TECHNISCHE
UNIVERSITÄT
DARMSTADT

Definition of a Type System for Generic and Reflective Graph Transformations

Vom Fachbereich Elektrotechnik und Informationstechnik
der Technischen Universität Darmstadt
zur Erlangung des akademischen Grades
eines Doktor-Ingenieurs (Dr.-Ing.)
genehmigte Dissertation

von

Dipl.-Ing. Elodie Legros

Geboren am 09.01.1982 in Vitry-le-François (Frankreich)

Referent: Prof. Dr. rer. nat. Andy Schürr
Korreferent: Prof. Dr. Bernhard Westfechtel
Tag der Einreichung: 17.12.2013
Tag der mündlichen Prüfung: 30.06.2014

D17
Darmstadt 2014

Schriftliche Erklärung

Gemäß §9 der Promotionsordnung zur Erlangung des akademischen Grades eines Doktors der Ingenieurwissenschaften (Dr.-Ing.) der Technischen Universität Darmstadt

Ich versichere hiermit, dass ich die vorliegende Dissertation allein und nur unter Verwendung der angegebenen Literatur verfasst habe. Die Arbeit hat bisher noch nicht zu Prüfungszwecken gedient.

Frederiksberg (Dänemark), den 17.12.2013

. . . . . .

Elodie Legros

Acknowledgment

My very special thanks go to my advisor, Prof. Dr. rer. nat. Andy Schürr, who has supported me during all phases of this thesis. His many ideas and advices have been a great help in this work. He has always been available every time I had questions or wanted to discuss some points of my work. His patience in reviewing and proof-reading this thesis, especially after I left the TU Darmstadt and moved to Denmark, deserves my gratitude. I am fully convinced that I would not have been able to achieve this thesis without his unswerving support. For all this: thank you!
Another person I am grateful to for his help is Prof. Dr. Bernhard Westfechtel. Reviewing a thesis is no easy task, and I want to thank him for having assumed this role and helped me in correcting details I missed in my work.

I had the chance to work in a friendly atmosphere with very nice colleagues in the Real-Time Systems Lab at the TU Darmstadt. It helped me in persevering in the long journey this thesis has been, and I feel very thankful for it. I would like to address a special thank to Carsten Amelunxen and Felix Klar as co-writers of publications. They helped me a lot in developing ideas, concept and examples which are now integrated in this thesis. I also want to thanks Martin Wieber, not only for the cooperation on the MAJA project, but also for having been such a kind and cheerful colleague.
I also want to thank Dr. Ingo Stürmer, Anna Trögel and Jae-Won Choi (Model Engineering Solutions) for the collaboration in the MATE/MAJA projects, i.e. the context and motivation for my work. Speaking about the MAJA project, I am grateful to the Federal Ministry of Education and Research (BMBF) who supported this project by providing research funding.
My thanks go to the Software Engineering Research Group Kassel for having been such a great help for all questions concerning the SDM graph transformations. I am grateful to the Technische Universität Darmstadt too for the research grants which allowed me to start this thesis.

Finally, I want to thank my boyfriend for his patience each time I had to say *"no"* to a trip or just a walk because I had to work on my thesis. Last but not least, my parents deserve my gratitude for their patience and support. They heard the word *"thesis"* for so many years almost every time I called them, and every time they cheered me up. *Merci à tous les deux, Papa et Maman!*

Abstract

This thesis presents the extension of the graph transformation language SDM (Story Driven Modeling) with generic and reflective features as well as the definition of type checking rules for this language. The generic and reflective features aim at improving the reusability and expressiveness of SDM, whereas the type checking rules will ensure the type-safety of graph transformations.
This thesis starts with an explanation of the relevant concepts as well as a description of the context in order to provide the reader with a better understanding of our approach. The model driven development of software, today considered as the standard paradigm, is generally based on the use of domain-specific languages such as MATLAB Simulink and Stateflow. To increase the quality, the reliability, and the efficiency of models and the generated code, checking and elimination of detected guideline violations defined in huge catalogues has become an essential, but error-prone and time-consuming task in the development process. The MATE/MAJA projects, which are based on the use of the SDM language, aim at an automation of this task for MATLAB Simulink/Stateflow models. Modeling guidelines can be specified on a very high level of abstraction by means of graph transformations. Moreover, these specifications allow for the generation of guideline checking tools. Unfortunately, most graph transformation languages do not offer appropriate concepts for reuse of specification fragments - a MUST, when we deal with hundreds of guidelines. As a consequence we present an extension of the SDM language which supports the definition of generic rewrite rules and combines them with the reflective programming mechanisms of Java and the model repository interface standard JMI.
Reusability and expressiveness are not the only aspects we want to improve. Another fundamental aspect of graph transformations must be ensured: their correctness in order to prevent type errors while executing the transformations. Checking and testing the graph transformations manually would ruin the benefit obtained by the automation of the guideline checking and by the generic and reflective features. Therefore, we propose in this work a type-checking method for graph transformations. We introduce a new notation for rules of inference and define a type system for SDM. We also proposed an algorithm to apply this type system.
We illustrate and evaluate both contributions of our work by applying them on running examples. Proposals for other additional SDM features as well as for possible improvements of our type checking open new perspectives and future research to pursue our work.

Keywords: Graph transformations, SDM, generic, reflective, type checking, rules of inference

Zusammenfassung

Diese Arbeit stellt die Erweiterung der Graphtransformationssprache SDM (Story Driven Modeling) mit generischen und reflektiven Features sowie die Definition eines Typsystems vor. Die generischen und reflektiven Features haben zum Ziel die Verbesserung der Wiederverwendung und Ausdrucksfähigkeit von SDM, während die Ableitungsregeln (oder Inferenzregeln) des Typsystems die Typsicherheit der Graphtransformationen gewährleisten.
Diese Dissertation fängt mit einer Beschreibung der relevanten Begriffe sowie des Kontextes an, um dem Leser ein besseres Verständnis unseres Ansatzés zu ermitteln. Die modellgetriebene Softwareentwicklung, die heutzutage als Standard gilt, basiert generell auf domainspezifischen Sprachen wie MATLAB Simulink und Stateflow. Um die Qualität, die Verlässlichkeit und die Effizienz von Modellen und von dem generierten Code zu garantieren, sind die Überprüfung und die Behebung der verletzten Modellierungsrichtlinien eine notwendige, aber fehleranfällige und zeitaufwändige, Aufgabe während des Entwicklungsprozesses. Das Ziel der MATE/MAJA-Projekte, die auf der Verwendung der SDM-Transformationssprache basieren, ist die Automatisierung dieser Aufgabe für MATLAB Simulink/Stateflow Modelle. Die Modellierungsrichtlinien können auf einem hohen Abstraktionsniveau mit Hilfe von Graphtransformationen spezifiziert werden. Diese Spezifikationen ermöglichen dazu die Generierung von Werkzeugen zur Richtliniensanalyse. Leider bieten die meisten Graphtransformationssprachen die zur Wiederverwendung geeigneten Konzepte nicht an - ein MUSS, wenn man sich mit Hunderten von Richtlinien beschäftigt. Deshalb stellen wir eine Erweiterung der SDM-Sprache, die die Spezifikation von generischen Graphersetzungsregeln unterstützt, und die diese mit den reflektiven Programmierungsmechanismen von Java und dem Standard JMI kombiniert, vor.
Wiederverwendung und Ausdrucksfähigkeit sind nicht die einzigen Aspekte, die wir verbessern möchten. Ein anderer wichtiger Aspekt der Graphtransformationen muss geprüft werden: ihre Korrektheit, um Typfehler bei der Ausführung der Transformationen zu unterdrücken. Eine manuelle Überprüfung der Graphtransformationen würde den Vorteil der automatisierten Richtliniensanalyse und der generischen und reflektiven Features zunichte machen. Deshalb schlagen wir in dieser Arbeit einen Ansatz zur Typüberprüfung von Graphtransformationen vor. Wir führen eine neue Notation für die Inferenzregeln ein und definieren ein Typsystem für SDM.
Wir veranschaulichen und evaluieren die beiden Beiträge, indem wir diese auf konkreten Beispielen anwenden. Vorschläge für weitere SDM-Features sowie mögliche Verbesserungen von unserem Typsystem bieten Anregungen für künftige Arbeite an.

Stichwörter: Graphtransformationen, SDM, Generizität, Reflektivität, Typsicherheit, Inferenzregeln

Contents

Chapter 1

Introduction

Nowadays, model-driven development (MDD) is common practice within a wide range of software development projects. According to this approach, models are not simply blueprints or sketches, but artifacts integrated in the different phases of the development process. In accordance with the increasing complexity of the systems to be developed, the models have become huge and intricate. In addition, the development process does not include a single step and a single model. Consequently, model transformations are required to ensure the traceability from model to model, as well as to ensure consistency when manipulating a model. Model transformations can be expressed and executed in different ways and in various contexts. The work for this dissertation concentrates on the graph transformation language SDM (Story Driven Modeling) [Zün01]. Our work namely aims at improving different aspects of this language: reusability, expressiveness and type-safety. Before describing our contribution, let us describe the context which motivated our work.

1.1 Context and Motivation

In the context of automotive embedded software development projects, the standard modeling language UML does not meet the requirements of the developers and, therefore, is neglected in favor of the MathWorks Matlab Simulink/Stateflow (Matlab SL/SF) environment which is better adapted for specifying, designing, implementing, and checking the functionality of new control functions. To improve the correctness and the efficiency of models and prevent typical modeling problems, generally accepted modeling guidelines such as the MathWorks Automotive Advisory Board catalogue [MAA] are usually adopted. These modeling guidelines are however numerous and, for the huge models which are common nowadays, this can add up to a few hundreds or even thousands of violations that must be corrected manually by the modeler. As a consequence, the analysis and correction of models can become a time-consuming and error-prone tasks. The MATE/MAJA projects which are described in this work have originally be motivated by the urgent need

for automation of these tasks. In addition to the urgent need for such a tool, another motivation for starting the MATE project was the observation of the very low level of abstraction concerning the implementation of modeling guidelines for which imperative programming languages are generally used. The MATE/MAJA projects are based on the use of graph transformations which offer a well-suited support for the specification and implementation of modeling guidelines and refactorings. More precisely, the language used to define graph transformations is the SDM language.
SDM is pretty well-suited to specify modeling guideline analysis and correction. Though, we are not completely satisfied with the specification of some kinds of modeling guidelines with the currently used graph transformation language SDM. For instance, numerous guidelines require the definition of very similar graph transformations, which is a repetitive and time-consuming task for the developer. We are convinced that reusability and expressiveness of graph transformations could be significantly improved by adopting the concepts of genericity and reflection from standard programming languages such as Java.

Reusability and expressiveness are not the only aspects of SDM we want to improve. Another fundamental aspect of SDM must be ensured: the correctness of graph transformations. More precisely, we want to inspect the model transformations statically in order to prevent type errors at runtime when the model transformations are executed.
Type error is a well-known concept in the context of programming languages, but it is not the case in the context of model transformations. Let us explain what we call type error in this work.
A DSL (Domain-Specific Language) such as MATLAB Simulink/Stateflow whose metamodel will be described in this work has syntax and semantics. The abstract syntax, i.e. the metamodel, defines the structure of its models, and contains the elements that can be used to create syntactically correct models. The static semantics corresponds to well-formedness rules for the models, i.e. constraints to create semantically correct models The DSL's meaning and behavior is described by its dynamic semantics (or execution semantics). Please note that the definition of dynamic semantics for the MATLAB Simulink/Stateflow language is out-of-scope in the context of this work.
The transformation rules are defined over the metamodel whose instances are sources and targets of the transformations. Thus, we have to ensure that the transformation produces a metamodel-compliant target model from a metamodel-compliant source model. Else, an error will occur. For instance, if the metamodel defines a class with a String attribute, the corresponding attribute's value in a metamodel-compliant source or target model must be a String. Then, if we define a transformation which assigns a Boolean value to this String attribute, the resulting target model will not be semantically correct. This error propagates to the code generated from the graph transformation and type errors occur at runtime when executing this code.

Although the adoption of genericity concepts would improve the reusability of graph transformation, we can notice that it increases the risk of type errors. In addition, it makes the detection of type errors more difficult because type information, namely the type of the parameterized elements, are only known when the model transformation is called. Check and testing the graph transformations manually would ruin the benefit obtained by the generic feature, namely the time and effort spared in drawing diagrams. Therefore, we need to define a formal method to detect type errors in graph transformations statically, i.e. before they propagate to the generated code and errors occur when executing the transformations.

1.2 Contribution

The standard process in MDD consists in generating code automatically from models instead of implementing code manually. The generated code may be complete, or may be a skeleton to be completed, the completeness of the generated code depending on the completeness of the model. In the context of the MATE/MAJA projects, we generate Java code from the SDM specifications of the modeling guidelines.
Genericity and reflection are concepts which have become part of Java for a long time. Generic Java has been created in 1998 as an extension to the Java languages to support generic types [gen]. Generics have been added to the Java programming language in 2004 in the third edition of the Java Specification Language [JGB05] This edition of the Java Specification Language introduced the `java.lang.reflect` package too which allows for the use of reflection by means of reflective interfaces.
In the context of the MATE/MAJA projects, the code is generated from the metamodel and the model transformations using JMI (Java Metadata Interface). JMI is a standard mapping between MOF and Java, and provides so-called reflective interfaces which support the manipulation of a metamodel even if its tailored interface is unknown. Consequently, JMI provides a support for genericity and reflection in the code generated from the metamodel and the model transformations. Though, the current SDM syntax does not provide any way to express generic or reflective graph transformations, and thus does not benefit from the generic and reflective support in the generated code.
That is why we need to modify or extend the SDM syntax in order to allow for the specification of generic and reflective graph transformations. Reusable and more expressive specification of modeling guidelines would be of benefit in the context of the MATE/MAJA projects.

In addition to the extension of the SDM syntax with generic and reflective features, we define a type system in order to inspect the graph transformation and detect type errors. A type system is the part of a typed language which keeps track of the type of variables and, in general, of the types of all expressions in a program.

We need to develop a formal method to statically check the type-safety of graph transformations. The approach we present in this work is based on inference systems, i.e. on the application of so-called type rules which are specified as rules of inference. A collection of type rules form a formal type system. By means of these rules of inference, we can extract type information and thus ensure that the graph transformation and the method calls cannot cause any error. Because we are working with a visual language, the premises of a type rule can be visual information from the metamodel or the graph transformation and/or a textual type information derived from the application of another rule. In addition, as we will see in this thesis, type errors can have different causes: incorrect graph transformations, or arguments which do not respect the graph transformation specifications or which cause type errors. In order to determine more precisely the kind of type error as well as to combine the visual and textual premises, we present in this thesis a new notation for the rule of inference of the type system we are defining.

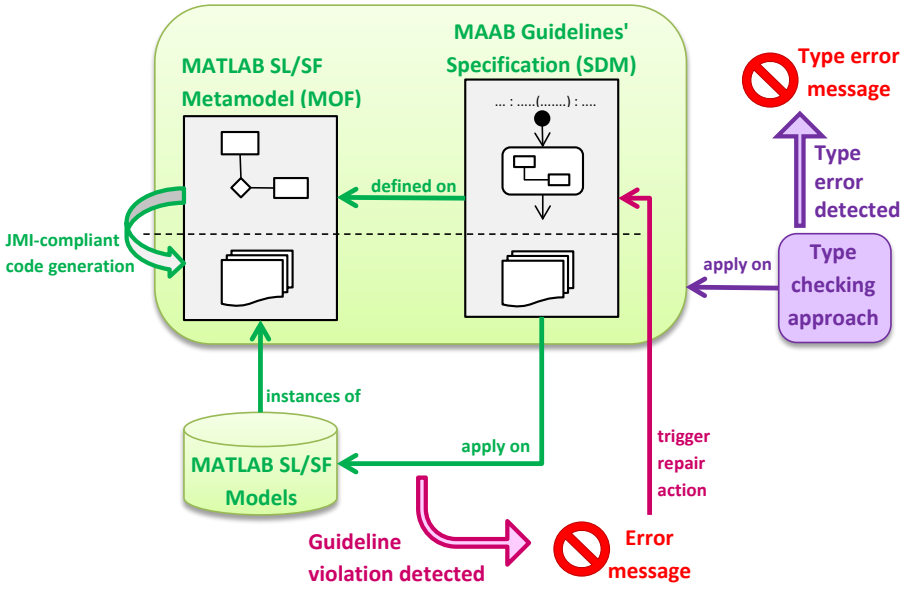


Figure 1.1: Overview of our Application and Contribution

Fig. 1.1 illustrates the context of our work as well as the approach described in this dissertation. The MATLAB SL/SF metamodel is specified as MOF-compliant metamodel, and the guidelines are defined on this metamodel by means of SDM graph transformations. A JMI-compliant code is generated from the metamodel and the graph transformations. The MATLAB SL/SF models are instances of the metamodel. When applying the graph transformations on these models, we can detect guideline violations and produce the corresponding error messages. Our type checking approach applies on the metamodel and the graph transformations. Error messages are created when type errors are detected. Our approach will be described more precisely in this work.

1.3 Overview

We present in Chapter 2 relevant concepts for this work in order to provide the reader with a better understanding of our approach. We first introduce the Model-Object Facility (MOF) and its standard mapping to Java called the Java Metadata Interface (JMI). We then describe what model transformations, and particularly graph transformations, are. The third section describes the language MOSL. This description is essential since our work consists of the extension of its transformation language SDM, and of the definition of its type system. The fourth section presents related transformation languages in order to offer a comparison with other tools and languages Finally, since the second aspect of our work is the definition of a type checking system for MOSL, we provide an overview of the main concepts about type systems and type checking approaches in a last section.
Chapter 3 describes the MATE/MAJA projects which motivated our work and will be used in the following as application of our approach. We first explain the context and the purpose of these projects, i.e. the analysis and correction of MATLAB Simulink/Stateflow model according to modeling guidelines. We describe in a second section the MATLAB metamodel which is at the core of the project. In the last section, we illustrate by means of some examples how guidelines can be specified as SDM graph transformations on the MATLAB metamodel. We then compare SDM with other specification languages.
This comparison aims at pointing out the need for the extension of SDM with the additional features which are described in Chapter 4. The first section of Chapter 4 presents the new language constructs for generic transformations, whereas the second section presents the new language constructs for reflective transformations. We compare these language extensions with related works in a third section.
Chapter 5 is dedicated to the second aspect of our work, namely our type checking approach of SDM graph transformations. In a first section, we explain by means of examples to what extent the definition of such a type checking system is necessary. We then present our approach in a second section, before we describe the rules of inference composing the type system in a third section. We explain in a fourth section the way our type checking approach can be executed and illustrate it with an application example. We finally compare our approach with related works in a fifth section.
Chapter 6 presents in a first section an application of both aspects of our work, namely the type checking system of a graph transformation which is using the new generic feature. The second section proposes additional features for SDM which could be useful before evaluating the benefits and limitation of the extended SDM syntax. The third section is dedicated to the evaluation of our type checking approach.
Chapter 7 concludes this thesis with a summary and an outlook.

Chapter 2

Fundamentals

In this chapter, we explain the following concepts in order to provide a better understanding of our approach presented in this thesis. We first introduce Model-Driven Development, and more precisely the Meta-Object Facility (MOF) and its standard mapping to Java which is called the Java Metadata Interface (JMI). In a second section, we describe what model transformations, and particularly graph transformations, are. The language MOSL is introduced in a third section.The description of MOSL is essential in this thesis since our works consists of the extension of this language's transformation language, i.e. SDM, and the definition of its type system. We present in a fourth section related transformation languages in order to provide a comparison with other tools and languages. Finally, since our work aims at defining a type checking for the graph transformation of MOSL, we provide an overview of the main concepts about type systems in a last section.

2.1 Meta-Object Facility and Java Metadata Interface

Nowadays, model-driven development (MDD) is common practice within a wide range of software development projects.
Although standard solutions are not necessary optimal solutions, standardization provides two mains advantages. On the one hand, the standardization of languages and technologies facilitates their broad application that contributes to their further development and improvement. On the other hand, the standardization allows for the data exchange between models and facilitates team work. Therefore, in order to encourage an efficient use of MDD, the Object Management Group (OMG) [OMGb], an international organization which provides standards widely used in the industry, introduced the Model Driven Architecture (MDA) as software design approach for the development of software systems.
MDA implies the use of modeling languages, e.g. the Unified Modeling Language (UML), as well as the use of model transformations. A formal definition for the UML is the so-called Meta Object Facilities (MOF).
Java is an object-oriented programming language, which is widely used in the in-

dustry, and the Java Metadata Interface (JMI) is the standard mapping of MOF to Java. Thus, it provides a standardized implementation for MOF-compliant models and metamodels.

2.1.1 Model Driven Engineering

Model Driven Engineering (MDE) is a software development methodology based on the use of models [Sch06]. Models are not simply blueprints or sketches, but artifacts integrated in the different phases of the development process. The models used in MDE make sense from the user's point of view and serve as basis for implementing systems. Code may be written by hand from a detailed model in a separate step. Though, the software systems to be developed are more and more complex. As a consequence of this increasing complexity, the models have become huge and intricate. Therefore, the standard process in MDE consists in generating code *automatically* from models instead of implementing code manually. The generated code may be complete, or may be a skeleton to be completed. The completeness of the generated code depends on the completeness of the model. For instance, it is possible to generate a deployable product from a complete model including executable actions.
The Model Driven Architecture (MDA) [MDA] is an initiative of the Object Management Group (OMG) [OMGb]. The OMG is an international organization considered as the most legitimate standardization instance in the domain of modeling. It provides a wide range of standards which are widely used in industry.

The MDA is a proposal to support the MDD, aiming at portability, interoperability and reusability [GY06]. To this purpose, MDA defines three kinds of models:

- **Computation-Independent Model (CIM)** - sometimes called domain model or business model: it describes the situation in which the system will be used, independently of its implementation.
- **Platform-Independent Model (PIM)**: it describes the system functionality using an appropriate domain specific language. There are no information about the technical details or the building of the solution for a specific platform. A PIM is namely suitable to any kind of platform.
- **Platform-Specific Model (PSM)**: it describes a solution from a particular platform perspective.

Fig. 2.1 illustrates the concept of MDA and the position of the MDA models within the development process. The development process includes not necessarily a single PIM and a single PSM. If the system to be developed is complex, the gap between models can be too large to perform a direct transformation, and several successive CIM, PIM and PSM may be necessary.
The main advantage of the formalization proposed by the MDA concerns the transformation from model to model. The use of formal models improves greatly the

efficiency of these transformations since, without formal models, it is not possible to define a formal transformation which can be (partly) automated. In Section 2.2, we will present model transformations based on graph rewriting.

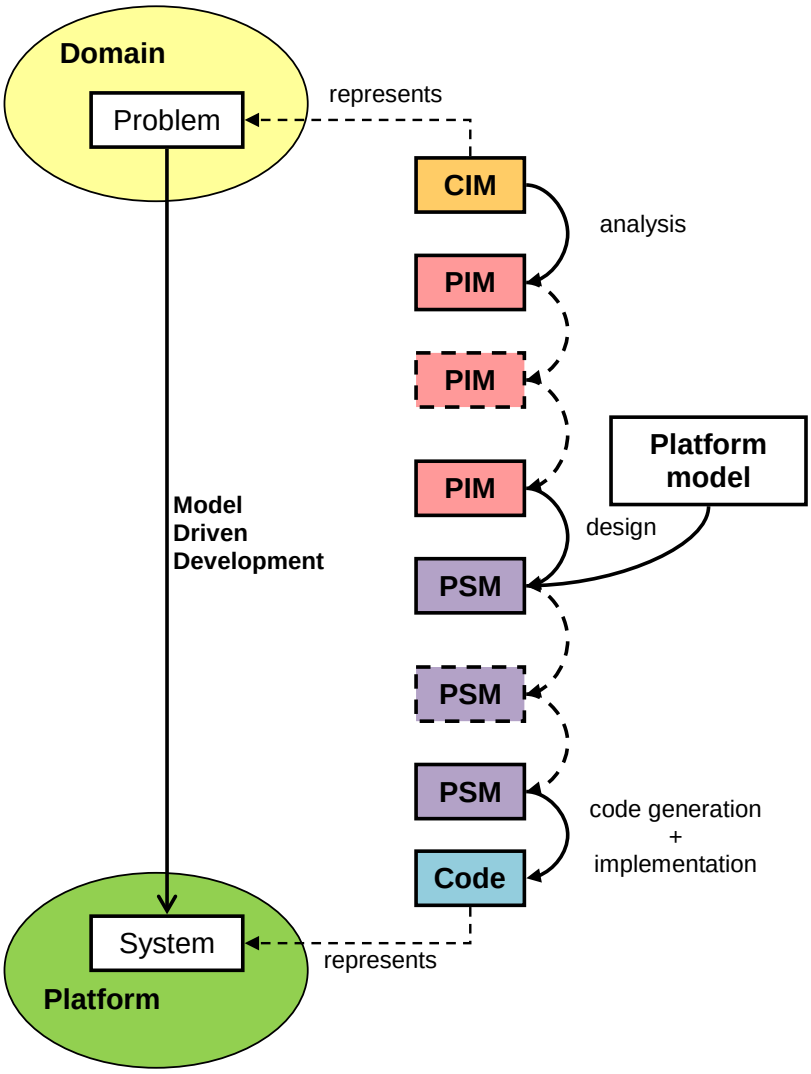


Figure 2.1: Concept of the MDA

2.1.2 Meta Object Facilities

Maybe one of the most popular modeling languages developed by the OMG is the Unified Modeling Language UML [Obj05a]. The UML is composed of a large set of visual modeling languages for the specification of the structure and behavior of a system by means of use-case diagrams, sequence diagrams, etc. One of the most

popular UML sublanguages is the widely used and accepted class diagram. The first versions of UML have not been formally defined, allowing too much room for interpretation. The OMG proposed the Meta-Object Facility MOF as response to the need for a formal definition of UML [OMGa].

MOF is designed as a four-layered architecture. Fig. 2.2 illustrates such a metamodeling hierarchy that extends the traditional object-oriented modeling over several levels. The MOF architecture provides a metametamodel at the top layer, called the M3 layer. This M3-model is the language used by MOF to build metamodels, called M2-models. The most known M2-model is the UML metamodel. These M2-models describe elements of the M1-layer, and thus M1-models. The last layer is the M0-layer, and is used to describe real-world objects. Because the consistency of the definition of all lower levels depends on the formal definition of the highest level, the M3-level, i.e. MOF as language, needs also to be formally specified. Because the demands on a formal description of UML do not considerably differ from the demands on formally describing MOF, MOF can be described in the same way as UML. Consequently, the M3 layer describes itself instead of defining an additional higher level very similar to this M3 level (and, recursively, an infinite number of almost identical abstraction level). MOF, as specification language on M3, is denoted to be *self-describing*, and MOF, as architecture, is a *closed* metamodeling architecture.

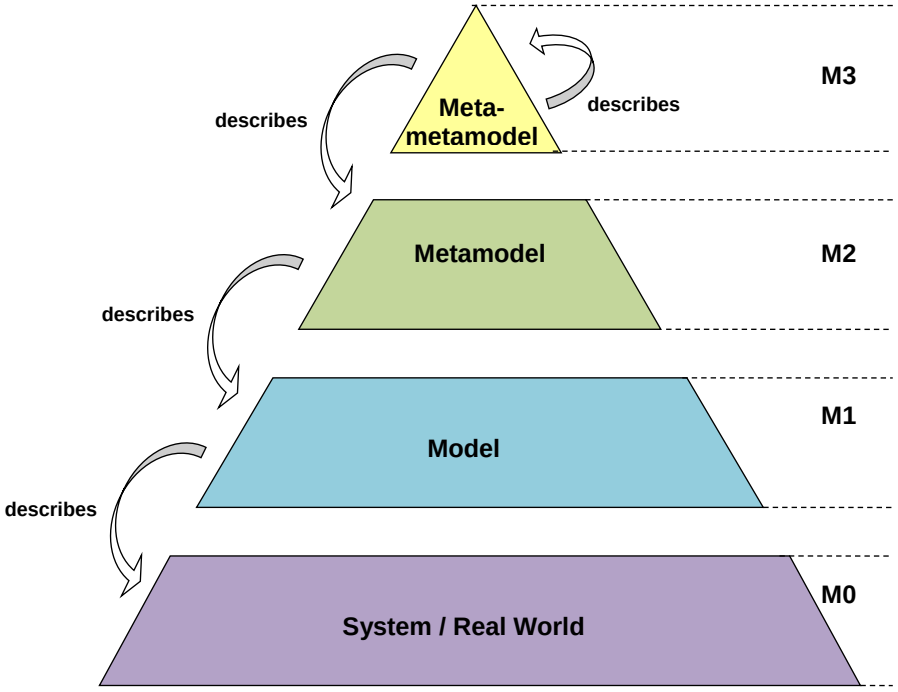


Figure 2.2: Metamodeling Hierarchy

MOF is a *strict* metamodeling architecture. This means that every model element on every layer is strictly in correspondence with a model element of the layer above. The original purpose of MOF was to provide a formal specification for UML. In fact, MOF allows for more because it provides a means to define the structure, or abstract syntax, of modeling languages, and especially of domain-specific languages (DSL). A DSL is a specification language dedicated to a particular domain or a particular problem, and hence, satisfy better the needs of software developers. An example of DSL (Matlab Simulink/Stateflow) and its specification as MOF-compliant metamodel will be presented in Section 3.2.

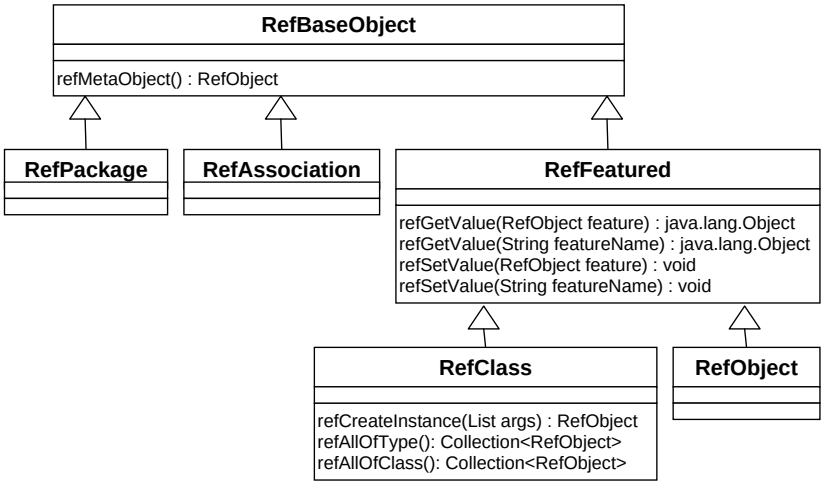


Figure 2.3: JMI reflective interfaces

2.1.3 Java Metadata Interface

The Java Metadata Interface (JMI) [Dir02] is a standard mapping between MOF and Java. It provides two kinds of interfaces: the tailored and the reflective interfaces. A metamodel is JMI compliant if, and only if, the tailored interfaces extend the JMI reflective interfaces. The tailored interfaces are generated for a given metamodel while the reflective interfaces are the same for all metamodels. They also support the manipulation of a metamodel even if its tailored interface is unknown, and hence, facilitate the communication between tools. An example of application of JMI is the adapter for MATLAB presented in Section 3.
Fig. 2.3 represents the JMI reflective interfaces and the reflective methods that are the most relevant for this work.

- **RefBaseObject** is the interface extended by all the other reflective interfaces. It represents any element of the metamodel. Its method *refMetaObject* returns the metaclass of the calling object.

- **RefPackage** is the interface of the packages.

- **RefAssociation** is the interface providing the metaobject description of an association. It also provides generic methods to query and update the links belonging to the association.
- **RefFeatured** provides the metaobject description of instances and of class proxy objects. Its methods *refGetValue* and *refSetValue* are getter and setter providing a generic manipulation of the objects' properties.
- **RefClass** is the interface of the proxy, i.e. of the "factory" which is responsible for the creation (by means of the method *refCreateInstance*) and management of objects.
- **RefObject** corresponds to instance objects.

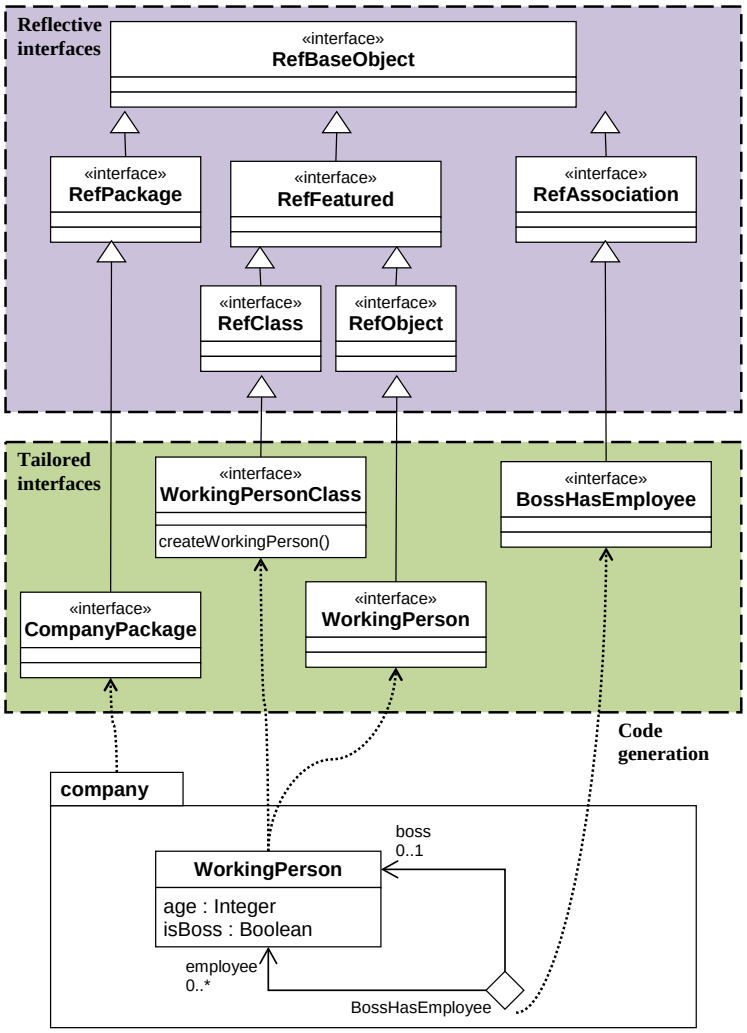


Figure 2.4: JMI example

The JMI standard defines not only this concept of interfaces, but describes also precisely which patterns the name of the interfaces and their methods must respect. Let us consider the example of Fig. 2.4. For the sake of clarity, only few tailored methods are depicted in this figure. We use the simple example previously used in Section 2.3. The metamodel is composed of a package called `company` and containing the class *WorkingPerson* connected to itself by an association called *BossHasEmployee*. As explained, both kinds of interfaces are generated. According to the JMI standard, the tailored interfaces extend the reflective interfaces which are metamodel-independent. The tailored interfaces are generated according to the JMI templates. Therefore, the interface generated for the package `company` is called *CompanyPackage*, i.e. the name of the package followed by the suffix *Package*. This interface extends the reflective interface *RefPackage*. In the same way, the interface *BossHasEmployee*, which extends *RefAssociation*, is generated for the association *BossHasEmployee*. Two interfaces are generated for the class *WorkingPerson*. On the one hand, the proxy interface which is responsible for the creation and management of instances, and on the other hand, the interface corresponding to an instance of the class. The proxy interface is *WorkingPersonClass*, i.e. the name of the class followed by the suffix *Class*, and it extends the reflective interface `RefClass`. The instance interface has simply the name of the class, i.e. *WorkingPerson*. JMI defines also a range of methods and naming convention for these methods. For instance, the method of the proxy *PersonWorkingClass* which is responsible for the creation of an instance of `WorkingPerson` is called *createWorkingPerson*, i.e. the prefix *create* followed by the class name.
A description of the complete range of JMI templates is here out-of-scope and can be found in the JMI documentation [Dir02].

2.2 Model Transformations based on Graph Rewriting

As explained in Section 2.1.1 and shown by Fig. 2.1 in the context of model-driven architecture, models are not simply sketches, but artifacts integrated in the different phases of the development process. Thus, model transformations are required to ensure the traceability from model to model, as well as to ensure consistency when manipulating a model. There are different kinds of model transformations. In the context of this thesis, we will concentrate on graph rewriting systems.

2.2.1 Model Transformations

[CH03] presents model transformation approaches in the form of feature diagrams as depicted in Fig. 2.5. A transformation rule consists of two parts: a Left-Hand Side (LHS), and a Right-Hand Side (RHS). The LHS accesses the source model, and the RHS expands the target model. LHS and RHS are composed of variables and logical expressions. Variables hold elements from the source and/or target models, and logical expressions define computations and constraints on model el-

ements. The logic may be non-executable or executable. Executable logic can be expressed in a declarative as well as in an imperative form. LHS and RHS may be defined with the help of patterns which are model fragments. Other variation aspects of transformation rules are bidirectionality, i.e. whether a rule may be executable in the inverse direction, and rule parameterization.

