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

Berlin
March 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

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 basically consists in keeping all the models of a software application 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 properly forwarded to all the models regarding this software. For large software applications it is clearly inviable to perform such synchronization procedure manually, therefore, it is desired the creation of automatic methods able to synchronize the software's models. We do not explore this problem for any kind of software, instead we limit our domain to the Java technological space, so that the scope of this thesis still remains feasible. This thesis thus proceeds by (1) identifying and formally defining some models of the Java technological space, (2) identifying and formalizing some relations between them, creating a network of metamodels supposed to be kept synchronized, and showing through a representative showcase how these relations work, and finally (3) discussing the synchronization of this network of metamodels. 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 ist in der letzten Jahren erheblich gewachsen. Als es zunimmt, nimmt auch die Relevanz eigener auf den Bereich bezogenen Probleme zu. Eines davon ist das Modell-Synchronisationsproblem, das grundsätzlich in 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 an alle Modelle weitergeleitet werden. Für größere Anwendungen ist es offensichtlich unmöglich, solches Synchronisationsverfahren manuell zu unternehmen. Daher ist die Schaffung von automatischen Verfahren, die in der Lage sind, Synchronisation durchzuführen, wünschenswert.

Wir untersuchen dieses Problem nicht für jede Art von Software, stattdessen beschränken wir unsere Domäne auf den technologischen Raum von Java derart, dass der Umfang dieser Arbeit immer noch realisierbar bleibt. Die vorliegende Arbeit geht in folgender Weise vor: (1) Einige Modelle des Java technologischen Raums werden identifiziert und definiert; (2) Einige Relationen zwischen denen werden identifiziert und formalisiert, sodass ein Netzwerk von Metamodellen aufgebaut wird, das synchronisiert aufbewahrt werden soll, und ein repräsentatives Beispiel gezeigt wird; und letztendlich (3) wird die Synchronisation dieses Netzwerkes von Modellen diskutiert. 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

Figure 2.1 The summary of the definitions of system, model, metamodel, metametamodel and modeling language 14
Figure 2.2 An example of a network of models very similar to the one developed in this work 16
Figure 2.3 The morphism $m : G \to H$ is a triple graph $m = (m^S, m^C, m^T)$. 16
Figure 2.4 An example of two triple rules 17
Figure 2.5 Illustration of the terms of model relation, transformation and synchronization and triple graph grammars (TGG) 18

Figure 4.1 A network of metamodels in the Java technological space 22
Figure 4.2 Simplification of the *UMLClassDiagram* metamodel 25
Figure 4.3 An example of a model *UMLClassDiagram* visualized in two different ways 25
Figure 4.4 Simplification of the *UMLSequenceDiagram* metamodel 26
Figure 4.5 An example of a model *UMLSequenceDiagram* visualized in two different ways 27
Figure 4.6 Simplification of the *UMLContract* metamodel 28
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. 28
Figure 4.8 The Java metamodel created 30
Figure 4.9 The equivalent *Java* version of the *UML* models from the Figures 4.3, 4.7 and 4.5 in abstract syntax. 31
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. 31
Figure 4.11 The triple type graph for *UMLClassDiagram* and *Java* 33
Figure 4.12 The triple rule *uClassAttribute2jClassField* 34
Figure 4.13 The triple rule *uInterfaceRealization2jInterfaceImplementation* 34
Figure 4.14 The triple type graph for *UMLSequenceDiagram* and *Java* 36
Figure 4.15 The triple rule *uLifeline2jMethodAnnotation* 36
Figure 4.16 The triple rule *uMessageSequence2jInteractionAnnotation* 37
Figure 4.17 The triple rule *uMessageSequence2jInteractionAnnotation_2* 37
Figure 4.18 The triple type graph for *UMLContract* and *Java* 38
Figure 4.19 The triple rule *uClassInvariant2jClassInvariant* 39
Figure 4.20 The triple rule *uOperationPreInterval2jMethodPreAssert* 39
Figure 4.21 Example of forward synchronization from a *UMLClassDiagram* model (left) and a *Java* model (right) 40
Figure 4.22 Example of forward synchronization from a *UMLSequenceDiagram* model (left) and a *Java* model (right) 42
Figure 4.23 Example of forward synchronization from a *UMLContract* model (left) and a *Java* model (right) 43

Figure 5.1 Synchronization scheme of the *MoTE* tool 44

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 **8**
1.1 Background **8**
1.2 Motivation **9**
1.3 Objective **10**
1.4 Methodology **10**
2 FOUNDATIONS **12**
2.1 Models and MDE **12**
2.2 Model Synchronization **14**
2.3 Triple Graph Grammar **16**
3 STATE OF THE ART **19**
4 METAMODEL RELATIONS IN THE JAVA TECHNOLOGICAL SPACE **21**
4.1 Network of Metamodels **21**
4.2 Metamodel Definitions **23**
4.2.1 UML Class Diagram 24
4.2.2 UML Sequence Diagram 26
4.2.3 UML Contract 28
4.2.4 Java 29
4.3 Metamodel Relations **32**
4.3.1 Relations between *UMLClassDiagram* and *Java* 33
4.3.2 Relations between *UMLSequenceDiagram* and *Java* 35
4.3.3 Relations between *UMLContract* and *Java* 38
4.4 Evaluation **40**
5 SYNCHRONIZATION OF MODELS IN THE JAVA TECHNOLOGICAL SPACE **44**
5.1 Synchronization of a Network of Models **45**
6 DISCUSSION **48**
7 CONCLUSION **51**
REFERENCES **53**

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 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 large number of developers working in a common software artifact have required the use of models in software engineering, what is referred to as Model-driven Engineering (MDE) or Model-driven Software Development (MDSD) (France & Rumpe, 2007). Although the use of models may contribute positively to this new context, it also introduces new problems. 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 regarding this software.

1.1 Background

According to Czarnecki & 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 are all 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 terms *model* and *artifact* 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. To exemplify, suppose one has 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 names updated in the sequence diagrams and in the source-code. It turns out, though, that a generic and automatic model synchronization tool able to be applied in the practice is not known by us to exist, even though an expressive effort has been made by the academic community to create solutions for this problem.

1.2 Motivation

In general, synchronized models enhance the documentation quality, since 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, since 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, since 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 user runs the risk of not being able to use a big part of their set of models, what in turn lowers the quality of the software. This discussion points out to the motivation of **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.

Generally, the amount of inconsistencies tend to increase 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.

1.3 Objective

Based on these facts, the general goal of this thesis would be explore the problem of model synchronization for complex technological spaces by analyzing the models contained in them and the relations between these models, and establishing synchronization techniques for them. Nonetheless, to work with a reasonable scope we restrict our domain to the **Java technological space**, mainly due to the popularity of the technology and the existence of well-established standard models like the UML.

More specifically, this bachelor thesis aims to (1) present and define formally some **models** from the Java technological space, that might require synchronization, explaining their objectives and some of their basic elements; (2) formalize and explain some **relations** between these models, creating a network of metamodels, so that synchronization is possible; and finally (3) discuss how **synchronization** may be accomplished in this network.

This thesis 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. Firstly, a collection of common metamodels used in the Java technological space is identified, this is done through a state-of-the-art research. Then the formal definition of some of these metamodels are presented. So for example, in this phase the choice of the used metamodels (i.e. *UML Class Diagram*, *Java Code*, etc.) is done and their formal definitions are shown.

Later on, given these defined metamodels, relations between them can be written. So for example, in this phase the inherent relations between the *UML Class Diagram*

metamodel and the *Java Code* metamodel are 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 and constitute our developed network of metamodels.

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, as well as 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. Nevertheless, the results of this thesis contribute to a better understanding of the problem of model synchronization of complex technological spaces.

The remainder of this document is structured as follows. Chapter 2 presents a literature review comprising the basic definitions 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 them and the Chapter 5 proposes an algorithm for the synchronization of a network of models. Finally, Chapter 6 and 7 discuss the results and give an overview of the work.

2 FOUNDATIONS

Before describing the development of this thesis, it is important to review some important definitions regarding 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.

2.1 Models and MDE

The items below introduce general definitions of terms regarding MDE.

- **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 instance, 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, 2004b, p. 13). One example of a system, according to this definition, is a Java program, since 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 (2004b), a model is a possible role that 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 a model M, it is meant a system that represents (or abstracts) another system S. Moreover, 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*, the documentation of a system in *Java Doc* or even a Java source-code.

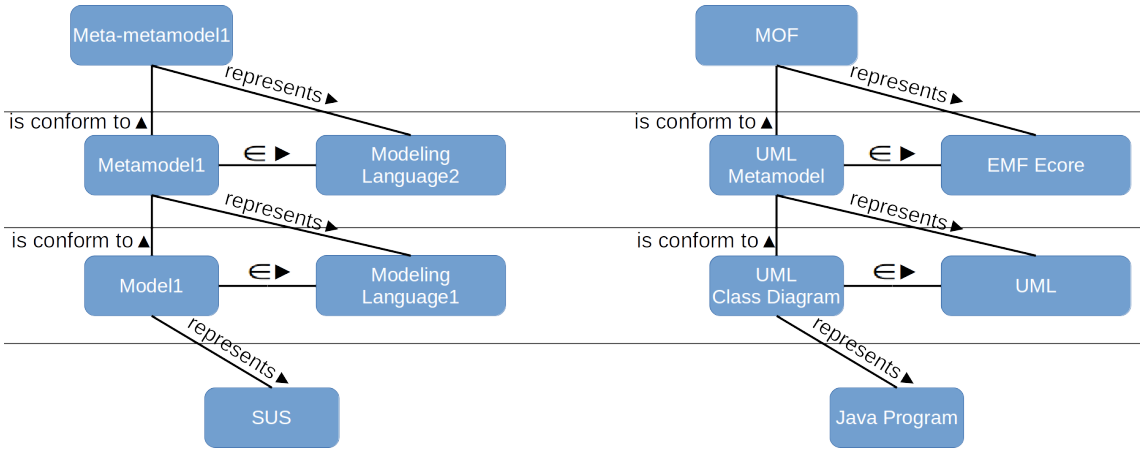
- **Modeling Language:** A model is expressed, using a modeling language. "A modeling language (L) is a set of models" (Favre, 2004b, p. 13), that contains all the models M_i expressed in that language, i.e. $L \ni M_i$. Examples of modeling languages are the *UML*, the *diagram notation for relational database diagram* or the Java language.
- **Metamodel:** Favre (2004a, 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, if and only if, M belongs to the language specified by MM. 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).
- **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 metamodels. An example of meta-metamodel is the *Meta Object Facility (MOF)* (OMG, 2015), which the *EMF Ecore* is supposedly 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.
- **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). According to our definition, the MOF is a meta-metamodel, that inherits much from the UML and deals with the ideas of classes, properties and associations, providing an extensible but simple fashion to define metamodels.

