the left, and 19 to 20, as well as 27 to 29 of the right model; *uOInv2jMAInv* is used

on the lines 12 and 13 of the left, and 30 to 32 of the right model; finally, *uOPPos2jMAPos* is used on the lines 14 and 15 of the left, and 24 to 26 of the right model. The rest of the elements are transformed by other rules, just like in the TGG *umlClassDiagram2java*.

Figure 4.23: Example of forward synchronization from a *UMLContract* model (left) and a *Java* (right) model

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

This TGG does not seem to present big problems, except that it is not complete, since neither *invariant interval constraints* for methods, nor non-integer *interval constraints* have been developed. Moreover, the implemented rules for *interval constraints* support only *great-or-equal-than expression*. On the other hand, the extension of the current TGG version in direction to completeness seems to be possible, so is the extension in the direction of building unit test cases based on the contracts.

5 SYNCHRONIZATION OF MODELS IN THE JAVA TECHNOLOGICAL SPACE

In the last chapter the relations between some metamodels 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 section (namely 4.3.1, 4.3.2 and 4.3.3) and represents a different edge of the network of models built (see Figure 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 metamodels according to Getir et al. (2015), 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
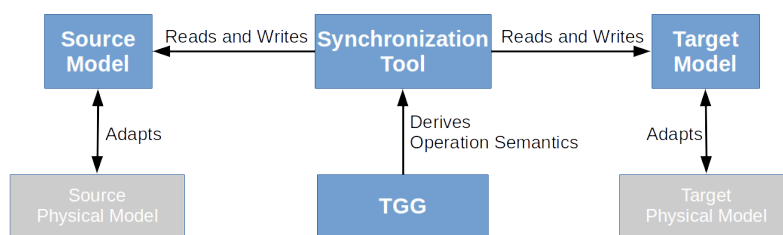
Source: Adapted from (Giese et al., 2010b).

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, since *MoTE* is used to execute each transformation implemented in Chapter 4.3 (see discussion in Chapter 4.4).

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 Figure 4.9) to physical models (presented so far in form of concrete syntax, e.g Figure 4.10). Such situation is out of the scope of this thesis.

As stated before, there exists one instance of the synchronization scheme in Figure 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. The goal of this chapter is not the definition of the synchronization algorithm or the implementation of a tool such as *MoTE*, but the definition of an algorithm for the synchronization of a whole network of models that uses an already proposed synchronization method.

5.1 Synchronization of a 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 $sync : S \times S \times \Delta_S \times T \to T \times \Delta_T$, 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 modifications in s_1 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.

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.

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

Listing 5.1: neysync imperative algorithm definition

Firstly, the initial network is updated with the new vertex s_1, then every neighbor n_i of s is synchronized according to the modifications d_s. If it causes modifications on n_i, then the network is recursively further synchronized starting from n_i. Extra assumptions were made by this algorithm:

- **Supposition 3:** The input network of models (V_0, E_0) is finite.
- **Supposition 4:** The input network of models (V_0, E_0) has no cycles.

- **Supposition 5:** s_1 *is synchronized to* $t_0 \Leftrightarrow sync(s_0, s_1, \delta_s, t_0) = t_0$. To put in words, if s_1 is synchronized with respect to t_0, then $sync$ returns the unmodified t_0 (and vice-versa).

  This implies the following properties:

- **Theorem 1:** The algorithm always terminates. Because supposition 2 is made, it suffices only one call to $sync$ on line 4, representing the synchronization of s_0 and $ni -$ i.e the edge (s_0, n_i)), 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). Thus the termination of the algorithm is guaranteed.

- **Theorem 2:** The time complexity of $netsync((V_0, E_0), s_0, s_1, d_s)$ is in the worst case $O(\Delta(V_0, E_0)|E_0|)$, where a $sync$ call is taken as elementary operation.

- **Theorem 3:** The algorithm is deterministic. This means, that the definition of the result of $netsync((V_0, E_0), s_0, s_1, \delta_v)) = (V_1, E_1)$ depends only on the inputs (V_0, E_0), s_0, s_1, δ_v. As $E_1 = E_0$, then E_1 is trivially deterministic. V_1 can also be determined, once that (1) for $v_0 \in V_1$ holds that $v_0 = s_1$ (i.e. the first modification of the network, line 2 of the algorithm); (2) $\forall v_i \in V_1, i > 0 : v_i = sync(s_j, s_{jnew}, \delta_j, s_i)$ (further modifications of the network); and (3) $sync$ is deterministic. The affirmation (2) is true, because v_i is whether "not modified" $-$ condition on line 6 never holds true for any call $sync(s_i, s_{inew}, d_s, v_i) -$ or v_i is assigned the deterministic value $sync(s_j, s_{jnew}, \delta_j, s_i)$ on the line $2 -$ note that, this assignment is executed only once in the whole execution of the synchronization. Therefore V_1 and E_1 can be defined deterministically and thus $netsync$ is deterministic.

6 DISCUSSION

In the previous chapters the development of this work is shown, so in this chapter this development is discussed in regard to what was achieved, what are the handicaps, and what is required to use the suggested approach. Firstly, the creation of a Java metamodel usable for writing TGG seems to be a promising result, despite it has some simplifications and is not complete. For instance, this metamodel does not deal properly with behavioral aspects, like source-code statements. The careful reader should notice the possible difficulties to write triple rules that deal with control and conditional structures (i.e *while*, *if*).