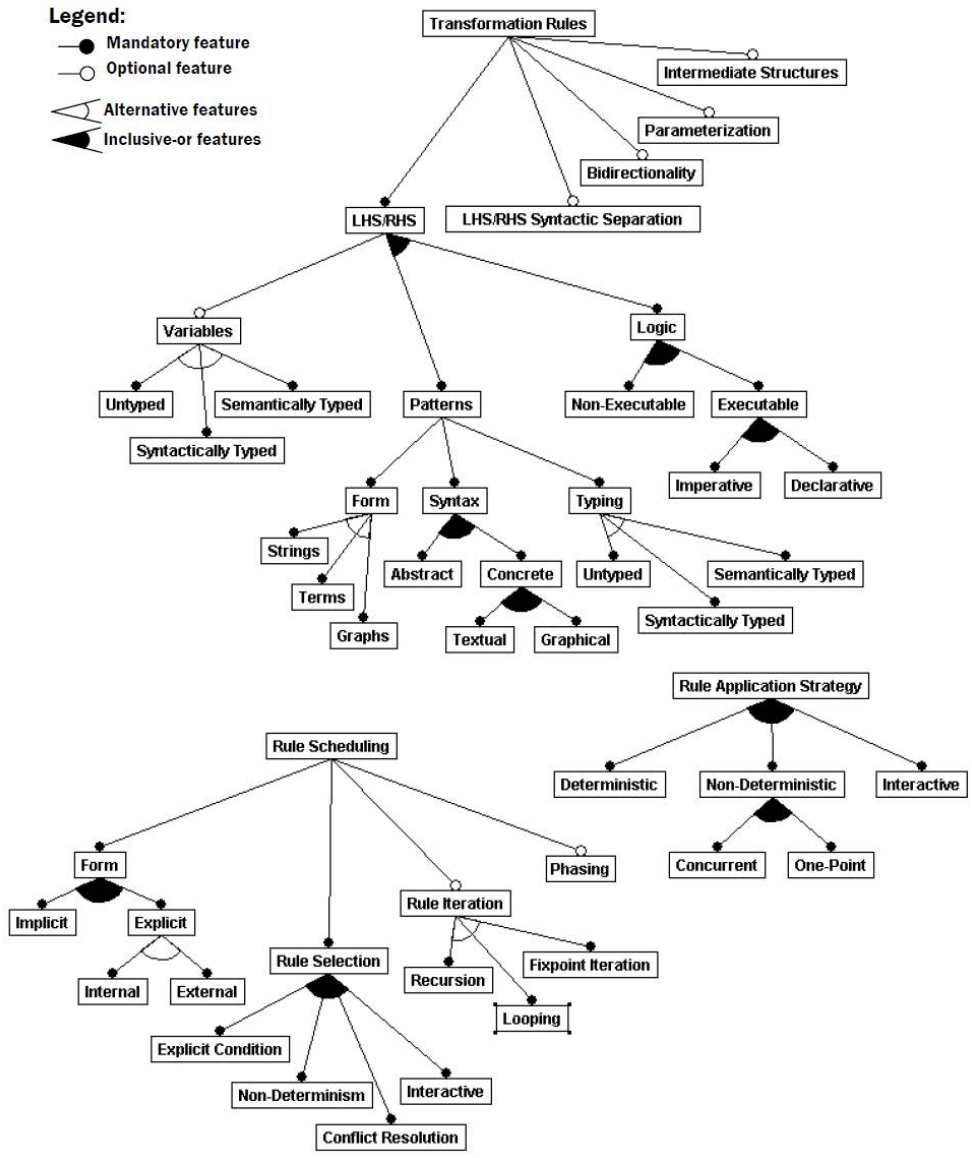


Figure 2.5: Feature Diagrams of Transformation Rules, Rule Scheduling, and Rule Application Strategy [CH03]

Scheduling mechanisms determine the order in which the rules are applied. The scheduling can be specified implicitly or explicitly. Implicit scheduling can be system-defined or user-defined. Explicit scheduling requires dedicated constructs to control the execution order. The rules can be selected by an explicit condition or by means of resolution mechanisms (e.g. based on priorities). Rule iteration mechanisms consist of recursion, looping, and fixpoint iteration (i.e. repeated application as long as changes are possible).
Multiple transformation rules may be organized by means of modularity or reuse mechanisms. Modularity mechanisms support rule packaging into modules which can import other modules. Reuse mechanism allows for the definition of rules based on one or more existing rule.
Model transformations may be qualified as being endogenous vs. exogenous, and horizontal vs. vertical [MCG05]. In order to transform models, the models need to be expressed in some modeling language whose syntax and semantics is defined by a metamodel (Cf. the principle of MOF in Section 2.1.2). A model transformation between two models with a common metamodel is called *endogenous*. The counterpart of endogenous model transformations are *exogenous* model transformations, i.e. between two models with different metamodels. Refactoring or specification of method behavior are typical examples of endogenous model transformations. A PIM-to-PSM transformation in the MDA usually is an example of an exogenous model transformation. Model transformations may also be differentiated between horizontal and vertical transformations. The source and target models of a *horizontal* transformation reside at the same abstraction level, whereas a *vertical* transformation occurs between two different abstraction levels. For instance, a model refinement of a model is a vertical transformation. A graph transformation can be *multilevel* too. Please note that multilevel does not mean vertical. The source and target models of a multilevel transformation can reside at the same or different abstraction levels. A multilevel transformation is a transformation which combines several meta-levels in the same transformation, e.g. giving a read access to next higher meta-level in order to get meta-information.
A model transformation language can be defined by means of a metamodel. That means that a model transformation can be a model, and, thus, be an input or an output of a model transformation. A transformation which has a model transformation as input or output belongs to the so-called Higher-Order Transformations (HOTs) [TJF+09]

2.2.2 Graph Rewriting Systems

Graphs and diagrams provide a powerful approach to visualize and structure a wide range of problems, especially in the domain of software engineering. Well-known visual notations such as entity-relationship diagrams [CC02], Petri nets [Mur89], and the different kinds of UML diagrams [Obj05b], are nowadays a standard in computer science. These notations produce models that can be easily considered as graphs. As a consequence, specification of model behavior and model manip-

ulation are defined as graph transformations. The first proposals in this direction already appeared in the early seventies [Pra71].
A *graph* is an ordered pair *G=(V,E)*, comprising a set V of *vertices* (also called *nodes*) and a set E of *edges* [Die10]. The nodes represent the objects, and the edges model the relationships between objects. Graphs representing UML-like models are *typed*, *attributed* and *labeled* [Hec06]. A labeled graph is a graph whose edges and nodes have given labels. The concept of typed graphs implies the existence of two kinds of graph: a *type graph* which represents the type level, and its *instance graphs* which are the individual snapshots. In the context of modeling, the type graph corresponds to the metamodel, and the instance graphs to the models. A graph is designed as attributed when it contains attributes to store values of predefined data types. These attributes possess a type-level declaration *a:T* (where *a* = attribute name, and *T* = attribute type), and an instance level occurrence *a=v* (where *v* = value assigned to the attribute *a*). The relation between a type graph and an instance graph must comply with the following compatibility conditions:

1. for each object *o:C* in the instance graph, there must be a node type *C* in the type graph.

2. for each edge between objects *o1:C1* and *o2:C2*, there must be a corresponding edge between the nodes *C1* and *C2* in the type graph.

3. for each attribute *a=v* associated to the object *o:C*, there must be a declaration *a:T* in a node of type *C*, and *v* must be of type *T*.

Graph transformations are executed on instance graphs, but are defined at the level of the type graph. Consequently, they are applicable on *any* instance graph for a fixed type graph. A *graph grammar* is composed of a starting graph and a set of production rules. There are several approaches of graph rewriting such as the classical algebraic approach, the triple graph grammars, the recursive graph pattern matching [EGdL$^+$05a], the node replacement graph grammars, or the hyperedge replacement graph grammars [RG97]. We describe here shortly the algebraic approach, which is the most common. This approach is called “algebraic” because graphs are considered as special kinds of algebras. This approach is divided into sub-approaches, mainly the single-pushout approach and the more frequently used double-pushout approach. A pushout is an “algebraic construction” in the category of graphs and total graph morphisms, and defines here the gluing for graphs [RG97].
In the double-pushout approach (DPO), a production $gt = (L \supseteq K \subseteq R)$ is composed of three instance graphs *L*, *K*, and *R* over a given type graph *TG* whose structure is compatible. Compatible means that nodes with the same identity in *L*, *K*, and *R* have the same type and attribute in *TG*, and edges with the same identity in *L*, *K*, and *R* have the same type, source, and target in *TG*. The graph *L* is the LHS of the rule, i.e. the graph to be matched, and represents the pre-conditions. The graph *K* is the so-called *gluing* graph, or *invariant*. It is a subgraph of both *L* and

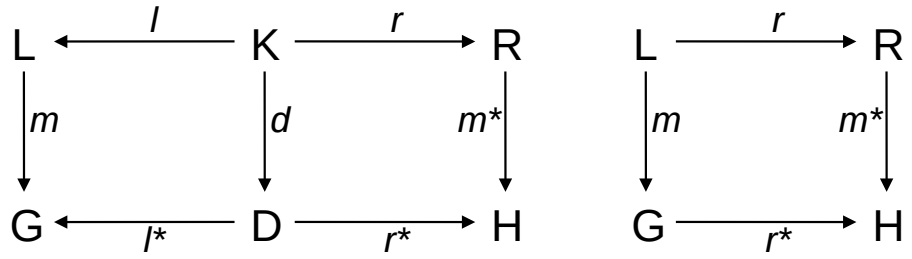


Figure 2.6: Double and Single Pushout - Morphism Diagrams

R, and is, therefore, composed of all elements common to the LHS and the RHS of the transformation rule. The graph *R* is the RHS of the rule, i.e. the graph *L* should be replaced with, and describes the post-conditions. Fig. 2.6.a shows the principle of DPO rule applied on an instance graph (the host graph) *G* and resulting in the graph *H*. The labeled arrows correspond to morphisms which are the algebraic representation of graph matching [RG97]. More precisely, the graph transformation occurs in three steps:

1. Find an occurrence (= a *match*) of the LHS pattern *L* in the host graph *G* (also, a morphism *m*).

2. Delete from *G* all nodes and edges that belong to L\ K, resulting into the graph *D*.

3. Add to the graph *D* a copy of $R \backslash L$, resulting into the derived graph *H*.

In the single-pushout approach (SPO), a production does not need a gluing graph. Thus, a rule can simply be written as $gt : L \rightarrow R$, and the corresponding partial morphism diagram is depicted by Fig. 2.6.b. When executing a graph transformation according to the SPO, a match of *L* in *G* has to be found similarly to the DPO approach. Then, this match is replaced by the image of *R*. Nodes and edges which do not occur in the RHS are rigorously deleted.
Because a graph transformation may result in the deletion of nodes, it must be ensured that the remaining structure after the transformation is still a graph, i.e. that no edges are left dangling after the deletion of their source or target nodes. The main difference between SPO and DPO is precisely the so-called dangling edge condition. The DPO approach requires that a node must not be removed if there are still edges to this node, whereas all dangling edges are just deleted when applying the SPO approach.

The next section describes a concrete example of transformation language which follows the SPO approach. The reader will also find other concrete examples of transformation tools and languages in Section 2.4.

2.3 MOSL

The MOFLON Specification Language (MOSL) is a language defined in [Ame09]. The interested reader can find an overview of this language in [AKRS06].
MOSL is composed of the Meta Object Facility (MOF 2.0 [Obj06a]) as schema language, of the Object Constraint Language (OCL 2.0 [Obj06b]) as constraint language, and of the Story Driven Modeling (SDM [Zün01]) as transformation language. MOSL supports also the Triple Graph Grammars (TGG [Kön05]) which is a technique for defining the correspondence between two different types of models. Though, considering the approach presented in this thesis, TGG is is out-of-scope, and will not be presented here.
We first give a short overview of MOF and OCL as modeling and constraint languages, before we describe more precisely the syntax of SDM in a next section. Finally, we present in a last section the meta-CASE-tool MOFLON which supports MOSL.

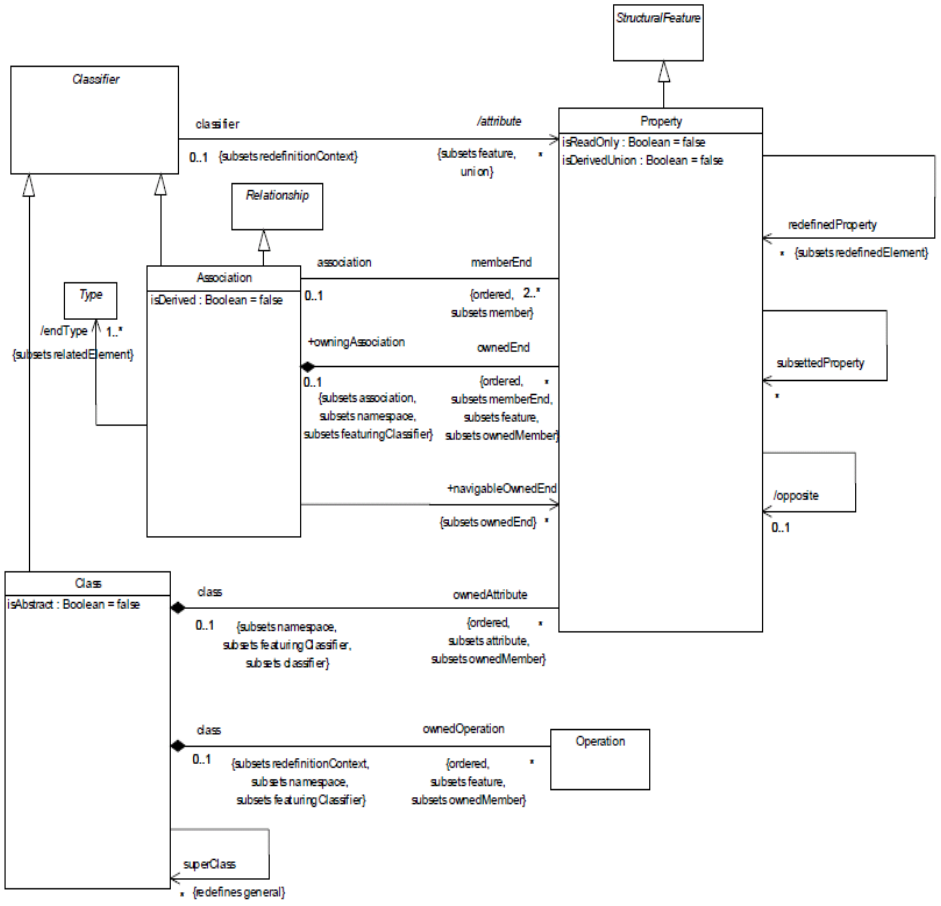


Figure 2.7: Classes diagram - UML Infrastructure [Obj05a]

2.3.1 MOSL Schema and Constraint Languages

Rather than defining a completely new specification language, Amelunxen composed MOSL from suitable existing ones in order to make MOSL as familiar, and hence as easy to use, as possible for the users. MOF 2.0 was adopted as schema language, and OCL 2.0 as constraint language.

MOF 2.0 as schema language

As explained in Section 2.1.2 with the concept of metamodeling, the original purpose of MOF is to provide (meta)modeling constructs for the specification of modeling languages. Basically, MOF 2.0 is a slightly extended subset of UML 2.0 class diagrams, and, hence, provides an optimized version of UML class diagrams for the purpose of metamodeling. The specification of UML class diagrams is sourced out in an own specification document called Infrastructure [Obj05a]. Due to reasons of synergy especially concerning maintenance, this specification is used by UML as well as by MOF 2.0.

Fig. 2.7 shows the specification of the UML/MOF class diagrams. The most important elements of the class diagram are *Class*, *Association*, *Operation*, and *Property*. In the following, we add the prefix MOF to design unambiguously the elements *Class*, *Association*, *Operation*, and *Property* of the MOF metametamodel (level M3). A *MOFClass* describes a set of objects that share the same specifications of features, constraints, and semantics. An instance of *MOFClass* may have any number of instance of *MOFOperation*. A *MOFOperation* specifies the name, type, parameters, and constraints for invoking an associated behavior. A *MOFProperty* may be related to *MOFClass* or to *MOFAssociation*. A *MOFProperty* related by *ownedAttribute* to *MOFClass* represents an attribute, and might also represent an association end. A *MOFProperty* related by *memberEnd* to *MOFAssociation* represents an end of the association. The type of the *MOFProperty* is the type of the end of the *MOFAssociation*.

Fig. 2.8 shows a concrete example of MOF metamodeling. The model in level of modeling M1 is composed of 3 instances of the class *WorkingPerson* which belongs to the level of metamodeing M2. This model represents a boss which has two employees. One instance of *WorkingPerson* is connected by links, instances of the association *BossHasEmployee*, to the both other instances. This class and this association are defined in the level of metametamodeling M3 which is itself defined using MOSL. The association *BossHasEmployee*, instance of *MOFAssociation*, is represented by a diamond with two association ends called *boss* and *employee*, instances of *MOFProperty*. The class *WorkingPerson* is instance of *MOFClass*, and possesses two attributes which are instances of *MOFProperty*: *age* of type Integer, and *isBoss* of type Boolean.

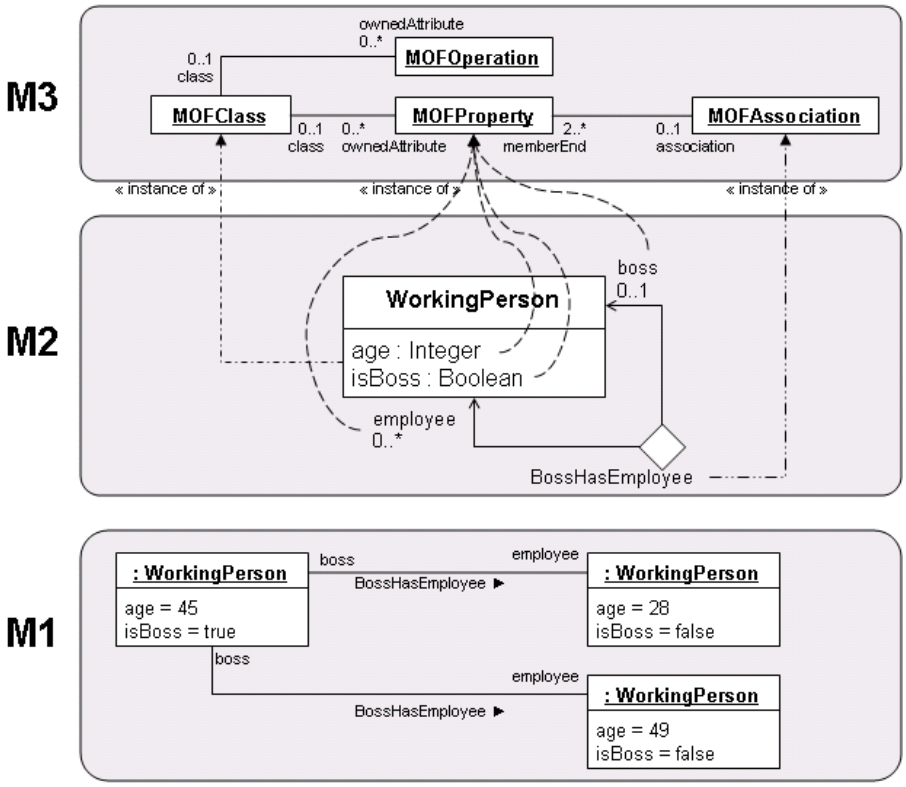


Figure 2.8: Example of metamodeling

OCL 2.0 as Constraint Language

OCL is a declarative language for describing constraints applying to UML models, and is now part of the UML standard. It may be used with any MOF compliant metamodel, including UML. In other words, OCL 2.0 integrates naturally with MOF 2.0, and hence, has been chosen as constraint language for MOSL.
OCL queries can be used in various contexts, such as the specification of invariants, derivation rules, pre- and post-conditions, etc. Each OCL constraint has to be specified in a particular context. The context defines the limited situation in which the statement is valid. An OCL expression is evaluated to a Boolean value that must be true.

Let us consider the example of Fig. 2.9 where two OCL invariants are defined with the class *WorkingPerson* as context. OCL expressions that define invariants are of the type Boolean, and have to be true for all the instances of this class. The reserved word *self* refers to the contextual instances, in the example of Fig. 2.9 all the instances of *WorkingPerson*. The operator "**.**" is used to query the properties (attribute or association end) linked to the object preceding the operator. For instance, in the first OCL expression of Fig. 2.9, *self.age* in the context of *WorkingPerson*

returns the attribute *age* of all the instances of *WorkingPerson*. OCL provides a set of operations applicable to the primitive types Integer, Real, String and Boolean. For instance, the *and* in the first OCL expression of Fig. 2.9 is the logic operator between both Boolean expressions *self.age >= 18*, and *self.age <= 65*. This first OCL constraint expresses namely a restriction on the age of a *WorkingPerson* which must be of age, but cannot be a pensioner. In other words, the attribute age must be between 18 AND 65. That is why the OCL statement is composed of two Boolean expressions related by the logical operator *and*. Each Boolean expression is composed of a query on the *age* of the *WorkingPerson* (*self.age*) and a check of this value (*>= 18*, *<= 65*). OCL provides also specific operations that can be called on collections by means of the "->" operator. For instance, the *notEmpty()* in the second OCL expression of Fig. 2.9 is an operation that checks if the collection preceding the "->" operator contains at least one element. This second constraint defines the correlation between the value of the Boolean attribute *isBoss* and the number of instances of *WorkingPerson* that are employees of the self instance of *WorkingPerson*. A *WorkingPerson* is namely a boss (*self.isBoss*) if, and only if, its set of employees is not empty (*self.employee- >notEmpty()*). The implication is expressed by the reserved word *implies*.

A more detailed presentation of OCL syntax and semantic is out-of-scope in this thesis. The interested reader finds a complete description of OCL in [Obj06b].

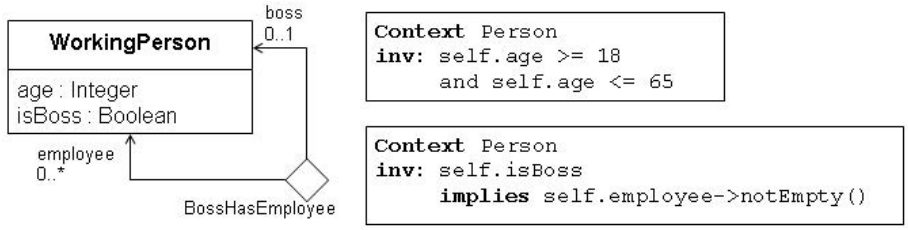


Figure 2.9: OCL expressions - Example

2.3.2 MOSL Transformation Language

Similarly to the schema and constraint languages of MOSL, the transformation language of MOSL is not a new language, but an existing one called SDM. Here again, the visual notation has been preferred to a textual language. An SDM diagram (also called story diagram) combines a UML activity diagram with graph transformations. An activity diagram is used to specify the behavior of exactly one operation of a schema class by specifying the control flow concerning the execution of several graph transformation rules. SDM is an endogeneous, in-place transformation language with both declarative (pattern matching within the activities) and operational elements (specification of control flow by means of the activity diagram).

Story Patterns

A story pattern describes a graph replacement that is embedded into an activity of an SDM diagram. SDM graph transformation rules differ from classical approaches since they are not described by two separated graphs representing the left- and right-hand side of a transformation. A transformation rule is rather described by a single graph which simultaneously describes the left- and right-hand sides of a transformation by means of annotations and different colors. The subgraph that is common to both sides is depicted in black without any annotations, whereas those parts of the left-hand side which are not part of the right-hand side and as such deleted by the transformation are annotated with *destroy*. Consequently, those parts of the right-hand side that are not part of the left-hand side (and as such created by the transformation) are annotated as *create*, except for the attribute assignment which is not annotated, but is identifiable by the use of the *assign*-operator depicted by the symbol *:=*.

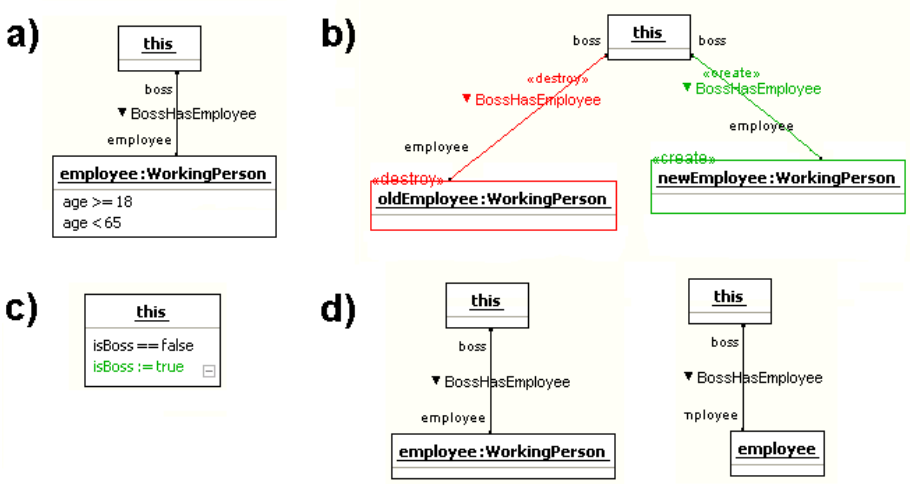


Figure 2.10: Story Pattern Examples

Fig. 2.10 provides examples of story patterns. The SDM diagrams which contain these story patterns describe the behavior of methods of the metaclass *WorkingPerson* (see Fig. 2.8). Consequently, the keyword *this* represents the object in which the method is called, i.e. here an instance of *WorkingPerson*.
In Fig. 2.10.a, an instance of *WorkingPerson* which is connected to the *this*-node is matched. It is a simple matching, without transformation, or, in other words, the left-hand side and the right-hand side are the same. Therefore, this pattern is depicted in black, without any label. The working person must be older than 18 years old and younger than 65 years old. This constraint is expressed by both attribute constraints $age >= 18$ and $age < 65$.
In Fig. 2.10.b, an *oldEmployee* is pensioned off, and is replaced by a *newEmployee*. The left-hand side consists of the nodes *this* and *oldEmployee:WorkingPerson* con-

nected by the link *BossHasEmployee*. The right-hand side consists of the nodes *this* and *newEmployee:WorkingPerson* connected by the link *BossHasEmployee*. The node *this* belongs to both left- and right-hand side of the transformation, and, thus, is depicted in black without any label. The node *oldEmployee:WorkingPerson* and its link *BossHasEmployee* to *this* belong only to the left-hand side and, thus, are depicted in red with the label *destroy*. In the same way, the node *newEmployee:WorkingPerson* and its link *BossHasEmployee* to *this* are depicted in green with the label *create* since they belong only to the right-hand side.
The Fig. 2.10.c shows a simple attribute check and assignment. If the Boolean value *isBoss* of *this* equals *false*, it is set to *true*. The value check is executed by means of the operator "==" whereas the value assignment is depicted in green with the operator ":=".
Another important aspect of SDM is the concept of *bound* and *unbound* pattern objects. Both patterns of Fig. 2.10.d seem pretty similar. Though, *employee* in the first pattern is unbound whereas it is bound in the second pattern. An unbound transformation object is depicted in the story pattern according to the following template: *objName:objType*, i.e. with the indication of the type. An object is bound because it has been matched or because it has been given as a parameter. Once a transformation object is bound to its matched object, it can be reused as such by the object's name in further activities. Concerning the type of an object, it must be noticed that the type model of the story pattern supports inheritance and polymorphism. This means that a node of a given type T may match not only objects that are exactly of this type, but also any object whose type is a subtype of T (about typing: see Section 2.5).

An overview of the syntax of the elements usable within a story diagram can be found in Appendix A. A more detailed description of the syntax and the semantic of the SDM language can be found in [Zün01].

Control Flow

As explained, SDM diagrams can be considered as UML activity diagrams embedding graph transformations within their activities in form of story patterns. Thus, the activity diagram allows for the specification of the control flow.

Fig. 2.11 shows two story diagrams. An SDM diagram always has one single start activity (depicted by a black circle), and at least one stop activity (depicted by a black point inside a circle). Since a story diagram specifies the behavior of an operation, it is possible to define a return value. For instance, in Fig. 2.11.a, the method *addEmployee()* returns an Integer value, namely the age of the employed person. This is specified by the expression *employee.age* below the stop activity.
An activity is connected to the next one by a transition represented by an arrow. A transition may be labeled by a guard condition expressed between squared brackets. Fig. 2.11.a contains such guard conditions: *[success]* and *[failure]*. A transition

labeled with [success] is traversed only if the story pattern of the preceding activity has been successfully executed, whereas a transition labeled with [failure] is traversed if the preceding story pattern failed. We can notice that a transition with the guard condition *[success]* has a similar semantics as a transition without a guard condition. Though, a *success*-transition is always used in combination with a *failure*-transition. A guard condition can also be a user-defined Boolean expression. In this case, it is always used in combination with an *else*-transition in order to ensure a default transition. If a story pattern fails, and the owning activity has no outgoing failure-transition, the execution of the story diagram is interrupted.
There are two kinds of activities: simple activities and *for-each*-activities. A simple activity is represented by a simple rounded-corner rectangle, whereas a *for-each*-activity is depicted by a double rounded-corner rectangle. Contrary to a simple activity which is executed at most once if there is a match, a *for-each*-activity will be executed for each match that is found when the activity is initially activated.

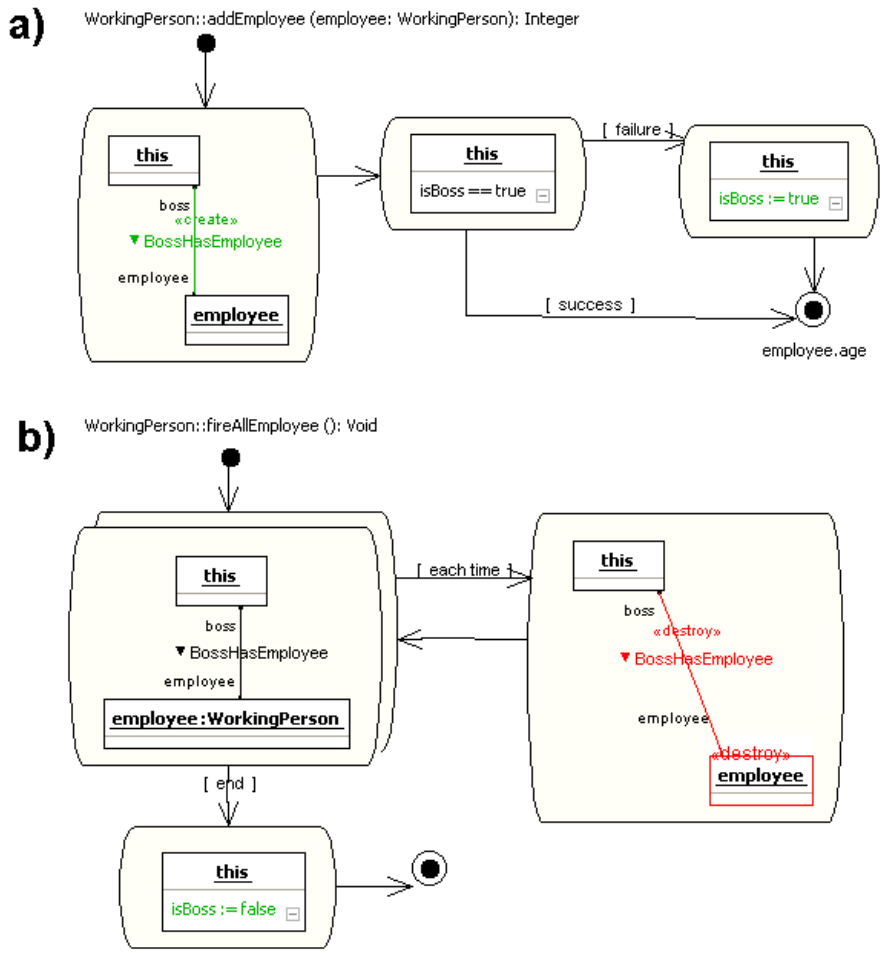


Figure 2.11: Control flow in SDM

To better understand how activities must be interpreted, let us consider the examples of Fig. 2.11.
The story diagram of Fig. 2.11.a specifies the behavior of the method *addEmployee(employee:WorkingPerson)* which models the employment of a *WorkingPerson*, and returns the age of this new employee. This *WorkingPerson* named *employee* is given as a parameter. In other words, it is already bound. The transformation of the first activity consists of the creation of a link instance of the association *BossHasEmployee* between *this* and the parameter *employee*. If the transformation is successfully executed, the transition is traversed to the next activity. Here, the value of the Boolean attribute *isBoss* of *this* is checked if it is set to *true* (*isBoss* == *true*). If this pattern is matched, the transition with the guard condition [success] is traversed to access directly the stop activity. Else, the *failure*-transition is traversed to the activity where the value *true* is assigned to the attribute *isBoss* (*isBoss* := *true*), before the next transition is traversed to the stop activity.
The story diagram of Fig. 2.11.b illustrates the use of a *for each*-activity. This diagram specifies the behavior of the method *fireAllEmployee()*. This method must delete all the employees of the calling *WorkingPerson*, and set its attribute *isBoss* to false. In the first activity, a *for each*-activity, an *employee* of *this* is matched. For each matched pattern, the activity is left via the transition with the guard *each time* to the next activity where the bound *employee* and its link to *this* are deleted. A transition is traversed afterwards back to the *for each*-activity. This cycle is repeated as long as the pattern contained in the *for each*-activity can be matched, i.e. here, as long as the calling *WorkingPerson* has an employee. Then, the for each-activity is left via the transition with the guard *end*. The last activity sets the attribute *isBoss* to false since the calling *WorkingPerson* has no employee anymore.

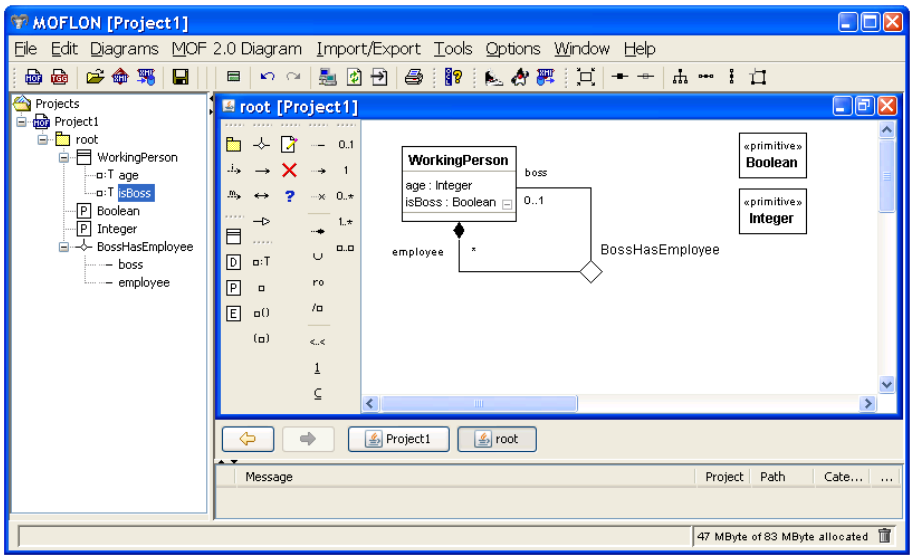


Figure 2.12: Screenshot of MOFLON

2.3.3 MOSL Tool Support

MOSL is supported by the meta-CASE-tool MOFLON [MOF11] [1]. MOFLON allows for the specification of metamodel and metamodel transformations using MOSL, and for the generation of JMI-compliant code (about JMI: see Section 2.1.3). Fig. 2.12 is a screenshot of this tool. The left-hand side of the screenshot shows the project tree, and the right-hand side represents the class diagram editor with the running example of Section 2.3.1.

MOFLON has not been built from scratch, but is based on the open-source CASE-tool FUJABA (From UML to Java And Back Again) [FUJ11]. After a redesign in 2002, FUJABA has become the "FUJABA Tool Suite" with a plug-in architecture. Thus, FUJABA offers an extensible platform, and allows developers to add functionality easily while retaining full control over their contributions. Many components are reused by MOFLON, completed by additional elements.

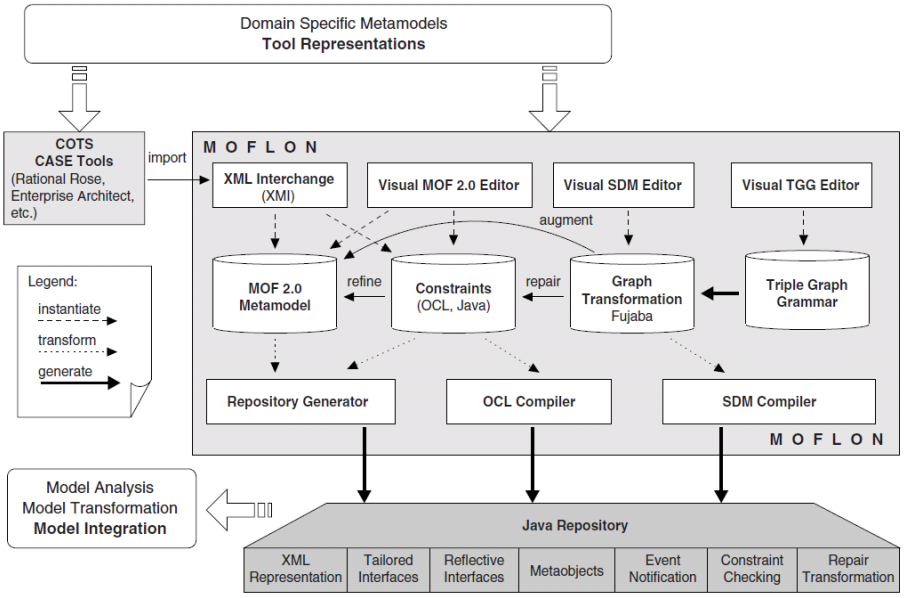


Figure 2.13: MOFLON Architecture [AR06]

Fig. 2.13 represents the architecture of MOFLON. The different elements and their role are described in the following. MOFLON provides a set of visual editors. It provides a MOF 2.0-compliant class diagram editor, with a textual OCL 2.0 editor, to specify metamodels (*Visual MOF 2.0 Editor*). In addition, the *visual SDM editor* of FUJABA allows for specifying dynamic behavior and model transforma-

[1] MOFLON has been reengineered during this thesis, in 2011, and is now called eMOFLON. It uses the professional CASE tool Enterprise Architect as frontend, and is based on Eclipse Modeling Framework (EMF) [ALPS11]

tions. Finally, a Triple Graph Grammar editor supports the specification of model integration patterns (*Visual TGG Editor*) Moreover, it is possible to import metamodels from an external visual editor (e.g. Rational Rose [Rat11] or Enterprise Architect [EA11]) by means of the XMI (XML Metadata Interchange [Obj05c]) import module.

MOFLON can be decomposed into four major parts.

- In the center of MOFLON there is the *MOF 2.0 metamodel*. The import module or the visual MOF 2.0 editor instantiates this metamodel. A *Repository Generator* called MOMoC [Bic03] generates JMI-compliant code from the instances of the metamodel. MoMOC is presented more precisely in the following.
- The second part is the OCL constraint parser and metamodel. MOFLON reuses the Dresden OCL Toolkit [Dre11]. It provides support for the validation of OCL constraints as well as for the generation of constraint-checking Java code (*Constraints*, *OCL Compiler*)
- The third part concerns the SDM model transformations. The transformations are visually specified by means of the *Visual SDM Editor* from FUJABA. These transformations are made persistent in the SDM metamodel as provided by Fujaba (*Graph Transformation Fujaba*). MOFLON uses the code generator from FUJABA, called CodeGen2 [GSR05], which solely generates transformation code (*SDM Compiler*). CodeGen2 is described more precisely in the following.
- The fourth and last part is the Triple Graph Grammar extension. It provides a special graphical editor (Triple Graph Grammar). The rule generator translates declarative triple graph grammars into executable SDM rules (*Triple Graph Grammar*).

As explained above, MOFLON integrates three code generators: MOMoC, CodeGen2, and the *OCL Compiler* of the Dresden OCL Toolkit. The combined output of all three code generators forms a sophisticated Java repository implementation (Java Repository). Fig. 2.14 explains how MOMoC and CodeGen2 work, and how the MOFLON Compiler brings them together.
MOMoC generates JMI-compliant code from the MOF 2.0 class diagram. This compiler is principally composed of four components: a parser, a set of modules, a so-called XML Generator, and a code generator. The instances of the MOF 2.0 metamodel are preprocessed by an arbitrary set of individual modules. These modules execute all necessary modifications which are essential for the task of code generation, e.g the assignment of default names for unnamed elements. Then, the XMLGenerator translates the metamodel instances in the form of Java runtime objects into a straightforward XML representation. The code generator transforms theses XML artifacts into Java code by applying an XSLT (Extensible Stylesheet

Language Transformation) [Dre07] transformation, a template based transformation approach. Finally, the pretty printer formats the generated code.
CodeGen2 is the code generator of Fujaba. A tokenizer transforms the input, an abstract syntax tree of the project, into a token graph, each token representing a code fragment that shall be generated. Mutators rearrange the token graph to optimize the code generation. The code generation is based on the use of Velocity templates [Vel07]. CodeGen2 generates originally Fujaba-compliant code. Therefore, a new set of Velocity templates has been defined so that the generated code conforms to the JMI standard. The Code Writer implements a "*Chain of Responsibilities*" design pattern where each elements knows a template and generates code for a given token for which it is responsible.
The MOFLON Compiler combines the code generated by MOMoC and the code generated by CodeGen2 as follows. It calls both compilers, but redefines the XMLGenerator of MOMoC. This XMLGenerator defines an additional element called *body*, and insert it in the XML Data representing an operation. This element possesses an attribute called *code*. The code generated by CodeGen2 from the story diagrams, i.e. the body of the corresponding operation, is then set as value of the attribute *code*. Finally, the code generator of MOMoC generates the rest of the code.