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

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

The Figure 2.1 illustrates our understanding of the definitions above. On the left is a depiction of the theoretical definitions of system, model, metamodel, meta-metamodel and modeling language. Like stated before, a system is represented by models, which themselves are expressed in languages and are conform to 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.

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



2.2 Model Synchronization

The items below introduce terms regarding the relation between models and the model synchronization.

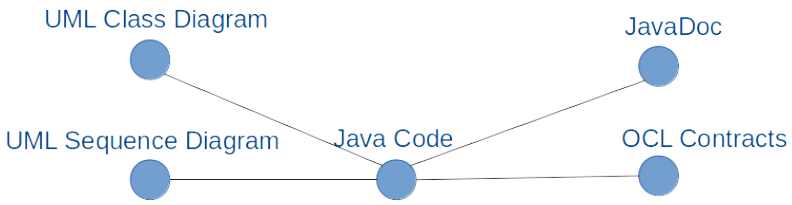
- **Model Relation:** Model relation is abstractly defined here as every relationship or

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

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) has a relation to the *JUnit* model, since the former has to be correspondingly tested by 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 & Helsen, 2006). More specifically, model transformation is defined here as a function $t : M \rightarrow N$, where $t(m) = n$ means that a target model $n \in N$ is created from a source model $m \in M$, M and N being respectively the modeling languages 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 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, as updates are performed over them (Diskin, 2011). More specifically, model synchronization is defined here as a function $s : M \times M \times \Delta_M \times N \times N \times \Delta_N \rightarrow M \times N$, where $s(m_0, m_1, \delta_m, n_0, n_1, \delta_n) = (m_2, n_2)$ means that final synchronized models m_2 and n_2 are created from the initial synchronized models m_0 and n_o and the modified non-synchronized models m_1 and n_1, considering the modifications (respectively δ_m and δ_n) performed over both. Practical example: Modification of a method name (δ_m) in the *UML class diagram* (m_0) has to be forwarded to the Java code (n_0), without losing extra information of it (e.g. comments). 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 $G = (V, E)$, where 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 & Van Gorp (2006).

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



2.3 Triple Graph Grammar

The items below introduce terms regarding the theory used to code the relations developed in this work, namely triple graph grammars (TGG).

- **Triple Graph:** With the use of a triple graph a relation between a source model S and a target model T are abstracted into a triple (G^S, G^C, G^T) – where G^S is the graph representation of source model elements, G^T is the graph representation of target model elements, and G^C represents the correspondence between the two set of model elements – together with two mappings $s_G : G^C \rightarrow G^S$ and $t_G : G^C \rightarrow G^T$, which bind the three graphs together (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 $m : G \rightarrow H$, with $m = (m^S, m^C, m^T)$. According to the Figure 2.3.

  Figure 2.3: The morphism $m : G \rightarrow H$ is a triple graph $m = (m^S, m^C, m^T)$.

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

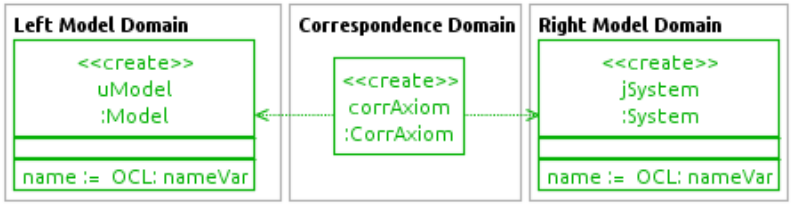
  Source: (Hermann *et al.* , 2011)

- **Triple Rule:** A triple rule is a triple graph morphism $t = (t^S, t^C, t^T) : L \rightarrow R$, where L and R are called respectively the left-hand the right-hand sides (respectively, LHS and RHS) (Ehrig *et al.* , 2007).
- **Triple Axiom:** A triple axiom is a triple rule $t_a = (t^S, t^C, t^T) : \varnothing \rightarrow R$. 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, respec-

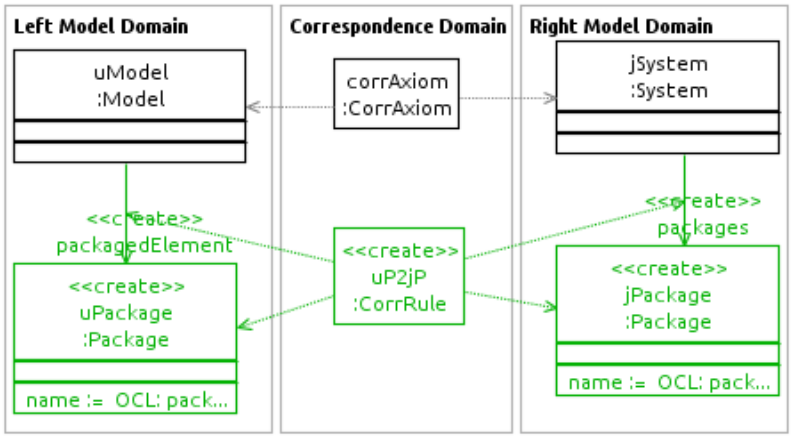
tively, the source model elements, the correspondence between source and target and, finally, the target model elements. A triple rule in turn is represented by a triple graph in black (left-hand side) plus a triple graph in green (right-hand side) (see 2.4b). Because an axiom is a triple rule with empty left-hand side, only green graph occurs in an axiom (see 2.4a).

Figure 2.4: 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*

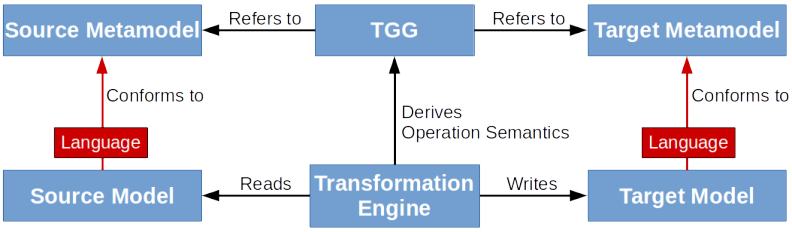


- **Triple Graph Grammar:** A triple graph grammar $TGG = (t_a, T_{rules})$ consists of a triple axiom t_a and a set of triple rules T_{rules} (Giese *et al.* , 2010b, p. 4). While one triple graph can be used as a description of a relation between two metamodels, one TGG describes the language of these 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.* , 2010b).

Figure 2.5 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 2.5: Illustration of the terms of model relation, transformation and synchronization and triple graph grammars (TGG)



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

3 STATE OF THE ART

Some endeavors have been made in order to code relations between some metamodels and mainly to develop theoretical results and synchronization methods. Heidenreich *et al.* present in (2009) and (2010) a Java metamodel using *EMF 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) come 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 through the use of the same modeling language and transformation method as us, namely *EMF Ecore* (Steinberg *et al.* , 2008) and *MoTE* (Giese *et al.* , 2010b). Finally, Giese *et al.* (2010a) 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* tool is the most adequate option for our needs, since literature about the subject is widely available (see also Giese & Hildebrandt, 2009 and Hildebrandt *et al.* , 2012). Nevertheless, there are other attempts to build a model synchronization tool, such as 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. Hildebrandt *et al.* (2013) also published 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 & Wagner (2006), Ivkovic & 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 & 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, similarly, in Mens & Van Gorp (2006) a taxonomy for model transformation is proposed, what helps to carry out more precise analysis. In Czarnecki & Helsen (2006) a survey was undertaken and a framework for

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

classification of model transformation approaches was presented. In Diskin *et al.* (2014) and (2016) a taxonomy for a network of models is presented and in Diskin (2011) a theoretical algebraic basis is proposed.

Additionally, one can judge by the date of publication of these works, that the topic of model synchronization is extremely active and is indeed on the edge of current academic research, what motivates even more the development of this thesis. All in all, differently from the other approaches, this thesis proposes the implementation of a whole network of metamodels for the Java technological space with formalized relations written in TGG, plus an evaluation of such implementation and, finally, a discussion about how synchronization may be done in this network.

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 to present the selected network of metamodels in the Chapter 4.1, by describing what each model represents and how they relate to each other. Then, each developed metamodel definition is exposed through diagrams and examples in the Chapter 4.2, and, ultimately, the formalization of the relations between them are partially shown in the Chapter 4.3, where an overview of the relations is given and some representative triple rules are presented. Attached to this thesis is the digital version of these metamodels and all the definitions of the relations.

4.1 Network of Metamodels

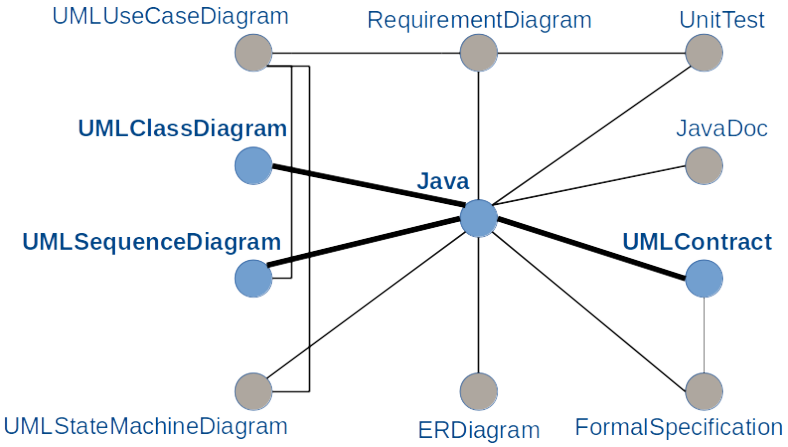
The choice of which models are used in a certain Java program may considerably vary depending on the context of the development and on the software requirements, which themselves can range from high dependability (e.g. airplane software) to continuous evolution (e.g. applications for cellphones), for 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*.

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*.

Since 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 ***UMLClassDiagram*** 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.