In regard to the relations, in general, they should be rather more extensive for the practical use, however they might serve as basis for further developments. For the relation *umlClassDiagram2java* it can be said, that it is simple, as it does not span all elements of the metamodels − e.g. *Parameters* are not covered − but generally speaking it works properly and conveys most of the relationship ideas between the two metamodels. The relation *umlSequenceDiagram2java* innovates, by transmitting the order of messages in a sequence diagram to the order of annotations in the Java source-code. On the other hand, it does not encompass the whole set of possibilities offered by sequence diagrams, expecting then some simplifications already explained in Chapter 4.3.2. Lastly, the relation *umlContract2java* is also not complete, but might be helpful for enhancing the network of metamodels towards the use of unit tests, for example.

The proposed $netsync$ algorithm is clearly an initial suggestion, yet it works properly for our simple showcase. An important supposition of $netsync$ is the absence of cycles in the input network of models, what hinders its use in practice, including the use for the complete network presented in the Figure 4.1. In fact, the presence of cycles requires a whole new $netsync$ algorithm, able to deal with bigger problems, like synchronization of two models with modifications in both sides. It is notable here also the need for more theoretical works able to deal with the problem of synchronizing a network. One alternative approach for this problem could be the application of *Graph Diagram Grammars* (Trollmann and Albayrak, 2015), that consists basically of a generalization of TGG for multiple models, in which one single triple rule may describe the relation between more than only two metamodels

In general, the approach proposed in this thesis does not solve the whole problem of synchronization of models in the Java technological space, neither does it synchro-

nization of a network in the practice. Instead, it can only treat transformations of pair of models (source and target) separately. For this case, our approach requires that (1) the input models be in conformity to our simplified metamodels, (2) they be in the *EMF Ecore* format, and (3) the *Eclipse IDE* be installed with the respective *MoTE plug-ins* and the plug-ins generated by *MoTE* representing our TGG's. Therewith, it is expected that the input models be transformed analogously to what was showcased in Chapter 4.4.

Some difficulties were found along this work, but they did not obstruct the success of the final result. The first one was the lack of openly available metamodels in the literature or from the vendors. For instance, *Oracle* does not publicize any standard metamodel for Java, nor are they easy 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 incompatible with the employed tool, what also hinders the progress of the work. The result of this thesis may be a partial solution for that, considering that metamodels of other technological spaces still lack a similar work. Another complication is 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 an 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 TGG 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 might 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 metamodels, expending the created network of metamodels, or by creating metamodels 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 creation of an *Eclipse* plug-in that implements the suggested synchronization algorithm $netsync$, able to synchronize a network of models, and comprise the whole synchronization scheme (see Figure 5.1) for each edge of the network could be

a next work. *MoTE* could be helpful for such task, since it already generates a plug-in for the execution of model transformation based on the input TGG.

And lately, a relatively big issue is the use of TGG's for non-MOF-compliant metamodels − or metamodels 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, rather than the MOF. A clear example is source-code, which may 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 a future challenge.

7 CONCLUSION

A network of metamodels of the Java technological space was developed in the Chapter 4 of this thesis, comprising a set of common metamodels used in Java software and the respective relations between them. These relations were showcased through a practical example, where a scenario with models were built, forward transformations were executed, and their results were evaluated. This evaluation showed their weaknesses and strengths, by describing the issues that we tried to deal with, and the issues that are still unsolved. The Chapter 5 endeavors to propose an algorithm for synchronization of a whole network of models based on already existing synchronization methods between two models. The Chapter 6 closes the work with a critical discussion about the contributions of this thesis and points to the need for future works.

In the first phase a discussion about some metamodels of the Java technological space is presented, followed by the definition of some of them, which are dealt more carefully in this work, namely *UML Class Diagram*, *UML Sequence Diagram*, *UML Contract*, and *Java*. In the second phase the three relations between these four metamodel were coded (*umlClassDiagram2java*, *umlSequenceDiagram2java*, *umlContract2java*). In regard to these both steps the main legacy are (1) the definition of a Java metamodel suitable for the use in synchronization with TGG, and (2) the novel relations between some elements of some metamodels, like the rules between *UML contracts* and *Java assertions*, that encode the relation between *pre* and *post-conditions* of the former and *assertions* and *annotations* of the latter.

The third phase serves to demonstrate briefly the application of transformation of these relations, in order to clarify their work, evaluate them, and show the contributions they bring. This is done through a representative example and the use of the *MoTE* transformation/synchronization tool. Therewith, we demonstrate that the implemented relation rules work properly for some cases, and that it leads towards the synchronization of such relations. The synchronization of a whole network of models is then handled broadly after the showcase of the transformations, where we suggest an algorithm for synchronization of a network of model, analyze some of its theoretical properties and conclude, that for the suppositions made, it always terminates and it is deterministic.

Despite the results of this thesis are not complete and we do not believe that the implemented artifacts are ripe enough to be put in practice, the total product including novel ideas, a summary of the state-of-art, and the discussion of difficulties and problems

found during the work are valuable and contributing. The success of this thesis brings the contribution towards the definition of a network of interconnected metamodels useful to both research and industry community. Therefore the availability of such a network might finally allow the extensive use of Model-driven Engineering in practice − helping bridging the gap between system abstractions and their concrete form − and foster the further development of more sophisticated model synchronization methods.

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, Tim Sanwald, and Matthias Tichy. Cowolf–a generic framework for multi-view co-evolution and evaluation of models. In *Theory and Practice of Model Transformations*, pages 34–40. Springer, 2015.

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. Technical report, Fakultät Informatik, Technological University of Dresden, Germany, 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.

I. Kurtev, J. Bézivin, and M. Akşit. Technological spaces: An initial appraisal. In *International Conference on Cooperative Information Systems (CoopIS), DOA'2002 Federated Conferences, Industrial Track, Irvine, USA*, pages 1–6, October 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*, pages 45–48, 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.

Frank Trollmann and Sahin Albayrak. Extending model to model transformation results from triple graph grammars to multiple models. In *Theory and Practice of Model Transformations*, pages 214–229. Springer, 2015.

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.