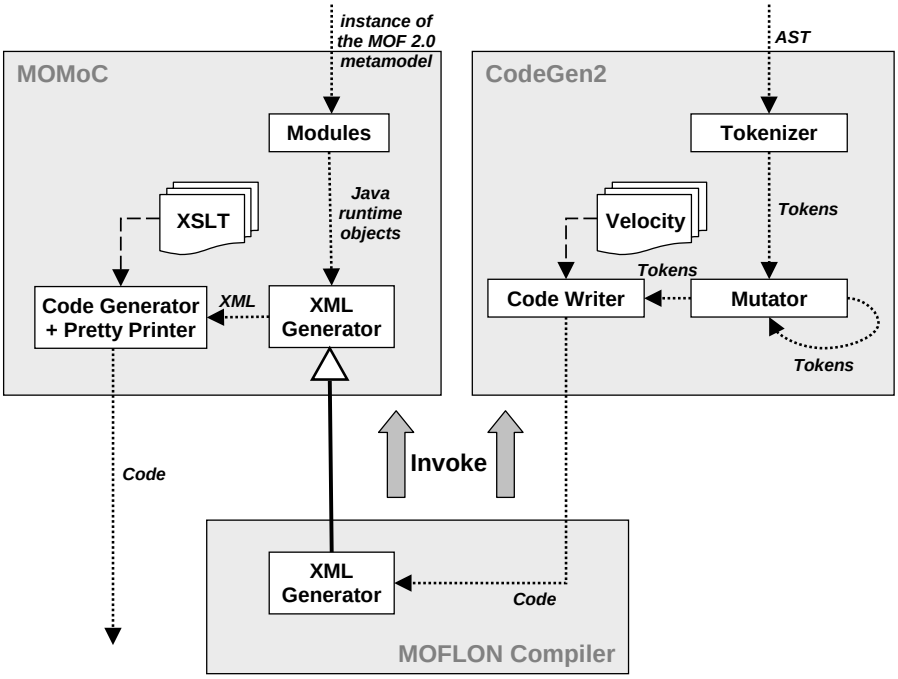


Figure 2.14: MOFLON Compiler

2.4 Related Transformation Languages

Several tools offer support for different kinds of model transformation languages: visual, textual, endogenous, exogenous, etc. There is a large number of tools and languages supporting model transformations: not only MOFLON with the SDM language, but also PROGRES [PRO11], VIATRA2 [Via11a], GReAT [BNvBK06], ATL [ATLa], ModGraph [Win12], DiaGen/DiaMeta [Dia13], etc.
Due to their large number, we cannot describe all existing languages in this work. Therefore, we limited ourselves to a representative subset which allows for presenting a relevant comparison between them.

2.4.1 PROGRES

PROgramming with **G**raph **R**ewriting **S**ystems (PROGRES) is a visual programming language which was developed at the University of Technology Aachen (Germany) since the late 1980s. It has been defined on the basis of the logic theoretic approach to graph grammars [PRO11]. Its programming environment consists of a syntax-directed editor, an analyzer, an interpreter, and a compiler.
The PROGRES language is based on directed, labeled and attributed graphs. It belongs to the category of visual programming languages since it has a graph-oriented data model and a graphical syntax for its most important language constructs. Though, it does not exclude textual syntax when it is more natural and concise. Similarly to MOSL (Cf. Section 2.3), the specification of a model with PROGRES presents two aspects. On the one hand, the graph scheme is the definition by means of nodes and edges of the model elements. On the other hand, the graph transformations regroups the definition of operation on the graph scheme.

Graph Scheme

A PROGRES graph scheme principally consists of *labeled and attributed nodes*, and *labeled edges*. It is also possible to define derived relations in the form of *paths* and *restrictions*.
PROGRES proposes a two levels typing system, with *node classes*, and *node types* which instantiate the node classes. Nodes can be attributed, i.e. may contain additional information in the form of attributes. An edge, which is a binary relation between two nodes, is defined by its label, and its source and target nodes. These source and target nodes may have cardinalities. PROGRES supports also binary complex relationships between two nodes, so-called *paths*, which can be a concatenation of different edge traversals, or may contain conditional expressions which influence the selection of the target node(s) of that relationship. The *restrictions* are an other kind of derived "relationships" which take effect on a set of nodes. It is also possible to define graph patterns which have to be present at any time by means of so-called *constraints*. A specific kind of constraint are the constraint attributes. They consist of derived attributes which have to be evaluated to the

Boolean value *true*, and may otherwise be *repaired*.
In PROGRES, the user can choose between two views on his specification. Next to the graphical scheme view, a textual view is available too. Fig. 2.15 is an example of a PROGRES graph scheme, the part a of the figure showing a visual specification and the part b showing its textual counterpart. This example is a simple modeling of an deterministic finite automata. The normal boxes represent the node classes whereas the rounded-corner boxes represent the node types. The node classes *STATE* and *TRANSITION* derive from the node classes *ENTITY* and *RELATIONSHIP*. This inheritance relationship between node classes is depicted by a triangle-headed arrow, and represented by the *is_a* operator in the textual notation. The node class *STATE* has an intrinsic (i.e. whose value is directly assigned and does not depend on any other attribute value) boolean attribute called *currentState*. The simple arrows between ENTITY and RELATIONSHIP are the labeled edges *src* and *trg*, and the label *[0:n]* and *[1:n]* are the cardinalities. As depicted by the dashed triangle-headed arrow, the node types *state* and *trans* specialize *STATE* and *TRANSITION*. Thus, the node types *state* and *trans* may have rules or instances, whereas node classes correspond to abstract classes in object-oriented languages.

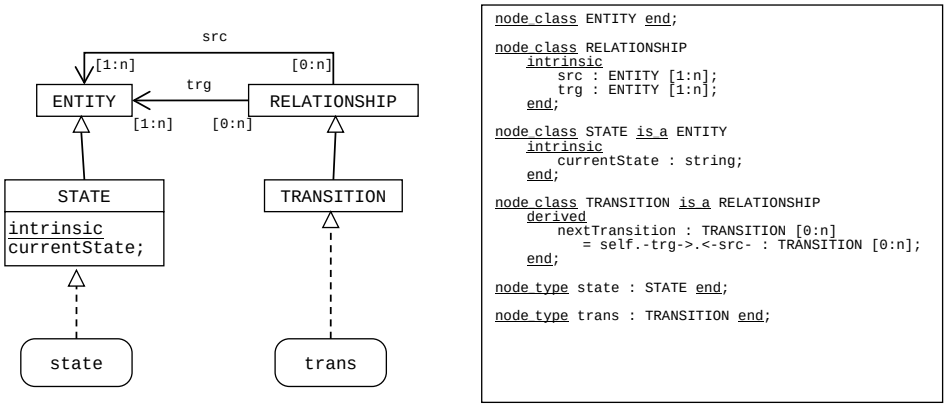


Figure 2.15: PROGRES Graph Scheme [Mue02]

Graph Transformations

The graph transformations in PROGRES have a left-hand side (LHS) and a right-hand side (RHS), depicted from top to bottom instead of the usual notation from left to right. The LHS is on top of the diagram while the RHS is found beneath the "*::=*"-sign.
The execution of a graph transformation is non-deterministic: two executions of the same transformation will not necessary affect the same part of the graph. Therefore, PROGRES supports the parameterization of graph transformations, e.g. if a node to be matched must be precisely determined. A parameter may be a node types as well as a variable. In the case of a variable as parameter, this variable

can contain a type as its value. In addition to the LHS and RHS, it is possible to define textually elements such as attribute conditions and assignments, or return statements.
The rules, as defined in a visual way, correspond to a declarative modeling style. An imperative modeling style would require specific mechanisms such as concatenation of transformations, conditional statements, calls of other transformations, etc. Similarly to the graph scheme, it is possible to define model transformations in a textual way which supports the definition of these mechanisms. Thus, the textual specification of a model transformation allows for the definition of a control flow. Finally, PROGRES supports the definition of textual pre- and postcondition, as well as the definition of queries which can be defined graphically as well as textually.

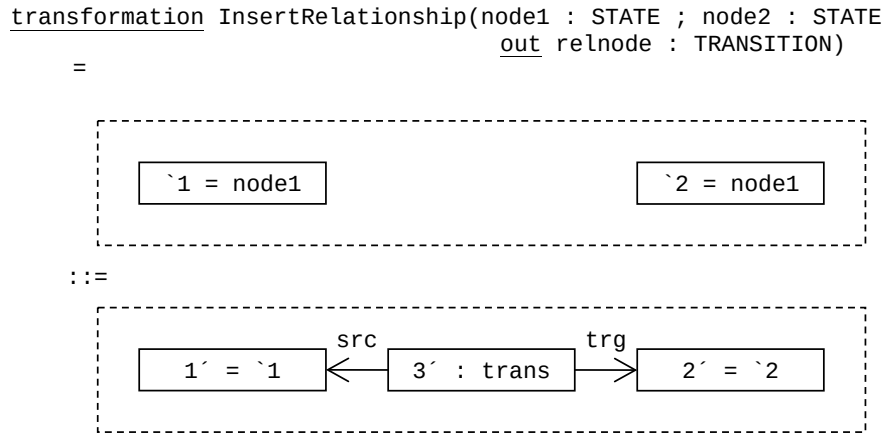


Figure 2.16: PROGRES visual Graph Transformation [Mue02]

Fig. 2.16 shows the visual specification of a graph transformation. This transformation, called *InsertRelationship*, takes two nodes of type *STATE* as input parameters, and returns a node of type *TRANSITION*. In PROGRES, nodes on the RHS are suffixed by a normal quote sign while nodes on the LHS are prefixed by a backquote sign. In the LHS, both input parameters, assigned to the node ` *1* and ` *2*, are matched first in the host graph. Since PROGRES is a context-sensitive graph transformation, the matched node of the LHS are also considered in the RHS. This is denoted by *x ´ = ` x* in the RHS. Then, the new node *3 ´ : trans* is created and connected to the nodes *1 ´* and *2 ´*. The textual part below the visual specification contains an attribute condition (keyword *condition*), an attribute assignment (keyword *transfer*), and a return value assignment (keyword *return*). The *condition* restricts the graph match on the LHS to a node ` *1* whose attribute *currentState* is

set to *waiting*. The *transfer*-statement sets this attribute to *active* after the evaluation of the RHS, and the *return*-statement assigns the node with the identifier *3 ´* to the output parameter *relnode*.

```
transformation ExecDiagram =
   use srcState : STATE;
      trans : TRANSITION [0:n]
   do
      GetActiveState( out srcState )
   & GetTransitions ( srcState, out trans
   )
   & for_all t := elem ( trans )
      do
         choose
            when (t.fireCondition = true)
            then
               FireTransition ( t )
            else
               skip
            end
         end
      end
   end;
```

Figure 2.17: PROGRES textual Graph Transformation [Mue02]

Fig. 2.17 is a textual specification of a model transformation. A detailed description of this transformation is here out-of-scope. Let us simply notice that this kind of specification combines several mechanisms that allow for an imperative modeling style. It is possible to call other graph transformations such as, in Fig. 2.17, *GetTransition*. The *use*-statement allows for the declaration of variable with a local scope. PROGRES offers control structures such as the conditional structure *if-then-else*, expressed here by the *choose-when-then-else*-statements. It is also possible to iterate a set: the *for_all*-loop is executed as long as there are elements in the set of transitions called *trans*, and each time an element of *trans* is assigned to the iteration variable *t*. Finally, the &-symbol corresponds to a deterministic concatenation, and, hence, allows for ensuring the order in which the commands are executed.

2.4.2 VIATRA2

The VIATRA (**VI**sual **A**utomated model **TRA**nsformations) framework [Via11a] has been developed at the Fault Tolerant Systems Research Group at the Budapest University of Technology and Economics. The first version, written in Prolog, has been developed between 2000 and 2003. Then, the second and current version, called VIATRA2, has been reengineered from scratch since 2004, written in Java, and fully integrated in the open source software development environment Eclipse [Ecl11].

This model transformation tool aims at providing an environment that supports the entire life-cycle of engineering model transformations. This includes not only the specification, design, and execution of transformations, but also their validation and maintenance within and between various modeling languages and domains. To this purpose, VIATRA2 offers a model space for the representation of models and metamodels, a transformation language based on the techniques of graph transformation and abstract state machines, a transformation engine, and a code generator [Via11b].

Model Space

VIATRA2, which is able to import models of several off-the-shelf industrial modeling tools, uses the VPM (Visual Precise and Multilevel) metamodeling approach [VP03]. The model elements of the VPM model space can either be *entities* which represent the basic concepts of the modeling domain, or *relations* which describe the associations between entities. In addition, VPM defines three specific relationships: *instanceOf*-relationship to represent the connection between model and metamodel elements, *supertypeOf*-relationship to create a type inheritance hierarchy, and *containment*-relationship to create an explicit containment hierarchy.
The VPM model space offers features which are not usual in most metamodeling environments such as:

- multi-level metamodeling
- possibility to assign multiple types to a model element
- possibility to retype elements dynamically
- representation of models and metamodels in the same model space

The representation of models and metamodels in the same model space allows for a simultaneous manipulation of models and metamodel, e.g. to enable generic/higher-order transformations or to access the class from an instance model element.
The textual language supporting VPM is called VTML (**V**IATRA **T**extual **M**etamodeling **L**anguage). Fig. 2.18 shows a simple example of modeling by means of VTML. We have two classes *Place* and *Transition*, and two relationships between them called *src* and *trg*, in a package called *Petri* (Fig. 2.18.a). Fig. 2.18.b is the textual specification of the example in VTML. The entity *Petri*, defined by the keyword *entity* and its name *Petri*, contains all the other elements. The entities Place and Transition are specified. Then, the relationships *src* and *trg* are defined by the keyword relation and, as parameters, the name of the relationship, its source, and its target. The keyword *multiplicity* is used to defined the multiplicity of a relation.
When opening a model, the content of the model space is represented in a tree editor, in a similar way as the EMF (Eclipse Modeling Framework [EMF11]) tree editor. The left part of Fig. 2.19 shows this tree editor in the case of a model of Petri nets. All the entities are displayed in a tree structure, with their attributes and

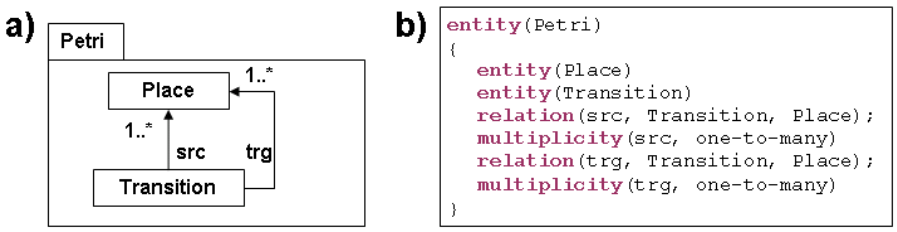


Figure 2.18: VTML - Simple example

relationships to other entities. Although the import of external (meta)models is recommended, the tree editor allows for the creation and edition of (meta)models. A graph visualization component can be used to display a selected subtree (Cf. right part of Fig. 2.19), and, thus, gives a more meaningful overview of the models. Though, it does not offer direct editing support as in the tree editor.

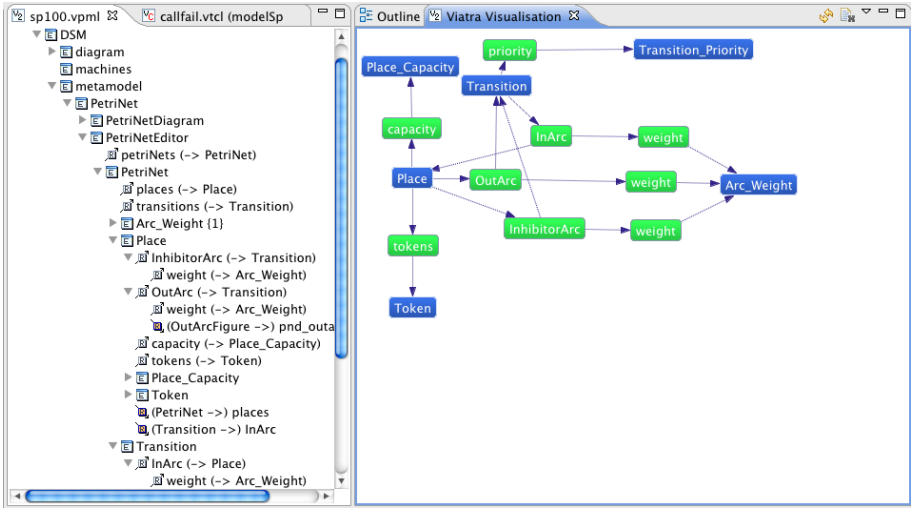


Figure 2.19: VIATRA2 Model Space [Via11b]

Model Transformations

The language offered by VIATRA2 for model transformations is the VTCL (**VIA**-TRA **T**extual **C**ommand **L**anguage), a textual language. It supports the transformation approach of VIATRA2 which consists of a combination of various specification formalisms. These constructs are:

- *Graph patterns*: atomic units of graph transformations.
  A graph pattern represents a condition or a constraint. This declarative query is satisfied if it can be matched to a subgraph of the model, and is introduced in VTCL by the keyword *pattern*. A graph pattern may call another graph

pattern by using the keyword *find*. It is possible to define generic graph patterns thanks to the *instanceOf*- and *superTypeOf*-relationships of VTML.

- *Graph transformation rules*: for the definition of elementary model manipulations.
  A graph transformation, which may have in-, out- or inout-parameters, is composed of a LHS pattern which matches the subgraph on which the rule will be applied, and a RHS pattern which determines in a declarative way the result of the transformation. The specification of a graph transformation is introduced by the VTCL keyword *gtrule*, the LHS pattern by *precondition pattern*, and the RHS pattern by *postcondition pattern*. Within a LHS or a RHS pattern, it is possible to use a predefined pattern by means of the keyword *find*. A graph transformation may contain an action sequence, introduced by the keyword *action*, that can by any ASM sequence (see bellow).

- *Abstract state machine (ASM)*: for the description of control structures.
  The ASM rules provide a set of commonly used imperative control structures: sequencing operator (*seq*), rule calls to other ASM rules (*call*), variable declarations and updates (*let* and *update* constructs) and *if-then-else* structures, non-deterministically selected rules (*random*) and executed rules (*choose*), iterative execution (applying a rule as long as possible *iterate*), etc.

Fig. 2.20 shows an example of transformation in the context of the simple model defined in Fig. 2.18.
Fig. 2.20.a is a pattern (keyword *pattern*) which is called *placesAreConnected*. It matched a subgraph where two places are connected to each other by a transition.
Fig. 2.20.b is a graph transformation (keyword *gtrule*) called *deleteTransition* that aims at deleting the transition between both places P1 and P2 which are given as input parameters. This transformation uses as LHS pattern the pattern *placesAreConnected* (keyword *find*) The RHS pattern consists of the entities P1 and P2, and of a negative (keyword *neg*) pattern. A negative pattern is a pattern that must NOT be matched. It ensures here that there is no transition between P1 and P2.
Fig. 2.20.c is an ASM (keyword *machine*) called *deleteAllTransition*, with a rule called *main*. The execution of this ASM must delete all instances of *Transition* contained in the model on which the ASM is applied. The keyword *seq* indicates that the operations in *main* are executed in a sequential order. The rule iterates over all pairs of entities P1 and P2 (keyword *forall*) that fulfill the condition (keyword *with*) expressed by the pattern *placesAreConnected*. For each pair P1 and P2, the graph transformation *deleteTransition* is called (keyword *apply*), and a text is printed out (keyword *println*). Finally, a message is written in the log. The ASM syntax allows for the print-out of text, as well as for printing in an error log with the indication of the error severity (here, *info*). The example of Fig. 2.20.c shows also the possibility to insert comments in the textual specification by prefixing them with "//", similarly to programming languages such as Java or C++.

a)

```
pattern placesAreConnected(P1,P2)
{
 Petri.Place(P1);
 Petri.Transition(T1);
 Petri.Place(P2);
 Petri.Transition.src(SRC, T1, P1);
 Petri.Transition.trg(TRG, T1, P2);
}
```

b)

```
gtrule deleteTransition(in P1, in P2)
{
 precondition pattern lhs(P1, P2) =
 {
  find placesAreConnected(P1, P2);
 }

 postcondition pattern rhs(P1, P2) =
 {
  Petri.Place(P1);
  Petri.Place(P2);
  neg pattern noTransition(P1, P2)=
  {
   Petri.Transition(T1);
   Petri.Transition.src(SRC, T1, P1);
   Petri.Transition.trg(TRG, T1, P2);
  }
 }
}
```

c)

```
machine deleteAllTransitions()
{
 rule main() = seq
 {
  // iterate over all couple of places
  forall P1, P2 with find placesAreConnected(P1, P2) do seq
  {
   // apply graph transformation
   apply deleteTransition(P1, P2);
   // print out some text
   println("Transition between "+ P1 +" and "+ P2 +"deleted");
  }
  // write to log
  log(info, "deleteAllTransitions - end of execution");
 }
}
```

Figure 2.20: VTCL - Simple example

2.4.3 Other Languages: GReAT and ATL

After having described PROGRES and VIATRA2, we now give a brief overview on some additional transformation tools.

Graph Rewriting And Transformation (GReAT)

Graph **Re**writing **A**nd **T**ransformation (GReAT) is a tool for building model transformation tools. It is based on the definition of the input and output modeling languages in form of metamodels, and the transformation itself as graph rewriting rules. The input and output metamodels are defined using the UML class diagrams. The model transformations are visual, and this language consists of three sub-languages: the pattern specification language, the transformation rule language, and the sequencing or control language [BNvBK06].

Similarly to SDM (Cf. Section 2.3.2), the *pattern graphs* are composed of nodes and edges which must have counterparts in the host graph. The basic transformation entity is a *production rule* which is composed of a pattern graph, actions, input

and output interfaces, a guard, an attribute mapping, and match conditions. The *actions* are a mapping of pattern edges and nodes to the set of actions which is composed of {*Bind, CreateNew, Delete*}. Thus, these actions combine in a single graph the LHS and the RHS of the transformation rule. The transformation rule language of GReAT allows for the reuse of graph objects from a rule to the next one by means of the input and output interfaces. An *output interface* is composed of a set of distinct output ports that transfer graph objects to the next rule. Here, they are received by means of the *input interface* which is composed of a set of distinct input ports. The *guard* of a transformation rule is a Boolean expression which determines whether the rule should be executed or not. The *attribute mapping* is executed for each valid match, and generates the values of the the edge and node attributes. Finally, a *match condition* is a flag that determines whether all matches are executed, or only a single (non-deterministically chosen) match is executed.

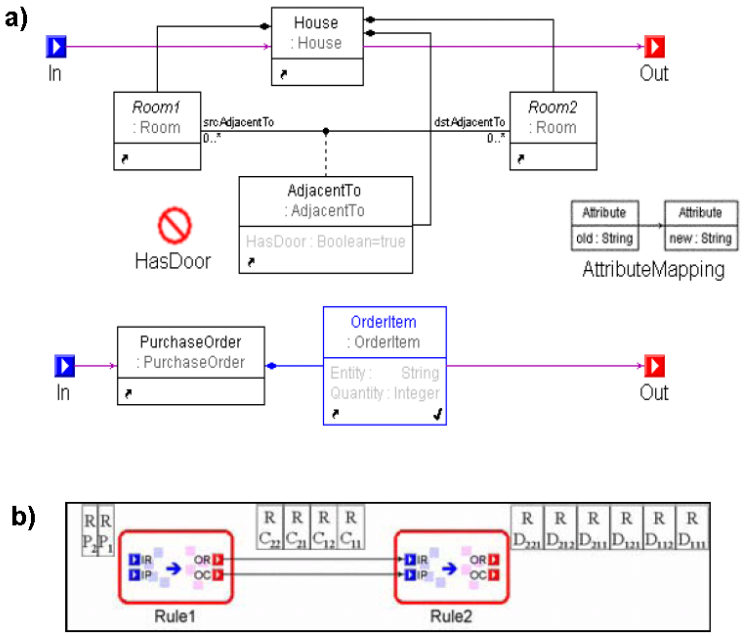


Figure 2.21: GReAT - Transformation Rule and Sequence of Rules [BNvBK06]

Fig. 2.21.a is a simple example of a transformation rule. The objects *House* and *PurchaseOrder* are bound to the input ports. This means that they are received from an previously executed rule. Then, the objects *Room1*, *Room2*, *AdjacentTo*, and the required links are searched. After theses elements have been matched, the guard condition *HasDoor*, which contains procedural code (C++), is checked. If the guard condition is true, the object *OrderItem* and its link to *PurchaseOrder* are created. Finally, the procedural code contained in the *AttributeMapping*-block is executed. The objects *House* and *OrderItem* are bound to the output port, which means that they are transferred to the next rule.

The control flow language of GReAT allows for constructing larger model transformations composed of several transformation rules. It provides different mechanisms: sequencing, non-determinism, hierarchy, recursion, and conditional execution. The sequencing is the sequential execution of rules, whereas the non-determinism corresponds to a parallel connection of the rules. For instance, the part b of Fig. 2.21 depicts a sequence of rules: *Rule1* executes first, and *Rule2* fires only after *Rule1* has successfully been executed, and has transferred its output packets to the input interface of *Rule2*. The hierarchical composition of rules is realized by means of a block which contains primitives rules or other blocks. The output of a rule can be connected to the input of a block higher in the containment hierarchy. As a consequence, the output packets are sent back as input packets to the preceding rule, which results in a recursive activation of the rule. Finally, GReAT provides so-called *Test*-blocks which can have multiple *Case*-blocks, and multiple outputs. The output packets are placed on the output ports which are determined by the successfully matched *Case*.

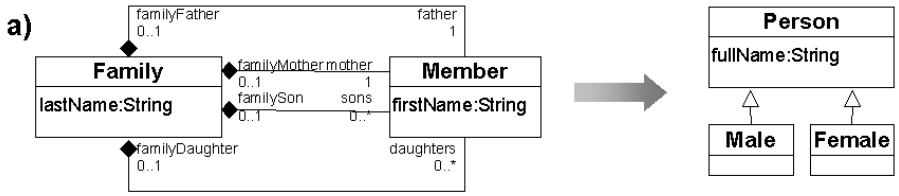
Atlas Transformation Language (ATL)

The Atlas Transformation Language (ATL)[ATLa] is a model transformation language developed by the AtlanMod team (previously ATLAS Group INRIA) [Atl11]. It was originally an answer to the Query/View/Transformation (QVT) request for proposal of the OMG in 2002. The purpose of this request was the definition of a standard compatible with the MDA recommendation suite (MOF, OCL, etc.). A query is an expression evaluated over a model. For instance, OCL is a query language. A view is a model derived from another model, and a transformation maps a source model to a target model. Thus, ATL, as an answer to the QVT request for proposal, aims at covering these aspects.
This language, specified both as metamodel as well as textual concrete syntax, allows for the specification of the transformation of a set of source models into a set of target models. As tool support, the ATL Integrated Development Tool (IDE), developed as Eclipse plug-in, provides standard options such as debugger, syntax highlighting, etc. to facilitate the specification of ATL transformations.

ATL focuses on model-to-model transformations, called *modules*, which are defined over MOF-compliant metamodels. A module is composed of an header section to define some attributes, an optional import section to import existing ATL libraries, a set of helpers which can be viewed as an ATL equivalent Java methods, and a set of transformation rules. The ATL language allows for the expression of transformations in a declarative as well as in an imperative way. This corresponds to the three kinds of rules provided by ATL: the matched rules (declarative and imperative programming), the lazy rules (similar to matched rules, but applied only when called), and the called rules (imperative programming, and executed only when called). As an answer to the QVT request for proposal, ATL allows for defining ATL queries which can be viewed as operations to compute a primitive

value from a set of source models. Finally, an ATL library defines a set of ATL helpers that are explicitly associated with a given context, and that will be made available in the ATL units which import this library.

a)



b)

```
module Families2Persons;
   create OUT : Persons from IN : Families;

helper context Families!Memeber
   def: isFemale() : Boolean =
   if not self.familyMother.oclIsUndefined() then
      true
   else
      if not self.familyDaughter.oclIsUndefined() then
         true
      else
         false
      endif
   endif;

rule Member2Male {
   from
      s : Families!Member (not s.isFemale())
   to
      t : Persons!Male (fullName <-s.firstName + ' '+ s.familyName)
   }

rule Member2Female {
   from
      s : Families!Member (s.isFemale())
   to
      t : Persons!Female (fullName <-s.firstName + ' '+ s.familyName)
   }
```

Figure 2.22: ATL - Simple example [ATLb]

Fig. 2.22 shows a simple ATL use case. The module called *Families2Persons* (Fig. 2.22.b) generates an instance of the metamodel *Persons*(Fig. 2.22.a - right) from a model instance of the metamodel *Families* (Fig. 2.22.a - left). The module is composed of an header, an helper called *isFemale()*, and two rules called *Member2Male* and *Member2Female*. The helper checks whether a member of the family is a man or a woman according to its role in the family, since a mother or a daughter is necessarily a woman, and a father or a son is necessarily a man. The rule *Member2Male* transforms a male family member into an instance of the class Male, and the rule *Member2Male* transforms a female family member into an instance of the class Female. These rules call the helper as a condition to determine whether the transformation can be executed or not.

We do not describe here thoroughly the syntax of the example, but this simple example shows that the ATL syntax integrates and extends the OCL syntax, e.g. the use of the OCL-method *oclIsUndefined()* or the operator $<-$ which represents a value assignment. The interested reader can find the complete documentation of ATL at [ATLc].

2.4.4 Comparison of the Languages

The increasing number of approaches and languages for model transformation made necessary the analysis of the nature of model transformation languages to avoid a waste of effort in their development [TC10]. As a consequence, several studies have been conducted, leading to different classification schemes and sets of characteristics, e.g. [GGKH03] [MCG05] [CH06] [DLC10] [EGdL$^+$05b].
For instance, [MCG05] presents a taxonomy to classify model transformations. The criteria have been established by investigating four aspects expressed by four questions:

- **"What need to be transformed into what?"**:
  1) Number of source and target, 2) Technological space, 3) Endogenous vs. exogenous, 4) Horizontal vs. vertical, 5) Syntactical vs. semantical

- **"What are the important characteristics of a model transformation?"**:
  1) Level of automation, 2) Complexity, 3) Preservation of properties.

- **"What are the success criteria for a transformation language or tool?"**:
  1) Suggesting when to apply, 2) Customizing and reusing, 3) Verifying and ensuring of correctness, 4) Testing and validating, 5) Dealing with incomplete or inconsistent models, 6) Grouping, composing and decomposing, 7) Genericity, 8) Bidirectionality, 9) Traceability and change propagation.

- **"What are the quality requirements for a transformation language or transformation tool?"**:
  1) Usability and usefulness, 2) Verbosity vs. conciseness, 3) Performance and scalability, 4) Extensibility, 5) Interoperability, 6) Acceptability, 7) Standardization.

These criteria (all or only a relevant part of) can be used to compare transformation tools and languages, such as in [MGVK06] which presents a concrete application example of this taxonomy.
As pointed out by these publications, comparison criteria can be determined according to different points of view. On the one hand, objective criteria such as "endogenous vs. exogenous" or "horizontal vs. vertical transformation" can be defined to differentiate various approaches. On the other hand, other criteria, designed in [MCG05] as *ideal characteristics*, need to be determined to compare transformation languages and to point out their advantages and weak points. The

choice of these criteria is subjective since it depends on the context of the comparison and/or the application domain of the transformations.
We will first categorize the above presented transformation languages (MOSL, PROGRES, VIATRA2, GReAT, ATL) according to a classical and objective classification scheme such as endogeneous vs. exogeneous. Then, we will compare them according to characteristics which we consider as relevant with regard to this thesis. These classifications have been defined with help of the above-cited works and our own observations.

Graph Scheme

A model transformation is defined in connection with a metamodel (endogenous transformation), or with a source metamodel and a target metamodel (exogenous transformation). These metamodels are expressed in a visual or textual language which can be related to the transformation language or not.

The graph scheme of MOSL, namely MOF, is part of MOSL and is a visual language. The input and output of GReAT are visual too. ATL has no dedicated metamodeling languages. The only application condition is that the input and output metamodels are MOF-compliant. The environment of VIATRA2 supports import- and export-modules so that various metamodeling languages can be used with VIATRA2. Though, the VIATRA2 language provides the metamodeling language VTML which is textual. Finally, contrary to the previous examples which are either visual or textual, the PROGRES graph scheme belongs to both categories due to the equivalence between the textual view and the graphical view (Cf. Fig. 2.15). Since the graph scheme of MOSL, ATL and GReAT are MOF-compliant, constraints may be defined using the standard textual constraint language OCL. Concerning VIATRA2, it is possible to define constraints on the VTML metamodels by means of graph patterns which are expressed in VTCL, the transformation language of VIATRA2. PROGRES provides also its own language to define constraints in form of graph pattern or attribute constraint.

Queries and Transformations

Because some languages aimed originally at answering the QVT request for proposals, they integrate explicitly the possibility to define queries, i.e. the search and retrieval of model elements. ATL allows for the definition of queries using OCL. The syntax of PROGRES and VIATRA2 provides queries too. Although MOSL and GReAT do not support explicitly queries, they allow for their specification by means of graph pattern, where the LHS of the rule equals the RHS. In the case of MOSL, a query corresponds to a story pattern without any *create*- or *delete*-label on the objects and links. In the context of GReAT, a query can be expressed as a graph pattern whose nodes and edges are only mapped to the *Bind*-action.

	MOSL	PROGRES	VIATRA2	GReAT	ATL
Endogenous vs. Exogenous	endogenous	endogenous	endogenous + exogenous	endogenous + exogenous	exogenous
Multilevel	-	X	X	-	-
Visual	X	-	-	X	-
Textual	-	X	X	-	X
Hybrid (= Visual + Textual)	-	X	-	-	-
LHS-RHS	combined	separated	separated	combined	separated
Declarative	X	X	X	X	X
Imperative	X	X	X	X	X
Operational (= combination with enriched programming language)	X	-	-	-	-
Parameterization by primitives, data types and objects	X	X	X	X	-
Parameterization by object types	-	X	X	-	-

Figure 2.23: Transformation Languages - Classical Taxonomy

Fig. 2.23 summarizes the results of a classical taxonomy applied on MOSL (more precisely, on SDM which is the transformation language of MOSL) and the four related transformation languages. The X-symbol indicates that a given language possesses a given characteristic.
The first criteria are endogenous vs. exogenous, and multilevel transformation. The concepts of endogenous and exogenous have already been presented in Section 2.2.1. The input and output models of an endogenous model transformation are instance of the same metamodel whereas the input and output models of an exogenous model transformation are instance of two different metamodels. A multilevel transformation is a transformation which combines several meta-levels in the same transformation, e.g. giving a read access to next higher meta-level in order to get meta-information. MOSL, PROGRES and ATL support only endogenous transformations, whereas VIATRA2 and GReAT allow for the specification of endogenous and exogenous transformations. Contrary to MOSL, GReAT and ATL, PROGRES and VIATRA2 model transformations can be multilevel thanks to multilevel modeling. MOSL and GReAT are visual whereas VIATRA2 and ATL are textual. PROGRES is a particular case due to the equivalence between textual and visual view. Thus, it is a textual as well as an hybrid language (the visual view comprises textual elements such as *condition* or *transfer*). The separation of the LHS and the RHS is pretty classical in transformation languages. Though, in the cases of MOSL and GReAT, both sides of a transformation rule are combined, which results in a compacter view.
Although definition of transformations in a declarative way are recommended, the table of Fig. 2.23 shows that all languages provide also imperative mechanisms. Only MOSL can be qualified of operational, i.e. enriched with programming language, thanks to the Java- and collaboration-statements.

All the model transformations can be parameterized with primitives, data types or objects, except the transformations defined using ATL. Though, only PROGRES and VIATRA2 support also the parameterization of object types.

Important Characteristics of Transformation Languages

Fig. 2.24 resumes the comparison between the related transformation languages according to "important criteria". The chosen criteria are: 1) *Reuse*, 2) *Group, compose and decompose*, 3) *Generic*, 4) *Higher-order*, 5) *Expressiveness*, and 7) *Understandability, Learnability*. These criteria belong to the ones suggested in [MCG05]. Although [MCG05] defines much more criteria, they are not all relevant in the context of this thesis. A comparison of the related transformation language according to these non-cited criteria would be out-of-scope.

	MOSL	PROGRES	VIATRA2	GReAT	ATL
Reuse	X	X	X	X	X
Group, compose, and decompose	X	X	X	X	X
Generic	-	X	X	-	-
Higher-order (= transformation as input and/or output)	-	-	X	-	-
Expressiveness	limited	very good	good	limited	limited
Understandability, learnability	easy	difficult	difficult	easy	easy

Figure 2.24: Transformation Languages - Important Characteristics

The first criterion, called *Reuse*, describes the possibility to reuse elements such as pattern or transformation rules. All considered languages fulfill this criterion mainly by use of parameterization, except for ATL. In the case of ATL, the *Reuse* is enabled by the definition of helpers, and the definition and import of ATL libraries. In the case of GReAT, not only the parameterization but also the definition of blocks allows for the reuse of transformation rules.
The second criterion is *Group, compose, and decompose*. Its is the ability to group several transformation rules into a model transformation, or to decompose a model transformation into several modules. The main way to achieve this characteristic is using controlled graph transformations, i.e. imperative control structure within a model transformation. Since all considered transformation languages support imperative mechanisms, they also fulfill the criterion *Group, compose, and decompose*. In the case of GReAT, this characteristic is realized not only by the control flow but also by the definition of blocks because blocks encapsulate rules, are hierarchical, and can participate in recursive calls.
The third criterion is the support of generic model transformations (about genericity: Cf. Section 2.5.1). Here, only PROGRES and VIATRA2 fulfill this criterion because they support the parameterization of object types.

The fourth criterion corresponds to the possibility to define higher-order transformations, i.e. transformations whose input and/or output are not models but model transformations. Only VIATRA2 among the considered languages possesses this characteristics.
The fifth criterion of Fig. 2.24 concerns the expressiveness of the language. Although the expressiveness of SDM (as transformation language for MOSL), of GReAT and of ATL is good, it remains limited and could be improved. On the other hand, VIATRA2's expressiveness is good [EGdL$^+$05b], and the one of PROGRES very good [FMRS07]. Though, this expressiveness has a drawback. Because it is supported by the definition of very specific language constructs contrary to SDM, GReAT and ATL which are pretty similar to standard languages. SDM and GReAT are namely UML-like, ATL is OCL-like, and, thus, they are easier to understand and to learn than PROGRES and VIATRA.

This comparison shows that SDM, the transformation language of MOSL, possesses interesting characteristics. Though, as we can see, PROGRES and VIATRA2 are "better", especially because they provide a support for genericity and are more expressive. Thus, it appears clearly that an improvement of these characteristics is desirable for SDM. Nevertheless, we can notice that the understandability and learnability of SDM are better than the ones of PROGRES and VIATRA2. Therefore, it would be important to keep this advantage.

2.5 Typed Languages and Type Checking System

The main purpose of type systems [Car97] is to prevent the occurrence of errors during program execution. We first define some basic concepts in order to provide a better comprehension of this work to the reader. Then, we describe general concepts about type systems.

2.5.1 Definitions

Syntax and Semantics

A DSL (**D**omain-**S**pecific **L**anguage) has syntax and semantics [KM08]. Semantics refers to the meaning of the language, as opposed to its form (syntax). More precisely, a language is defined by its concrete and abstract syntax, and by its static and dynamic semantics.

- **Static vs. dynamic semantics:** The dynamic semantics (or execution semantics) describes the meaning and behavior of the DSL, whereas static semantics essentially includes those semantic rules, also called *well-formedness* rules, that can be checked at compile time and determine well-formed models.

- **Concrete vs. abstract syntax:** The abstract syntax defines the structure of its models, and contains the elements that can be used to create syntactically correct models, whereas the concrete syntax is the actual notation presented to the user. Metamodeling provides a structural definition (ie. abstract syntax) of modeling languages. A *metamodel-compliant model* is an instance of the metamodel and thus complies with the abstract syntax.

Correctness

The concept of correctness for model transformations may have different definitions according to the use of these transformations. In the case of model refinement, model abstraction or model refactoring, a model transformation is syntactically correct if, given a metamodel-compliant source model, it produces a metamodel-compliant target model. In other words, the source model and the target model conform to their abstract syntax. In addition, it is considered as semantically correct if the source model and the target model have the same semantic properties [MCG05]. Though, in the context of this work, the input and output models are not necessarily semantically equivalent. In fact, we use the model transformation language SDM as a programming language in order to fix models according to modeling guidelines (Cf. Chapter 3). Thus, we will consider the correctness as defined for programming languages [SK95].
Syntax defines the formal relations between the elements of a language, and hence, in the case of a visual programming language, it is based on the spatial layout and

connections between symbols. Nevertheless, syntax deals solely with the form and structure of symbols without any consideration given to their meaning. Not all syntactically correct programs are semantically correct. For instance, if we compare a programming language to the natural language: *"Colorless green ideas sleep furiously."* is grammatically well-formed but has no generally accepted meaning [Cho57]. It appears clearly that not only the syntactical correctness but also the semantical correctness of a program or a model transformation are essential for its execution.
Let us resume the concepts of correctness we will use in this work:

- **Syntactically correct model transformation**: it respects the syntax of the model transformation language.
  In the context of our work, this implies that the story diagrams are correctly defined with respect to the syntax of the MOSL language.