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



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 through 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 is the Z Notation (Spivey & 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 of them is based on creating *test cases* in the latter according to the *contract annotations* (e.g pre and postconditions, invariants, 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 in information system analysis for description of features of a system, and also 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 have 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), which is used to construct data models and which is specially applied to describe database schemes through basically *entities*, that often correspond to *Java classes*, as well as to the *relationships* between them, that may be seen as *Java attributes*.

4.2 Metamodel Definitions

As stated in the last section, the vertices highlighted in the Figure 4.1, namely *UMLClassDiagram*, *UMLSequenceDiagram*, *UMLContract*, and *Java*, have its metamodels defined 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

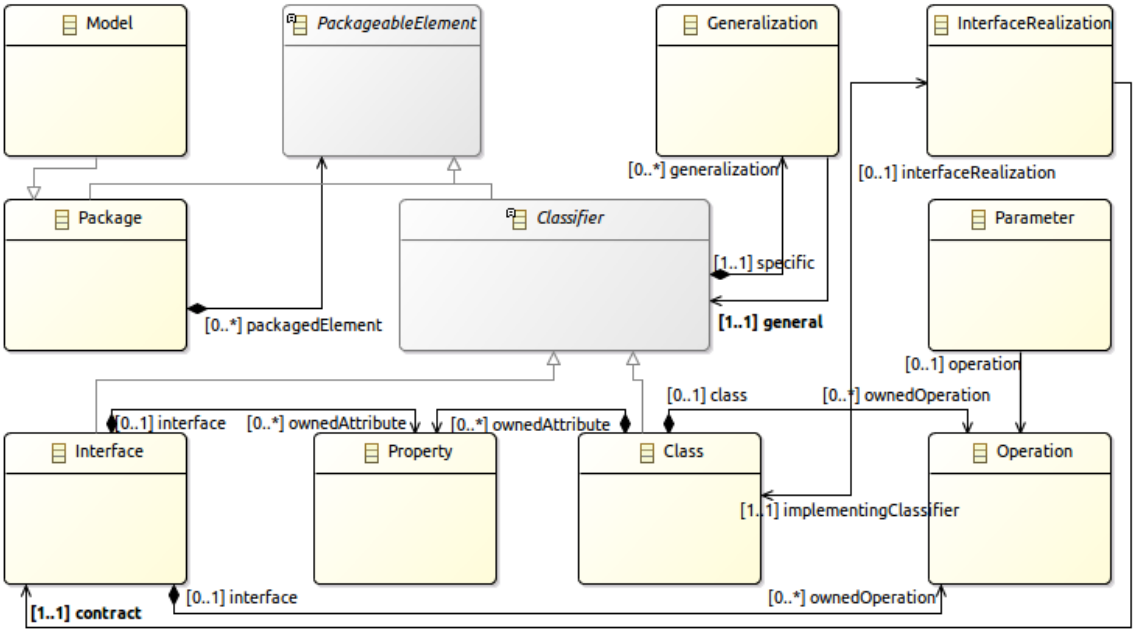
The metamodel utilized for *UMLClassDiagram* (also for *UMLSequenceDiagram*, *UMLContract*) represents the **version 2.0 of the UML standard**, is provided by the *EMF plug-in*[4] for Eclipse, integrates easily with the IDE and seems to be suitable for our needs. Alternatively, we could use the metamodel 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 ones. 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 the latter inherits *PackageableElement*. The two classifiers handled in the figure are *Class* and *Interface*. The model in our running example (Figure 4.3) 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



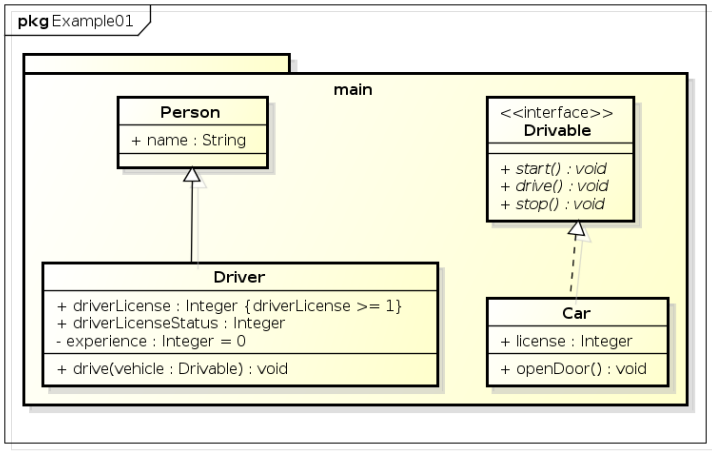
Source: Image created using the *Eclipse IDE*. Metamodel from *EMF plug-in*.

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

(a) Abstract Syntax

```
1  <Model> Example01
2    <Package> main
3      <Class> Person
4        <Property> name : String
5      <Class> Driver
6        <Generalization> Person
7        <Property> driverLicense : Integer
8        <Property> driverLicenseStatus : Integer
9        <Property> experience : Integer
10         <Literal String> 0
11       <Operation> drive (vehicle : Drivable)
12         <Parameter> vehicle : Drivable
13     <Interface> Drivable
14       <Operation> start ()
15       <Operation> drive ()
16       <Operation> stop ()
17     <Class> Car
18       <Property> license : Integer
19       <Interface Realization> Drivable
20       <Operation> openDoor ()
```

(b) Concrete Syntax



Source: Diagram created using the *Astah Software*.

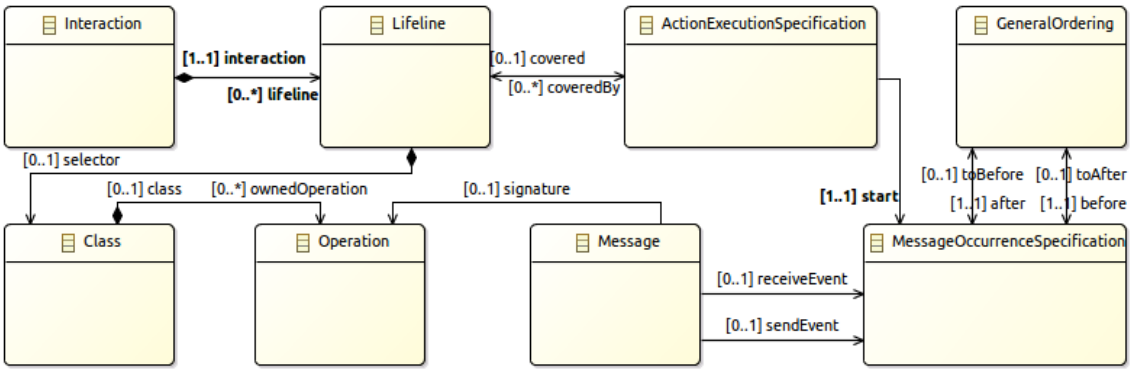
According to the metamodel, a *Classifier* may have a *Generalization* (i.e. inheritance), illustrated with a straight arrow from *Driver* to *Person* in the example on the Figure 4.3b, or an *InterfaceRealization* illustrated with a dashed arrow from *Car* to *Drivable*. Moreover, a *Class* is allowed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



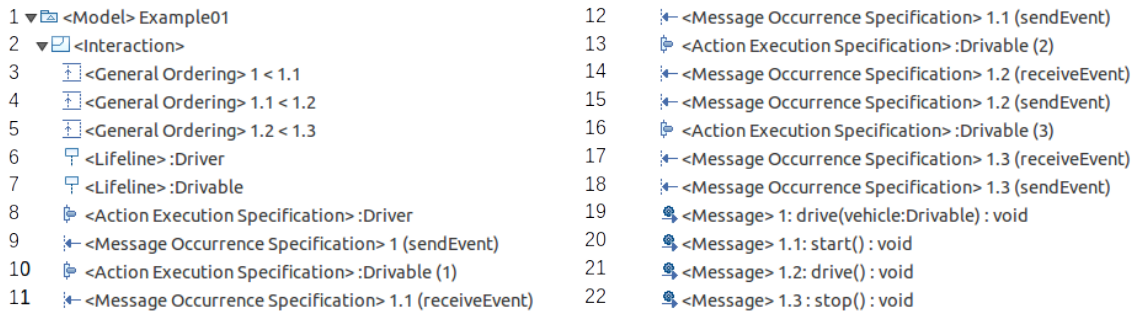
Source: Image created using the *Eclipse IDE*. Metamodel from *EMF plug-in*.

A *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' code 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

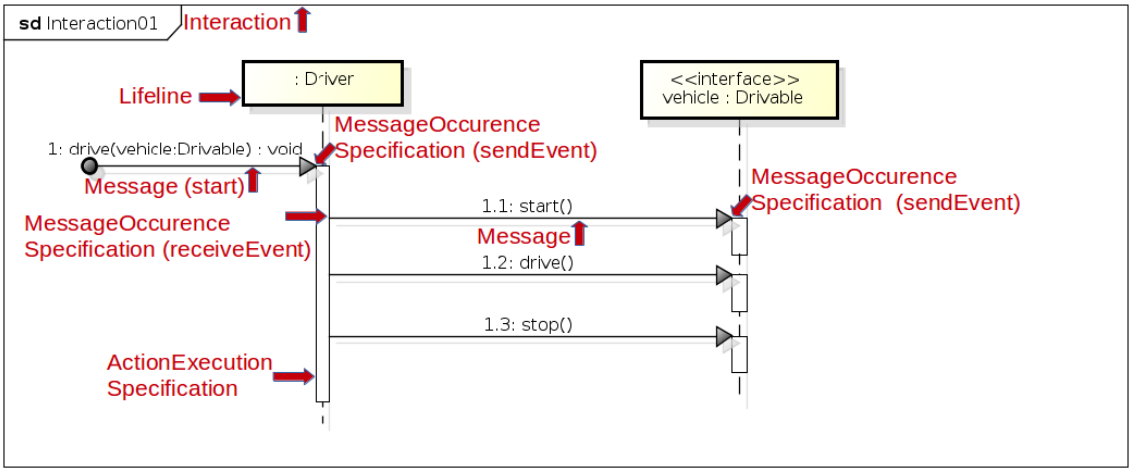
comprehension of this interpretation over *UMLSequenceDiagrams* requires the reader to grasp the Figure 4.5 containing one *Interaction* (*Interaction01*), two *Lifelines* (*:Driver* and *:Drivable*), and four *Messages*.

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

(a) Abstract Syntax



(b) Concrete Syntax



Source: Image created using the *Astah Software*.