- **Semantically correct model transformation**: it respects the static semantics of the model transformation language.
  In the context of our work, it respects the semantics of the MOSL language and, thus, does not violate the constraints of the type system we have defined. As a consequence, a semantically correct model transformation is well-defined, i.e. it produces a metamodel-compliant target model from a metamodel-compliant source model. In other words, the rule's LHS can only match a subgraph from a metamodel-compliant source model, and the rule's RHS can only match a subgraph from a metamodel-compliant target model. In addition, both source and target models fulfill semantic constraints, e.g. about redefinition of association ends.

Typed Languages

Accordingly to [PSW76], a type can be defined in different ways:
1) *Syntactic*: A type is a syntactic label associated with a variable.
2) *Representation*: A type is defined in terms of its composition of more primitive types.
3) *Representation and behavior*: The definition can include the behavior as a set of operators manipulating the representations of the types.
4) *Value space*: A type corresponds to the set of possibles values taken by a variable.
5) *Value space and behavior*: The previous definition is completed by the set of functions which can be applied on the values.

Due to the context of metamodel-based domain-specific languages in our work, we will consider the definition in terms of 'representation and behavior'. A range of values can be assigned to a variable during a method execution. The upper bound of such a range of a variable is the type of the variable. For instance, an attribute of type Boolean is supposed to accept only the Boolean values *true* or *false* during

every run of the program. Languages where variables can be given types are called *typed languages*. In contrast to typed languages, languages that do not restrict the range of variables are called *untyped languages*. Untyped languages do not have any type or, equivalently, that they have a single universal type which contains all values. The untyped λ-calculus is an extreme case of untyped language [Chu36].

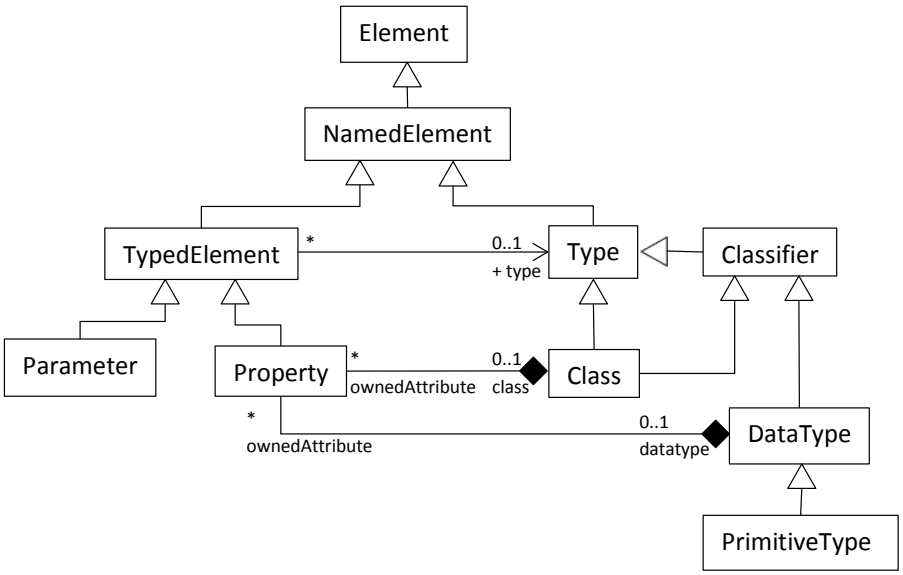


Figure 2.25: Type Concept as defined in [Obj05a]

In the context of this work, we will exclusively consider typed languages. Fig. 2.25 presents the concept of type as defined in the UML Infrastructure Specification [Obj05a].

An *Element* is a constituent of a model, used as the common superclass for all metaclasses, and a *NamedElement* simply represents elements with names. A *TypedElement* is a kind of *NamedElement* that represents elements with types. A *Type* is a *NamedElement* that is used as the type for a *TypedElement*. It represents the general notion of type and constrains the set of values that the *TypedElement* may refer to. A *Parameter* is a *TypedElement* (thus, has a type) and represents a parameter of an operation. When an operation is invoked, an argument may be passed to it for each parameter. A *Property* is a *TypedElement* and represents an attribute of a class or an association end. As *ownedAttribute*, a *Property* may belong to a *Class* or to a *DataType*. A *Classifier* is a *Type*. Its purpose is the classification of instances according to their features: thus, a*Class* or a *DataType* is a *Classifier*. A *Class* is a *Type* and has objects as its instances. The purpose of a class is to specify a classification of objects and to specify the features that characterize the structure and behavior of those objects. In other words, a class is the type of an object. A *DataType* represents different kinds of data types, i.e. a type whose instances are identified only by their value. In other words, a data type is the type of a value.

Another concept related to the concept of type in object-oriented metamodeling is *inheritance*. Classes can inherit attributes and behavior from their superclasses, or parent classes, and the resulting classes are known as subclasses, or child classes. The superclass establishes an interface which subclasses can inherit, modify, and supplement.
A reference to an instance of a class may actually be referring to one of its subclasses. One consequence of this property is the *universal polymorphism* as described in the next section.

Polymorphism

In the context of object-oriented programming, polymorphism is the ability to create a variable, a function, or an object, that has more than one form. For instance, a function that can evaluate to or be applied to values of different types is known as a polymorphic function. There are two main kinds of polymorphism as described by C. Stacey in [Str67]. On the one hand, the ad-hoc polymorphism, and on the other hand, the parametric polymorphism. The ad-hoc polymorphism refers to polymorphic functions which can be applied to arguments of different types, but which behave differently depending on the type of the argument to which they are applied. A function using parametric polymorphism will operate uniformly on a range of types without depending on their type.
Cardelli proposes in [CW85] a refinement of Strachey's classification. He distinguishes between universal and ad-hoc polymorphism as follows:

- **universal polymorphism:** the functions work on an infinite number of types that have some common structure.
  - **parametric polymorphism:** a polymorphic function has an implicit or explicit type parameter which determines the type of the argument for each application of that function.
  - **subtype or inclusion polymorphism:** an object can be viewed as belonging to many different classes that are not necessarily disjoint. In other words, there may be inclusion of classes.
- **ad-hoc polymorphism:** the functions work on a finite set of different and potentially unrelated types.
  - **overloading:** overloading allows multiple functions taking different types for being defined with the same name. The context decides which of the functions is denoted by that particular name. The compiler or interpreter automatically calls the right one.
  - **coercion:** it refers to different ways of, implicitly or explicitly, translating an entity of one data type into another.

A classical example of ad-hoc polymorphism concerns the operator "+":

(1) 1 + 1 = 2
(2) 1 + 1.0 = 2.0
(3) [1,2]+[3,4,5] = [1,2,3,4,5]
(4) "ab" + "cde" = "abcde"

This is ad-hoc polymorphism because the behavior of the operator "+" depends on the type of the operands. The case (1) is an integer addition, and the case (2) is a floating-point addition (with type coercion). In the case (3), the operator "+" acts as a list concatenation, whereas the case (4) invokes a string concatenation.

In the context of this work, we will mainly consider the universal polymorphism: the subtype and the parametric polymorphism. On the one hand, since inheritance belongs to metamodeling and object-oriented programming, the methods whose behavior is specified by means of SDM diagram can use subtype polymorphism. On the other hand, the purpose of this work is to improve the reusability of story diagrams, which can be realized with the help of parametric polymorphism.

2.5.2 Type System

A type system is the part of a typed language which keeps track of the type of variables and, in general, of the types of all expressions in a program. A language is not typed by virtue of type annotations in its syntax, but by virtue of the existence of its type system. Type annotations are not a requirement for a language's typing. We can consider two kinds of typing: manifest typing vs. latent typing. Manifest typing, also called explicit typing, refers to the explicit identification of the type of each variable being declared. Contrary to the manifest typing, latent typing, also called implicit typing, does not require explicit type declaration for each variable. There can be no type annotations, or only some type annotations.
In the context of manifest typing as well as latent typing, checking the type of variables allows for preventing type errors, i.e. errors which occur when the type of an expression does not correspond to the type it is supposed to have. The checking process is called *type checking*, and the algorithm performing this checking is called *type checker*.

Type Errors and Type Safety

A precise definition of type error depends on the language and type system. A type error occurs when the type of an expression does not correspond to the type it is supposed to have. Type errors can be trapped or untrapped. In the context of programming languages, trapped errors cause the computation to stop immediately whereas untrapped errors go unnoticed for a while. A program fragment is *safe* if it does not cause untrapped errors to occur. Consequently, a language is safe if any program fragment expressed using this language is safe. Type safety is determined

by two properties of the language: 1) *type preservation* or subject reduction and 2) *progress* [WF94]. A type system has the first property if evaluation of expressions does not cause their type to change. A type system has the second property if there are no untrapped errors. Consequently, the evaluation of an expression does not get stuck in any unexpected way: either we have a value (and are done), or there is a way to proceed. Since a safe language does not allow for untrapped errors, typed languages may enforce safety by statically rejecting any specification which is potentially unsafe.
Although safety is a crucial property, typed languages generally do not try to eliminate only untrapped errors, but also trapped errors along with the untrapped ones. It is not possible to rule out the complete set of trapped and untrapped errors. A subset composed of all untrapped errors and a subset of trapped errors is called forbidden errors. If no forbidden error occurs, the program is said to have a good behavior and to be *well-typed*. Thus, we can notice that safety contributes to well-typedness, but does not ensure it.

Type Checking

Typed languages can enforce well-typedness by performing type checking. In functional programming languages, type checking allows ensuring the type consistency, i.e. the matching of the function argument with the type of the specified parameter. More generally, type checking associates values or expression with type information in order to verify and enforce the type constraints of a type system.

Type enforcement can be *static*, or *dynamic*, or a combination of both.
Static type checking can catch potential errors *at compile time*, whereas dynamic type checking associates type information with values *at runtime* and consults them as needed to detect imminent errors. Both approaches possess their pros and their cons. On the one hand, stating typing generally results in compile code which can be executed more quickly and efficiently. In other words, the program is faster and less memory is required. On the other hand, static type enforcement is limited and conservative since some information are only available at runtime. Thus, some method specifications are rejected due to the possibility to determine at compile-time whether they are type-safe or not. In return to a slower execution, dynamic typing allows compilers to run more quickly and allow interpreters to load new code dynamically.

A type system can allow for a *weak typing* or a *strong typing*.
In [LZ74], a language is defined as strongly typed if, *"whenever an object is passed from a calling function to a called function, its type must be compatible with the type declared in the called function"*. A type system features strong typing when it places one or more severe restrictions on how operations involving values of different data types can be intermixed. For instance, an addition operation may not allow for adding an integer with a string value. The more type restrictions are imposed,

the more strongly typed a programming language is. If a type system is strongly enough typed, it can ensure type safety by the rejection of method calls which do not respect data types.
The opposite of strong typing is weak typing. Weakly typed languages support either implicit type conversion, ad-hoc polymorphism or both. In return to fewer errors catched at compile time compared to strongly typed languages, weakly typed languages require less effort on the part of the user because the compiler or interpreter implicitly performs certain kinds of conversions.

To ensure type safety, it is necessary to be able to determine compatibility and equivalence between expressions. For instance, if we consider two distinct type names associated with a similar type:
type X = Boolean
type Y = Boolean
On the one hand, we can consider that types X and Y match because both are associated to the same type. Though, on the other hand, we can consider that types X and Y are not equivalent due to their distinct names. Therefore, we can determine different classes of type systems:

- **Nominative type system** (also called *name-based type system*):
  The compatibility and equivalence of data types is determined by explicit declarations and/or the name of types. It means that two variables are type-compatible if and only if their declarations name the same type, or in other words, that types are not identified unless explicitly declared. In the same way, nominal subtyping means that one type is a subtype of another if and only if it is explicitly declared to be so in its definition.

- **Structural type system** (also called *property-based type system*):
  This class of type system is often considered as the opposite of the nominative base system. Type compatibility and equivalence are namely determined by the type's actual structure or definition, and not by other characteristics such as its name. An element is considered to be compatible with another if for each feature within the second element's type, there is a corresponding and identical feature in the first element's type, e.g. when an object can be cast to an interface.

- **Duck typing**:
  This kind of typing is pretty close to the structural typing since it is based on the objects' methods and properties. But the difference between structural typing and duck typing is that only those aspects of an object that are used, rather than with the type of the object itself, are relevant. In other words, a structural type system requires that the object possesses *all* properties and methods corresponding to a type T in order to infer that the object's type is the type T, whereas only a subset of relevant methods and properties will be enough to infer the same conclusion in the case of duck typing. The name

of the concept refers to the "duck test" which may be phrased as follows: *"When I see a bird that walks like a duck and swims like a duck and quacks like a duck, I call that bird a duck."(James Whitcomb Riley)*, i.e. that if a bird possesses *some*, but not necessary *all*, characteristics belonging to a duck, then, you can consider that this bird is a duck.

As already mentioned, typing can be manifest or latent. Latent typing is generally associated with duck typing in a context of dynamic typing. Though, latent typing can be achieved by static typing, too. If there are no type annotations, the type of an expression is automatically deduced. This approach is called *type inference*. If some (but not all) type annotations are present, this deduction of type at compile time is referred to as *type reconstruction*.

2.5.3 Type Reconstruction Formalism

Type inference/reconstruction systems are able to infer types from a program specification, with or without some type annotations. For simplicity, we do not distinguish between type inference and type reconstruction in the rest of this work.

A term *t* which has the type τ is noted: *t:*τ. Then, the general type reconstruction problem can be formulated as:

Given a well-formed term t without any types, does there exist a type τ *such that we can obtain t:*τ.

Terms and types are related by assertions in a typing environment. A typing environment can be expressed e.g. as a list of variables and their types. The empty environment is denoted by $\varnothing$. The description of a type system generally starts with a collection of *judgments* which have typically this form: $\Gamma \vdash \mathcal{J}$, which means that the typing environment Γ entails the assertion $\mathcal{J}$. For instance, a typing judgment which asserts that a term *t* has the type τ with respect to the environment Γ has this form: $\Gamma \vdash t : \tau$. Another classical judgment which simply asserts that an environment is well-formed is: $\Gamma \vdash \diamond$. An example of judgment entailed by the empty environment is $\varnothing \vdash true : Boolean$. Any judgment can be valid or invalid, and the well-typedness is formalized by valid judgments.
A *derivation* of an assertion *t:*τ is a finite sequence of assertions ending with *t:*τ. Every assertion in that sequence is either an instance of an axiom or is derivable from *type rules*. Type rules are specified as rules of inference (= inference rule, or transformation rule) They assert the validity of assertions on the basis of other assertions whose validity has already been ensured. Each rule has a name and is written as a number of premise judgments (zero, one or more) above a horizontal line with a single conclusion judgment below:

$$\frac{\Gamma_1 \vdash \mathcal{J}_1 ..... \Gamma_n \vdash \mathcal{J}_n}{\Gamma \vdash \mathcal{J}} \quad \textit{(RuleName)}$$

If all premises are valid, the conclusion must hold and we can infer the validity of the conclusion judgment. A collection of type rules form a formal type system. If a derivation is based on type rules, this derivation can be described as a tree of judgment with leaves at the top and a root at the bottom. Fig. 2.26 shows an example of derivation. The rules *Env* $\varnothing$ at the top are the start point. Since there are no premises, these rules always apply and we conclude that the empty set of variables is well-typed. The next rules are *Val n*. These rules infer that 1 resp. 2 are natural numbers ($\varnothing \vdash 1 : Nat$ and $\varnothing \vdash 2 : Nat$). These conclusions become the premises of the last rule *Val +* which concludes that the sum *1+2* belongs to the natural numbers *Nat*.

$$\dfrac{\dfrac{\dfrac{}{\emptyset \vdash \diamond}\ \text{by (Env } \emptyset\text{)}}{\emptyset \vdash 1 : Nat}\ \text{by (Val } n\text{)} \qquad \dfrac{\dfrac{}{\emptyset \vdash \diamond}\ \text{by (Env } \emptyset\text{)}}{\emptyset \vdash 2 : Nat}\ \text{by (Val } n\text{)}}{\emptyset \vdash 1{+}2 : Nat}\ \text{by (Val +)}$$

Figure 2.26: Type Rule Derivation - Example [Car97]

The discovery of a derivation, and thus of a type, for a term corresponds to the type inference problem. For instance, the derivation in Fig. 2.26 allows for inferring the type *Nat* for the term *1+2* in the empty environment. Depending on the rules composing the type system, derivations for a term can be found or not, which allows for concluding about the typedness.
If we add a rule with premise $\Gamma \vdash \diamond$ and conclusion $\Gamma \vdash true : Bool$ to the previous rules, this will not be enough to infer any type for the term *1+true* due to the absence of a rule for adding a natural numbers to a boolean. Because it is not possible to find any derivation for *1+true*, the term *1+true* is designed as *not typeable* or *ill typed*. In other words, there is a *typing error*.
Though, the term *1+true* becomes well-typed if it becomes possible to infer a type for the term *1+true*. This is the case if we interpret *true* as 1 and complete the type system with this rule:

$$\frac{\Gamma \vdash M : Nat \quad \Gamma \vdash N : Bool}{\Gamma \vdash M{+}N : Nat}\ \textit{Val n+b}$$

As we can see, the definition of the type system is crucial in the context of type inference to detect the presence or the absence of typing errors. In addition, the construction of derivations in order to infer the type of a term requires the definition of an algorithm. The practical use of a type system depends closely on the availability of an efficient algorithm. Depending on the type system, a type inference algorithm may be very easy, very hard, or even impossible to find. If found, its execution may be very efficient or, on the contrary, very slow. The type inference problem becomes particularly hard in the presence of polymorphism.

In the context of this work, we will present a type system for SDM as graph transformation language of MOSL in order to prevent type errors. We describe this approach in Chapter 5.

Chapter 3

Modeling Guidelines

As we all know the importance of model driven engineering principles for the development of safety-critical software is continuously increasing. Quite a number of standards have been developed or are still under development that ask for the application of semi-formal or even formal model-driven engineering concepts when software of a certain safety integrity level (SIL) is developed. Examples of this kind are the domain independent standard IEC 61508 [IEC10] or the automotive domain specific standard ISO 26262 [ISO10]. The MATLAB Simulink/Stateflow (SL/SF) [Sim10] environment is preferred in the automotive industry for model-driven software development purposes. It namely supports these standards, and better meets the developers requirements for specifying, designing, implementing, and checking the functionality of new control functions than other modeling languages like UML.
Generally accepted modeling guidelines are usually adopted to improve the correctness and the efficiency of models The MATE project and its successor the MAJA project aim at automatically ensuring the respect of the modeling guidelines. These projects will be used in this work as application of our approach since they are based on the specification of a MATLAB metamodel and guidelines implemented by means of SDM.

3.1 MATE and MAJA Projects

In model driven development of automotive embedded software, de facto standard modeling and simulation tools such as MATLAB/Simulink/Stateflow are used for specifying, designing, implementing, and checking the functionality of new control functions. To improve the correctness and the efficiency of models and prevent typical modeling problems, generally accepted modeling guidelines such as the MathWorks Automotive Advisory Board (MAAB) catalogue are usually adopted. Due to the increasing complexity of the developed systems, the models have become intricate and huge. As a consequence, the analysis and correction of models have become time-consuming and error-prone tasks. The MATE/MAJA projects

which are described in this section have originally be motivated by the urgent need for automation of these tasks. In the rest of this thesis, MATE/MAJA provides a concrete application example of our contribution for model transformations.

3.1.1 Context

In the automotive industry, the model-driven development has become a standard. MATLAB/Simulink/Stateflow, as domain-specific language, provides the suitable support for specifying, designing, implementing, and checking the functionality of new control functions. Embedded controller software is either manually developed by programmers using MATLAB SL/SF models as executable requirements specifications or generated automatically by code generators which translate MATLAB SL/SF models into rather efficient C code. In both cases the reliability, robustness, and efficiency of the developed code heavily depends on the quality of the specified models. The reliability and robustness of the code are essential in the context of safety-critical systems like many automotive embedded software. The efficiency is also important since the resources of the embedded system on which the generated code is running are limited. Therefore, generally accepted modeling guidelines such as the MathWorks Automotive Advisory Board (MAAB) guidelines [Mat07] are usually adopted. We will present some MAAB guidelines in Section 3.1.2. The guidelines support the developer in preventing typical modeling problems, for instance non-connected elements. They also define conventions such as naming convention or element setting that facilitate the team working and improve the reusability of the models.

The motivation for the MATE project [SSSL10] was to provide support for semi-automatic checking and enforcement of modeling guidelines as well as for version management, design pattern instantiation, and interactive model refactoring and beautifying operations. It was a joint project of two companies (DaimlerChrysler, Model Engineering Solution) and four universities (Technical University of Darmstadt, University of Kassel, University of Paderborn, University of Siegen).
This project was born out of an urgent need of the automotive industry for more sophisticated tool support in this area. The models have become intricate and huge due to the increasing complexity of the developed systems. The modeling guidelines support the developer in preventing typical modeling problems. Though, a guideline catalog such as MAAB contains more than fifty guidelines. The analysis of a model may lead to a report of up to a few hundred or thousands of violations. In addition, these guideline violations must be corrected manually by the modeler: a cumbersome, elaborative, and also expensive task. The significance of model reviewing is supported by a case study [SCFD06] which presents 146 critical model changes due to findings from a multi-iterative reviewing process on a model of 9308 blocks. All in all, the reviewing took 1600 minutes, nearly 27 hours. Since reviewing is such a time-consuming and thus expensive process, it is highly desirable to automate the review and correction of models. This was precisely the

purpose of the MATE project. MATE aimed at providing analysis of MATLAB SL/SF models as well as automated and interactive repair functions (i.e. functions that requires feedback or additional information from the user).
MAJA [MAJ] was the successor of the MATE project. It focused on the development of an online adapter, and aimed especially at the enhancement of the semi-automated adapter generation. The application of the generated adapter in the MAJA project is the checking and correction of MATLAB SL/SF models too.

3.1.2 MAAB Guidelines

The MAAB [MAA] was originally established in 1998 to coordinate feature requests from several key customers in the automotive industry such as Ford, Daimler or Toyota. One of the core output of the MAAB is its catalog of modeling style guidelines for MATLAB SL/SF [Mat07] which belongs nowadays to the industry standards. The catalog is divided into several guideline aspects:

- **Naming conventions**: these guidelines define conventions about the naming of elements, for instance which characters are allowed and which are forbidden.
- **Model architecture**: these guidelines indicates whether Simulink or Stateflow should be used to model a given kind of functionality, or which kinds of elements are allowed on which layer of the architecture.
- **Model configuration options**: these guidelines precise the recommended tool setting and parameter, e.g. model diagnostic settings.
- **Simulink**: theses guidelines concern specifically Simulink. This set of guidelines is also divided into aspects such as “diagram appearance”, “signals” or “block parameters”.
- **Stateflow**: theses guidelines concern specifically Stateflow. This set of guidelines is also divided into aspects such as “chart appearance” or “Stateflow data and operations”.

The MAAB guidelines are defined accordingly to the template of Fig. 3.1.a. The *priority* describes the importance of the guideline and, hence, determines the consequences of violations. If a mandatory guideline is not respected, this means that the model may not work properly, whereas the violation of a recommended guideline will simply lead to a non-conformance of the model appearance to other projects. The *scope* indicates if the guideline concerns the complete MAAB group, or only one of its subgroup J-MAAB (= Japan MAAB) or NA-MAAB (= North American MAAB). Rules with J-MAAB scope are local to Japan, whereas rules with NA-MAAB scope are local rules in Europe and USA. The application of a guideline may require fulfilled modeling rules as prerequisites. In this case, the ID and title of the guideline(s) are indicated in the *Prerequisites* field. The *Description*

field contains a detailed description of the guideline in natural language. If needed, images and tables can be added. The field called *Rational* indicates the reason(s) why the guidelines are recommended: *Readability*, *Workflow*, *Simulation*, *Verification and Validation*, or *Code generation*.

a)

ID: Title	**XX_nnnn: Title of the guideline (unique, short)**
Priority	One of mandatory / strongly recommended / recommended
Scope	MAAB, NA-MAAB, J-MAAB, Specific Company (for optional local company usage)
MATLAB® Version	all RX, RY, RZ RX and earlier RX and later RX through RY
Prerequisites	Links to guidelines, which are prerequisite to this guideline (ID+title)
Description	Description of the guideline (text, images)
Rationale	Motivation for the guideline
Last Change	Version number of last change

b)

ID: Title	**db_0081: Unconnected signals and block inputs / outputs**
Priority	Mandatory
Scope	MAAB
MATLAB Version	All
Prerequisites	
Description	A system must not have any: • Unconnected subsystem or basic block inputs. • Unconnected subsystem or basic block outputs • Unconnected signal lines • An otherwise unconnected input should be connected to a ground block • An otherwise unconnected output should be connected to a terminator block **Correct**  **Incorrect** 
Rationale	☑ Readability ☑ Verification and Validation ☑ Workflow ☐ Code Generation ☐ Simulation
Last Change	V2.0

Figure 3.1: MAAB Guideline - Template and Example

The part b of Fig. 3.1 shows an example of modeling guideline whose ID is *db_0081* and whose title is *Unconnected signal and Block Inputs/Outputs*. This guideline has a *mandatory* priority because it concerns not only the readability, but also the workflow, and the verification and validation. The respect of this rule ensures that no block input or output remains unconnected. The text of the description indicates

also how to repair the model in case of violation of this rule, i.e. by connecting an input to a ground block or an output to a terminator block. The text is completed by a screenshot which represents a correct and an incorrect case.
This example shows that the guideline description in the MAAB catalog is pretty understandable for a human, but cannot be used directly for automated analysis and repair of model. We will present in Section 3.3 how such guidelines may be formally specified by means of model transformations.

3.1.3 MATE Architecture

Analysis as well as refactoring of MATLAB SL/SF models demands full access to MATLAB's model repository. Such an access is provided by an API written in M-Script, a proprietary script language. Both the used C-like scripting language and the tool's API evolved over many years. As a consequence, it takes quite some time and efforts to learn how to program reliable model checks and transformations using this approach. The MATE project overcomes these problems by providing a layer of uniform API adapters on top of which visual graph queries and transformations can be developed on a considerably higher level of abstraction.

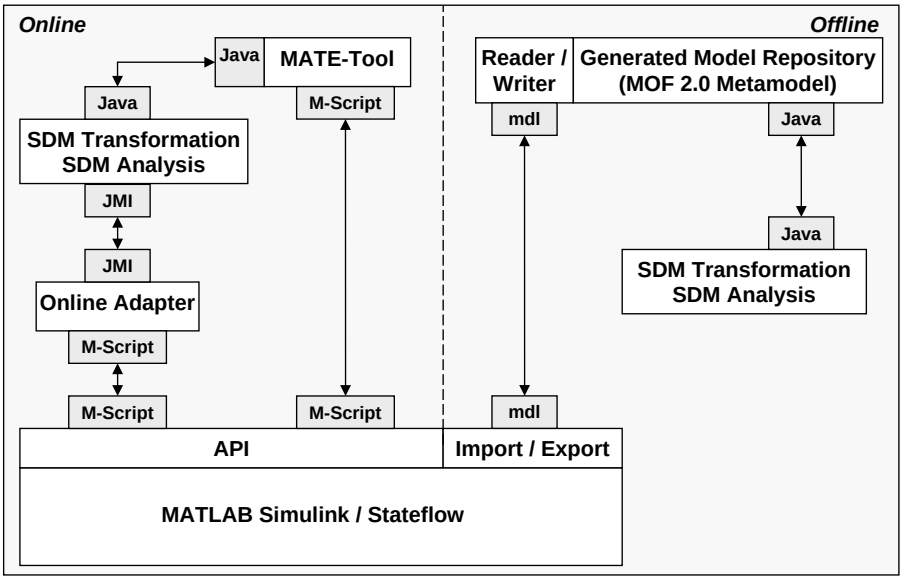


Figure 3.2: MATE Architecture

MATE provides two ways to analyze and repair models: *online* and *offline*. The corresponding architecture is represented in Fig. 3.2. The online-modus enables an interactive analysis of a model within MATLAB whereas the offline-modus works on the data representing the models to be checked and is generally used for complex analyses and corrections. The online-modus requires a communication between MATE and MATLAB. This communication is realized through an *Online-Adapter*

which supports all required read-and-write operations. The *MATE-Tool* represents the Java application controlling the execution of the analysis and transformation operations. To execute the offline-modus, the models are exported in their proprietary mdl-format, and imported in the *Generated Model Repository* through the *Reader/Writer* as instances of the MATLAB SL/SF metamodel. The module *SDM Transformation - SDM Analysis* defined in both parts (online and offline) of the MATE architecture represents the specifications of model analysis and correction.

The specification of analysis and repair actions by means of visual graph queries and transformations requires the metamodeling of MATLAB SL/SF. A simplified version of this metamodel is presented in Section 3.2.2. Section 3.3 describes the specification of guidelines as graph transformation on this metamodel.

3.2 MATLAB Simulink/Stateflow Metamodel

In the center of guideline specifications by means of graph transformations, there is a metamodel of the particular language, namely MATLAB Simulink/Stateflow in the case of MATE. We first give an overview of this language, before a description of the metamodel.

3.2.1 MATLAB Simulink/Stateflow

MATLAB (for **MAT**rix **LAB**oratory) is a numerical computing environment developed by MathWorks [MAT10c]. MATLAB allows for matrix manipulation, and for the development of algorithms and applications. It supports the entire data analysis process, from the data acquisition to the production of output. It also provides graphic features to visualize engineering and scientific data, such as 2-D and 3-D plotting functions or 3-D volume visualization functions.

Simulink is a tool that offers tight integration with the rest of the MATLAB environment. It can either drive MATLAB or be scripted from it. Simulink adds to MATLAB graphical multi-domain simulation and model-based design for dynamic and embedded systems.
The primary interface of Simulink is a graphical block diagramming tool and a customizable set of block libraries. The part a of Fig. 3.3 shows the Simulink Library Browser. There are several categories of block such as the *Commonly Used Blocks* shown on Fig. 3.3.a, the *Logic and Bit Operations*, the *Math Operations* or the *Ports & Subsystems*.

The part b of Fig. 3.3 depicts a very simple example of Simulink model. The function of the modeled system is the display of the product of the two constant inputs. The window “product” represents the root system, i.e. the highest level of the model. *Constant_1* and *Constant_2* are two constant blocks and represents the

sources of the system. *Scope* is the output of the system, displaying the result of the operation after a simulation. The blocks possess input and output ports, and are connected to each other by lines. These lines are signals and represent the dataflow within the system. A Simulink model can be defined over several level by encapsulating functions into subsystem blocks. The window "product/Subsystem" displays

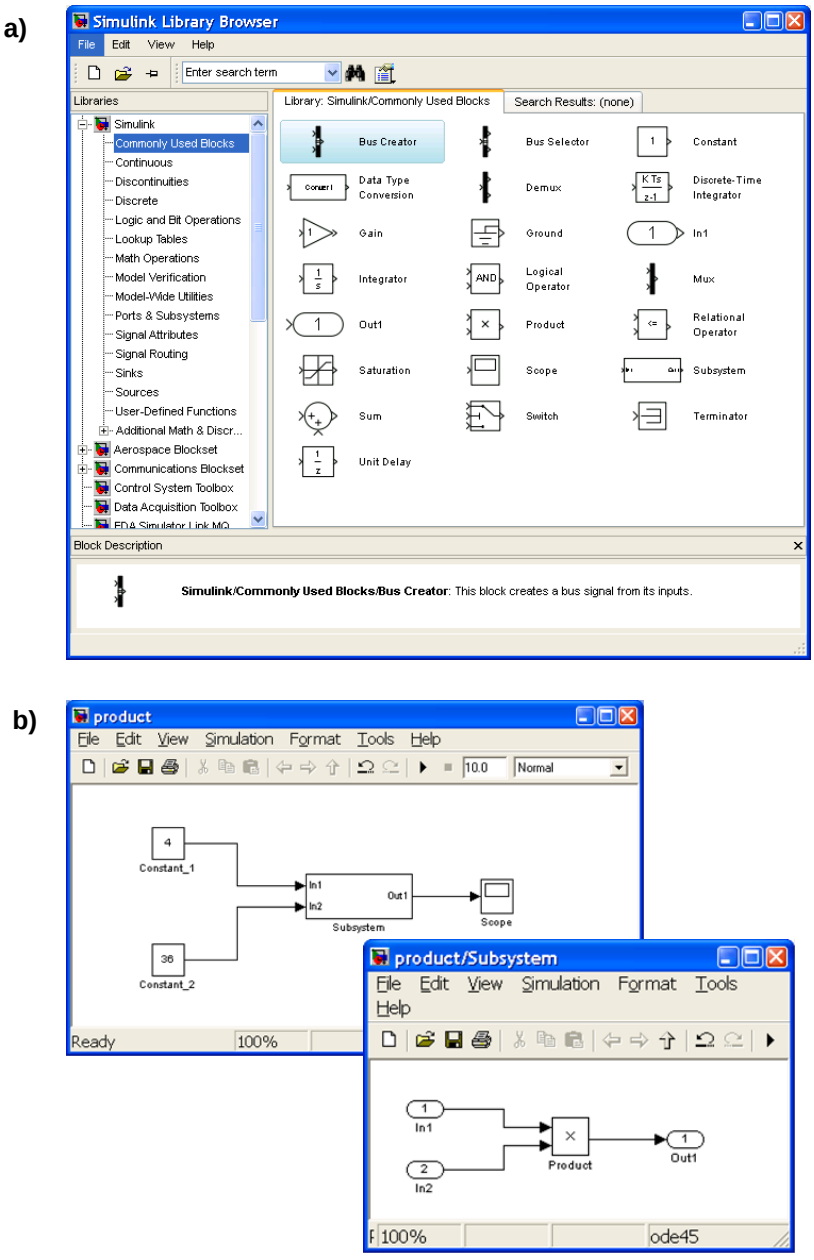


Figure 3.3: Simulink Libraries and Example

the system that is encapsulated in the subsystem block *Subsystem*. In this example, it is simply the product function. *In1* and *In2* are inport blocks, corresponding to the input port of *Subsystem*, and *Out1* is an outport block, corresponding to the output port of *Subsystem*.

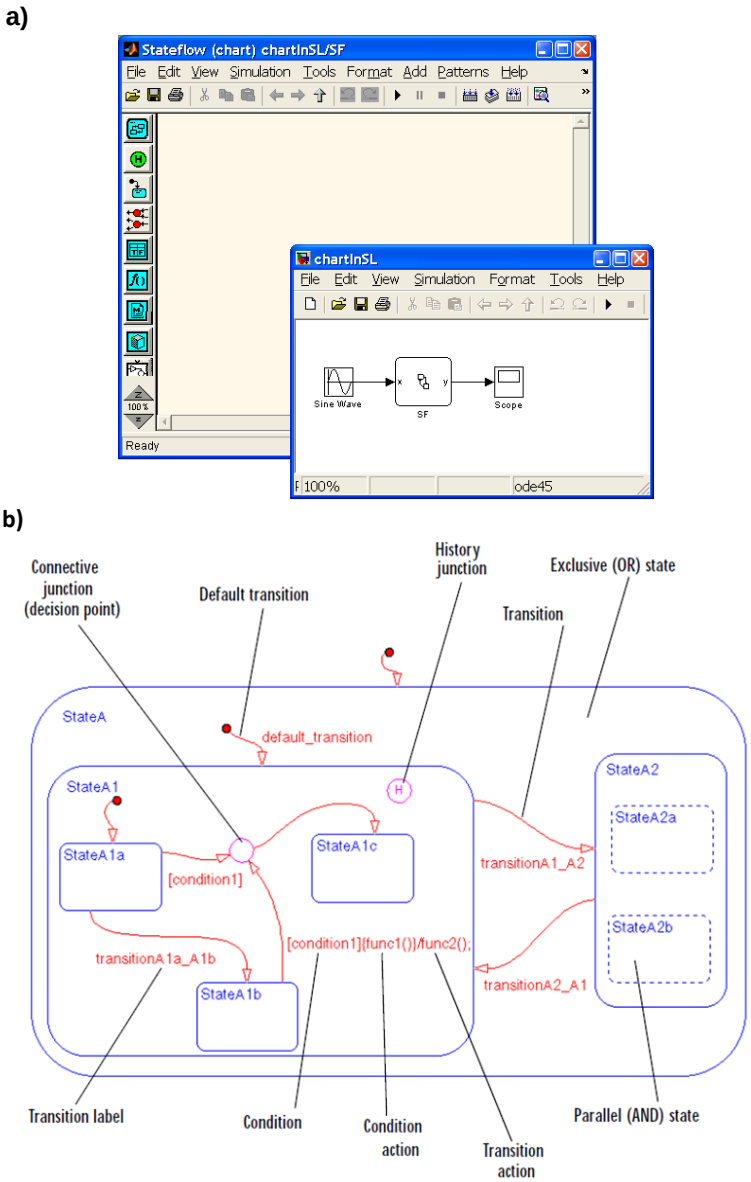


Figure 3.4: Stateflow - Graphical Objects [Mat00]

A number of hardware and software products are available for use with Simulink. Stateflow is one of these products. It is a control logic tool used to model reactive systems via state charts and flow diagrams within a Simulink model. The part a

of Fig. 3.4 shows how a Stateflow diagram can be integrated in a Simulink model. The Library Browser provides specific kind of block "Stateflow/Chart". By clicking on this block, the user opens the Stateflow editor (window "Stateflow(chart) chartInSL/SF"), and can add input data and events from Simulink as well as output data and events to Simulink. In the example of Fig. 3.4.a, *x* is an input data and *y* is an output data.
The part b of Fig. 3.4 gives an overview of the graphical Stateflow objects. These are typical feature for the specification of state machines, such as exclusive and parallel states, transitions, conditions and actions. We will not describes these features in detail because it would be out-of-scope in this document. The interested reader can find an extensive description in [Mat10b].

3.2.2 Metamodel

A metamodel of MATLAB SL/SF is the center of the specification of guideline analysis and correction within MATE. Because MATLAB SL/SF is a powerful and complex language, its complete metamodelling is itself intricate and extensive. MATE is in the context of this thesis only an application of our approach. Therefore, for the sake of clarity, we only present an simplified version of the metamodel.

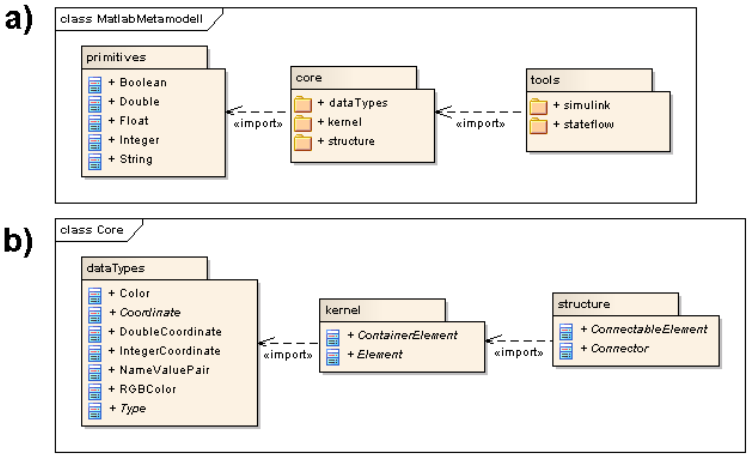


Figure 3.5: Simplified MATLAB SL/SF Metamodel

Fig. 3.5.a represents the structure of the metamodel. The package `primitives` contains the definition of primitives such as *String*, *Boolean* or *Integer*. The package `core` contains generic elements and relationships, usable for Simulink as well as for Stateflow. Therefore, the package `tools` which contains the packages `simulink` and `stateflow` imports the package `core`.
The part b of Fig. 3.5 depicts the subpackages of `core`. The subpackage `kernel` of `core` contains *Element* and *ContainerElement* The class *Element* is an abstract class that may represent an element of Simulink or Stateflow. *ContainerElement*

is a subclass of *Element*. An association between theses classes models the containment relationship. The subpackage `structure` of `core` contains *Connector* and *ConnectableElement Connector* and *ConnectableElement* are subclasses of *Element* too. These classes are bound to each other by two associations. This represents that an instance of *ConnectableElement*, as source, may be related to another, as target, by means of an instance of *Connector*. Fig. 3.7 depicts more precisely these relationships.

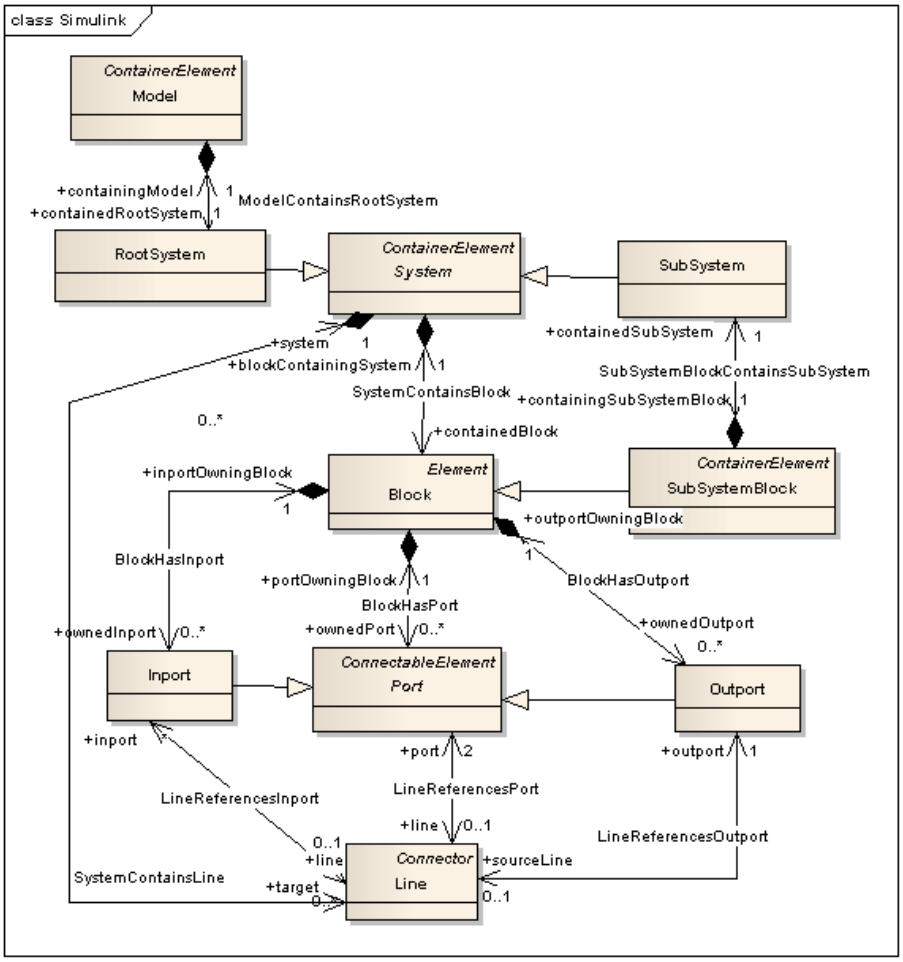


Figure 3.6: Simplified Simulink Metamodel

Fig. 3.6 is a view of the Simulink part of the metamodel. A *Model* contains a *RootSystem*. This *RootSystem* is the *System* on the highest level of the *Model*. In the example of Fig. 3.3.b, the system represented in the window "product" is the root system of the model. A *System* contains *Blocks* and *Lines*. *Lines* represents the signals between the Simulink blocks. *Blocks* have *Inports* and/or *Outports*. A *Port*, which is a *ConnectableElement*, is a connection point for a *Line*. The different

kinds of blocks contained in the Simulink libraries (Fig. 3.3.a) are modeled as subclasses of *Block*, e.g. *SubSystemBlock*. A *SubSystemBlock* contains a *SubSystem* which is a *System*. In the example of Fig. 3.3.b, the system represented in the window "product/Subsystem" is the subsystem contained in the block "Subsystem".

Our metamodel uses the concept of association redefinition [CG06]. An association has two ends that are represented by properties. A redefinition allows for defining an association end more specifically. The concept of redefinition involves not only association but also generalizations. Fig. 3.7 depicts some redefinition examples.
Fig. 3.7.a shows the redefinition of the association ends between *Connector* and *ConnectableElement*. Both classes belong to the package `core` which contains generic elements, usable for Simulink as well as for Stateflow. *Port*, which are specialized into *Inport* and *Outport*, are Simulink *ConnectableElement*, and *Line* are Simulink *Conector*. The association *LineReferencesInport* between *Inport* and *Line* corresponds to the association *TargetConnectableElementReferencesConnector*. Consequently, the property *targetLine* redefines the property *targetConnector*, and the property *inport* redefines the property *targetConnectableElement*. In the same way, *LineReferencesOutport* between *Outport* and *Line* corresponds to the association *SourceConnectableElementReferencesConnector*. Thus, *sourceLine* redefines *sourceConnector*, and *outport* redefines *sourceConnectableElement*. Because *Inport* is a *ConnectableElement* and consequently inherits all its properties , it has not only a *targetConnector* but also a *sourceConnector*. Though, according to the Simulink language, an *Inport* can be related only to a single *Line* which has the role of *targetConnector*. An *Inport* should not have another *Connector* as *sourceConnector*. The end *src* of the association between *Inport* and *UndefinedConnector* redefines *sourceConnector*. Because the class *UndefinedConnector*, subclass of *Connector*, is an abstract class, it cannot be instantiated. This ensures that instances of *Inport* in Simulink models will not have any *sourceConnector*. Similarly, the end *trg* of the association between *Outport* and *UndefinedConnector* redefines *targetConnector*. This prevents instances of *Outport* in Simulink models from having any *targetConnector*.
In Stateflow, *Transition* are *Connector* between states and conjunctive junctions, generalized by the class *Vertex*. The association *TransitionReferencesTarget* between *Transition* and *Vertex* corresponds to the association *TargetConnectableElementReferencesConnector*. Thus, the property *transitionToTarget* redefines the property *targetConnector*, and the property *target* redefines the property *targetConnectableElement*. Similarly, *TransitionReferencesSource* corresponds to *SourceConnectableElementReferencesConnector*. Hence, *transitionToSource* redefines *sourceConnector*, and *source* redefines *sourceConnectableElement*.
Each of the set of types connected by the redefining association end must conforms to a corresponding type connected by the redefined association end. For instance, *transitionToSource* could not redefine *sourceConnectableElement* since *Transition* is not of type *ConnectableElement*. In the case of association redefini-

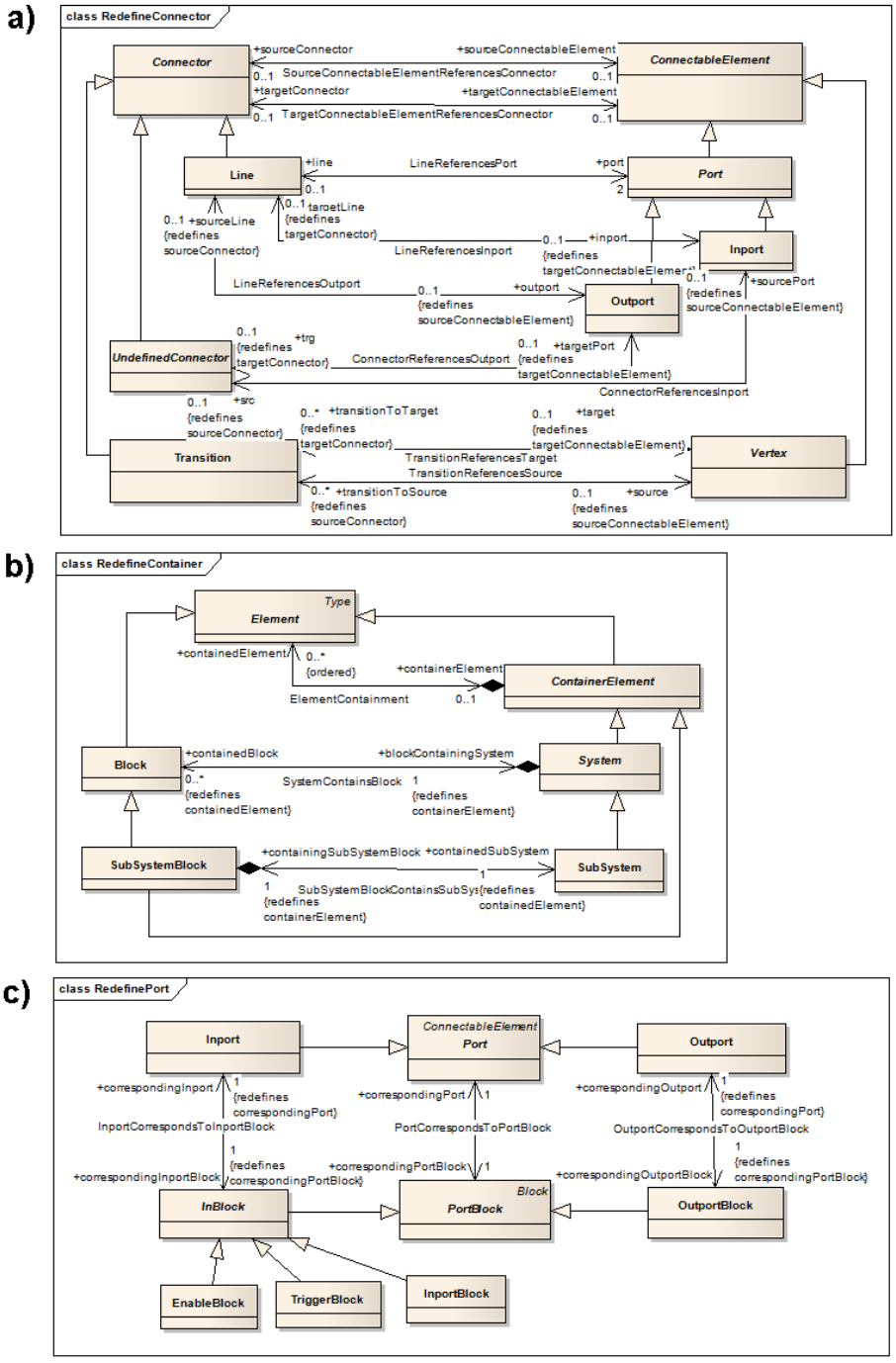


Figure 3.7: Redefinitions of Properties

tion, the collections denoted by the redefining and redefined ends are the same. In our metamodel, for an instance of *Vertex*, the set of *sourceConnector* is the same as the set of *transitionToSource*. The use of redefinition in the metamodel allows for specifying that *Inport* and *Outport* can be connected only by *Line*, and *Vertex* only by *Transition*, without having to define contraints such as OCL contraints.
Fig. 3.7.b depicts the containment relationships between *Block*, *SubSystemBlock*, *System*, and *SubSystem*. A *System* is a *ContainerElement*, and contains *Block* that is an *Element*. Consequently, the property *blockContainingSystem* redefines the property *containerElement*, and the property *containedBlock* redefines the property *containedElement*. A *SubSystem* belongs (indirectly) to the subclasses of *Element*. A *SubSystemBlock* is a subclass of *ContainerElement* which contains a *SubSystem*. Thus, the property *containingSubSystemBlock* redefines the property *containerElement*, and the property *containedSubSystem* redefines the property *containedElement*.
A *PortBlock* is the link from outside a *SubSystemBlock* into the contained *SubSystem*. In other words, a *PortBlock* in a *SubSystem* corresponds to a *Port* of the containing *SubSystemBlock*. In the example of Fig. 3.3.b, the blocks *In1* and *In2* are the inport blocks that correspond to the both input ports of the subsystem block *Subsystem*. The block *Out1* is the outport block that corresponds to the output port of *Subsystem*. Fig. 3.7.c shows the relationship between the classes *PortBlock* and *Port*, as well as between their subclasses. Since an *InBlock* (which can be an *InportBlock*, a *TriggerBlock* or an *EnableBlock*) must correspond to an *Inport*, and an *OutportBlock* to an *Outport*, the properties *correspondingPort* and *correspondingPortBlock* are redefined accordingly.

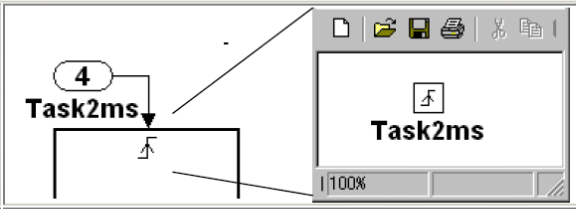
3.3 Guideline Specification with SDM

A key concept of MATE is the specification of guideline analysis and correction in the form of graph transformation. We describe in this section specification examples, expressed by means of SDM as well as by other languages. Hence, we can show to what extent SDM is suitable for the specification of modeling guideline. In addition, we can determine which properties should be improved or added to the current state of SDM.

3.3.1 Examples of Guideline Specifications

As explained in 3.1.2, the MAAB guidelines are described in natural language according to a given template. Let us consider the example of the guideline jc_0281 (Cf. Fig. 3.8.a). This guideline concerns the naming of trigger or enable block. An enable block in a subsystem makes it a enabled subsystem, and a trigger in a subsystem makes it a triggered subsystem. As a consequence, the containing subsystem block gets an enable or a trigger port. An enabled subsystem executes while the input received at the enable port is greater than zero. A triggered subsys-

a)

ID: Title	jc_0281: Naming of Trigger Port block and Enable Port block
Priority	strongly recommended
Scope	J-MAAB
MATLAB Version	All
Prerequisites	
Description	For Trigger port blocks and Enable port blocks • The block name should match the name of the signal triggering the subsystem.
	
Rationale	☑ Readability ☐ Verification and Validation ☐ Workflow ☐ Code Generation ☐ Simulation
Last Change	V2.0

b)

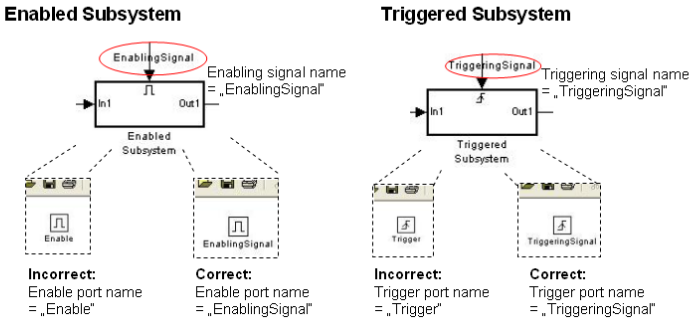


Figure 3.8: MAAB Guideline jc_0281

tem executes each time a trigger event occurs. The guideline jc_0281 improves the readability of the models by ensuring that the name of the enable or trigger block matches the name of the corresponding input signal. Fig. 3.8.b illustrates correct and incorrect cases. A violation of this guideline may be repaired by renaming the incorrect trigger or enable block.

The analysis and correction of this guideline within MATE is expressed in the form of a story diagram. Fig. 3.9.a shows the element from the MATLAB metamodel which are relevant for the analysis and the correction. The trigger and enable blocks are represented by the classes *TriggerBlock* and *EnableBlock* that specialize the class *Block* via the class *EnableTriggerBlock*. The association between this class and the class Inport represents the relation between the enable or trigger block contained in the subsystem and the corresponding input port of the containing subsystem block.

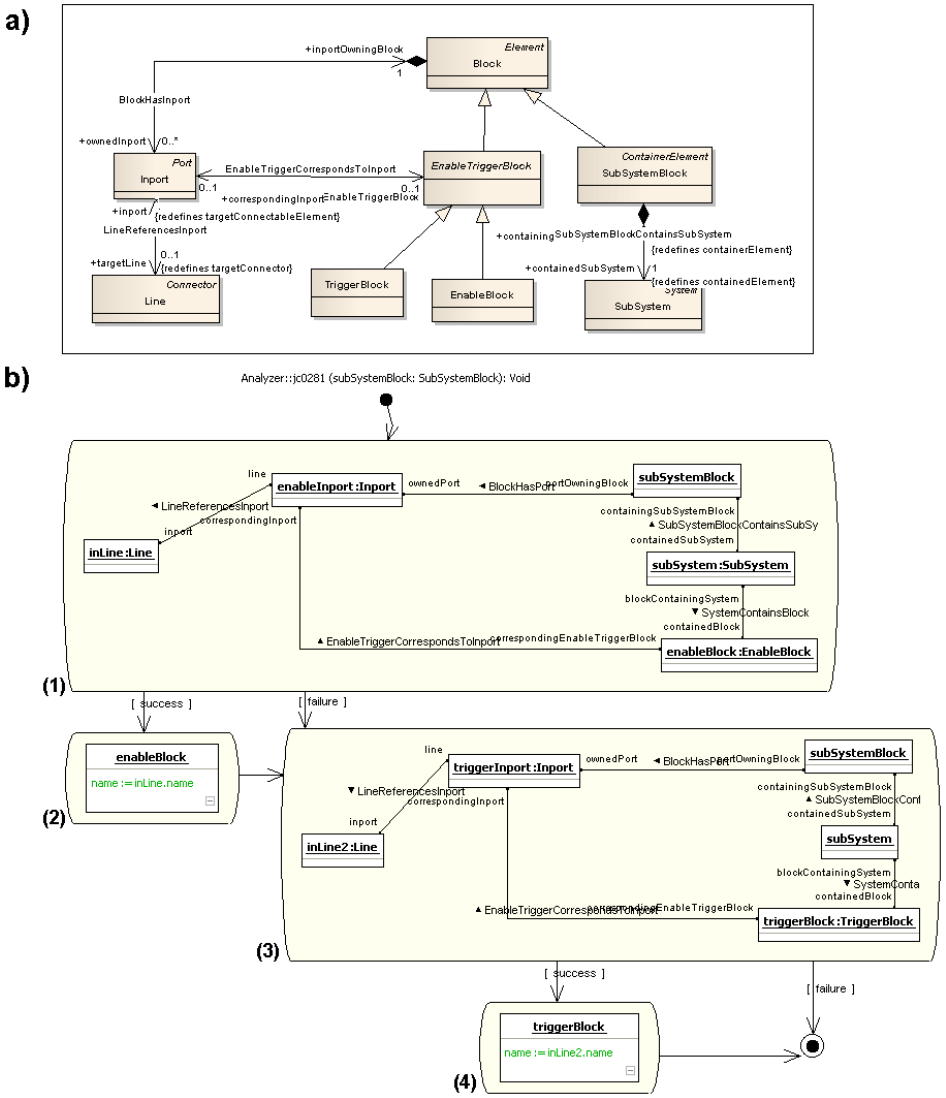


Figure 3.9: MAAB Guideline jc_0281 - SDM Diagram

Fig. 3.9.b represents the SDM diagram that executes the analysis and the correction of the guideline jc_0281. The method analyzes the subsystem block given as parameter. The first activity tries to match the following pattern. It checks if the contained subsystem is an enabled subsystem, i.e. if it contains an enable block (*enableBlock*) corresponding to an enable input port (*enableInport*). If this pattern is matched, the activity 2 is executed: the name of *enableInport* is, if necessary, corrected and set to the name of the incoming signal (assignment with the ":=´´-operator to the value *inLine.name*). If the subsystem is not an enabled subsystem, the activity 3 is directly executed. This activity is similar to the activity 1, except that it checks if the subsystem is triggered and not if it is enabled. Similarly to

the activity 2, the activity 4 corrects, if necessary, the name of the trigger block by assigning the name of the trigger signal.

Another example of modeling guideline is the guideline db_0081 (Cf. Fig. 3.1.b). This guideline ensures that every element of a Simulink model is connected. Unconnected subsystems, basic block inputs, outputs or unconnected signal lines are not allowed. Thus, this rule is rather important for the structural correctness of a model. A violation of this rule inevitably leads to an erroneous model. According to the guideline description, the repair action consists in connecting the unconnected inputs to ground blocks, and the unconnected outputs to terminator blocks.

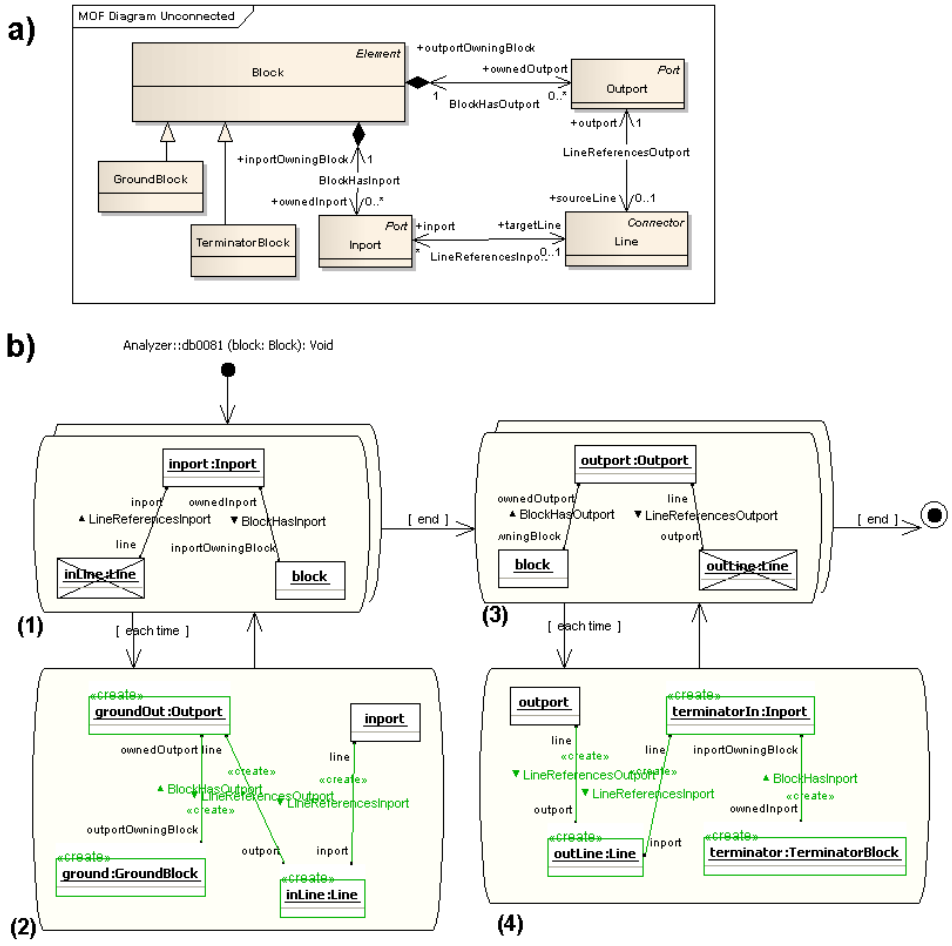


Figure 3.10: MAAB Guideline db_0081 - SDM Diagram

Fig. 3.10.a shows the elements from the MATLAB metamodel which are relevant for the specification of this guideline. The ground block and terminator blocks are represented by the classes *GroundBlock* and *TerminatorBlock*, subclasses of *Block*. Fig. 3.10.b represents the SDM diagrams that executes the analysis and

the correction of the guideline db_0081. This method get as parameter the block whose inputs and outputs will be checked. The activity 1 is a *for-each*-activity. It checks every input port of *block* if connected or not. An unconnected input port is modeled by the Negative Application Condition (NAC) on the object node *inLine*. The crossed out *inLine* means that the pattern is successfully matched if there is NO instance of Line relied to the instance of Inport. In such a case, the activity 2 is executed. It is the repair action for unconnected input. An instance of *GroundBlock* is created, as well as its *Outport* and a *Line*, and connected to the unconnected *Inport*. The activity 1 (and if necessary, the activity 2) is repeated till all input ports of *block* have been checked. The activities 3 and 4 are similar to the activities 1 and 2, except that they concern the inspection of the output ports of *block* and the creation, if needed, of a terminator block.

3.3.2 Comparison between SDM and other Languages

SDM is not the only language that allows for the specification of guidelines. The programming language M-Script of MATLAB is widely used for the specification of modeling guideline for MATLAB SL/SF, e.g. within the Model Examiner of Model Engineering Solutions [MXA10], or within the Model Advisor which is a tool integrated in Simulink [Mat10a]. Though, the implementation of guidelines by means of M-Script occurs on a very low level of abstraction.

```
function f_block_h = guideline_2(system, cmd_s)
  top_h = get_param(bdroot,'Handle');
  f_block_h = [];
  subsys = get_param(get_param(find_system(top_h, 'BlockType',
                  'EnablePort'), 'Parent'), 'Handle');
  for k=1:length(subsys)
    subsys_handle = get_param(subsys{k},'Handle');
    porth = get_param(subsys{k},'PortHandles');
    enable_port_name = get_param(porth.Enable,'Name');
    enableh = find_system(subsys{k},'SearchDepth',1,
                          'BlockType','EnablePort');
    enable_block_name = get_param(enableh,'Name');
    if ~(strcmp(enable_port_name, enable_block_name))
       f_block_h = [f_block_h;subsys_handle];
    end
    trigger_port_name = get_param(porth.Trigger,'Name');
    triggerh = find_system(subsys{k},'SearchDepth',1,
                          'BlockType','TriggerPort');
    trigger_block_name = get_param(triggerh,'Name');
    if ~(strcmp(trigger_port_name, trigger_block_name))
       f_block_h = [f_block_h;subsys_handle];
    end
  end % for
end % function
```

Figure 3.11: MAAB Guideline jc_0281 - M-Script

Let us consider the example of the guideline jc_0281 (Cf. Fig. 3.8.a) whose implementation as graph transformation is depicted by Fig. 3.9.b. Fig. 3.11 illustrates how the same guideline can be implemented by means of M-Script.
We skip the detailed explanation since the example is serving its purpose of giving an impression of what M-Script checks are suffering from. In fact, the implementation of model guidelines with M-Script is nothing else than traversing graph structures and implementing graph pattern matching operations with an imperative language. Thus, implementing guidelines with M-Script is rather a task of programming skills and detailed API knowledge than a task of a conceptual and well structured conversion of an informal description into a formal one.

The application of the Object Constraint Language (OCL) provides an approach which could in general act as a basis for the formalization of all kinds modeling guidelines. Since modeling guidelines represent constraints on model elements or relations between model elements which have to be respected, OCL can be used for a formal description of such rules. A concrete application of OCL for the specification of guidelines is described in [FHR06].
Let us demonstrate the application of OCL by the implementation of guidelines jc_0281 and db_0081. In the case of guideline db_0081, a port has to be connected to a line. Since the classes `Inport` and `Outport` are connected to the class `Line` by different associations, we write two different constraints for the two regarded classes. Both invariants are listed as follows:

```
context Inport                  context Outport
inv: targetLine != null         inv: sourceLine != null
```

As a consequence guideline db_0081 is formalized by a set of two OCL invariants. In fact, both invariants are quite trivial. The presented OCL specification has only one drawback: a single modeling guideline is translated into two different constraints instead of being a single piece of code. If a one-to-one correspondence of guidelines and constraints is an issue (e.g. for reasons of maintainability of guideline implementations) then we can resort to the following solution, where a single more complex OCL constraint enforces the same guideline.

```
context Block
inv: incomingLine->forAll( srcBlock != null ) and
     outgoingLine->forAll ( targetBlock != null )
```

The OCL expressions presented above probably give the reader the impression that it is straight-forward to produce and to understand logic-based specifications of modeling guidelines. But this is no longer true, when more complex patterns have to be specified. Let us consider the OCL expression for the modeling guideline jc_0281.

```
context SubSystem inv:
if self.containedBlock
  ->exists(b:Block | b.oclIsTypeOf(EnableBlock) )
then
(1) self.containingSubsystemBlock.ownedInport
    ->select( enableInport |
    enableInport.correspondingEnableTriggerBlock
    .oclIsTypeOf(EnableBlock))
    ->collect(line.name)
    -> intersection (self.containedBlock
    (2) ->select(b:Block | b.oclIsTypeOf(EnableBlock))
        ->collect(name)) -> notEmpty()
endif
and
if self.containedBlock
  ->exists(b:Block | b.oclIsTypeOf(TriggerBlock) )
then
(1) self.containingSubsystemBlock.ownedInport
    ->select( triggerInport |
    triggerInport.correspondingEnableTriggerBlock
    .oclIsTypeOf(TriggerBlock))
    ->collect(line.name)
    -> intersection (self.containedBlock
    (2) ->select(b:Block | b.oclIsTypeOf(TriggerBlock))
        ->collect(name)) -> notEmpty()
endif
```

This guideline requires that the enable or the trigger block name matches the name of the signal enabling or triggering the subsystem. The class `SubsystemBlock` that contains both the regarded block and its corresponding signal is an obvious choice as context for the to be defined OCL expression. The OCL invariant is composed of two similar expression related by the logical operator *and* because both constraints must be fulfilled. The first expression is the constraint on the enable block, and the second expression concerns the trigger block.
Let us describe the constraint on the enable block. First of all, we have to check that the regarded subsystem contains an `EnableBlock`. Then two elements of the subsystem must be determined and compared: the name of the enabling signal and the name of the corresponding enable block. To compute the name of the enabling signal, we select the instance of `Inport` connected to the `EnableBlock`, and collect the name of the `Line` connected to this instance of `Inport` (cf. subexpression starting at label (1) above). To find the name of the enable block, we must select the block instance of the class `EnableBlock` contained in the subsystem and return its name (cf. subexpression starting at label (2) above).
Please note that a subsystem neither may contain more than one enable block or more than one enabling signal. That means that the intersection of the computed sets of signal and block names is either the single common name (the guideline is respected) or empty (a violation of the guideline).

This example clearly shows that OCL is not very well-suited for the specification of complex patterns, where we have to navigate along different paths through a model and to compare their results.
In addition, only analysis operation can be implemented with OCL. This language provides no support for transformation, and, thus, no support to repair a violated guideline.
Correction of violated guideline would be possible by using ATL (Cf. Section 2.4.3) instead of OCL since ATL integrates and extends OCL to define transformations. Though, even with the possibility to repair violated guidelines, OCL, as part of ATL used for the specification of analysis, lacks in expressiveness. It is almost unfeasible to encode guidelines that require specific facilities, e.g. the manipulation of string or complex arithmetic operations. OCL offers some basic operators on integers and reals, and on strings, but very limited. For instance, it is not possible to use regular expressions, or to calculate two to the power of a negative value. Such computations must be delegated to a host programming language via method calls embedded in OCL expressions.

3.3.3 Evaluation of the SDM Syntax

The example of the previous sections has shown that SDM is pretty well-suited for the specification and implementation of modeling guideline an refactoring. Though, let us evaluate the SDM syntax to determine in which way it could be improved.

A set of MAAB guidelines such as the guideline jc_0201 in Fig. 3.12 concerns the naming conventions. This guideline jc_0201 defines restrictions on the use of characters to name a subsystem block, e.g. the allowed first character or the use of underscore. Using regular expressions [Fri02] to implement such guidelines appears as an evidence.
A regular expression allows for matching strings of text, such as particular characters, words, or patterns of characters. Regular expressions consist of constants and operators that denote sets of strings and operations over these sets. Operations are the concatenation RS, the alternation R|S and the Kleene star *. A number of special characters (or metacharacters) are used to denote actions or delimit groups. For instance, [], called bracket expression, matches a single character that is contained within the brackets. [a-z] specifies a range which matches any lowercase letter from *a* to *z*. The metacharacter ^ matches a single character that is not contained within the brackets. For instance, [^0-9] matches any single character that is not a number. The Kleene star indicates that there are zero or more of the preceding element, the "+" sign that there is one or more of the preceding element, and the "?" sign that there is zero or one of the preceding element. For instance, *ab** matches *a*, *ab*, *abb*, *abbb*, and so on. *ab+* matches *ab*, *abb*, *abbb*, and so on, but not *a*. *ab?* matches *a* and *ab*.

ID: Title	**jc_0201: Usable characters for Subsystem names**
Priority	strongly recommended
Scope	MAAB
MATLAB Version	All
Prerequisites	
Description	The names of all Subsystem blocks should conform to the following constraints: FORM: **name**: • should not start with a number • should not have blank spaces ALLOWED CHARACTERS: **name**: a b c d e f g h i j k l m n o p q r s t u v w x y z A B C D E F G H I J K L M N O P Q R S T U V W X Y Z 0 1 2 3 4 5 6 7 8 9 _ UNDERSCORES: **name**: • underscores can be used to separate parts • cannot have more than one consecutive underscore • cannot start with an underscore • cannot end with an underscore
Rationale	☑ Readability ☑ Workflow ☐ Simulation ☐ Verification and Validation ☑ Code Generation
Last Change	V2.0

Figure 3.12: MAAB Guideline jc_0201

The regular expression representing the allowed pattern for a subsystem name according to jc_0201 is: [a-zA-Z]([_]?[a-zA-Z0-9]+)*. Unfortunately, the current syntax of SDM does not provide any support for regular expressions, and, thus, for a direct specification of naming convention guideline.

As shown by the example of the guideline jc_0281, textual notation is less adapted than the visual notation for guidelines requiring the matching of a complex pattern or a navigation throughout the metamodel. Nevertheless, the textual notation would be more compact than the visual notation for very simple guidelines, especially if the context of the guideline is limited to one kind of element. For instance, for guidelines concerning solely the value of one attribute of a given class such as the guideline jc_0011 represented in Fig. 3.13.a. This guideline ensures that the option for boolean data type must be enabled. M-script is tightly integrated into MATLAB since it is the proprietary language. Though, as shown above, this language is on a too low level of abstraction, and, thus, we will rather consider OCL as example of textual notation. The part b of Fig. 3.13 represents the SDM specification of this guideline whereas the part c of this figure shows its OCL implementation. The context is the class *Model*, and the value of its attribute *booleanDataType* must be checked. It appears clearly that the OCL notation in such a case is more simple and compact than the SDM one.

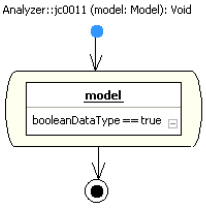
a)

ID:Title	jc_0011: Optimization parameters for Boolean data types
Priority	strongly recommended
Scope	MAAB
MATLAB Version	All
Prerequisites	na_0002: Appropriate implementation of fundamental logical and numerical operations
Description	The optimization option for Boolean data tyypes must be enabled (on).

MATLAB version	Option Name
R13SP2 and earlier	Boolean Logic signals Boolean logic signals On
R14 and later	Use logic signals as Boolean data. (vs. double) ☑ Implement logic signals as boolean data (vs. double).

Rationale	☐ Readability ☑ Workflow ☐ Simulation ☐ Verification and Validation ☑ Code Generation
Last Change	V2.0

b)



c)

```
Context Model
inv: self.booleanDataType
```

Figure 3.13: MAAB Guideline jc_0011

More generally, the question “textual vs. visual” is still a matter of debate. On the one hand, visual languages have become popular because they are considered as easier to understand. In the context of model-based engineering where most models and metamodels are graph-based, graph transformations are a direct and natural way to define model transformation. This explains that SDM, as visual language, is so well-suited for the definition of patterns to be matched. On the other hand, the textual notation is considered as more expressive. OCL has become necessary to complete UML models with textual constraints. This example illustrates the limited expressiveness of purely visual languages. In the same way, the most expressive languages compared in Section 2.4.4 are VIATRA2 and PROGRES, i.e. a textual and a hybrid language.

Another drawback of a visual notation is the necessity to manage the layout of the diagrams. Let us consider the example of Fig. 3.9. The activities 1 and 2 are very similar to the activities 3 and 4, but the user is forced to draw nearly the same diagram twice. We can see here a limitation of the current syntax of SDM because it is not possible to reduce the diagram or to reuse the activities 1 and 2. Because one drawback of a visual notation such as SDM is the necessity to manage diagram

layouts, it would be desirable to reduce the effort in drawing graph. Thus, we need to make the SDM as compact and efficient as possible, and improve the reusability of diagrams. As indicated in Section 2.4.4, the current SDM syntax does not support the definition of generic transformations, and more precisely the parameterization of object types. Though, generic transformation rules could be reused, and improve the composition and decomposition of model transformations. For instance, the story diagram depicted by Fig. 3.9 which specifies the analysis and correction of guideline jc_0281 consists of four story pattern. We can notice that the story patterns a and b are very similar to the story patterns c and d. Thus, if we can define generically a method whose story diagram consists of the story patterns a and b, it would greatly reduce the effort, and, hence, would improve the efficiency in drawing graph. Then, we would simply need to call the resulting generic method twice in order to execute the same action as the method depicted by Fig. 3.9.

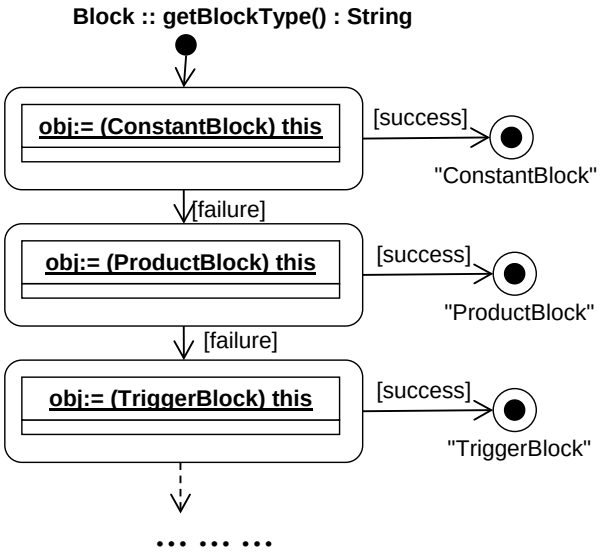


Figure 3.14: Checking of the Type of a Block using the current SDM Syntax

Another aspect which is currently not supported by SDM is the multilevel transformations. Though, multilevel transformations, in terms of having a read access (i.e. without creation or deletion of elements) to the next higher metalevel would provide information, and, thus, increase the expressiveness of SDM. Several guidelines require the knowledge of the type of the blocks contained in the analyzed model. As depicted by Fig. 3.14, it is possible to specify the behavior of a method which returns the type of a block by means of the current SDM syntax. Though, this story diagram shows that such a specification is cumbersome. In addition, the specification needs to be complete since only the specified cases can be considered. Since a modeling level is described by the next higher meta-level, a reading access to this meta-level could return directly the desired block type information.

Guidelines are not defined for a specific object, but for the complete range of an object type, e.g. all the product blocks. As a consequence, the specification of many guidelines requires to draw the exploration of the complete model. This is complicated by the nesting of elements which illustrated in MATLAB Simulink by the subsystems contained in the subsystem blocks. The effort to express checking and correction of models would be greatly reduced by a direct access to all instances of a metaclass.

As indicated in Fig. 2.24, SDM is understandable and pretty easy to learn due to its similarity with UML (activity and object diagrams). This is a strength of this language. Thus, we must take care in defining enhancement and extension for SDM to preserve this characteristic. In the next chapter, we will present with full details how SDM has been improved with regard to the evaluation presented in this section.

Chapter 4

Extension of the SDM Syntax

The previous section illustrates the benefits but also the limitations of SDM as graph transformation language. SDM is well-suited for the behavior's description of a method, especially when defined on a visually specified metamodel. For instance, patterns to be matched can be specified more easily in a visual way than in a textual way. Nevertheless, because graph drawing requires time and efforts, the reusability of story diagram should be improved by providing a support for the specification of generic methods.
Another aspect which is currently not supported by SDM is the multilevel transformations. Consequently, in addition to generic features, we propose to extend the SDM syntax with reflective features which provide a reading access to the next higher abstraction level. This access to additional information will increase the expressiveness of SDM.