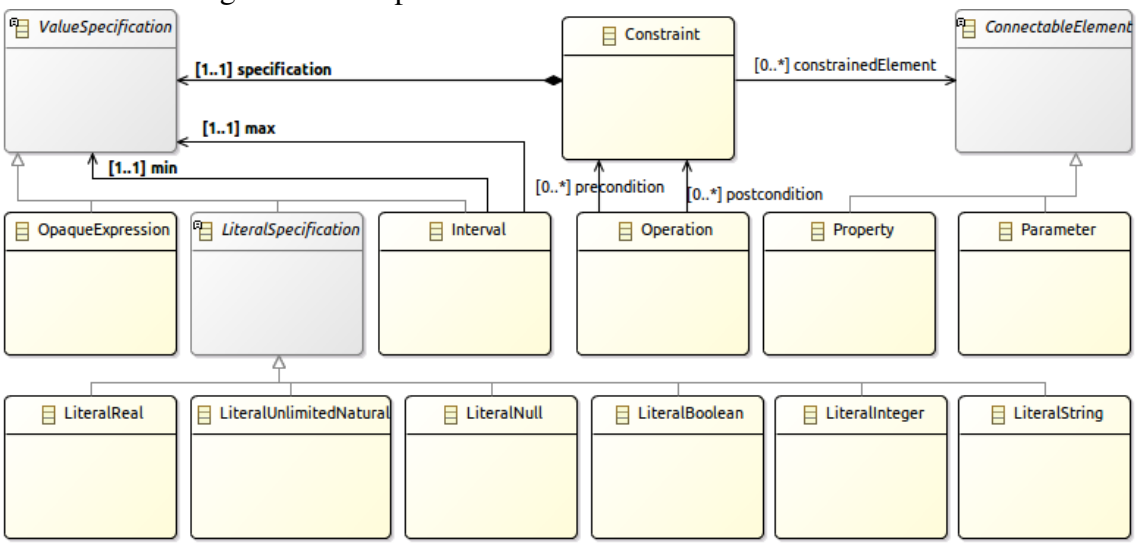
To model the order in which the *Messages* occur, a *GeneralOrdering* establishes an order between two *MessageOccurenceSpecifications*, by signalizing which of them occur *before* or *after* the other. So, in the running example there is a *GeneralOdering* instance holding the *MessageOccurenceSpecifications* 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

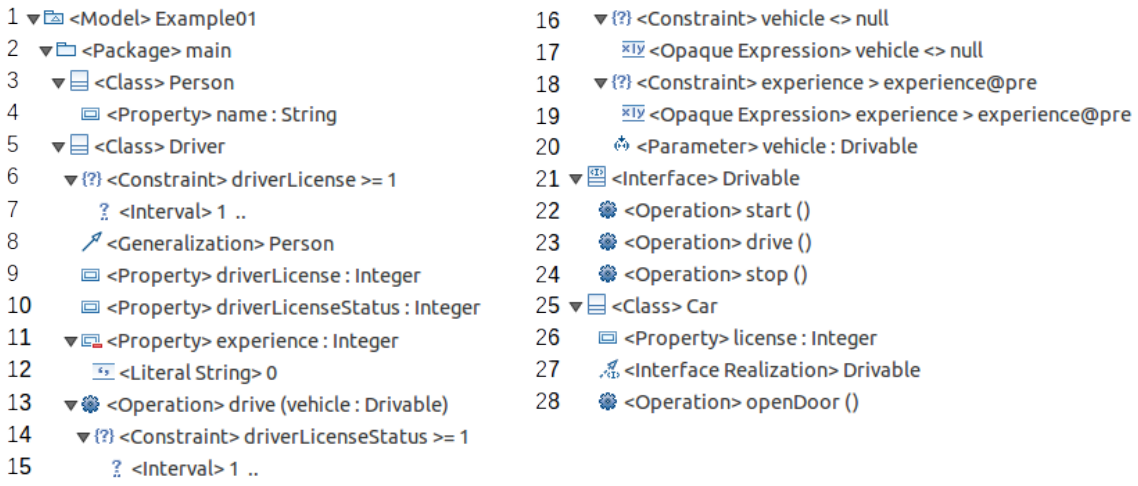
The *UMLContract* is a slice of the *UML* metamodel, that aims, basically, to provide **constraints** to *Operations* and *Properties* of *Classes*. *Operations* may have *pre* and *postconditions* as well as *Invariants*, which are modeled through the *EMF Ecore* class *Constraint* (on the top of the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

Figure 4.6: Simplification of the *UMLContract* metamodel



Source: Image created using the *Eclipse IDE*. Metamodel from *EMF plug-in*.

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.



Source: Image created using the *Eclipse IDE*.

A *Constraint* may have *constrainedElements*, which in the scope of this thesis are

either *Properties* or *Parameters*. In addition, it also has a *ValueSpecification*, defining the constraint itself, that in this thesis may be an *OpaqueExpression* or an *Interval*. The former is a free definition of the constraint that is composed by a String (see lines 16 to 19 in Figure 4.7). The latter defines an interval of values, in which the constrained element must lie. Therefore, it has one *ValueSpecification* for the minimal and one for the maximal value. Here, only *Intervals* with one *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*).

4.2.4 Java

A *Java* model is usually the main artifact of a network of models, since 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 as in the Figure 4.10. The first option is used here, since it seems to be more suitable than a plain-text source-code format, when handling the model for synchronization. Nevertheless, there are techniques[5] to transform one format into another.

There are currently some Java metamodels available in literature. Among them are the metamodel[6] provided by the *Eclipse IDE*, whose excess of simplicity hinders its use; or the one found in Heidenreich *et al.* (2010), which happens to be so extensive that could bring unnecessary complexity to this thesis. Therefore, a brand 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 being very shallowly modeled here, even though it could be further developed in future works. The Figure 4.8 reports this whole *Java* metamodel created. Elements in blue are *EMF Ecore* abstract elements, whilst elements in yellow are concrete ones.

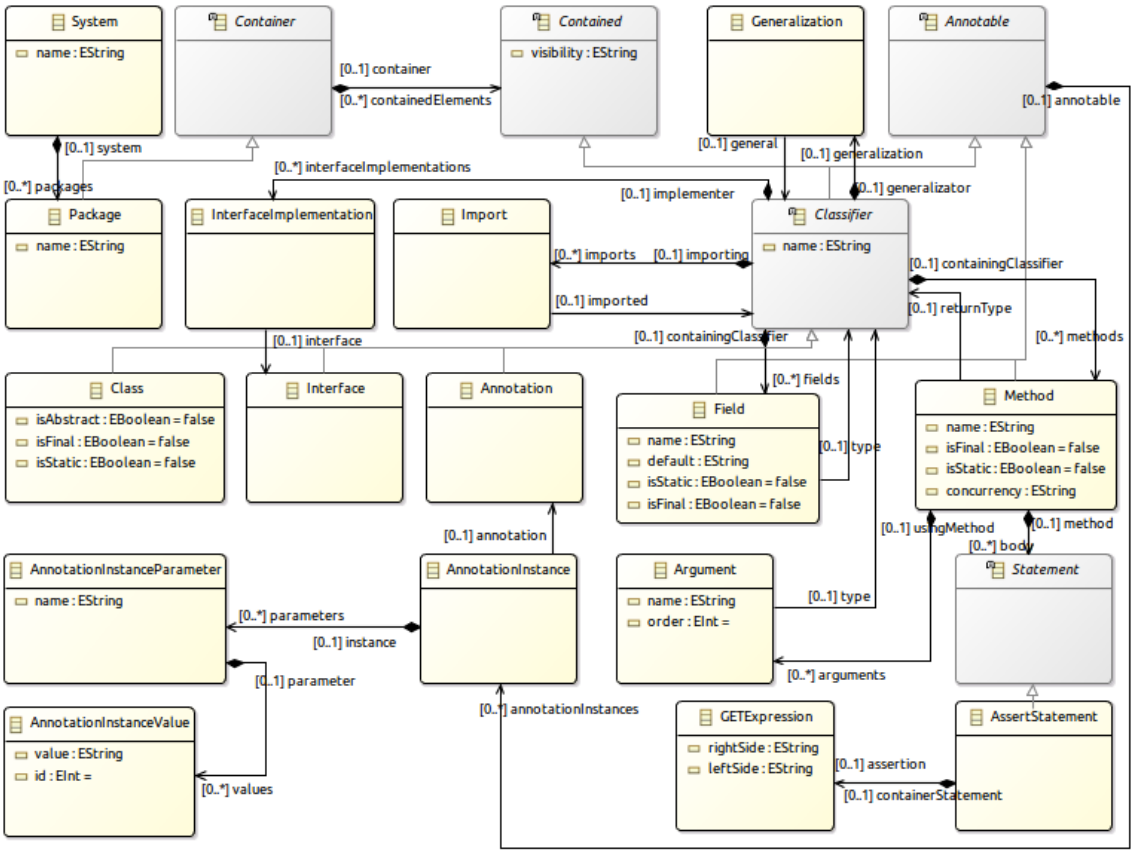
The root element is the *System* (on the left top of the Figure 4.8) and represents the whole Java program. It contains *Packages*, which, because of the *Container* inheritance, contain *Classifiers* (through the *Contained* inheritance). The example on the Figure 4.9 illustrates concrete types of *Classifiers* 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,

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

31, 33, 35, 37), *Imports* (lines 40 to 43), *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*.

Figure 4.8: The Java metamodel created



Source: Image created using the *Eclipse IDE*.

A *Method* may have zero or more *Arguments* – also known as parameters – as well as zero or more *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*, usually representing 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, since they are useful in the relation between *Java* and *UMLContract*. 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.

Source: Image created using the *Eclipse IDE*.