4.1 Generic Feature

As explained in Section 3.3.3, SDM, as visual model transformation language, is pretty well-suited for the specification of modeling guideline. Nevertheless, the efficiency could be greatly improved by adding a support for generic transformations. We propose an approach which is based on the genericity provided by the JMI interfaces. In order to illustrate our approach, we present a simple prototype implemented in our meta-CASE-tool MOFLON.

4.1.1 Generic Model Transformations

As described in Section 2.5.1, there are different kinds of polymorphism. In the following, we will only consider the universal polymorphism: the parametric and the subtype polymorphism. One primary purpose of polymorphism is the support of *dynamic binding* (also called *late binding*). Objects belonging to different types may respond to method calls of the same name, each one according to an appropriate type-specific behavior. Dynamic binding means that the exact behavior is

determined at runtime since the exact object type is only determined at runtime. The other advantage of using parametric polymorphism is the possibility to write functions generically so that they can handle values identically without depending on their type. Generic programming is namely a style of computer programming in which algorithms are written in terms of "to-be-specified-later" types that are then instantiated when needed for specific types provided as parameters [MVM10]. Consequently, genericity relates to reusability of implementations.
Story diagrams are used as specification of methods. As seen in the previous chapter, the lack of reusability is a drawback of the current SDM syntax. Therefore, we need to extend the SDM syntax in order to support the specification of generic model transformation, and, hence, of generic functions.

Reflective operations	Tailored operations

`refClass(String className)`
or
`refClass(RefObject type)`
	`get`*`ClassName()`*

`refCreateInstance()`	`create`*`ClassName`*`()`

`refAssociation(String "associationName")`
or
`refAssociation(RefObject association)`
	`get`*`AssociationName`*`()`

`refGetValue(String featureName)`
or
`refGetValue(RefObject feature)`
	`get`*`AttributeName`*`()`

`refSetValue(String featureName, java.lang.Object value)`
or
`refSetValue(RefObject feature, java.lang.Object value)`
	`set`*`AttributeName`*`(value)`

`refDelete()`	*`///`*

Figure 4.1: Equivalence Reflective/Tailored Operations

MOFLON generates JMI-compliant code. The Java Metadata Interface, presented in Section 2.1.3, provides two kinds of interfaces: the tailored (i.e. generated) strongly typed interfaces, and the reflective untyped interfaces.
Although the reflective interfaces provide the same functionality as the tailored interfaces, there is an important difference between both. The reflective interfaces, which correspond to the graph schema independent part of JMI, can be used on any model to provide access to its metainformation without having to know the generated interfaces. The methods of the reflective interfaces obtain all metamodel-related information as parameterized values (string or typed values), whereas in the case of the tailored interfaces all metamodel-related information is an integral part of the interfaces itself. Since tailored and reflective interfaces finally provide the same functionality, the parameterization of all metamodel-related information can be passed to the description of model transformations without any loss of functionality. Metamodel-related information for which the JMI interfaces provide reflective access are attribute names, class names and association names.
Fig. 4.1 shows the equivalence between tailored and reflective operations. The reflective *refClass* method returns the proxy of the class which is given as parameter, and *refCreateInstance* creates the corresponding object. *refAssociation* returns the corresponding association. The method *refGetValue* fetches the attribute or reference given as parameter, whereas its counterpart *refSetValue* assigns a value

to the attribute or reference given as parameter. Finally, the reflective operations *refDelete* has no counterpart among the tailored operations.
This equivalence between tailored and reflective methods allows for defining generic functions by means of model transformations. The proposed extension of the SDM syntax will provide visual counterparts in the graph schemata to these JMI facilities.

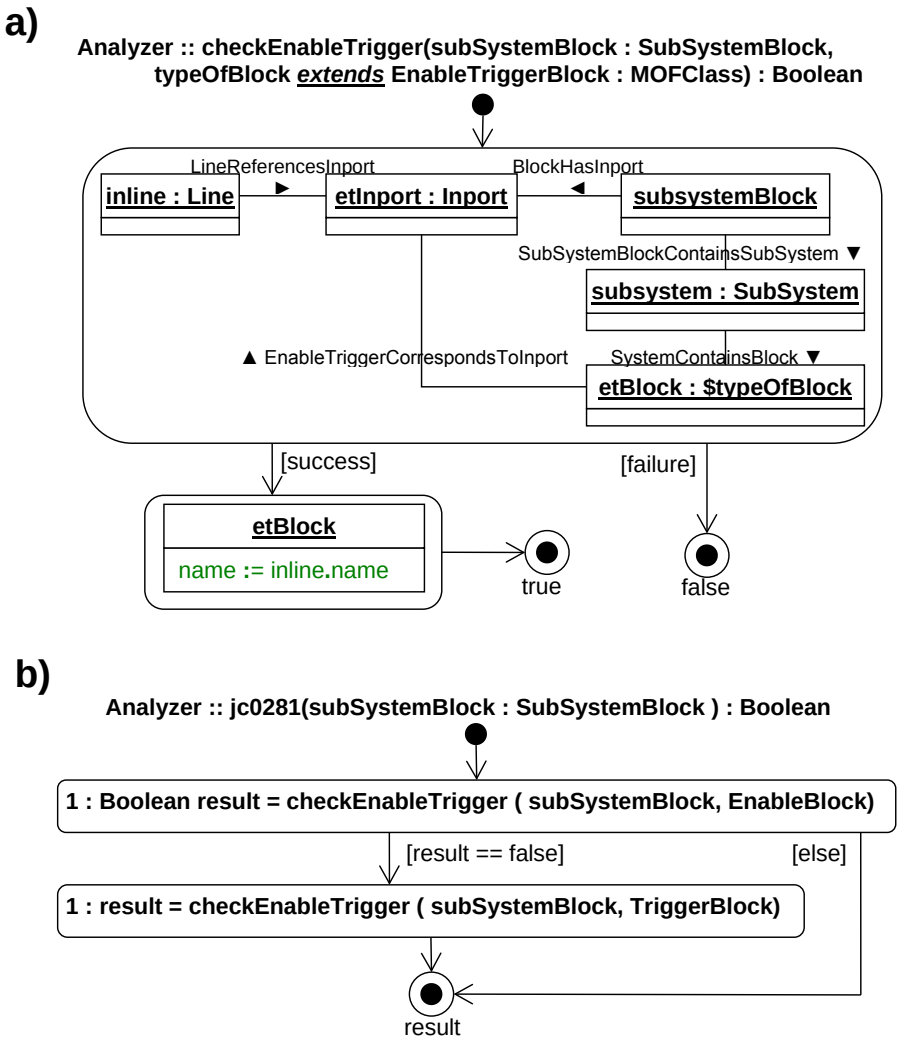
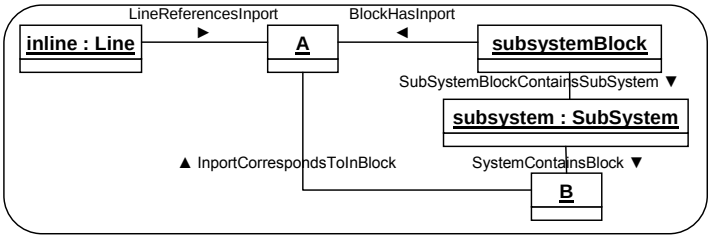


Figure 4.2: jc_0281 - SDM Diagrams with Generic Feature

The implementation of the guideline jc_0281 by means of a story diagram was shown in Fig. 3.9. As explained, the weak point of this diagram was the repetition of two very similar patterns, and the impossibility to reuse these patterns. Let us see how this can be expressed as generic graph transformation. Fig. 4.2 depicts the generic solution for the implementation of the guideline jc_0281.
Fig. 4.2.a is the method *checkEnableTrigger*. This method contains the patterns that will be reused. The parameter *typeOfBlock* defines the type of the object node *etBlock*. The $-symbol indicates that the following expression is a metamodel-related information (in this case, the type of the object node *etBlock*) which is evaluated when executing the graph transformation. The parameter is expressed as: *typeOfBlock* *extends* *EnableTriggerBlock : MOFClass*. The MATLAB metamodel is MOF-compliant, i.e. it is an instance of the MOF metametamodel. Consequently, any class of the MATLAB metamodel is an instance of *MOFClass*, and, thus, so does the parameterized class. The expression *extends* *EnableTriggerBlock* defines a type restriction on the parameter. The parameter is restricted in such a way that it must be a specialization of the class *EnableTriggerBlock*, i.e. the class *EnableBlock* or *TriggerBlock*.
Fig. 4.2.b is the method that implements the guideline jc_281. The single parameter of this method is the SubSystemBlock *subSystemBlock* which will be checked. The method consists of two successive calls (expressed as collaborations statements) of the generic method *checkEnableTrigger*. The first call has *subSystemBlock* and the class *EnableBlock* as arguments, and the second has *subSystemBlock* and the class *TriggerBlock* as arguments. Consequently, the execution of the first method call will check if *subSystemBlock* contains an *EnableBlock*, and in this case, if this block is correctly named. If the method returns false, this means that *subSystemBlock* contains no *EnableBlock*, and the second method call is executed, else, the method terminates. The second method call checks if *subSystemBlock* contains a *TriggerBlock*, and in this case, if this block is correctly named.

Fig. 4.3 presents a comparison between the code generated for a pattern that we want to reuse to implement the guideline jc_0281. In the first case, if the pattern is non-generic (cf. Story diagram of Fig. 3.9), and in the second case if the pattern uses the generic feature (cf. Story diagram of Fig. 4.2). For the sake of clarity, Fig. 4.3 shows only an excerpt of the code generated for this pattern, more precisely the matching of the node *B* (*enableBlock : EnableBlock* in the non-generic case, *etBlock : $typeOfBlock* in the generic case). The code consists of an iteration over the properties *containedBlock* of the association *SystemContainsBlock*. The type of each matched instances *tmpObject* is checked till an object with the correct type is found. As we can see, the main difference between the non-generic code (Fig. 4.3.a) and the generic code (Fig. 4.3.b) is this type checking operation.
In Fig. 4.3.a, the type is already known (here: *EnableBlock*) and we can directly use the java operator *instanceof*.
In Fig. 4.3.b, this is not possible because the real type is only known at runtime. We need a method to evaluate the value of the arguments *typeOfBlock* when the



a)

Non generic: A = enableInport:Inport B = enableBlock:EnableBlock

```
1  /*
2  *
3  * matched from subSystemBlock: enableInport and subsystem
4  *
5  */
6
7  Iterator iterSubSystemToEtBlock = subSystem.getContainedBlock().iterator();
8
9  while(iterSubSystemToEnableBlock.hasNext()){
10    try{
11        java.lang.Object tmpObject = iterSubSystemToEtBlock.next();
12        javaSDM.ensure(tmpObject instanceof
13            org.moflon.matlab.tools.simulink.EnableBlock);
14        enableBlock = (org.moflon.matlab.tools.simulink.EnableBlock) tmpObject;
15        javaSDM.ensure(etInport.equals(etBlock.getCorrespondingInport()));
16    }
17    catch(JavaSDMException internalException)
18 }
19
20 /*
21 *
22 *
23 */
```

b)

Generic: A = etInport:Inport B = etBlock:$typeOfBlock

```
1  /*
2  *
3  * matched from subSystemBlock: enableInport and subsystem
4  *
5  */
6
7  Iterator iterSubSystemToEtBlock = subSystem.getContainedBlock().iterator();
8
9  while(iterSubSystemToEnableBlock.hasNext()){
10    try{
11        java.lang.Object tmpObject = iterSubSystemToEtBlock.next();
12        javaSDM.ensure(
13        ((org.moflon.matlabPackage)refImmediatePackage().refImmediatePackage())
14           .getToolsPackage().getSimulinkPackage().refGetClass(typeOfClass)
15           .refAllOfClass().contains(tmpObject));
16        etBlock = (org.moflon.matlab.tools.simulink.InBlock) tmpObject;
17        javaSDM.ensure(etInport.equals(enableBlock.getCorrespondingInport()));
18    }
19    catch(JavaSDMException internalException)
20 }
21
22 /*
23 *
24 *
25 */
```

Figure 4.3: Code Comparison for a Pattern a)without and b)with Generic Feature

method is called. Therefore, we need to explore the metamodel. The methods provided by JMI reflective interfaces makes this navigation possible. With the two successive calls of *refImmediatePackage*, we have access to the root package of the metamodel. Then, with *getToolsPackage().getSimulinkPackage()*, we obtain the *Simulink* package. The method refGetClass(typeOfClass) returns the proxy of the parameterized class. The proxy of a class manages all the instances of this class. Thus, the method refAllOfClass() called on the proxy returns a collection containing all the objects. With the method *contains*, we can check if *tmpObject* is contained in this collection, and, hence, if it is an instance of the parameterized class.

4.1.2 Prototype and Application

In order to illustrate the feasibility of our approach, we implemented the generic feature in MOFLON as a prototype. Nevertheless, our work occurred in a phase of important refactoring for MOFLON [1]. Consequently, we restricted our implementation to a simple case, namely the parameterization of modeling elements as string. In a first section, we explain how we have implemented the prototype. Then, we illustrate our implementation by an example.

Adaptation of the Code Generator

As shown by the table of Fig. 4.1, reflective methods may have indifferently a string value or a type value as parameter. We extended our meta-CASE-tool MOFLON in order to support the parameterization of elements in form of string values. Because MOFLON already provides a support for string parameters, we extended the dialog box to indicate the use of the generic feature, and focused our effort on the code generation.
The concept of code generation within MOFLON is described in Section 2.3.3. As explained, CodeGen2 and its Velocity Templates are in charge of the code generation from the story diagram. More precisely, the main part to be modified to allow for the generation of generic code is the set of Velocity templates.

First of all, we must shortly explain the principle of the Velocity template language. The Velocity Template Language (VTL) is based on the definition and the use of references to add dynamic content to the code. There are three kinds of references:

- **Variable:** A variable is denoted by a leading $-character followed by a VTL identifier. For instance, *$foo* is a VTL variable. Such a variable gets its value from either a *set*-directive in the template, or from the Java code. If

[1] MOFLON has been reengineered during this thesis, in 2011, and is now called eMOFLON. It uses the professional CASE tool Enterprise Architect as frontend, and is based on Eclipse Modeling Framework (EMF) [ALPS11]

the value of the variable *$foo* is set by the Java code, and the value *bar* is bound to the variable *$foo* at the time the template is requested, bar replaces all instances of $foo in the generated code. A setting statement within a template is expressed as following: `#set { $foo = "bar"}`. Then, the output will be the same for all instances of *$foo* that follow this directive.

- **Method:** A method is defined in Java code, allowing for running calculation. It consists of a leading $-character followed by a VTL Identifier, a dot, and a method body. An example of a VTL method is: `$object.getName()`.
- **Properties:** The shorthand notation consists of a leading $ character followed by a VTL Identifier, followed by a dot character, and another VTL Identifier. An example of a property is the following: `$object.Name`. It is an abbreviate way of writing `$object.getName()`.

Velocity uses these references to embed dynamic content in a document, e.g. in the generated code in the case of the code generation by CodeGen2. The Velocity syntax provides directives which always start with the # symbol, e.g. the *set*-directive presented above that establishes the value of a reference. The *parse*-directive renders a local template that is parsed by Velocity. Directives may also define control structures: the *foreach*-directive loops through a list of objects, and the *if*, *elseif* and *else* output conditional on truth of statements. A control structure ends with the *end*-directive. Velocity provides other directives, but their description would be out-of-scope here. The interested reader can find additional information in the Velocity user guide [Vel07].

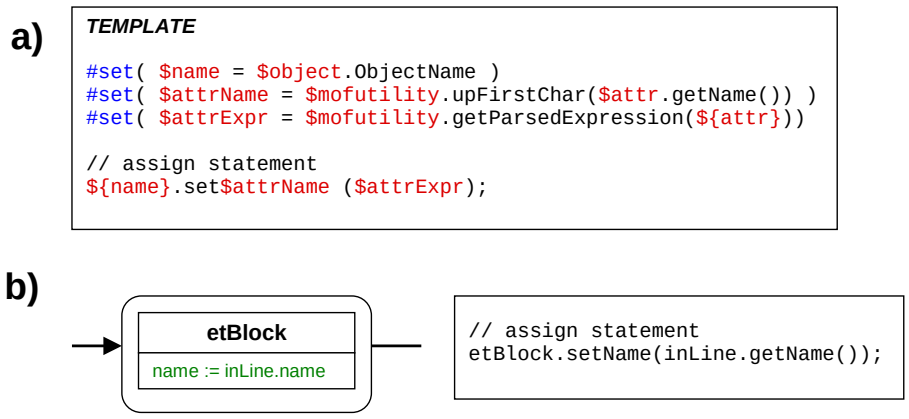


Figure 4.4: Velocity Template - Simple Example

Fig. 4.4.b shows a very simple application of the Velocity templates of Fig. 4.4.a. for the setting of an attribute's value. The template defines three references: *name*, *attrName*, and *attrExpr*. The reference *name* is the object's name. The reference *attrName* is the name of the attribute. An attribute, according to the MOF standard, starts with a lowercase character. Though, we want to use this string within

a setting method. Therefore, we need the attribute's name, but starting with an uppercase character. We use an appropriate method (*upFirstChar*) of a utility class which is referenced by the variable *mofutility*. In the same way, the third reference is defined with help of the utility class that parses the value of the attribute assignment. For instance, in Fig. 4.4.b, the value which is assigned to the attribute *name* of the object *etBlock* is given as *inLine.name*. Though, this value cannot be used directly in the generated code, and is then parsed into *inLine.getName()*. The generated code of Fig. 4.4.b shows how the references *name*, *attrName*, and *attrExpr* are replaced by their concrete values.

Figure 4.5: Generic Feature - Adaptation of the Velocity Templates

As explained, the directives *if*, *elseif*, and *else*, allow for parsing VTL statements according to the truth of condition statements. We use this facility provided by Velocity to modify the templates in order to generate generic code when needed.

Fig. 4.5 shows the three templates in charge of the code generation for attribute checking and assignment. For the sake of clarity, we only show the lines added in the templates to implement the generic feature. The template called *import* is a template which is always parsed when code for an attribute checking or assignment must be generated. A new reference called *isGeneric* is defined in this template. This reference has a Boolean value, set to *true* in the case of a generic attribute checking or assignment. The *check* template is called for an attribute checking. The reference called *method* is the expression giving access to the attribute. In the case of a non-generic attribute, it would be simply expressed as *objectName.attributeName*. As previously explained, the generic access to an attribute is provided by the JMI method *refGetValue*. Thus, this method is used to define the reference called *method*. Then, using the *if* and *else* directive, we can define the part of the template which is parsed only if the reference *isGeneric* is true (i.e. in the case of a generic attribute) The *assign* template is called for an attribute assignment (use of the ":="-operator). The generic method that is required for the assignment of an attribute value is the JMI method *refSetValue* whose arguments are the attribute or its name, and the value to be set. Here again, we can add a specific case for the assignment of a value to a generically defined attribute by means of the *if* and *else* directives. The reference called *name* is the name of the owning object, *genericAttrName* is the parameterized attribute name, and *attrExpr* is the value assigned to the attribute. A concrete application of these modified templates is presented in the next section.

Application

Fig. 4.6 represents the specification of the guideline na_0004. This guideline defines the value of some settings for a Simulink model such as the screen color or the zoom factor.

We can naturally implement this guideline with the previous version of the SDM syntax. There are two possibilities, both of them are represented by Fig. 4.7. The part a of Fig. 4.7 shows the implementation of this guideline in a single diagram. More precisely, it shows only an excerpt of this story diagram because the complete diagram is too large to be represented in this document. The model settings are modeled as attributes of the class *Model*. The guideline defines 19 settings that must be checked, and, if necessary, corrected. Every setting requires two activities, the first to check the value of the attribute representing this setting, and the second to assign, if necessary (= the *failure*-transition is traversed), the correct value. That means that the complete story diagram is composed of 38 (very repetitive) activities.
Fig. 4.7.b shows the second variant. Each checking and correction are implemented by a method, e.g. *checkAndRepairValue1(model:Model)*. The method *na0004(model:Model)* calls all these checking and correction methods. These variant reduces the effort on the layout of the diagrams since they are smaller. Though,

ID: Title	**na_0004 Simulink model appearance**
Priority	Recommended
Scope	MAAB
MATLAB Version	All
Prerequisites	
Description	The model appearance settings should conform to the following guidelines when the model is released. The user is free to change the settings during the development process.

View Options	**Setting**
Model Browser	unchecked
Screen color	white
Status Bar	checked
Toolbar	checked
Zoom factor	Normal (100%)
Block Display Options	**Setting**
Background Color	white
Foreground Color	black
Execution Context Indicator	unchecked
Library Link Display	none
Linearization Indicators	checked
Model/Block I/O Mismatch	unchecked
Model Block Version	unchecked
Sample Time Colors	unchecked
Sorted Order	unchecked
Signal Display Options	**Setting**
Port Data Types	unchecked
Signal Dimensions	unchecked
Storage Class	unchecked
Test point Indicators	checked
Viewer Indicators	checked
Wide Non-scalar Lines	checked

Rationale	☑ Readability	☐ Verification and Validation	
	☑ Workflow	☐ Code Generation	
	☐ Simulation		
Last Change	V2.0		

Figure 4.6: MAAB Guideline na_0004

we need to define as many story diagrams as model settings. Since these SDM diagrams are very similar, this variant is absolutely not efficient, and may be greatly improved by the new generic SDM feature.

Similarly to the variant of Fig. 4.7, the generic variant depicted by Fig.4.8 requires a *method na0004 (model:Model)* calling the checking methods and, if necessary, the correcting methods. These methods (*checkAndRepairModelStringValue*, *checkAndRepairModelBooleanValue*, and *checkAndRepairModelIntegerValue*) have three

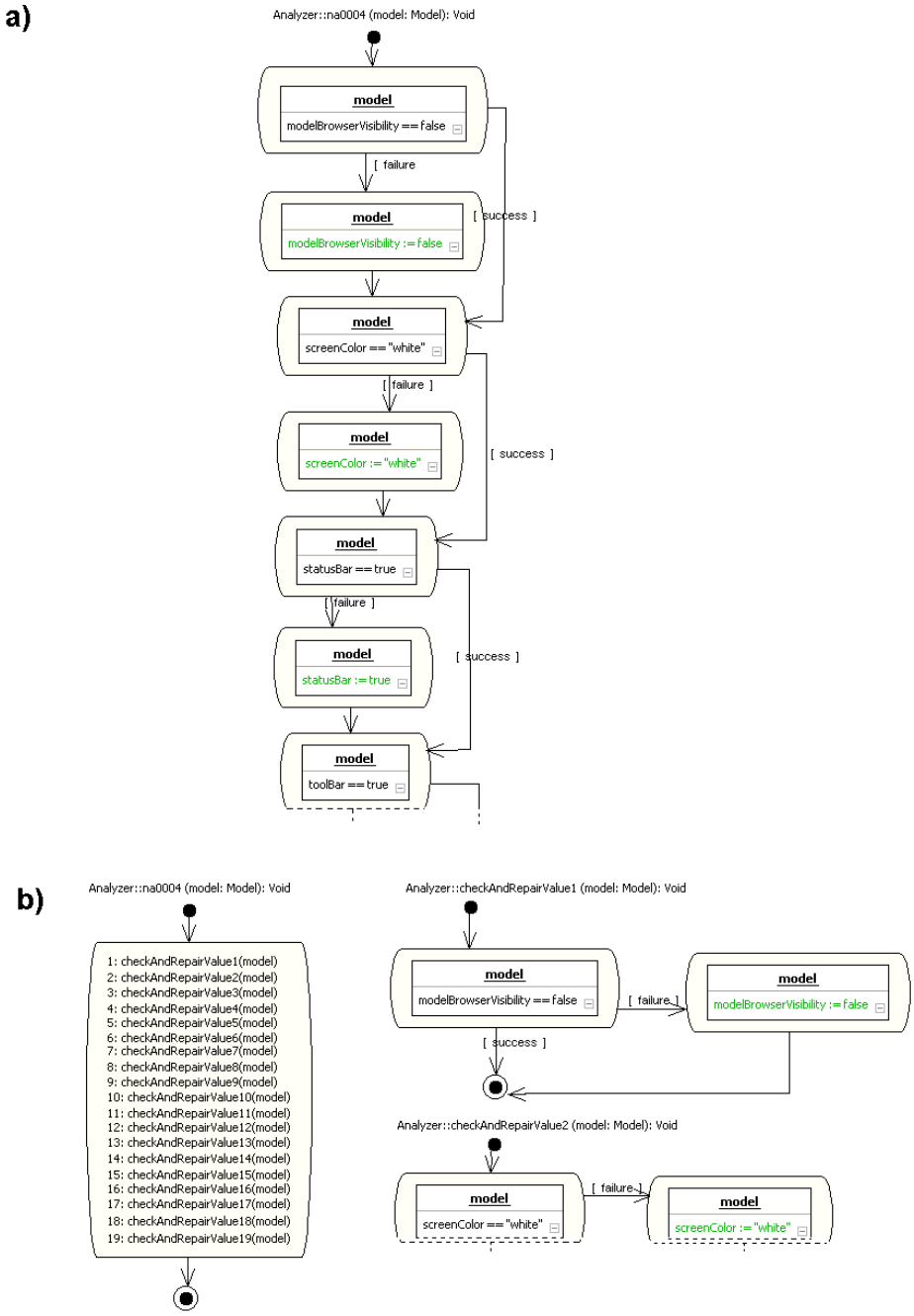


Figure 4.7: MAAB Guideline na_0004 - SDM Diagrams

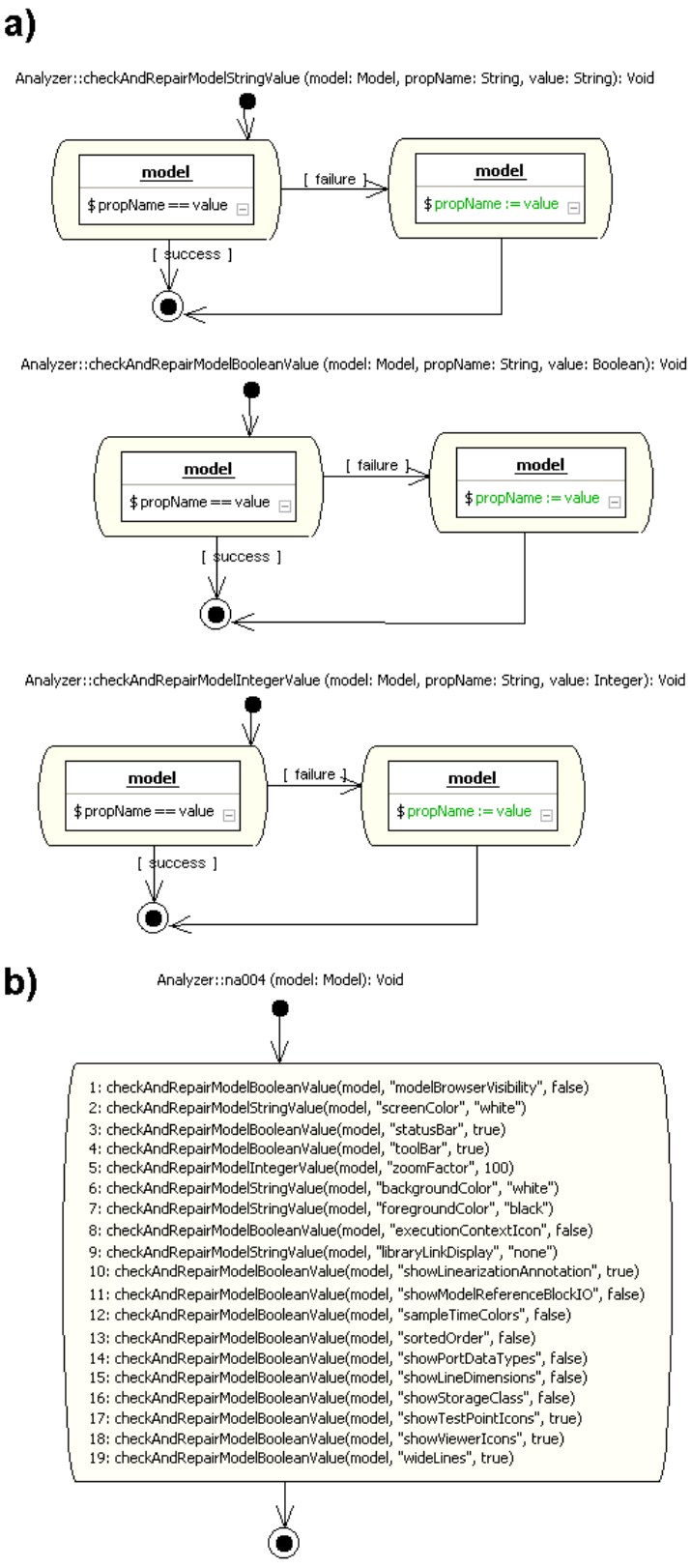


Figure 4.8: MAAB Guideline na_0004 - SDM Diagram with Generic Feature

attributes. The second attribute, *propName*, is used as parameterized attribute as indicated by the use of the $-symbol in the story diagram. The type of the setting value, represented by the third attribute value, must correspond to the type of this attribute. Therefore we define three methods to distinguish three cases: when the model setting is a String, a Boolean, or an Integer.

Fig. 4.9 is a screenshot of MOFLON showing concretely how a generic attribute is set. In this case, this is the assignment of the correct value in the method *checkAndRepairModelBooleanValue*. An option called *GenericType* has been added to the dialog box. If this option is ticked off, this means that the attribute whose name entered in the field *AttributeName* is a parameterized one. As a consequence, the attribute appears in the story diagram preceded by the $-symbol.

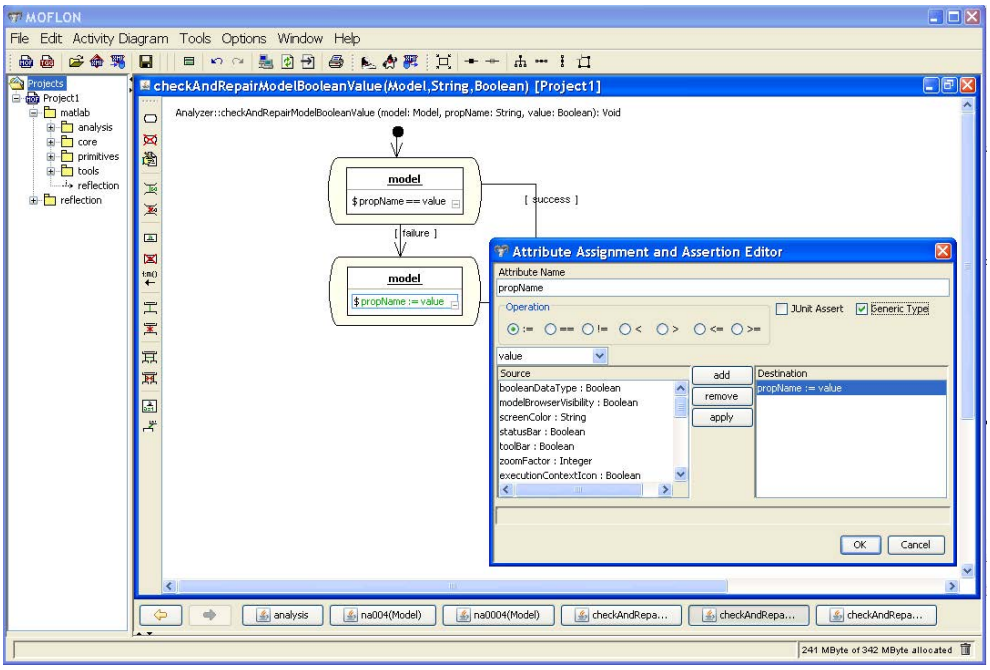


Figure 4.9: SDM Generic Feature - Prototype

4.2 Reflective Feature

Not only generic but also multilevel transformations are not supported by the old version of SDM. Though, having an access to the next higher abstraction level would provide additional information, and, thus, increase the expressiveness of SDM. Therefore, we want to extend the SDM language with a so-called reflective feature. We first explain the concept of reflection and make clear how it must be understood in the context of this thesis. Then, we present our proposed extension for SDM before describing the implementation of a prototype for MOFLON.

4.2.1 Terminology

Let us see what we mean by the term of *reflection* in the context of this thesis. Brian Cantwell Smith introduced in [Smi82] the notion of computational reflection in programming languages. Reflection is the process by which a computer program can observe and modify its own structure and comportment. A reflective system supports a representation of itself (or self-representation). This self representation makes the system able to get access to information on itself, and to support action on itself. In addition, this self-representation is *causally-connected*. This means that changes made to the self-representation are immediately mirrored in the underlying system's actual state and behavior, and vice-versa. Hence, the status and computation of the system are always in compliance with this representation.

Reflection can be used for observing and/or modifying program execution at runtime. In this case, the reflective computation does not directly contribute to the object-computation, i.e. computation about the external problem domain. Instead, it contributes to the internal structure of the system in order to improve the effectiveness and smooth functioning of the object-computation. Reflection is also a key strategy for metaprogramming [Bar05]. Metaprogramming is the writing of computer programs that write or manipulate other programs (or themselves). The language in which the metaprogram is written is called the *metalanguage*, whereas the language of the programs that are manipulated is called the *object language*. Thus, reflection is the ability of an object language to be its own metalanguage.

A reflective system has always an accurate representation of itself. *Reification* is the process by which an abstract idea is turned into an explicit data model or object, and thus is a key concept in the context of reflection. For instance, it makes abstract aspects available to a program which can inspect them as ordinary data. [Tan04] defined several aspects of the reflection for a program:

- **Structural Reflection**: ability of a program to access a representation of its structure.
- **Behavioral Reflection**: ability of a program to access a dynamic representation of itself, i.e. of its operational execution.

- **Introspection**: ability of a program to reason about reifications of otherwise implicit aspects of itself.
- **Intercession**: ability of a program to act upon reifications of otherwise implicit aspects of itself.

It must be noticed that introspection allows only for reasoning on reifications. Thus, this is simply a read-only form of reflection, without any modification of the program and the structure.

In object-oriented programing languages such as Java, reflection allows for inspection of classes, interfaces, fields and methods at runtime without knowing the names of the interfaces, fields, methods at compile time. We want to exploit precisely this aspect of reflection, i.e. the introspection, for SDM.
In the context of the model driven architecture, the meta-information are contained in the next higher meta-level. The current version of SDM provides no access to these information. Nevertheless, an access to these information would greatly improve the expressiveness of the story diagrams as explained in Section 3.3.3 and illustrated by the example of Fig. 3.14. Therefore, we propose a so-called reflective feature in order to visually specify a read access to the next higher metalevel in a story pattern.
Please note that we limit the reflective feature to a read-only access, i.e. without creation or deletion of elements. If we modify element in the next higher metalevel, the metamodel will be affected when executing the graph transformation. This means that the code generated from the metamodel must be updated, i.e. regenerated, after the execution of this graph transformation. In addition, the graph transformations are defined on the metamodel. If the metamodel is modified, graph transformations can become erroneous. For instance, if a class is deleted from the metamodel, a graph pattern trying to match/create/delete instances of this class becomes invalid.

4.2.2 Reflective Model Transformations

As explained in Section 2.1.3, the JMI-compliant code which is generated by MOFLON implements tailored as well as reflective interfaces. Fig. 2.3 of Section 2.1.3 gives an overview of the JMI reflective interfaces. At the top of the interface hierarchy, the interface *RefBaseObject* provides the common reflective method *refMetaObject()*. This method is particularly relevant in the context of reflection because it returns the metaobject of the calling object.

Let us explain what a *metaobject* is. According to the MOF architecture (Cf. Fig. 2.2), the M3-level describes the M2-level which defines the M1-level. This means that each MOF compliant metamodel can be described as an instance of the MOF metametamodel. An example of such a description is depicted in Fig. 4.10. We can see that the M2-level on this figure is composed of two parts. The left part

is an excerpt of the MATLAB metamodel as defined in Section 3.2.2. The right part is the equivalent of the left part, but expressed as an instance of the MOF metametamodel. The classes *Block* and *Outport* are two instances of *MOFClass* called *mb* and *mo*. The association *BlockHasOutport* is an instance of *MOFAssociation*, and both association end *outportOwningBlock* and *ownedOutport* are instance of the class *MOFProperty*. The M0-level of Fig. 4.10 is a very simple Simulink model since it is composed of a single constant block and its output. It is modeled in the M1-level by means of an instance *b* of the class *Block*, and an instance *o* of the class *Outport*. These both objects are connected to each other by a link, instance of the association *BlockHasOutport*. Thus, the instance called *b* instantiates the class *Block* which is also represented as the metaobject *mb*, instance of the metaclass *MOFClass*.

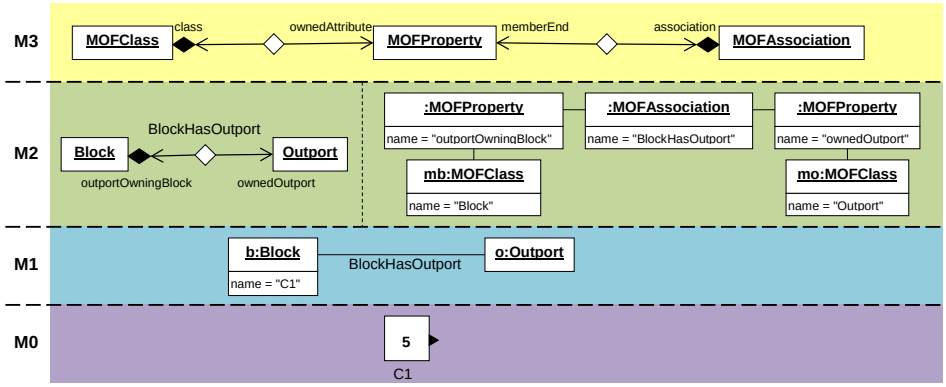


Figure 4.10: Relations between metamodels

Beside the interfaces for the creation, access and storage of models, JMI also provides a mechanism for the discovery of a model's metadata. As explained, each metamodel can be described as instance of the MOF metametamodel. In case of a JMI repository such as in MOFLON, this information is available at runtime in such a way that each element of a metamodel is aware of its description as metaobject, i.e. as an instance of the MOF metametamodel.
The MATLAB metamodel is generated as a JMI compliant metamodel, and the MOF metametamodel itself as a JMI compliant metamodel which is instantiated by the MATLAB metamodel. The method *refMetaObject()* belongs to the interface *RefBaseObject* which is at the top of the JMI interface hierarchy. This means that it is possible to get the metaobject from any element of the M1-metalevel, i.e. not only the objects but also the links. Once a metaobject for a class, package or association is determined, the complete metamodel can be explored based on that metaobject. Thus, with the information provided by the metametamodel instances, it is possible to explore and discover the MATLAB metamodel without prior knowledge of its interfaces.

The JMI mechanism for the discovery of metamodel's metadata makes possible the extension of the SDM syntax with a reflective feature. Linking the metamodel of a repository and the appropriate instances of the next higher metalevel allows for matching metadata at runtime. Due to the linking that is depicted in Fig. 4.10, there are also graph structures that result from a metaobject which is linked with the class of a given object. As the metaobject itself is part of a further (meta)model, there are finally metalevel spanning graph structures. The matching of those metalevel-spanning structures can be very beneficially integrated into the existing matching of intra-metalevel structures as demonstrated in the example of Fig. 4.11.b.
The example of Fig. 4.11.b matches an instance of the class *Block* (denoted by the object *this* since the rule is defined as operation of the class *Block*) and its metaobject which is an instance of the class *MOFClass*. The object *this* is element of the M1-metalevel, whereas the (meta)object *c* belongs to the M2-metalevel. The connector between the object and its corresponding metaobject is represented by a dashed arrow with the label *instance of*. This method called *checkBlockType* returns the type of the calling block at runtime, e.g. if it is a constant block, or a product block, or a trigger block, etc. This information is contained at runtime in the attribute *name* of the metaobject which is linked to the class of the calling object. Thus, we do not need to define such a method for every type of *Block*, but only one single generic method. In addition, it makes the type checking of an object pretty compact.