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

```java
package main;

import de.silvawb.utils.*;

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

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

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

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

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

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 of model 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.

4.3 Metamodel Relations

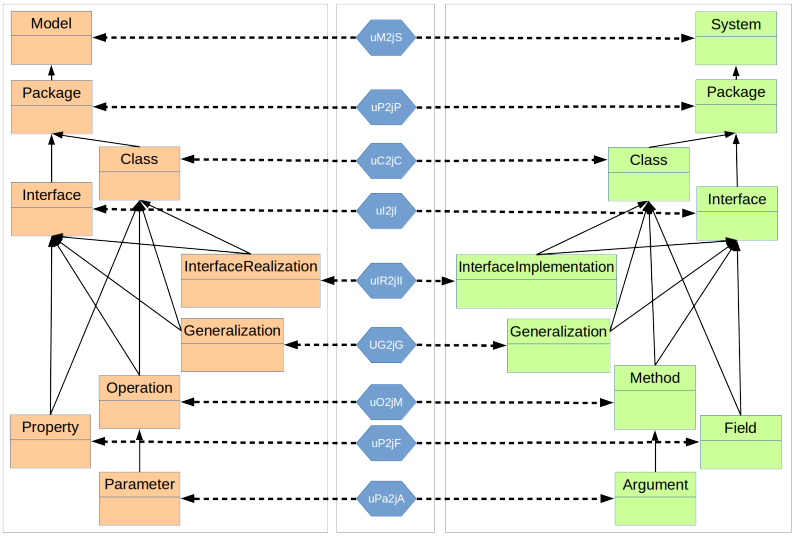
With all the metamodels defined, the definition of the **relations between their elements** can be made. In order to accomplish it, triple graph grammar (TGG) is used with the *MoTE* tool to code such relations, due to the extensive use of TGG in current academic research and to its wide support by several synchronization tools. Other options included the *ATL* (Jouault *et al.* , 2008), which does not seem to be ripe enough for our purpose; or *Graph Transformation* (see an implementation in Arendt *et al.* , 2010), which is not widely supported for the best synchronization tools (Hildebrandt *et al.* , 2013). A theoretical basis of TGG has been given in Chapter 2.

In order to introduce the general ideas of each relation, the respective **triple type graphs** are shown first, superficially presenting how each element in the source metamodel relates to elements in the target metamodel. The the most representative triple rules, developed with the *MoTE* tool, are shown afterwards. It is important to note that these triple rules diverge slightly from the theoretical definitions used here, anyhow, they serve to discuss some issues found out during the development of this work. The complete explanation of the implementation details are not included in this thesis, since we intend to focus on the general ideas over the rules. The complete version of the TGGs are attached to this thesis in digital form.

4.3.1 Relations between *UMLClassDiagram* and *Java*

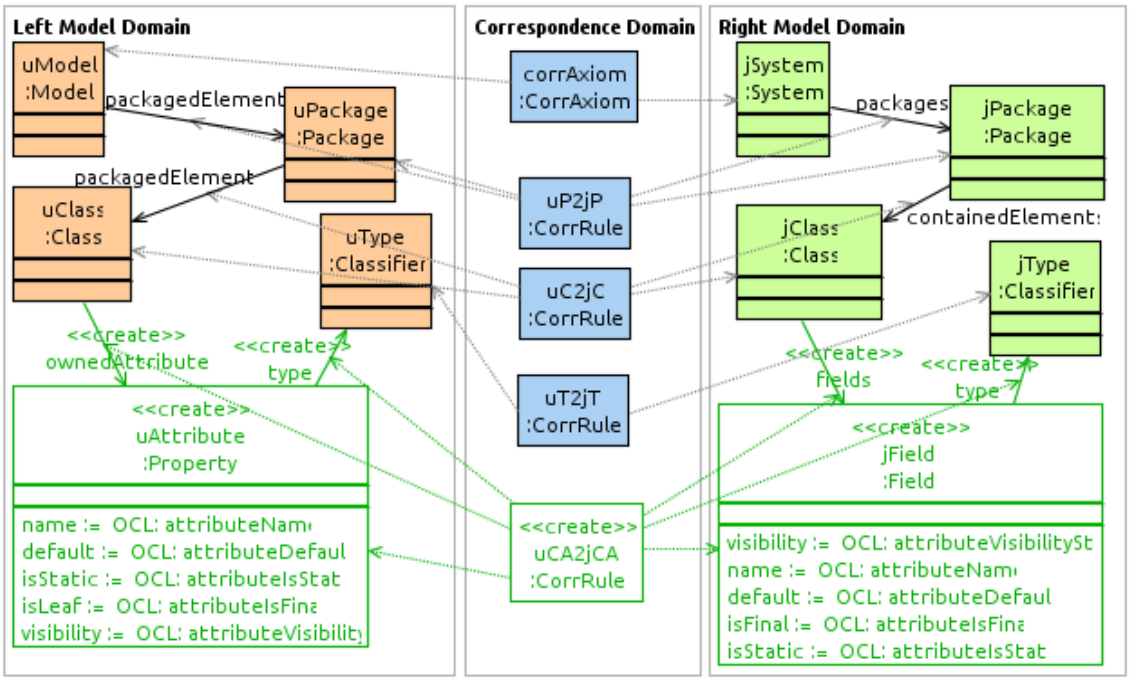
Figure 4.11 shows the triple type graph for the relations between the *UMLClassDiagram* (left) and the *Java* (right) domains, with the correspondence domain being in the middle. In this graph, elements from the left domain are connected to the elements on the right domain, which they have a relation with. So, the element *Model* in *UMLClassDiagram* has a relation to the element *System* in *Java*; analogously, a *Property* in *UMLClassDiagram* is related to a *Field* in *Java*, to be specific, 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*



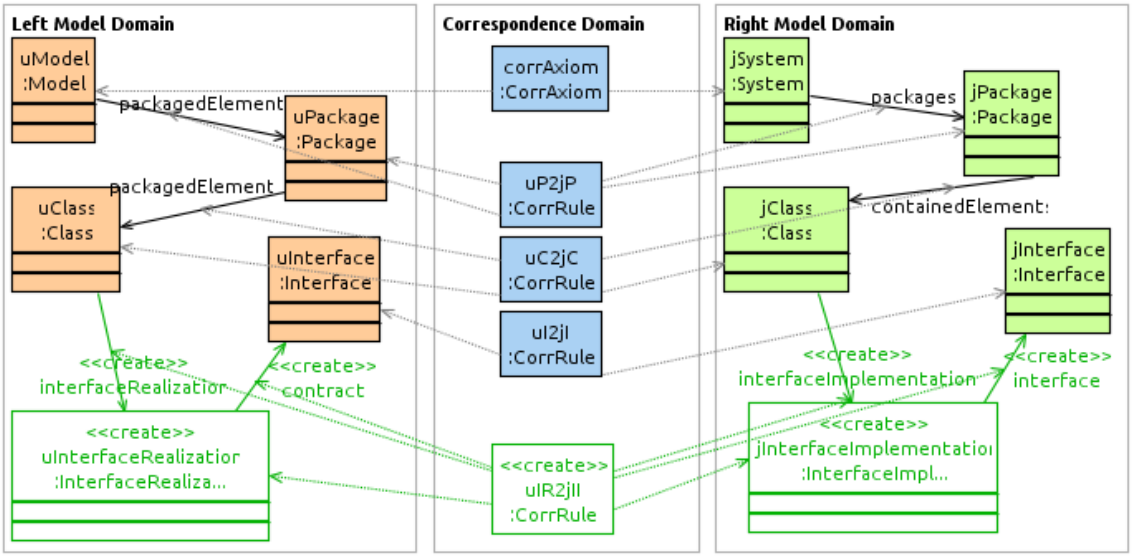
The triple rules describing the relation between *Model* and *System*, as well as *Package* (UML) and *Package* (Java) is available on Figure 2.4b. The triple rule encoding the relation between an *Attribute* of a *Class* in *UMLClassDiagram* and a *Field* of a *Class* in *Java* is shown on the Figure 4.12. In essence, this rule formalizes the fact that for every *Class Attribute* (*uAttribute*) in *UMLClassDiagram* a correspondent *Class Field* (*jField*) in *Java* is supposed to exist with corresponding values for the respective meta-attributes. These are *name*, *default*, *isStatic*, *isLeaf* and *visibility* in *uAttribute*, which correspond, respectively, to *name*, *default*, *isStatic*, *isFinal* and *visibility* in *jField*.

Figure 4.12: The triple rule *uClassAttribute2jClassField*



Source: Image created using the *Eclipse IDE*.

Figure 4.13: The triple rule *uInterfaceRealization2jInterfaceImplementation*



Source: Image created using the *Eclipse IDE*.

Note that, the *MoTE* tool uses a more relaxed definition of triple graphs, allowing elements of the correspondence model to map more than one element from the source to more than one element from the target model. Moreover, for this tool it is not necessary to distinguish elements of the correspondence model using types. As a consequence, the

types *CorrRule* and *CorrAxiom* are enough for our implementation.

Analogously, the Figure 4.13 shows the triple rule $(L^S, L^C, L^T) \rightarrow (R^S, R^C, R^T)$ for *UML InterfaceRealization* and *Java InterfaceImplementation*. In this case, *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^S, L^C, L^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 ($uIR2jII \in R^C$) in the state S_{i+1} 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* with regard to a 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 an *AnnotationInstanceParameter* (*jAnnInstParam*) belonging to an already existing method (*jMethod*) and referencing an already existing *Annotation* (*InteractionSequenceAnnotation* named *Interaction*).