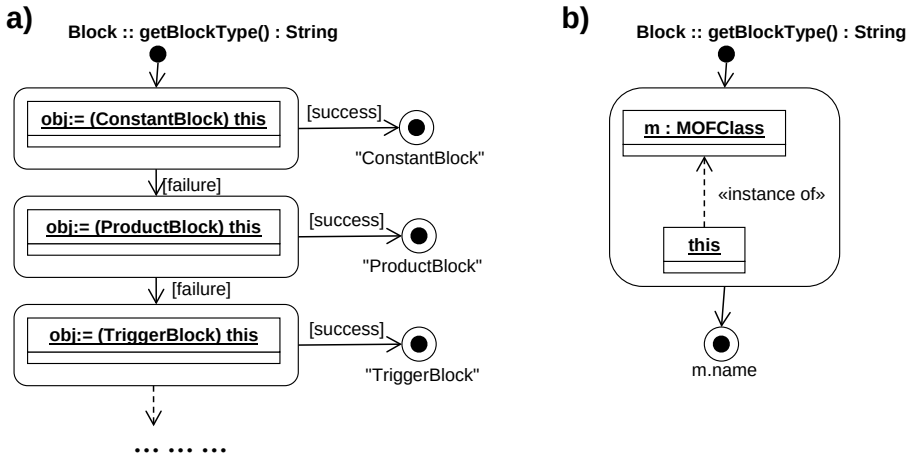


Figure 4.11: Checking of the type of a block - without and with reflective feature

Fig. 4.11.a presents an alternative for the method *checkBlockType* that resorts to the old SDM syntax, remaining on the single M1-metalevel. It is thus possible with SDM to cast the type of an object, and, thus, to check it. If the type checking is successful, the *success*-transition is traversed, and the type name is returned. Else, the failure-transition is traversed, and an other typecasting is tried is the next activity. In this case, as many typecasting (and, hence, SDM activities) as block types are

necessary. The implementation of this variant costs too much effort comparatively to the variant of Fig. 4.11.b. In addition, it is error-prone compared to the reflective variant. Because it must always be compliant with the underlying metamodel, the method specification of Fig. 4.11.a should be updated every time a block type is added, deleted, or modified in the metamodel.
Finally, not only the SDM diagram, but also the generated code is more compact thanks to the reflective feature. Instead of generating the code corresponding to all checking, the method which is able to return the metaobject with the relevant information about the object type is simply called: *MOFClass m = this.refMetaObject()*.

4.2.3 Prototype and Application

We extended our meta-CASE-tool MOFLON in order to support the specification of multilivel graph transformations, i.e. with access to the next higher metalevel such as in the example of Fig. 4.11.b.
Similarly to the generic feature, the most important aspect to be adapted is the code generation, and more precisely, the Velocity templates. We explained in Section 4.1.2 the principles of the Velocity template language. Fig. 4.12 shows the modifications executed on the templates to generate a correct code when using our reflective SDM feature.

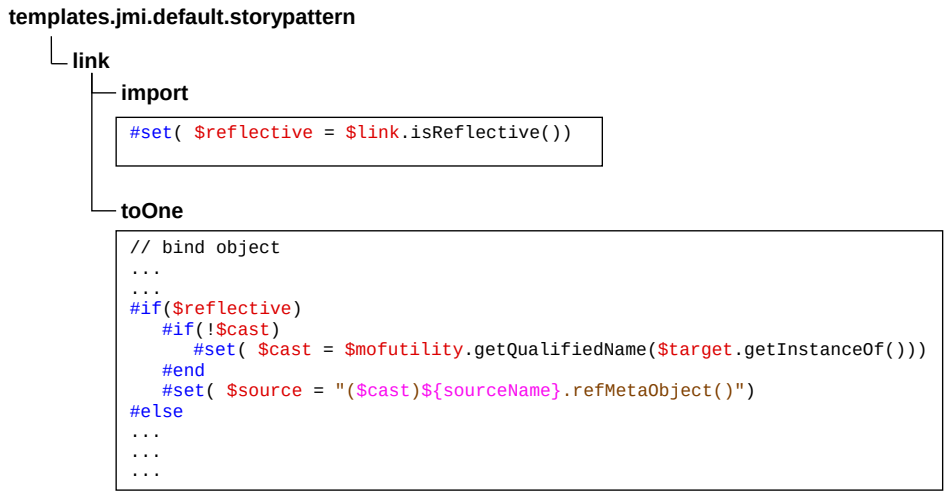


Figure 4.12: Generic Feature - Adaptation of the Velocity Templates

The template modification concerns two templates dedicated to the code generation for a link (*import.vm*, and *toOne.vm*). Instead of modifying the parsing mechanism of CodeGen2 by defining a new kind of element, we process the dashed arrow labeled with *instance of* as a simple link.
The template called *import* is a template which is always parsed when code for a link must be generated. A new reference called *isReflective* is defined in this template to differentiate the case in which code for a real link must be generated,

and the case in which the processed link represents the metalevel-spinning. This reference *isReflective* has a Boolean value, set to *true* in the case of a link between an object and a metaobject.
The template *toOne* is applied when matching an element at the end of an association with the cardinality *1* or *0..1*. We added in this template a conditional structure *#if...#else...* in order to generate specific code in the case of metalevel-spinning. Because the variable is defined at the beginning of the method as a *java.lang.Object* (Cf. the application example presented bellow), a typecast is required, and is defined in the template with the reference called *cast*. The reference *sourceName* is the name of the object at the one end of the connection *instance of* whereas the reference source relates to the matched metaobject.

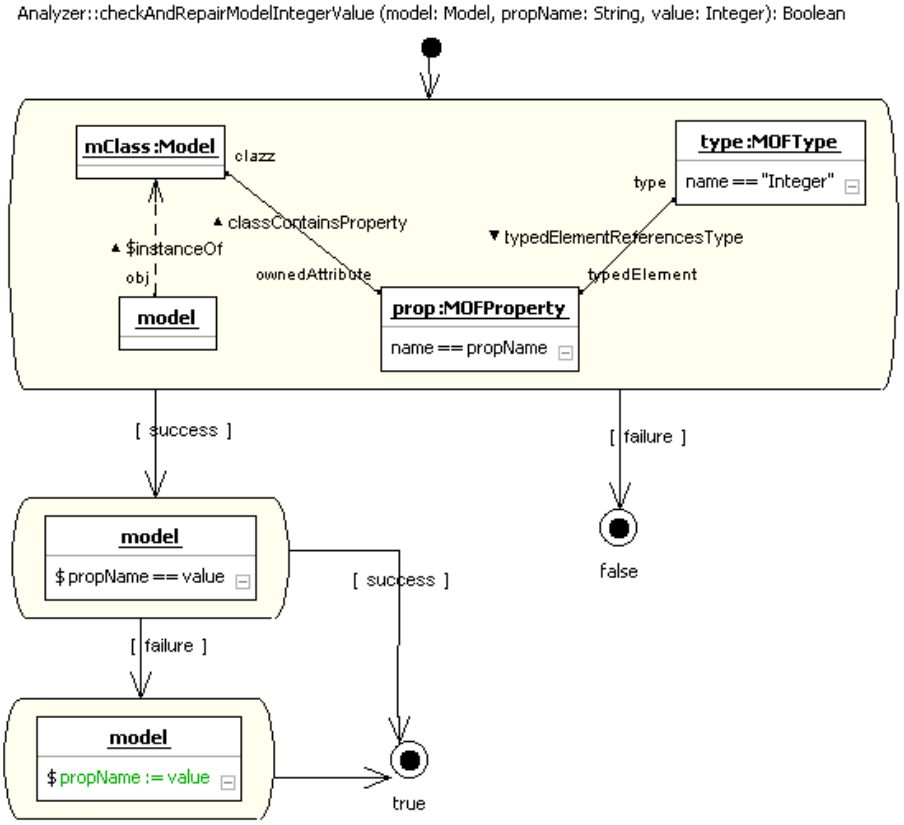


Figure 4.13: Using reflective SDM feature - Example

Let us see a concrete application example of the new reflective feature. In Fig. 4.8, we showed a generic variant to implement the analysis and correction of the guideline na_004. This solution consisted of a main method that calls all necessary check-and-repair methods generically defined. Though, it was necessary to define as many methods as types of attribute: for Boolean attributes, for Integer attributes, and for String attributes. In addition, it suppose that these methods are correctly

called, i.e. that the type of the attribute whose name is parameterized (*propName*) corresponds to the type of the parameter *value*.
In Fig. 4.13, which is a screenshot of MOFLON, we extended the method *checkAndRepairModelIntegerValue* with an additional activity to check if the method has been adequately called. This activity checks if the type of the attribute whose name is parameterized as *propName* is conform to the type of the parameter *value*. This checking is allowed by the reflective feature which gives access to the meta-information. Concretely, the start point is the object *model*. The corresponding metaobject called *mClass* is matched. It is possible to explore the metamodel from this metaobject. The *MOFProperty* corresponding to the attribute whose name is parameterized as *propName* is matched. Then, the corresponding *Type* is matched, and its name is compared with the type of the parameter *value*, i.e. here *Integer*. If the attribute can be matched, and has the correct type for the method execution, the *success*-transition is traversed, the checking and, if necessary correction, of the model is executed, and the Boolean value *true* is returned. Else, the model transformation terminates, and the Boolean value *false* is returned.

4.3 Related Works

The extension described in this chapter is not the first one which has been proposed to enhance the SDM syntax. We can cite for instance [Gor08] that proposes among others a *copy*-operator for copying subgraphs within model repositories [GJ06]. The main topic of this thesis is not such new features, which are only application examples, but the definition of new language constructs as extensions to a small transformation modeling profile. [Gor08] does not propose any generic or reflective features for controlled graph transformations. Though, this work may be relevant for the implementation of additional SDM extensions. Our enhancement proposal for the graph transformation rules used by MATE must provide a support for the definition of generic graph query and rewrite rules combined with the reflective JMI mechanisms.
Reusability is a desirable property for graph transformation, as proved by different works aiming at the improvement of this characteristic. For instance, [Pha12] presents an approach in order to reuse graph transformations which have been defined for a given metamodel. Even if the graph transformations are not generic templates as in [dLG12], this approach also aims at using a given graph transformations on different metamodels. It is based on a type mapping instead of a language extension such as our new features for SDM. [AVK$^+$05] presents another way to improve the reusability of graph transformations. Contrary to our language extension for domain-dependent graph transformations, this approach consists in defining domain-independent design patterns which apply in the context of graph transformations.
Generic rules have been proposed quite early for textual languages, e.g. for the Van Wijngaarden Grammars [vWMP$^+$] that allows for the definition of potential

infinite grammars with a finite number of rules, or later in [VP04] as successor of Göttler's two level grammar. The main drawback of this approach is its complexity. The textual counterpart to the parameterization of modeling elements described in Section 4.1.1 corresponds to parametric polymorphism as supported e.g. by programming languages like Ada or Java, just to mention a few.
Similarly to our work, the approach presented in [dLG12] is based on concepts from generic programming in order to define generic graph transformation. Contrary to our generic extension for SDM, which is defined on a metamodel, these generic graph transformations are reusable templates which apply to different metamodels. The authors of [dLG12] use so-called concepts to gather the requirements of a family of metamodels, needed by a reusable graph transformations to work. The concepts' elements (nodes, edges, attributes) are interpreted as variables to be bound to elements of specific metamodels. Another generic approach is described in [SMM+12]. Whereas our approach consists in extending the graph transformation language, the approach in [SMM+12] is based on a generic metamodel and the Pull Up method refactoring [Fow99].
Regarding visual graph transformation languages, the most usual mechanism to improve the reusability of graph transformations consists of the definition and use of parameters. The PROGRES language, which is described in Section 2.4.1, allows for the definition of generic rewrite rules [Mue02], but in a limited way only. Types are first-order objects that may be used to parametrize graph transformation rules, but no means are offered to parameterize rules with attributes or associations. Furthermore, PROGRES does not offer any means for runtime reflection. There are also approaches [DHJ+07] which provide genericity by the creation of concrete rewriting rules from generic ones rather than by parameterized rules. Nevertheless, reflectivity is also out-of-scope. [CM97] proposes an interesting approach for reflectivity in rewriting logic and its applications in several areas. One application that may be relevant for our work is the possibility of formally specifying novel reflective languages. [Kur08] presents an application of reflection in model transformation languages. Though, contrary to our features which provide access to the metalevel of the model on which the transformation is applied, this approach defines a metalevel of the transformation language itself. The Action Semantics is an extension for the UML language in order to annotate UML models with action statements as well as a model of execution for these statements. It has been integrated to the specification of UML 1.5. The authors of [PJMS01] explain how to complete the Action Semantics in order to add a behavioural reflective dimension to the UML. They propose links between elements from one metalevel to the previous or the next one. Thus, this proposal is pretty close to our own reflective feature, except that [PJMS01] only presents a textual notation whereas we propose a graphical notation. [Kur10] presents a reflective extension for the textual model transformation language MISTRAL [Kur05]. Whereas our reflective feature for SDM only supports a structural reflection with introspection, the approach in [Kur10] proposes a complete reflection, i.e. a behavioral reflection which support the ability of intercession.

We are not aware of any other kind of graph transformation approach which combines visual graph transformation with reflectivity and genericity, except of VIATRA2 which is presented in Section 2.4.2. The generic model transformation approach of the graph transformation tool VIATRA2 also supports generic as well as reflective model transformations. The main differences and drawbacks compared to the MOFLON approach presented here are the following. First of all generic attribute and association parameters are modeled as first-order (node) objects instead of using a slightly modified more lightweight standard notation, which is hence easier to understand, for generic attributes and associations. Furthermore, metatransformations are used to translate generic transformations into efficiently executable standard transformations at design time. MOFLON solves the efficiency problem in a rather different way. Its JMI standard compatible model repository already offers very efficient access to metamodels as well as reflective model manipulating operations with negligible runtime overhead. As a consequence each generic/reflective MOFLON transformation is directly translated into a single efficiently executable parameterized Java method instead of generating a separate model transformation (implementation) for each instantiation of a generic model transformation.

Chapter 5

Analysis of Graph Transformations

The generic and reflective SDM features presented in the previous chapter improve greatly the expressiveness and reusability of story diagrams. Though, not only these characteristics need to be improved. Another fundamental aspect of SDM must be ensured: the correctness of graph transformations.
The example of use of the reflective feature in Fig. 4.13 underlines a drawback of the new generic feature, namely the risks of type error at runtime since the type of the parameterized elements is only known when the model transformation is called and processed. As shown by the example, we may use the reflective feature to get the necessary information for the "manual" specification of a type checking function. Nevertheless, such a solution would ruin the benefit brought by the generic feature, namely the time and effort spared in drawing diagrams. Therefore, we propose here a formal method to statically check the type-safety of graph transformations. Such a type checking approach for SDM has not only the advantage of reducing type errors, but also to detect them earlier in the system development process. As explained in Section 2.3.3, the MOFLON compiler generates Java code from story diagrams. Generated code must fulfill the type constraints of the programming language (here, the Java language). Type errors in model transformations are propagated to the generated code. Thus, detecting such errors at this early step of the development life cycle prevents their propagation, and, consequently, reduces the costs. In fact, "*Prevention is cheaper than cure*" [DRP99].

We first present in this chapter a running example that illustrates which kind of type errors can occur at execution time and, thus, should be detected by means of static type checking. Our approach is based on the extraction of semantic information from the syntactic elements of the metamodel, the model transformations and the methods' calls by means of rules of inference. We namely want to detect type errors by means of natural deduction. Instead of using the classical notation with premises and conclusion, we define a new notation which combines syntactic

and semantic premises, constraints, and conclusions. After having explained our approach, we introduce our new notation and describe our type system. It is composed of semantic domains, relationships, and applications whose semantics can be defined by means of rules of inference. Then, we present and describe in a third section the set of rules whose application on the metamodel, the model transformations and the methods' calls should enable static type checking. The complete type system can be found in the Appendix B, too. A rule application algorithm is introduced in the fourth section of this chapter. In order to illustrate this algorithm and allow for a better comprehension, we apply it concretely to the running example from Section 5.1.1. Finally, we compare our approach with related work in a last section.

5.1 Motivation

Before we describe our approach to ensure correctness of graph transformations, we first present by means of concrete examples different kinds of errors. We show in this section how model transformations can be source of errors at runtime, e.g. if they are executed with the wrong arguments. In addition, we explain why the metamodel and the story diagrams should be inspected, too.

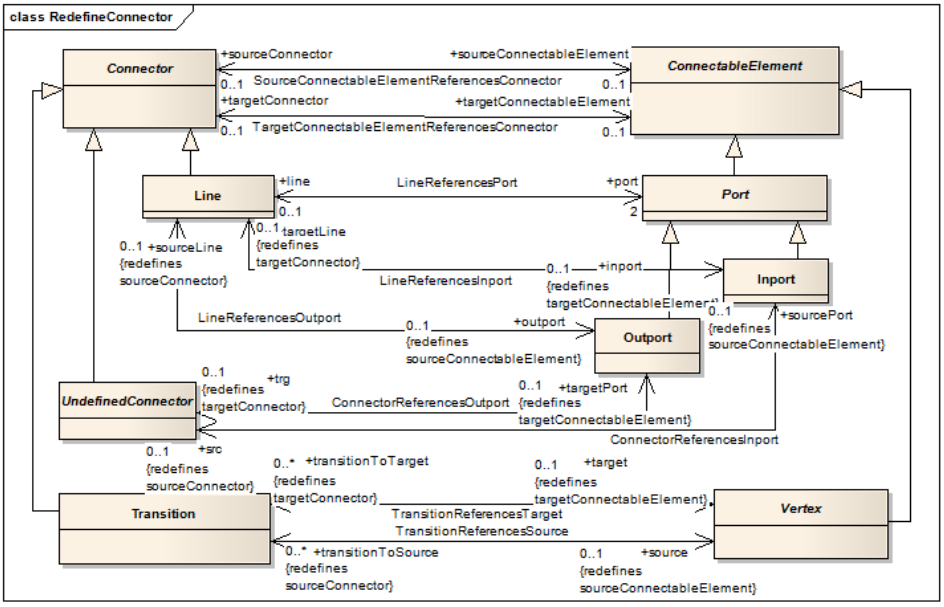


Figure 5.1: Redefine *Connector* and *ConnectableElement*

5.1.1 Type Errors in Graph Transformations: Examples

As explained in Section 3.2.2 and illustrated by Fig. 3.7, we used the concept of redefinition of association ends instead of OCL to define some constraints in the MATLAB metamodel. Fig. 5.1 depicts the part a of Fig. 3.7. *Connector* and *ConnectableElement* represent the connection between elements. *Inport* and *Outport* are Simulink *ConnectableElement*, and *Line* is a Simulink *Connector*. *Vertex* represents Stateflow *ConnectableElement*, and *Transition* is a Stateflow *Connector*. The ends of the associations between *Inport*/*Outport* and *Line*, and between *Vertex* and *Transition*, redefine accordingly the ends of the association between *ConnectableElement* and *Connector*.

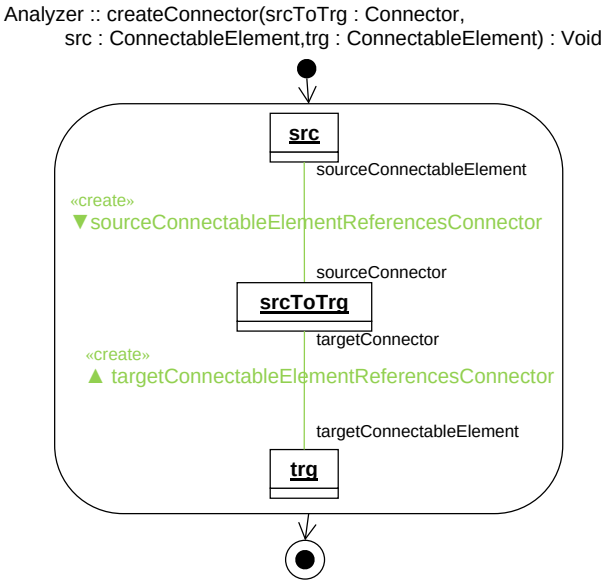


Figure 5.2: Method *createConnector*

Let us consider a method called *createConnector* which binds together an instance of *Connector* to two instances of *ConnectableElement*. Fig. 5.2 represents the behavior specification of this method by means of SDM. The source *src* and the target *trg*, elements of type *ConnectableElement*, as well as *srcToTrg*, elements of type *Connector*, are the parameters of this method. The method creates links between *srcToTrg* and *src*, and between *srcToTrg* and *trg*. Although this story pattern is syntactically and semantically correct, it is error-prone.
For instance, what will happen if the argument for the parameter *srcToTrg* is instance of *Line*, the argument for the parameter *src* is instance of *Outport* and the argument for the parameter *trg* is instance of *Vertex*? We can call the method since *Line* is of type of *Connector*, and *Outport* and *Vertex* are subclasses of *ConnectableElement*. Though, an error at runtime will occur. If *src* is instance of *Outport*, its property *sourceConnector* must be of type *Line* due to the redefinition

of the association end. This is the case since *srcToTrg* is an instance of *Line*. For the same reason, an instance of *Vertex* can only be connected to an instance of *Transition*. Consequently, if *trg* is instance of *Vertex*, a link between *trg* and *srcToTrg* can be created only if *srcToTrg* is instance of *Vertex*. This is not possible because *srcToTrg* is instance of *Line* and there is no inheritance relationship between *Line* and *Vertex*.
Of course we could define two distinct rules, the one for Simulink and the other for Stateflow. However, it would gainsay the purpose of our work, i.e. the reusability of story diagrams, and the reduction of effort in specifying transformations.

The extension of SDM with generic features provides other solutions as depicted by Fig. 5.3 by parameterizing the type of *src* and *trg* in the method's signature. Both versions differs slightly from Fig. 5.2, and illustrate the benefice of the generic SDM feature. We can namely provide as parameter the type of *srcToTrg* instead of an instance. This allows for creating the instance *srcToTrg* of type *Connector* before binding it to *src* and *trg*. Such a method is even more useful to repair models in the context of MAJA.
In the story diagram of Fig. 5.3.a , we add the type of *ConnectableElement* as parameter called *connected*. The type of both parameters *src* and *trg* is defined by *connected*. If the value of *connect* is *Line*, and the value of *connected* is *Port*, the arguments corresponding to the parameter *src* and *trg* can be instances of *Port*. Then, executing this method call allows for creating a Simulink connector. Similarly, if the value of *connect* is *Transition*, and the value of *connected* is *Vertex*, the arguments corresponding to the parameter *src* and *trg* can be instances of *Vertex*. Executing this method call allows for creating a Stateflow connector. Such a method specification is powerful, but can be error-prone. In the case the value of *connect* is *Line*, but the value of *connected* is *Vertex*, the execution of this method call is not possible since no *Line* can be created between two *Vertex* according to the metamodel. Even if the the value of *connect* is *Line*, and the value of *connected* is *Port*, an error will occur if *src* is of type *Inport* and *trg* of type *Outport*. Such a method call would respect the method signature since *Inport* and *Outport* are subclasses of *Port*. Nevertheless, if *src* is instance of *Outport*, its property *sourceConnector* is not of type *Line*, but of type *UndefinedConnector*, due to the redefinition of the association end. Similarly, if *trg* is instance of *Inport*, its property *targetConnector* is of type *UndefinedConnector* too. *UndefinedConnector* is an abstract class and cannot be instantiated. Moreover, there is no inheritance relationship between *Line*, which should be the type of *srcToTrg* according to the SDM diagram, and *UndefinedConnector*.
Fig. 5.3.b depicts another solution. The type of the *Connector* is parameterized, it is the parameter called *connect*. The type of *src* and *trg* is derived of *connect*. With this solution, the type definition of *src* and *trg* will comply with the MATLAB metamodel if both cases: if the *Connector* is a *Vertex* as well as if it is a *Transition*. Nevertheless, no error occurs only if the method call respects the signature. The method call must be inspected to ensure that it complies with the signature.

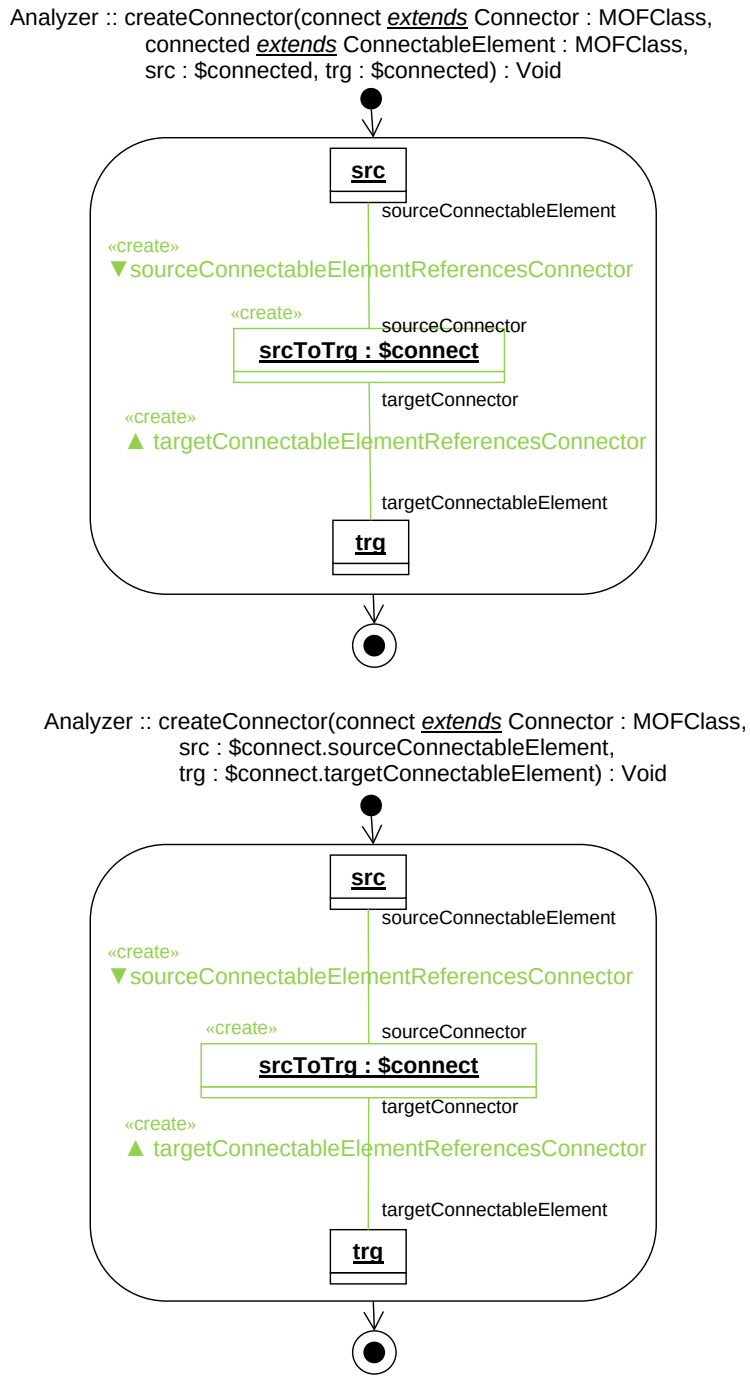


Figure 5.3: Method *createConnector* with parameterized type

The patterns of the story diagrams have a chance to match models' subgraphs only if these patterns are metamodel-compliant. Even if we specify such metamodel-compliant graph transformations and respect the SDM syntax, the examples in this section show that it cannot always prevent type errors at runtime. Therefore, we want to introduce a new approach of static type checking for SDM, and, more generally, for graph transformations.

5.1.2 Metamodeling and Specification Errors

The type errors described above concern only the method calls. Nevertheless, errors at runtime cannot always be prevented by only ensuring adequate arguments. A graph transformation produces a metamodel-compliant target model only from a metamodel-compliant source model. In addition, the patterns composing the transformation rules must be metamodel-compliant too so that subgraphs in the source model can be matched. Therefore, it is necessary to check the metamodel as well as the story diagram, and not only the arguments of the transformation. We describe here error examples concerning the metamodel and the story diagrams.

Metamodeling Errors

The metamodel excerpt of Fig. 5.4 looks like the one of Fig. 5.1. Nevertheless, it contains errors: on the one hand, an inheritance cycle error, and on the other hand, an association end redefinition error.
The inheritance cycle is due to the inheritance relationship between the class *ConnectableElement* and the class *Inport*. Then, we have an inheritance cycle composed of *Inport -> Port -> ConnectableElement -> Inport*.
The association end redefinition error is caused by the missing inheritance relationship between *Vertex* and *ConnectableElement*. As explained in Section 3.2.2, the concept of redefinition involves not only association but also generalizations. Each of the set of types connected by the redefining association end must conforms to a corresponding type connected by the redefined association end. Thus, the properties *source* and *target* can redefine *sourceConnectableElement* and *targetConnectableElement* only if *Vertex* is of type *ConnectableElement*. In other words, only if *Vertex* is connected to *ConnectableElement* by an inheritance relationship, which is not the case here in Fig. 5.4.
Because the model transformation are defined over the metamodel, a precondition for a correct model transformation is a correctly specified metamodel. Therefore, errors such as the ones described above must be detected.

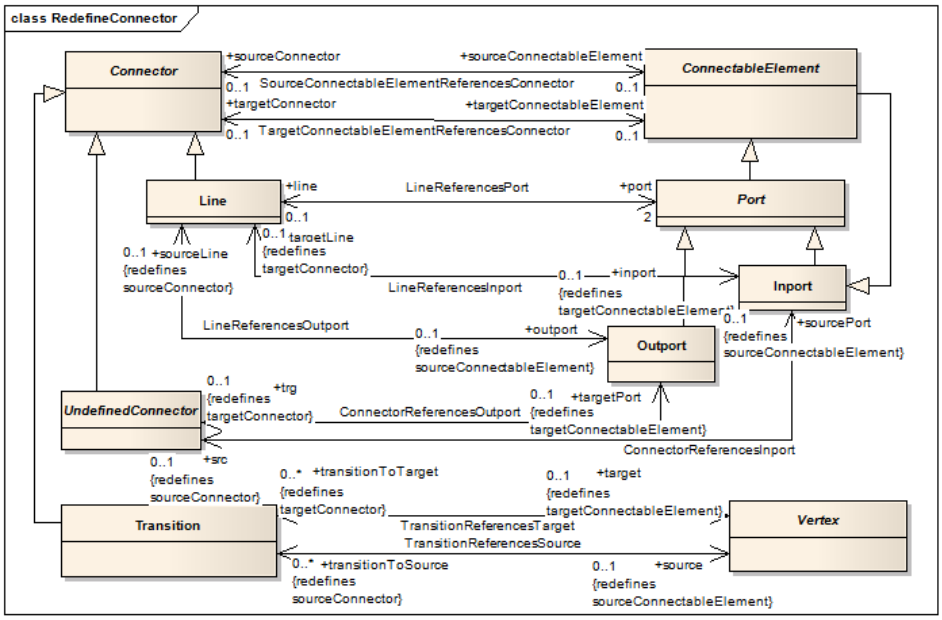


Figure 5.4: Metamodeling Errors

Specification Errors

The model transformation of Fig. 5.5 looks like the one of Fig. 5.2. Nevertheless, it contains errors: the method signature and the story diagram are not metamodel-compliant.

The parameter *src* is defined as an object of type *Connector*. Though, the object *src* is connected to the object *srcToTrg* of type Connector by a link. According to the metamodel, there is no association from *Connector* to itself. Thus, this link is invalid. Because the metamodel contains a property called *sourceConnector* corresponding to the type *Connector* and a property called *sourceConnectableElement* corresponding to the type *ConnectableElement*, we can also consider that not the link but the type specification of *src* is incorrect.

Another error concerns the link between *srcToTrg* and *trg*. The property called *targetConnectableElement* corresponds to the type *ConnectableElement* in the metamodel, and not to the type *Connector* like in the story diagram. Similarly, the property called *targetConnector* corresponds to the type *Connector* in the metamodel, and not to the type *ConnectableElement* like in the story diagram. Thus, we can conclude that the link is specified in an inverted way, or that the types of srcToTrg and trg should be inverted.

In both cases, the errors can be interpreted, and, hence, corrected, in two different ways. It depends on the method's behavior as desired by the developer. Consequently, a correction would require a human-machine interaction. Nevertheless, an analysis of the diagram based on the type information can at least point out incorrect model transformations, and, thus, prevent error at runtime.

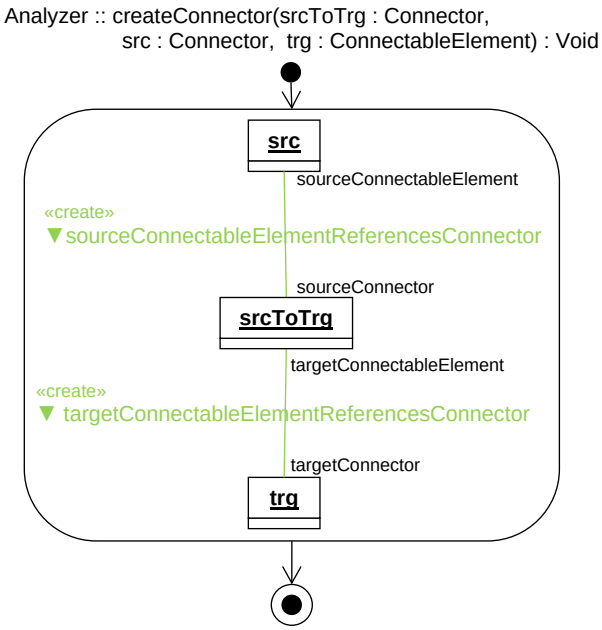


Figure 5.5: SDM Specification Errors

Another scenario which would lead to an error at runtime deals with the modifications of the metamodel after the model transformation has been specified. The renaming of a metamodel element can lead to inconsistencies. For instance, if the class *Connector* is renamed into *Connection*, the story diagrams of Fig. 5.2 and Fig. 5.3 which originally was metamodel-compliant are not valid anymore.

A correct graph transformation produces a metamodel-compliant target model only from a metamodel-compliant source model. In addition, the patterns composing the transformation rules must be metamodel-compliant too so that subgraphs in the source model can be matched. If a story pattern is not metamodel-compliant, no matter the model on which the graph transformation will be executed, it will namely be impossible to find a host graph. Thus, the detection of such an error will improve the efficiency of the graph transformation by detecting which graph transformation will always fail.
Let us repeat the concepts of correctness we will use in this work as defined in Section 2.5.1:

- **Syntactically correct model transformation**: it respects the syntax of the model transformation language. In the context of our work, this implies that the story diagrams are correctly defined with respect to the syntax of the MOSL language.

- **Semantically correct model transformation**: it respects the static semantics of the model transformation language. In the context of our work, it respects the static semantics of the MOSL language and, thus, does not violate the constraints of the type system we have defined. As a consequence, a semantically correct model transformation is well-defined, i.e. it produces a metamodel-compliant target model from a metamodel-compliant source model. In other words, the rule's LHS can only match a subgraph from a metamodel-compliant source model, and the rule's RHS can only match a subgraph from a metamodel-compliant target model. In addition, both source and target models fulfill semantic constraints, e.g. about redefinition of association ends.

5.2 Definition of a Type System

In order to detect (potential) type errors, we need to extract type information from the metamodel and the story diagrams. We first provide an overview of our approach in this section. According to this approach, we obtain type information by means of rules of inferences from the syntactic domain to the semantic domain. We then describe these semantic domains as well as some operators and functions before introducing a new notation for rules of inference. Finally, we define the semantics of the operators and functions by means of rules of inference.

5.2.1 Overview of our Approach

Endogenous graph transformations such as SDM diagrams consist of a chain of one or more transformation rules. Each transformation rule (in SDM: each story pattern) maps a source model to a target model which becomes the source model of the next rule. The transformation rules are defined over the metamodel whose instances are sources and targets of the transformations. As basis of the model transformations, the correctness of the metamodel specification is essential. For instance, the metamodel must not contain any inheritance cycle. The correctness of the metamodel specification is the first aspect our analysis must consider.
Then, the correctness of the graph transformation specification must be ensured. A match of a rule means that a graph homomorphism can be found from the left-hand side of the transformation to the model. A rule is semantically correct if it transforms a metamodel-compliant source model into a metamodel-compliant target model. Consequently, any matched subgraph should belong to a metamodel-compliant source model, and, thus, the story patterns composing the rule must be metamodel-compliant. For instance, any object pattern must match a class instance, or a link between two pattern objects must correspond to an association in the metamodel. Besides, the application of a transformation rule must result in a metamodel-compliant target model. Therefore, not only the left-hand side, but also the right-hand side of the transformation must be metamodel-compliant.

Finally, as shown by the examples of Section 5.1.1, depending on the arguments, a syntactically and semantically correct graph transformation applied on a metamodel-compliant source model does not ensure an error-free execution at runtime when it is called.
Therefore, the analysis of graph transformations must consider several aspects:
1) the correctness of the metamodel which is the basis of the graph transformations.
2) the syntactical and semantical correctness of the graph transformations.
3) the arguments of the method calls.

A DSL (**D**omain-**S**pecific **L**anguage) such as MATLAB Simulink/Stateflow, whose metamodel is presented in Section 3.2, has syntax and semantics [KM08]. More precisely, the abstract syntax, i.e. the metamodel, defines the structure of its models, and contains the elements that can be used to create syntactically correct models. The concrete syntax is the actual notation presented to the user. The dynamic semantics describes the meaning and behavior of the DSL, whereas static semantics corresponds to well-formedness rules for the models. Thus, in order to prevent errors when executing model transformations, we need to extract the static semantics of the metamodel and the graph transformation rules from their concrete syntax. In other words, we need to extract type information from visual modeling elements. To this end, we base our approach on *natural deduction* [BE02].
We express our logical reasoning by means of rules of inference. The standard notation for rules of inference consists of premises placed above conclusions, and separated by a horizontal line. Taking syntactical elements as premises, we can derive semantical conclusions. Though, relevant semantical information can not necessarily be directly obtained from syntactical elements, but from the combination of semantical information or from the combination of syntactical and semantical elements. In addition, natural deduction allows for demonstrating that a reasoning is correct, but is not explicitly suitable to prove invalidity [Lab05]. Consequently, it appears that we have to extend the notation and the semantics of the rules of inference, on the one hand to account for the syntactical and the semantical premises, and on the other hand to express invalidity. We describe this new notation in Section 5.2.3.

Because model transformations are based on the metamodel, we first extract semantic information by applying rules of inference on the metamodel: set of classes contained in the metamodel, inheritance relationships between them, set of owned properties. These metamodel information are the basis for defining the story diagrams.
Besides, we need to inspect the methods' signature for two reasons. On the one hand, the objects declared as parameters are bound in the story diagram, and the properties and values are used to define attribute checks or assignments. Thus, the signature's inspection must ensure that the declared parameters comply to the metamodel, i.e. parameters' types must be defined in the metamodel. For instance, a link between two objects must be an instance of an association between both

corresponding classes, etc. On the other hand, the semantic information extracted from an operation's signature will allow for checking the arguments and eventually relationships between arguments when this operation is called.
The inspection of a story diagram must ensure that the specified patterns comply to the metamodel. This is necessary if we want that a transformation rule can match a metamodel-compliant source model. For instance, a pattern containing an object whose type is not defined in the metamodel will never match any metamodel-compliant source model. Thus, detecting such errors in a story diagram ensures a more effective specification development process by eliminating in the early steps invalid model transformations. In addition, the conclusions of the applied rules of inference define relationships between pattern elements of the story diagram.
Finally, methods can be instantiated within a story diagram (by means of a collaboration statement). Whereas the type information contained by the method signature and the story diagram are only type restrictions, the arguments assign an exact type to the elements. These method calls must be inspected to ensure that they conform to the corresponding method signatures. The arguments must respect the type restriction or the relationship between parameters as defined in the signature. Whereas the type information contained by the method signature and the story diagram are only type restrictions, the arguments assign an exact type to the elements.

5.2.2 Semantic Domain

As explained in the previous section, our approach is based on the extraction of type information from visual modeling elements. More precisely, we want to extract the static semantics of the metamodel and the graph transformation rules from their concrete syntax. To this end, we have to define a type system.

We first define sets which represent different elements of the static semantics, and we call these sets *semantic domains*.