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 4.14: The triple type graph for *UMLSequenceDiagram* and *Java*

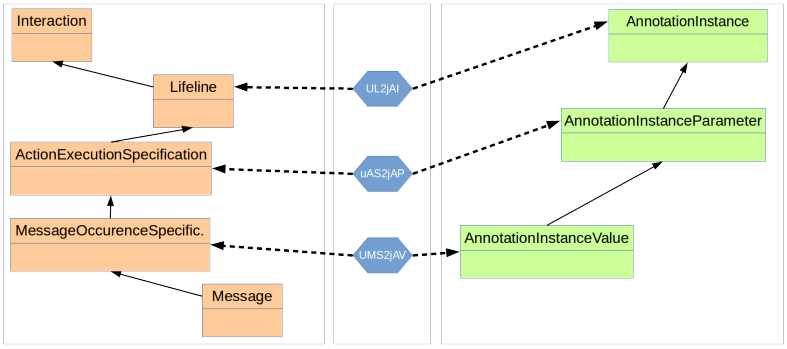
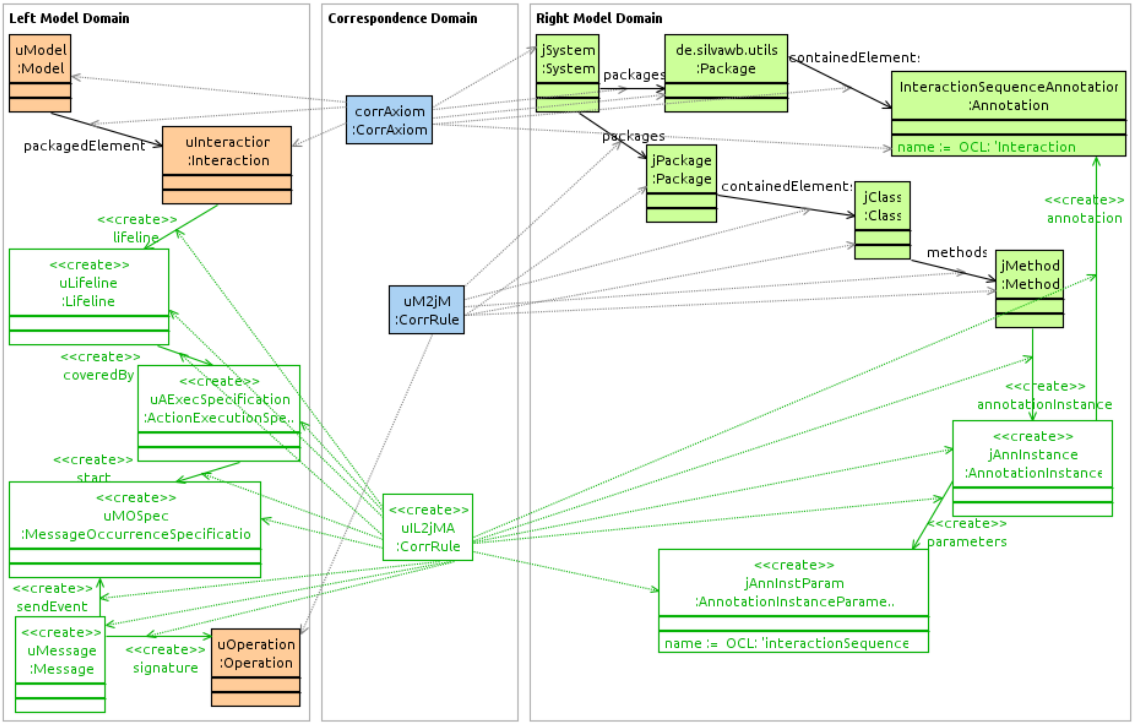


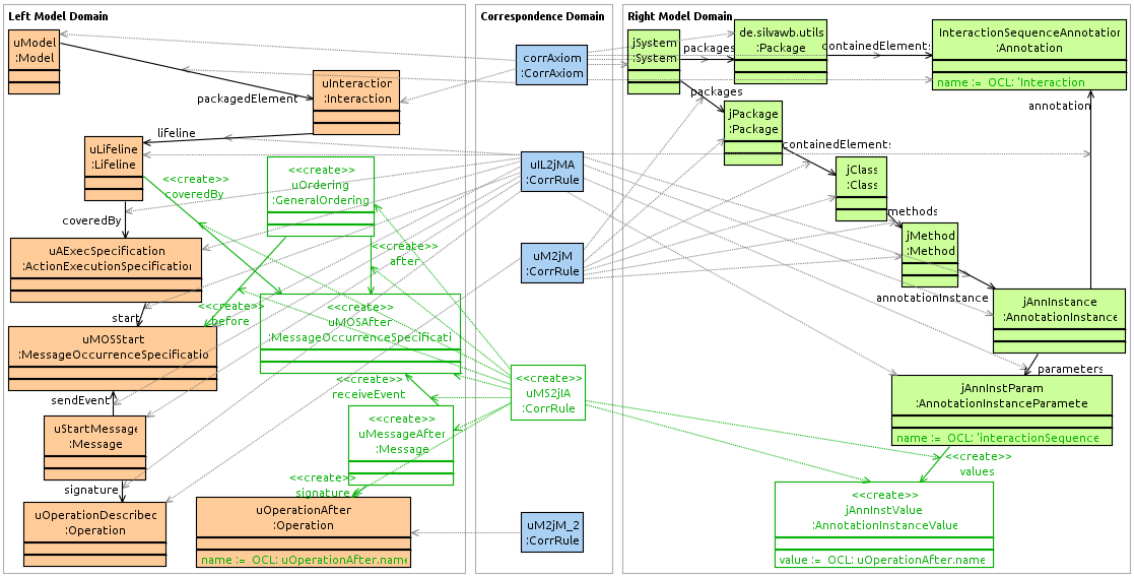
Figure 4.15: The triple rule *uLifeline2jMethodAnnotation*



Source: Image created using the *Eclipse IDE*.

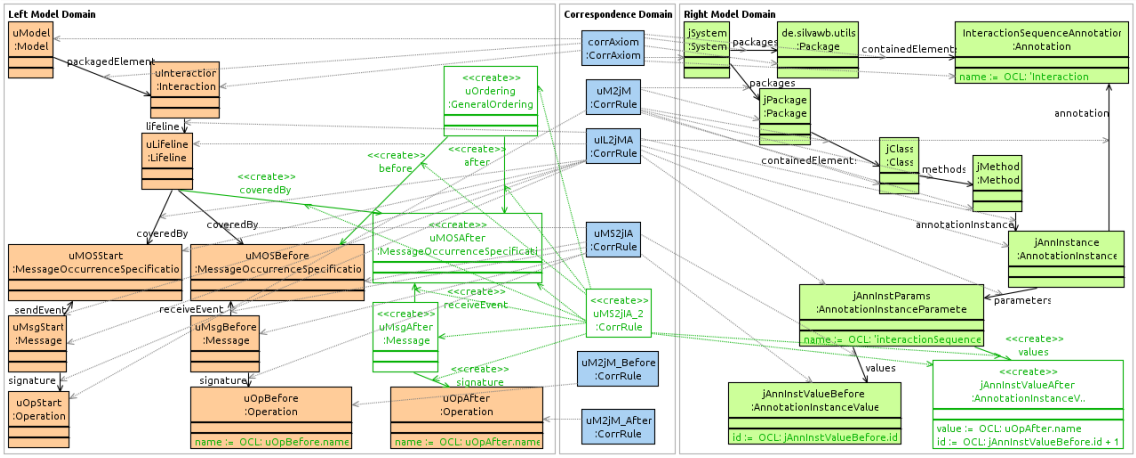
The rule *uMessageSequence2jInteractionAnnotation* encodes the fact that every *Message* (*uMOSAfter*) coming right after the initial *Message* (*uStartMessage*) of *uLifeline* has a correspondent *AnnotationInstanceValue* (*jAnnInstValue*) containing the same *name* as *uOparationAfter*, and being child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

Figure 4.16: The triple rule *uMessageSequence2jInteractionAnnotation*



Source: Image created using the *Eclipse IDE*.

Figure 4.17: The triple rule *uMessageSequence2jInteractionAnnotation_2*



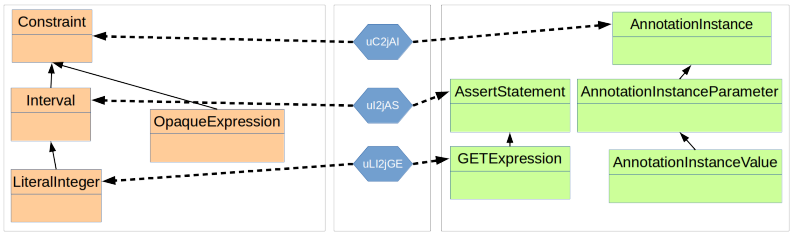
Source: Image created using the *Eclipse IDE*.

The occurrence order of the *Messages* is handled by the rule *uMessageSequence2jInteractionAnnotation_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.

4.3.3 Relations between *UMLContract* and *Java*

The 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 contain (1) an *OpaqueExpression* or (2) an *Interval* and is connected always to an *AnnotationInstance*. Examples of *OpaqueConstraints* are shown in the lines 29 and 31 of the Figure 4.10. The triple rule for the case of an *OpaqueConstraint* 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* named *Inv*.

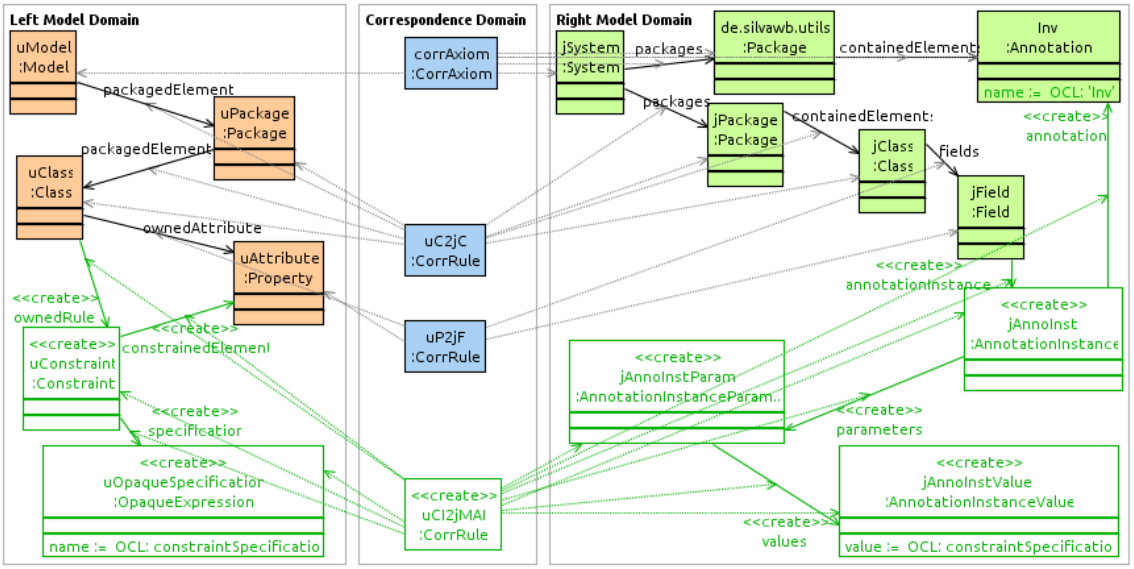
Figure 4.18: The triple type graph for *UMLContract* and *Java*



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 creates not only *Annotation* elements in the Java model, but also *AssertStatements* in the form of *GETExpressions* belonging to a previously created *check Method*.

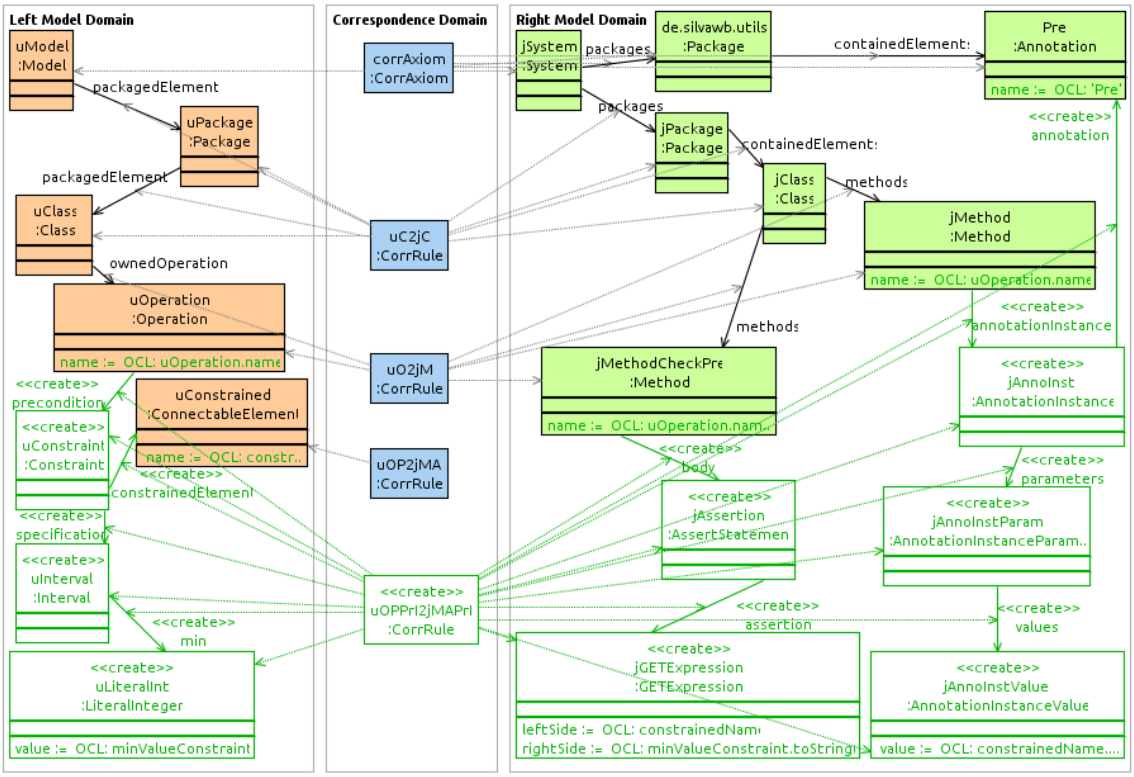
In our implementation, every *Method* of the Java model has three *check Methods*, namely *checkPreConstraint*, *checkPosConstraint* and *checkInvConstraint*. By being so, the rule *uOperationPreInterval2jMethodPreAssert* is able to transform *UML interval preconditions* into *Java GETExpression*, by 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 *uClassInvariant2jClassInvariant*



Source: Image created using the *Eclipse IDE*.

Figure 4.20: The triple rule *uOperationPreInterval2jMethodPreAssert*

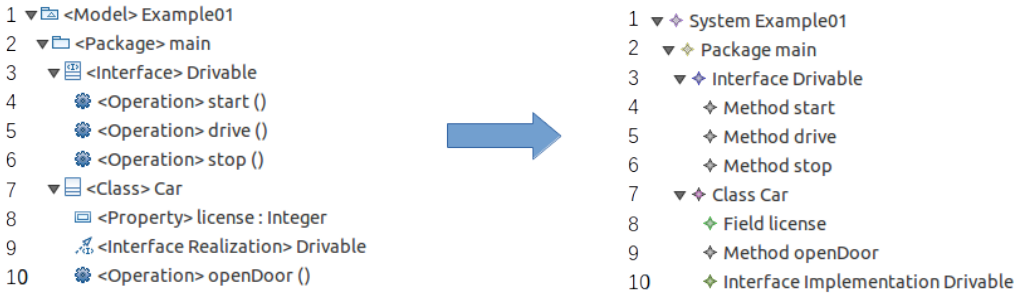


Source: Image created using the *Eclipse IDE*.

4.4 Evaluation

The three TGGs (*umlClassDiagram2java*, *umlSequenceDiagram2java* and *umlContract2java*) 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 TGGs are executed. We used a set of example cases, some of which 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 TGG *umlClassDiagram2java*. This TGG includes the rules presented above (*uClassAttribute2jClassField*, *uInterfaceRealization2jInterfaceImplementation*) plus the following rules, all comprised in digital form, attached to this thesis: *uModel2jSystem* (relation between *UML Model* to *Java System*), *uPackage2jPackage* (relation between *UML Package* to *Java Package*), *uClass2jClass* (relation between *UML Class* to *Java Class*), *uClassOperation2jClassMethod* (relation between *UML Operation* to *Java Method*), *uInterface2jInterface* (relation between *UML Interface* to *Java Interface*), *uInterfaceAttribute2jInterfaceField* (relation between *UML Property* of an *Interface* to *Java Field* of an *Interface*), *uInterfaceOperation2jInterfaceMethod* (relation between *UML Operation* of an *Interface* to *Java Method* of an *Interface*) and *uCGeneralization2jCGeneralization* (relation between *UML Generalization* to *Java Generalization*). 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; *uInterfaceRealization2jInterfaceImplementation* in line 9; and *uClassOperation2jClassMethod* in line 10.

Figure 4.21: Example of forward synchronization from a *UMLClassDiagram* model (left) and a *Java* model (right)



Source: Image created using the *Eclipse IDE*.

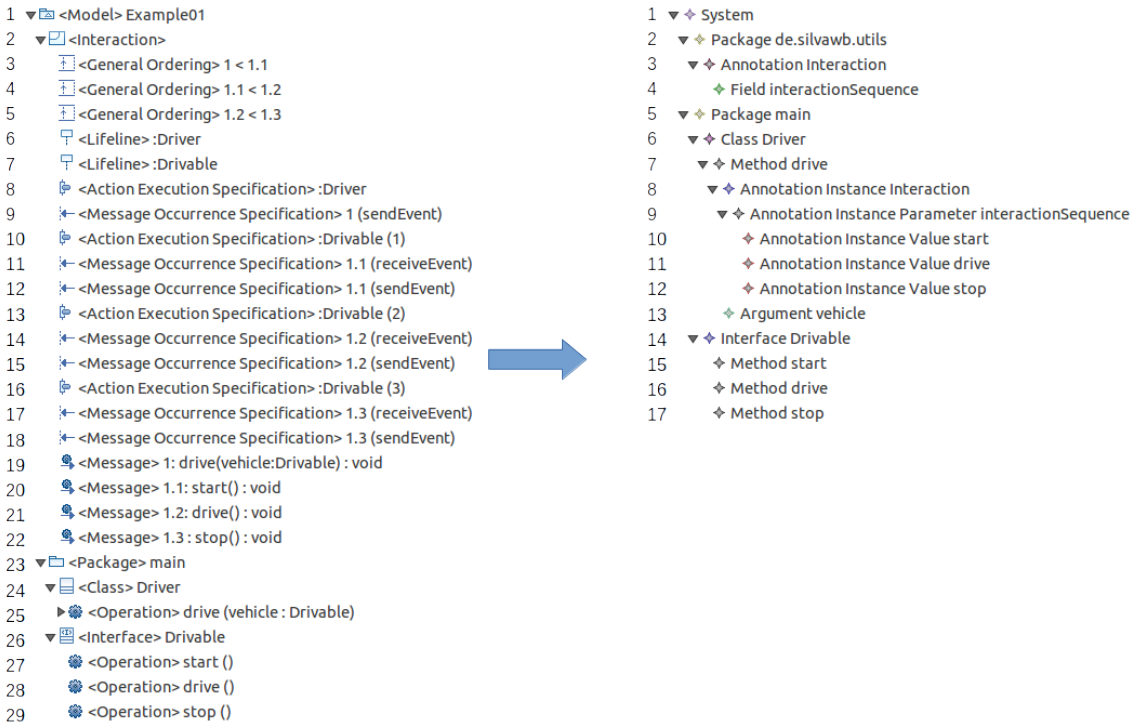
The problem with the TGG *umlClassDiagram2java* is that it does not encode the

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, since each element in L_s can be transformed only once in the operational semantic scheme proposed by Giese *et al.* (2010b, p. 9) and used by *MoTE*. In the same scheme, the creation of such rule would entail a critical pair, given that two different rules have the same object *uInterfaceRealization* in the R_s. A definite solution is not known 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 TGG *umlSequenceDiagram2java*, which comprises the rules shown above (*uLifeline2jMethodAnnotation*, *uMessageSequence2jInteractionAnnotation*, *uMessageSequence2jInteractionAnnotation_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, *uMessageSequence2jInteractionAnnotation_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 TGG *umlClassDiagram2java*. 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, since 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

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



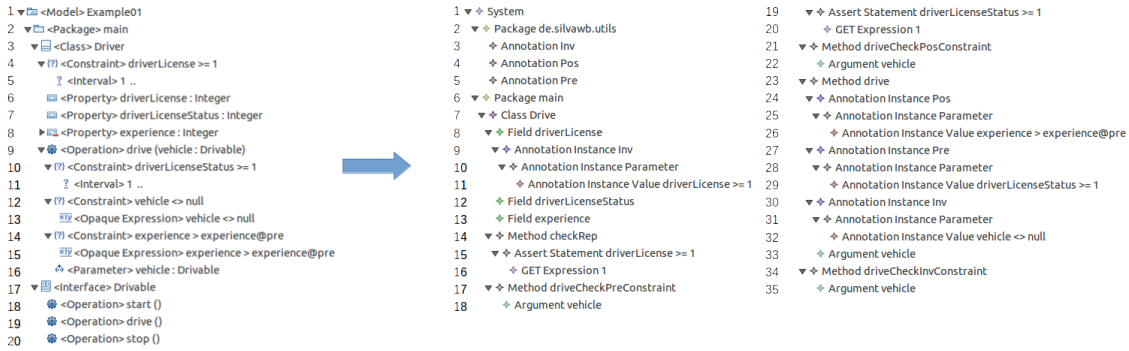
Source: Image created using the *Eclipse IDE*.

broader view, one could say, that the rule is too simple, since it only captures the order of the message occurrences and transforms it into annotations in the Java model (and vice-versa). Indeed, it exposes the problem of TGGs to deal with non MOF-appropriate models, since more powerful tools to handle method calls inside a Java method would be desired to build a more comprehensive transformation from sequence 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, thus, 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 TGG *umlContract2java*. Beyond the two rules exposed in the previous section (*uClassInvariant2jClassInvariant*, *uOperationPreInterval2jMethodPreAssert*), this TGG comprises basically one Axiom; three more rules that encode the relation between *UML Opaque Constraints* and *Java Annotations* (*uOPPre2jMAPre*, *uOPPos2jMAPos*, *uOInv2jMAInv*, respectively, for *pre*, *post* and *invariant* constraints); and two more 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; *uOperationPreInterval2jMethodPreAssert* 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* model (right)



Source: Image created using the *Eclipse IDE*.

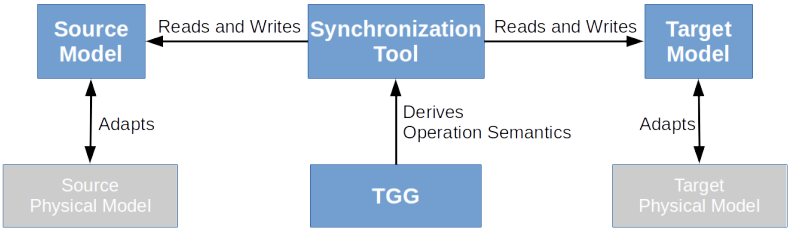
To summarize, this TGG does not seem to present big problems, except that it is not complete, since neither *invariant interval constraints* for methods, nor non-integer *interval constraints* have been developed. Moreover, the implemented rules for *interval constraints* support only *great-or-equal-than expression*. On the other hand, the extension of the current TGG version towards 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 presented in a different section (namely 4.3.1 for the TGG *umlClassDiagram2java*, 4.3.2 for the TGG *umlSequenceDiagram2java* and 4.3.3 for the TGG *umlContract2java*) and each one represents a **different edge of the network of metamodels** built (see Figure 4.1). By being so, each edge corresponds to a different synchronization problem and can be solved, 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 the problem of model synchronization between two models using TGG.

Between them are the *FUJABA* (Nickel *et al.* , 2000), a standalone application that uses TGG to code the relations; the *CoWolf*, an extensible framework based on *Eclipse Plug-ins* that supports several metamodels and that uses *Henshin* (Arendt *et al.* , 2010) to encode the relations (Getir *et al.* , 2015); and, finally, there is the *MoTE* tool, a series of *Eclipse Plug-ins* for creating TGGs, both graphically and textually, and for transforming models based on these TGGs. We deem *MoTE* the most adequate option for our needs, mainly due to the extensive literature about the subject (Giese & 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.* (2010a).

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 four first components (highlighted in the picture) are treated in this thesis. Note, however, that the final practical application of the synchronization is only possible through the

adaptation of the logical models (presented so far in form of abstract syntax, e.g. 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 network. 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 a synchronization method already proposed.

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 and analyze an algorithm to propagate such modifications.

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

Assumption 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 restriction may be treated with help of a version control system, that maintains models centralized, dealing with eventual problems such as conflicts, and observing them for changes.

Assumption 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 do not ripple back to the source, instead they go 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 $netsync : (V, E) \times V \times V \times \Delta_S \rightarrow (V, E)$, 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 as follows.

Algorithm 1 netsync Algorithm

```
1:  function NEYSYNC((V_0, E_0), s_0, s_1, δ_s)
2:      (V_i, E_i) ← ((V_0 \ s_0) ∪ s_1, E_0)          ▷ New net with first modification
3:      for all n_i = N(s_0) do
4:          (n_{i_new}, δ_n) ← sync(s_0, s_1, δ_s, n_i)          ▷ Update neighbor
5:          if δ_n not empty then                          ▷ If modified the neighbor
6:              (V_i, E_i) ← netsync((V_i, E_i), n_i, n_{i_new}, δ_n)   ▷ Update net starting from it
7:          end if
8:      end for
9:      return (V_i, E_i)
10: end function
```

Firstly, the initial network is updated with the new vertex s_1, then every neighbor n_i of s_0 is synchronized according to the modifications δ_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:

Assumption 3 *The input network of models* (V_0, E_0) *is finite.*

Assumption 4 *The input network of models* (V_0, E_0) *has no cycles.*

Assumption 5 s_1 *is synchronized to* t_0 *if, and only if,* $sync(s_0, s_1, \delta_s, t_0) = t_0$. *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 three theorems:

Theorem 1 *The algorithm always terminates.*

Because Assumption 2 is made, it suffices only one call to $sync$ on line 4, representing the synchronization of s_0 and n_i − i.e. the edge $\{s_0, n_i\}$, to synchronize both models (see Corollary 1). Furthermore, every edge synchronization is followed by at most one recursive call on line 6 − in the case that the synchronization of n_i entails the need for synchronization of neighbor vertices of n_i. Because the set of edges is finite (Assumption 3) and there is no cycle in the input graph (Assumption 4), the amount of recursive calls (line 6) and of loops iterating over any vertex's neighbors (lines 3 to 8) are also finite. Hence, the termination of the algorithm is guaranteed.

Theorem 2 *When a $sync$ call is taken as elementary operation, the time complexity of $netsync((V_0, E_0), s_0, s_1, \delta_s)$ is, in the worst case, $O(\Delta(V_0, E_0)|V_0|)$, i.e. the* maximum degree *of the graph (V_0, E_0) times the amount of vertices of the same graph.*

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, since (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 either *"not modified"* − in the case that condition on line 5 never holds true for any call $sync(s_i, s_{inew}, \delta_s, v_i)$ − or v_i is assigned the deterministic value $sync(s_j, s_{jnew}, \delta_j, s_i)$ on 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 deterministically defined 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 its simplifications and incompleteness. For instance, this metamodel does not properly deal 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 (e.g. *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 (explained in Chapter 4.3.2). Lastly, the relation *umlContract2java* is also not complete, although it might be helpful for the enhancing of 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 assumption 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 & 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 synchroniza-

tion of network in the practice. However, it can treat transformations of pairs 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 TGGs. 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 publish any standard metamodel for Java, nor are they easy to find in the literature. One might find alternative versions in the source code of IDE's, but it still requires some effort. Moreover, they are sometimes incompatible with the tool employed, what also impedes the desired 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 insufficiency 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 and broad documentation of the plug-in for the *Eclipse IDE* is still needed. At last, but not least, is the performance problem of such tools. Both the *EMF* and *MoTE plug-ins* 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 are in many cases not familiarized with 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, expanding the created network of metamodels, or by developing metamodels that satisfy completeness. Not to mention that 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, 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 devel-

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

Lately, a relatively big issue is the use of TGGs for non-MOF-compliant metamodels − or metamodels that are not naturally seen as MOF-compliant, e.g. the complete Java metamodel − because, under certain conditions, it could be interesting to analyze 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 to be solved. The Chapter 5 endeavors to propose an algorithm for synchronization of a whole network of models based on 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 briefly demonstrate the application of transformation of these relations in order to clarify how they work, evaluate them, and show the contributions that 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 broadly handled 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 assumptions made, it always terminates and it is deterministic.

Despite the fact that 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-the-art and the discussion of difficul-

ties 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

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

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

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

Blouin, Dominique, Plantec, Alain, Dissaux, Pierre, Singhoff, Frank, & Diguet, Jean-Philippe. 2014. Synchronization of Models of Rich Languages with Triple Graph Grammars: An Experience Report. *Pages 106–121 of: Theory and Practice of Model Transformations*. Springer.

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

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

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

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

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

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

Ehrig, Hartmut, Ehrig, Karsten, Ermel, Claudia, Hermann, Frank, & Taentzer, Gabriele. 2007. Information preserving bidirectional model transformations. *Pages 72–86 of: Fundamental Approaches to Software Engineering*. Springer.

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

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

Foss, Luciana, Costa, S, Bisi, Nicolas, Brisolara, Lisane, & Wagner, Flávio. 2011. From UML to SIMULINK: a Graph Grammar Specification. *Pages 37–42 of: 14th Brazilian Symposium on Formal Methods: Short Papers*.

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

Getir, Sinem, Grunske, Lars, Bernasko, Christian Karl, Käfer, Verena, Sanwald, Tim, & Tichy, Matthias. 2015. CoWolf–A Generic Framework for Multi-view Co-evolution and Evaluation of Models. *Pages 34–40 of: Theory and Practice of Model Transformations*. Springer.

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

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

Giese, Holger, Hildebrandt, Stephan, & Neumann, Stefan. 2010a. Model synchronization at work: keeping SysML and AUTOSAR models consistent. *Pages 555–579 of: Graph transformations and model-driven engineering*. Springer.

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

Greenyer, Joel, Kindler, Ekkart, Rieke, Jan, & Travkin, Oleg. 2008. *TGGs for Transforming UML to CSP: Contribution to the AGTIVE 2007 Graph Transformation Tools Contest*. Tech. rept. Department of Computer Science, University of Paderborn Paderborn, Germany.

Heidenreich, Florian, Johannes, Jendrik, Seifert, Mirko, & Wende, Christian. 2009. *Jamopp: The java model parser and printer*. Tech. rept. Fakultät Informatik, Technological University of Dresden, Germany.

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

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

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

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

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

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

Kurtev, I., Bézivin, J., & Akşit, M. 2002 (October). Technological Spaces: An Initial Appraisal. *Pages 1–6 of: International Conference on Cooperative Information Systems (CoopIS), DOA'2002 Federated Conferences, Industrial Track, Irvine, USA*.

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

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

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

Nickel, Ulrich, Niere, Jörg, & Zündorf, Albert. 2000. The FUJABA environment. *Pages 742–745 of: Proceedings of the 22nd international conference on Software engineering*. ACM.

Noack, T. 2013. Automatic Linking of Test Cases and Requirements. *Pages 45–48 of: 5th International Conference on Advances in System Testing and Validation Lifecycle, Venice, Italy*.

OMG. 2015. OMG Meta Object Facility (MOF) Core Specification Version 2.5. *Final Adopted Specification (June 2015)*.

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

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

Seidewitz, Ed. 2003. What models mean. *IEEE software*, 26–32.

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

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

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

Trollmann, Frank, & Albayrak, Sahin. 2015. Extending Model to Model Transformation Results from Triple Graph Grammars to Multiple Models. *Pages 214–229 of: Theory and Practice of Model Transformations*. Springer.

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