- $\mathcal{C}$: set of classes contained in the metamodel.
- $\mathcal{D}t$: set of data types (e.g. *String*, *Boolean*, *Integer*)
- $\mathcal{O}p$: set of operations : $\{:=, ==, <, >, \leq, \geq\}$
- $\mathcal{P}$: set of properties.
- $\mathcal{O}$: set of bound objects (more precisely, bound pattern objects).
- $\mathcal{V}$: set of concrete values (e.g. *true*, *false*, *0.01*, *"Hello World"*).
- $\mathcal{P}a$: set of parameters.
- $\mathcal{A}rg$: set of a method's arguments.

We introduce the following operator: [[ ]]. This operator maps an element from the syntactical domain to the corresponding element from the semantical domain. For instance, if we consider the class called *Line*, *[[Line]]* is the corresponding semantic element and belongs to $\mathcal{C}$.

In addition, we define functions on our semantic domains in order to get additional type information.

- *type:* $\mathcal{O} \rightarrow \mathcal{C}$:
  It returns the class of an object. If $o \in \mathcal{O}$, type(o) returns the class the object o is an instance of.

- *dType:* $\mathcal{V} \rightarrow \mathcal{D}t$:
  It returns the data type of a value. For instance, dType(`true`) returns the data type `Boolean`.

- $\leq$: $\mathcal{C} \times \mathcal{C} \rightarrow \{$ *true, false* $\}$:
  It is the transitive, reflective closure of the class inheritance relationships. For instance, in the metamodel excerpt of Fig. 5.1, *[[Inport]]* $\leq$ *[[ConnectableElement]]* is *true* whereas *[[Connector]]* $\leq$ *[[Line]]* is *false*. More precisely, this binary relation builds the basis for the type polymorphism. If we consider an instance *aLine* of class *Line*, we can write *type([[aLine]])* = *[[Line]]*. Though, we can also write *type([[aLine]])* $\leq$ *[[Connector]]*, i.e. that an instance of *Line* is also an instance of *Connector*.

- $\circ : \mathcal{C} \times \mathcal{P} \rightarrow \mathcal{C} \cup \mathcal{D}t$:
  This operator returns the type of an owned property of a given class. For instance, *[[Transition]]* $\circ$ *[[target]]* returns *[[Vertex]]*, with *[[Transition]]*, *[[Vertex]]* $\in \mathcal{C}$, and *[[target]]* $\in \mathcal{P}$.

- $[[m]]_i \in \mathcal{P}a$ *with* $i \in \mathbb{N}$:
  This is the notation for the indexing of parameters. Here, $[[m]]_i$ represents the i^{th} parameter of the method m. Because a method can possess several parameters whose order is part of the method's signature, we defined this indexing.

- $\Theta : \mathcal{P}a \rightarrow \mathcal{C} \cup \mathcal{P} \cup \mathcal{D}t$:
  It returns type information about the parameter it is applied to. The detailed operator's semantics is described by the rules of inference in Fig. 5.11

- *arg:* $\mathcal{P}a \rightarrow \mathcal{A}rg$:
  It returns the argument to which a method's parameter corresponds in the context of a given method call.

The semantics of these operators and functions will be defined more formally in the next sections by means of rules of inference. Please note that there are no semantic domains for the methods and method calls. Thus, there is no function to connect

parameters to a given method or arguments to a method call. If we only consider the semantics domains and their functions, there is no explicit separation between global information (i.e. the metamodel and all methods and method calls) and local information (i.e. for a single method or method call). Nevertheless, this separation is ensured by the application algorithm described in Section 5.4.2.

5.2.3 New Rules of Inference

Once we have defined semantic domains, and additional operators and relations, we can define the rules of inference of our type system. The standard notation for rules of inference consists of premises placed above conclusions, and separated by a horizontal line. Because we want to obtain semantic information from syntactic elements, we want to use syntactic patterns as premises. Though, we noticed that a single syntactic pattern is not always sufficient to define a rule. Therefore, we divide the fields over the horizontal line into two parts. The first part at the top left corner contains the syntactic pattern, and the second part at the top right corner contains semantic premises.
In addition to the premises and conclusions, we need to specify semantic constraints. Thus, we also divide the fields under the horizontal line into two parts. The part at the bottom left corner contains semantic constraints, and the last part at the bottom right corner contains the conclusions of the rule. This last field which contains derived information is separated by a double line from the other fields which contain information to be checked. Fig.5.6 illustrates this new notation.

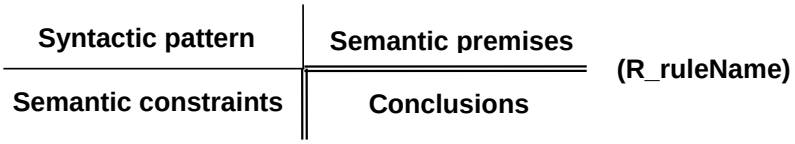


Figure 5.6: Rule notation

Our approach is defined in a syntax-directed way in order to attach semantics to syntactic language elements. Derived semantic information are collected in a semantic knowledge base. A rule is interpreted as follows. The syntactic pattern and the semantic premises must match for the rule to be considered. If they match, the semantic constraints can be checked. In the case the semantic knowledge base does not contain the information necessary to fulfill the semantic constraints, this means that the matched pattern is potentially not type-safe. Finally, if the premises match and the constraints are fulfilled, the rule can apply and the conclusions can be derived.
Unfulfilled semantic constraints mean only that the matched pattern is *potentially* type-unsafe. A single rule application is not enough to assert the existence of an error. In fact, a constraint can be unfulfilled either because there actually is an error or because semantic information are not available yet in the semantic know-

ledge base. As explained, if a rule can apply, additional semantic information are derived. Thus, the missing information can be available later after the application of other rules. Therefore, we cannot affirm that the specification contains an error only because a constraint's validity cannot be proven, but we can *suppose* it. It appears clearly that we need to define an algorithm to use the rules of inference in order to detect errors. This algorithm will be presented in Section 5.4.
The specification of a rule does not require to complete every field. Missing syntactic patterns (resp. missing semantic premises) mean that the rule always matches from the syntactical (resp. semantical) point of view. No semantic constraints means that the matched pattern is always type-safe. Missing semantic conclusions means that the rule does not derive any new semantic fact.

We can use this new kind of inference rules not only for type checking purposes, but also to specify properties and semantics of the above described operators. Fig. 5.7 and Fig. 5.8 depict such specifications in form of rules.

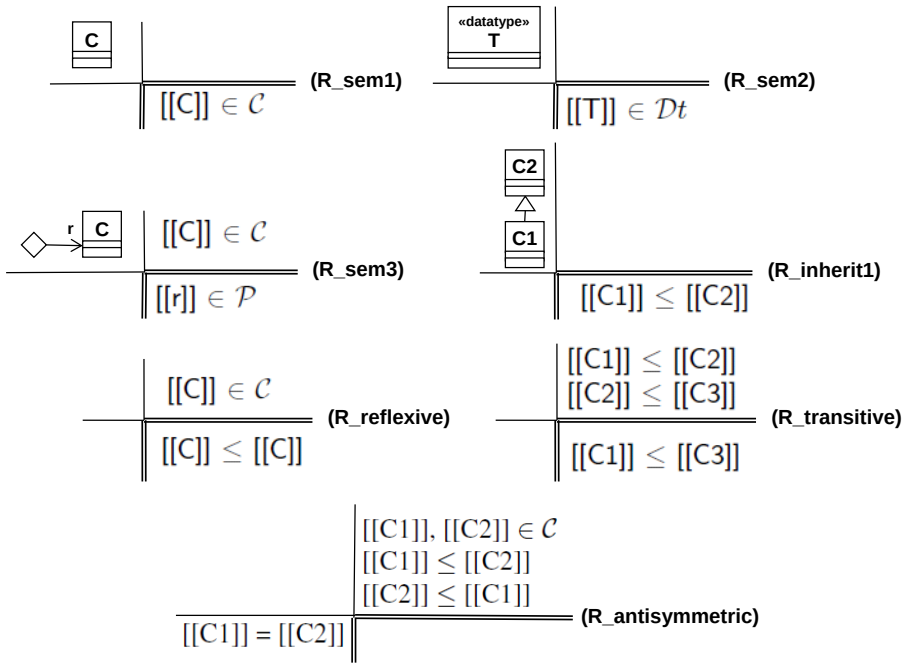


Figure 5.7: Operators' Semantics 1/2

The rules `R_sem1`, `R_sem2`, and `R_sem3` describe the application of *[[]]* on classes, data types and properties. Applied on a class called *C*, we can conclude that *[[C]]* belongs to the set $\mathcal{C}$ (rule `R_sem1`). Applied on a data type called *T*, we can conclude that *[[T]]* belongs to the set $\mathcal{D}t$ (rule `R_sem2`). Finally, applied on a property called *r*, we can conclude that *[[r]]* belongs to the set $\mathcal{P}$ (rule `R_sem3`).

The rule R_inherit1 is applied to inheritance relationship between two classes *C1* and *C2*. As a consequence, relations $\leq$ between *[[C1]]* and *[[C2]]* are established.
The relation $\leq$ is a reflexive relation, i.e. a binary relation for which every elements of a set relates to itself. The rule R_reflexive represents this property for the operator $\leq$.
The rule R_transitive defines another property of the relation $\leq$, namely the transitivity: if whenever a class *C1* is related to a class *C2*, and *C2* is in turn related to a class *C3*, then *C1* is also related to *C3*.
Because inheritance cycles are not allowed, two distinct classes cannot be subclasses of each other. As a consequence, the relation $\leq$ is not only reflexive and transitive, but antisymmetric too. This property is represented by the rule R_antisymmetric. A relation is antisymmetric if there is no pair of distinct elements each of which is related to the other. In other words, if *C1* and *C2* belong to the set of classes (*[[C1]], [[C2]]* $\in \mathcal{C}$), the semantic premises *[[C1]]*$\leq$*[[C2]]* and *[[C2]]*$\leq$*[[C1]]* imply that *C1* and *C2* are equal (*[[C1]]* = *[[C2]]*).

The rule R_prop1 represents the semantics of the operator $\circ$. For two classes *C1* and *C2* connected to each other by an association with the association end *r1* and *r2*, *r2* is a property owned by *C1* and is of type *C2*. This information is derived by means of the rule R_prop1 in form of *[[C1]]* $\circ$ *[[r2]]* which is equal to *[[C2]]*. In the same way, *r1* is a property owned by *C2* and is of type *C1*. This information is derived in form of *[[C2]]* $\circ$ *[[r1]]* which is equal to *[[C1]]*.

The rule R_generic defines the application of [[]] to a parameterized metamodel element, which is depicted as a String starting with a $-symbol. This rule indicates that the $-symbol plays no role in the semantic domains. Thus, *[[$Exp]]* is equal to *[[Exp]]*, where *Exp* represents any parameterized expression.

Syntactically, the navigation through the metamodel by accessing class properties is expressed by means of "." (dot symbol). Therefore, we define the rule R_compos which is applicable to a syntactic expression *Exp.r*. Here, *Exp* is an expression which semantically belongs to the set of classes C (*[[Exp]]*$\in \mathcal{C}$), i.e. *Exp* is not necessary an class name, but it must be at least evaluated into a class. The element *r* is a property name (*[[r]]* $\in \mathcal{P}$). The application of this rule requires the semantic constraint checking expressed as *[[Exp]]*$\circ$*[[r]]* $\in (\mathcal{C} \bigcup \mathcal{D}t)$. This constraint ensures that the property *r* is really owned by the class expressed by *Exp*. In this case, we can decompose *[[Exp.r]]* into *[[Exp]]*$\circ$*[[r]]*.
This rule can be applied iteratively on specification of complex navigation. For instance, considering the metamodel excerpt of Fig. 5.1, we can convert *[[Outport.sourceLine.inport]]* into *[[Outport.sourceLine]]*$\circ$*[[inport]]*, and finally into *[[Outport]]*$\circ$*[[sourceLine]]*$\circ$*[[inport]]* thanks to successive application of the rule R_compos.

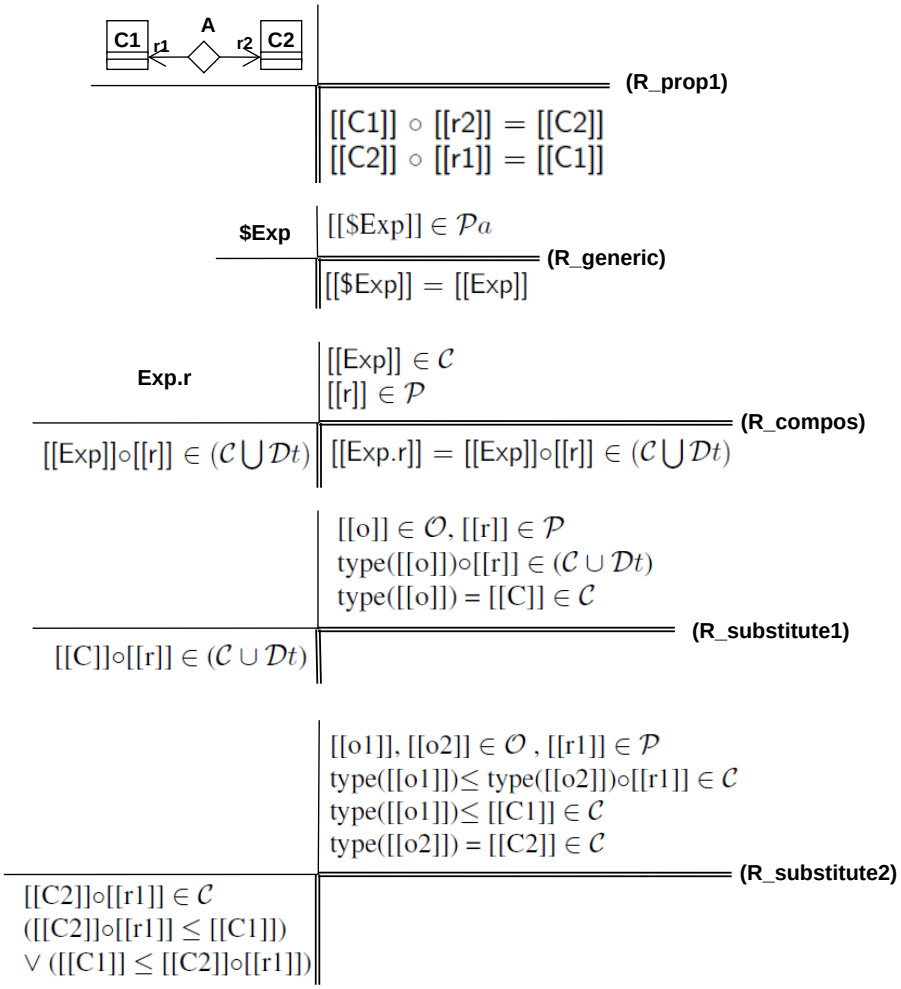


Figure 5.8: Operators' Semantics 2/2

We must ensure that applying a substitution to an expression will preserve its semantics. This is especially relevant when inspecting a method call. We must ensure that the relationships collected in the semantic knowledge base after the analysis of the metamodel and SDM diagrams still are respected when substituting a parameter by its actual value in the method call. Both rules `R_substitute1` and `R_substitute2` allows for checking it.

In the case of `R_substitute1`, the semantic knowledge base contains the information about the property *r* belonging to the type of the object *o* (*type([[o]])∘[[r]] ∈ (C ∪ Dt)*). Then we must ensure that this relationship is still fulfilled if the exact type of the object *o* is known, and more precisely if *C* is assigned to the type of *o* (*type([[o]]) = [[C]]*). In other words, we must ensure that the property *r* belongs to the class *C*, as expressed by the constraint: *[[C]]∘[[r]] ∈ (C ∪ Dt)*.

In the case of the rule of inference R_substitute2, the semantic premise, expressed as *type([[o1]]) ≤ type([[o2]])∘[[r1]]*, means that the type of the object *o1* is restricted by the property *r1* belonging to the class the object *o2* is instance of. If the object *o1* is instance of a subclass of *C1* (*type(([[o1]])) ≤ [[C1]]*), we must ensure that this relationship is still fulfilled if the exact type of the object *o2* is known, and more precisely is the class *C2* (*type([[o2]]) = [[C2]]*). Therefore, if we substitute *[[C2]]* for *type([[o2]])* in *type([[o2]])∘[[r1]]*, we define constraints which ensure that the property *r1* belongs to *C2* (*[[C2]]∘[[r1]]* $\in \mathcal{C}$), and that there is an inheritance relationship between the type of this property and *C1* as expressed by *([[C2]]∘[[r1]] ≤ [[C1]]) ∨ ([[C1]] ≤ [[C2]]∘[[r1]])*.
The role of both rules R_substitute1 and R_substitute2 will be illustrated by the application examples in Sections 5.4.4 and 6.1.

5.3 Rules of Inference

Once we have defined the semantics domains and the predicates and functions, we can determine the rules of inference which will be used as type system. This set of rules, which is expressed using the new notation introduced in Section 5.2.3, can be sorted according to the syntactic elements on which they are applied. We can distinguish rules applied on the metamodel, rules applied on the story patterns, rules applied on the method signature, and rules applied on the methods calls.

5.3.1 Rules of Inference for the Metamodel

Graph transformations must respect the metamodel on which they are defined. Therefore, we define a first set of rules which complete the rules R_sem1, R_sem2, R_sem3, R_inherit, and R_prop1 presented above. The application of the rules R_sem1, R_sem2, and R_sem3 convert syntactic elements (classes, datatypes and properties) into their semantic counterparts. The rule R_inherit provides information about the inheritance relationships between the classes, and R_prop1 allows for discovering the set of properties (association ends) owned by each class. Fig. 5.9 depicts the other rules which can be applied on the metamodel to extract semantical information.

The rule R_prop2 applies on the attributes. The syntactic pattern is a class *C* with an attribute called *attr* of type *T*. We first must ensure that the type *T* is defined in the metamodel (*[[T]]* $\in \mathcal{D}t$). If this condition is fulfilled, we can conclude that *[[attr]]* belongs to the set of properties (*[[attr]]* $\in \mathcal{P}$), and that *[[C]]∘[[attr]]* returns the attribute type *[[T]]*.

The rule R_redefineProp is applied on redefinitions of association ends. In this rule, *r2'* is a property which is owned by the class *C1'*, and which *redefines r2* (*[[r2]]* $\in \mathcal{P}$). Semantic constraints must be checked. According to the concept of

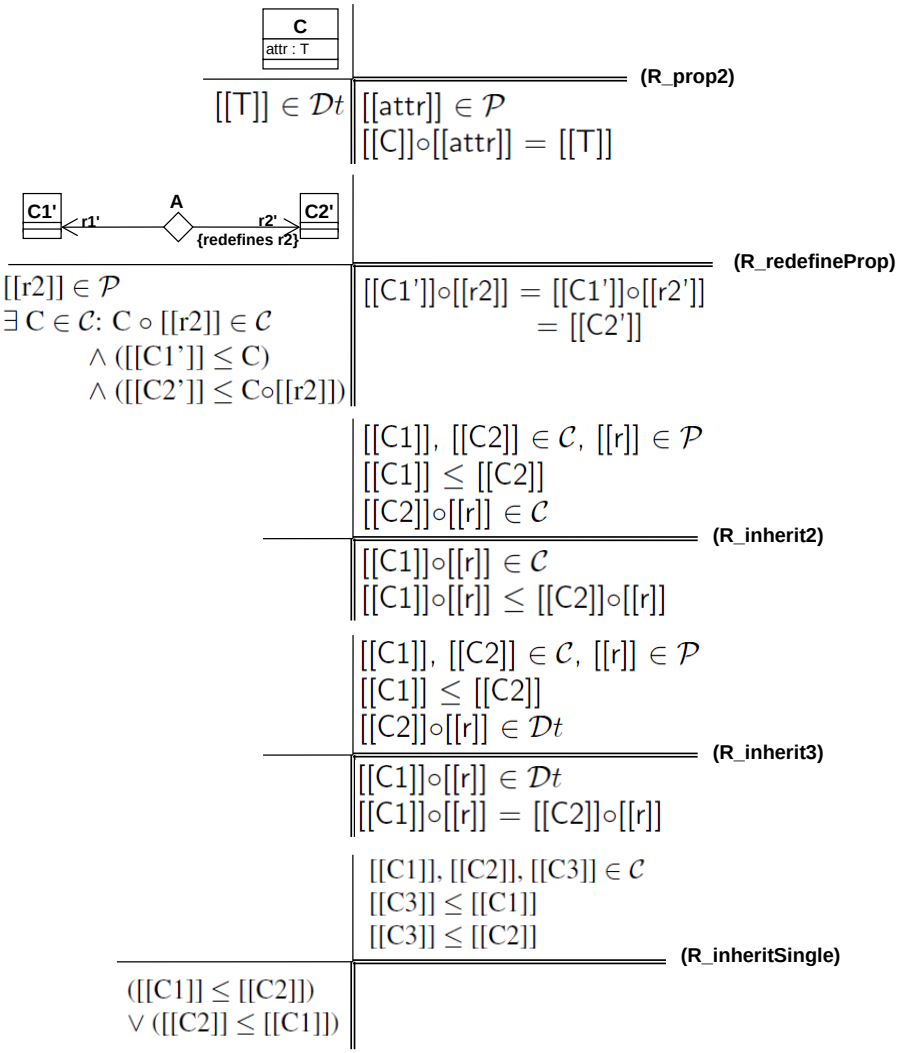


Figure 5.9: Rules Applicable on the Metamodel

redefinition, we must ensure that the class *C2'* connected by the redefining property *r2'* conforms to a corresponding type connected by the redefined property *r2*. In other words, there exists a class *C* which is connected to the redefined property *r2* (*C∘[[r2]]*). *C1'* is part of the subclasses of *C* (*[[C1']]≤C*), and *C2'* is a subtypes of property *r2* owned by *C* (*[[C2']]≤ C∘[[r2]]*). If theses constraints are fulfilled, we can conclude that, according to the concept of redefinition, the collections denoted by the redefining and redefined properties are the same, which is expressed by *[[C1']] ∘ [[r2]] = [[C1']] ∘ [[r2']]*.

The rules R_inherit2 resp. R_inherit3 concern the inheritance of properties (association ends resp. attributes). If we consider two classes *C1* and *C2* where *C1* belongs to the subclasses of *C2* and *C2* possesses a property *r*, then *C1* inherits this property. Semantically, this pattern is expressed as following in R_inherit2. *[[C1]]* and *[[C2]]* belong to the set of classes $\mathcal{C}$, and *[[r]]* belong to the set of properties $\mathcal{P}$. *[[C1]]* $\leq$ *[[C2]]* represents the inheritance relationship between *C1* and *C2*, obtained when applying the rules R_inherit1 and R_transitive. *[[C2]]*$\circ$*[[r]]* $\in \mathcal{C}$ in the semantic pattern of R_inherit2 means that *r* is an association end which belongs to the properties of *C2* whereas *[[C2]]*$\circ$*[[r]]* $\in \mathcal{D}t$ in the semantic pattern of R_inherit3 means that *r* is an attribute of *C2*. Then wen can conclude that the property *r* belongs also to *C1* (*[[C1]]*$\circ$*[[r]]* $\in \mathcal{C}$ resp. *[[C1]]*$\circ$*[[r]]* $\in \mathcal{D}t$). More precisely, in the case of inherited attribute (rules R_inherit3), the type of *[[C1]]*$\circ$*[[r]]* is the same as the type of the inherited attribute (*[[C2]]*$\circ$*[[r]]*). In the case of inherited association end (rules R_inherit2), we can only indicate that the type of *[[C1]]*$\circ$*[[r]]* belongs to the subtypes of the property owned by *C2*. In fact, we cannot conclude that it is exactly the same due to the eventuality of an association end redefinition (Cf. Rule R_redefineProp). This is expressed by the relation $\leq$ between *[[C1]]*$\circ$*[[r]]* and *[[C2]]*$\circ$*[[r]]*.

The rule R_inheritSingle enforces single inheritance. We can apply it in this work since the code generated from the metamodel and model transformations is Java, i.e. an object-oriented language which only supports single inheritance. Accordingly to the single inheritance, if a class *C3* is a subclass of both classes *C1* and *C2* (*[[C3]]*$\leq$*[[C1]]* and *[[C3]]*$\leq$*[[C2]]*), theses classes *C1* and *C2* must be connected to each other by an inheritance relationship (*[[C1]]*$\leq$*[[C2]]* $\vee$ *[[C2]]*$\leq$*[[C1]]*).

5.3.2 Rules of Inference for the Method Signatures

Fig. 5.10 and Fig. 5.11 depict rules which are applicable on the method signatures. More precisely, we focus here on the method's parameters. The rules depicted by Fig. 5.10 describe the application of *[[]]* on the parameters whereas the rules depicted by Fig. 5.11 define more precisely the semantics of the function Θ.

The rule R_sem4 illustrates how a parameter p_i of a method *m* is mapped to its semantical counterpart *[[*p_i*]]* which can also be noted *[[m]]*$_i$ as the i^{th} parameter of the method *m*. In addition, this rules adds *[[*p_i*]]* to the set of parameters $\mathcal{P}a$.

The rule R_sem5 applies on the case where the i^{th} parameter of a method *m* is an object *o* of type *C*. As indicated by the semantic pattern *[[C]]*$\in \mathcal{C}$, *C* is a class. This rule adds the i^{th} parameter to the semantic set of parameters $\mathcal{P}a$. We also can reduce the semantic parameter notation to the single object name as represented by the equality *[[o:C]]* = *[[o]]* $\in \mathcal{P}a$.

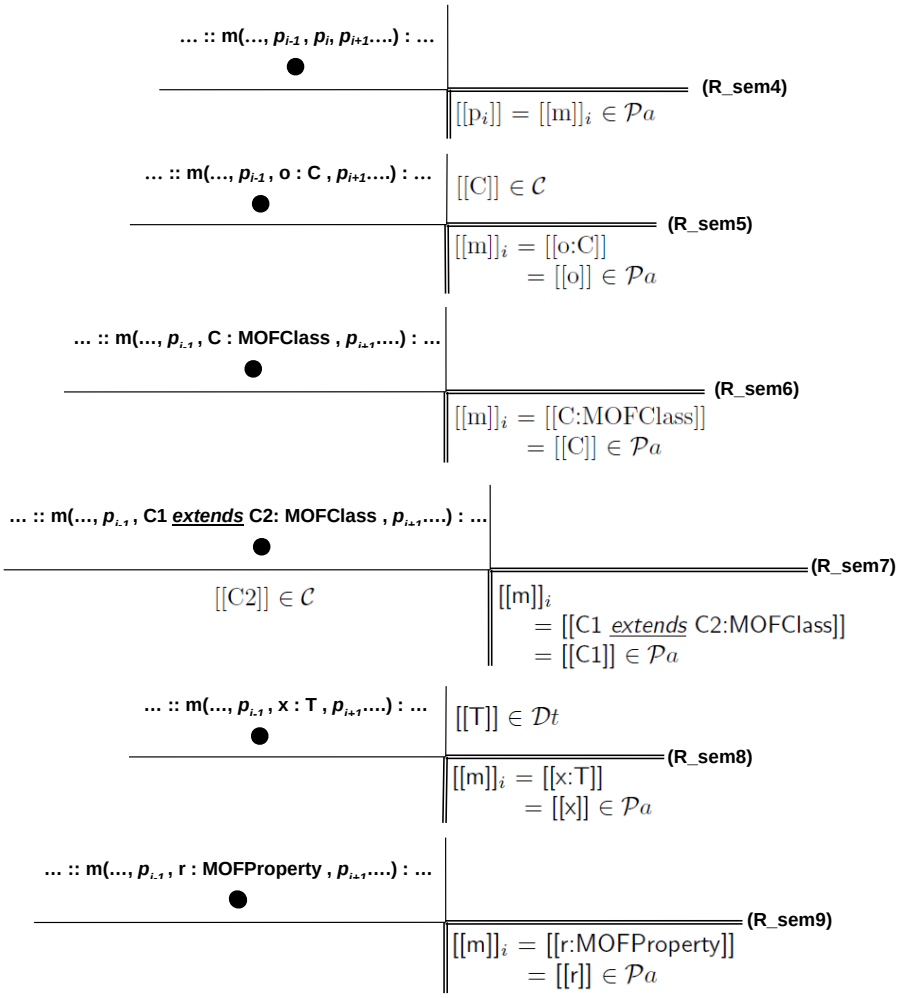


Figure 5.10: Rules Applicable on the Methods' Signature 1/2

The rules `R_sem6` applies on the case where the i^{th} parameter of a method *m* is an class *C* and, thus, instance of *MOFClass*. The rule `R_sem7` corresponds to the parameterization of a class too, but with a type restriction expressed by use of the keyword *"extends"*. The application of these rules allows for concluding that *[[C:MOFClass]]* resp. *[[C1 extends C2: MOFClass]]* belongs to the set of parameters $\mathcal{P}a$. In addition, we can reduce the notation by only indicating the name of the parameterized class (*[[C:MOFClass]] = [[C]]* resp. *[[C1 extends C2: MOFClass]] = [[C1]]*).

The rule `R_sem8` applies on data type parameters (*[[T]]* $\in \mathcal{D}t$). Here again, we can add the semantic counterpart of this ith parameter to the set of parameter $\mathcal{P}a$, and we reduce the notation to the single name of the attribute (*[[x:T]] = [[x]]* $\in \mathcal{P}a$).

Finally, the rule R_sem9 is pretty similar to R_sem6 except that it applies on parameterized properties, instance of *MOFProperty*, and not on parameterized classes. This rules allows for adding this *[[C:MOFProperty]]* to the set of parameters $\mathcal{P}a$, and we can reduce the notation by only indicating the name of the parameterized property (*[[r:MOFProperty]]* = *[[r]]* $\in \mathcal{P}a$).

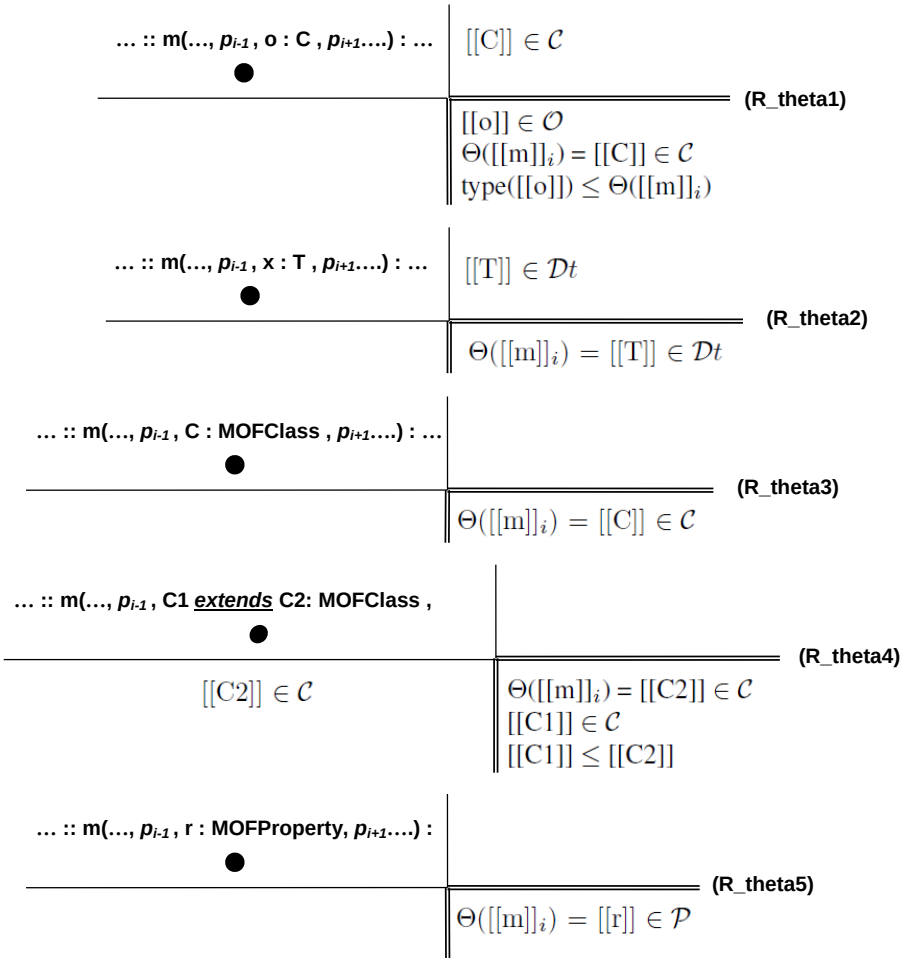


Figure 5.11: Rules Applicable on the Methods' Signature 2/2

The function Θ returns type information about a given parameter. The different application cases of the function Θ are depicted in Fig. 5.11 in form of several rules: R_theta1, R_theta2, R_theta3, R_theta4 and R_theta5.

The first rule, called R_theta1, corresponds to the case where the i^{th} parameter of a method *m* is an object *o* of type *C*. Then, the application of Θ on this parameter is equivalent to the application of the function *type* on the object *[[o]]*, and returns the class *[[C]]*. In addition, because an object is bound when it is parameterized, *[[o]]* is added to the set of bound objects $\mathcal{O}$. We can also deduce that the object's type is restricted by the type returned by Θ as expressed by *type([[o]])* $\leq \Theta$*([[m]]$_i$)*.
The second rule R_theta2 defines the case where the i^{th} parameter called *x* of a method *m* is a variable of type *T*, with *[[T]]* element of the set of data types $\mathcal{D}t$. In this case,we can derive that Θ applied on *[[m]]*$_i$ returns the data type *[[T]]*.
The third rule R_theta3 corresponds to the parameterization of a class, i.e. where the i^{th} parameter of the method *m* is an instance called *C* of *MOFClass*. In this case, Θ returns this instance of MOFClass, *[[C]]*, which then is assumed to be a class ([[C]]$\in \mathcal{C}$).
The rule R_theta4 corresponds to the parameterization of a class, too, but with a type restriction expressed by use of the keyword *"extends"*. In the syntactic pattern matching, *C2* restricts the type of the i^{th} parameter of the method *m* called here *C1*. When applying this rule, a type constraint must be checked: the restriction type *C2* must belongs to the metamodel's classes, i.e. semantically expressed, *[[C2]]* $\in \mathcal{C}$. Then, if this type constraint is fulfilled, we can derive that Θ applied on *[[m]]*$_i$ returns the type restriction *[[C2]]*. In addition, we can add *[[C1]]* to the set of classes and indicate that *C1* belongs to the subclasses of *C2* as expressed by *[[C1]]* $\leq$ *[[C2]]*.
The last rule R_theta5 applies on a parameterized property: the syntactic pattern to be matched is an i^{th} parameter of the method *m* which is an instance called *r* of *MOFProperty*. In this case, Θ returns this instance of MOFProperty, *[[r]]*, which then is assumed to be a property ([[r]]$\in \mathcal{P}$).

5.3.3 Rules of Inference for the Story Patterns

Fig. 5.12 and Fig. 5.13 depict the rules which are applicable on story patterns.

The rule R_bound is applied on a object *o* which is depicted as "bound" in the story diagram. This rules checks whether this object has been previously bound, i.e. semantically whether this object already belongs to the set of pattern objects $\mathcal{O}$. This is expressed by *[[o]]* $\in \mathcal{O}$ in the field "semantic constraints" of the rule.

The rules R_unbound1 and R_unbound2 match unbound objects and add them to the set of pattern objects $\mathcal{O}$. More precisely, R_unbound1 applies on an unbound object whose class *C* is directly specified whereas R_unbound2 applies on an unbound object whose class is parameterized (*[[cName]]* $\in claPa$).
When applying R_unbound1, we must ensure that *C* is a class belonging to the metamodel ([[C]] $\in \mathcal{C}$). If this type checking is correct, we can add the object *o* to the set of bound objects by concluding *[[o]]* $in\mathcal{O}$.

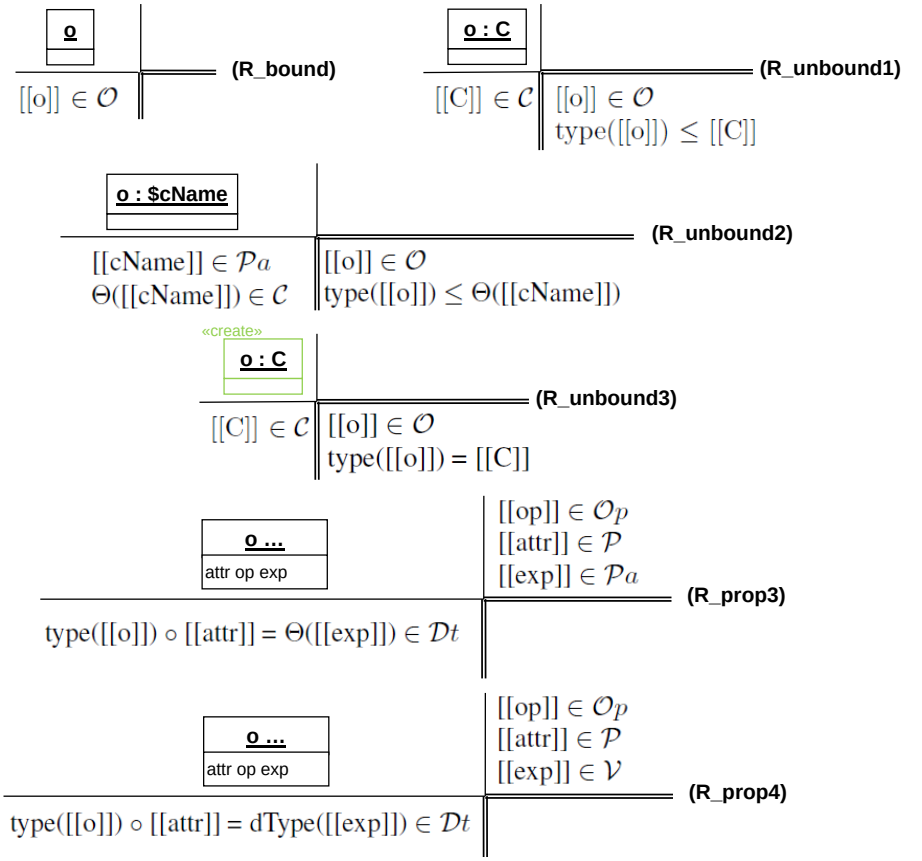


Figure 5.12: Rules Applicable on the Story Patterns 1/2

When applying `R_unbound2`, we must ensure that the type of the parameter *className* belongs to the set of classes contained in the metamodel (Θ*([[className]])* $\in \mathcal{C}$). If this type checking is correct, we can add the object *o* to the set of pattern objects by concluding *[[o]]* $in\mathcal{O}$.
If we can apply `R_unbound1` resp. `R_unbound2`, we can state that the type of the object *[[o]]* is restricted by *[[C]]* (*type([[o]])* $\leq$ *[[C]]*) resp. by the type returned by the application of Θ on [[cName]] (*type([[o]])* $\leq \Theta$*([[className]])*). Though, we cannot affirm in both cases that the type of the object *o* is exactly the specified one. This type may namely be restricted by the relationships between this objects and other objects in the story diagram or, later, by the arguments' value when the method is called and executed. For instance, in the example of Fig. 5.2, the unbound object *srcToTrg* is specified as instance of the class Connector. Though, according to the type of the objects *src* and *trg*, *srcToTrg* matches an instance of *Line* or an instance of *Transition* which are subclasses of *Connector*. This is explicitly illustrated by application examples in the appendix C.

The rule R_unbound3 looks like the rule R_unbound1, but applies when creating an object *o* of type *C*. In such a case, the type *[[C]]* is exactly assigned to the new object. This is specified by the conclusion *type([[o]]) = [[C]]*. In addition, the new object is added to the set of bound objects (*[[o]]* $\in \mathcal{O}$).

R_prop3 and R_prop4 concern the attribute assignment and/or condition. The syntactic pattern is composed of a object *o*, bound or not as represented by the ellipsis after the object's name. *attr* represents the attribute name (*[[attr]]* $\in \mathcal{P}$) and *op* is an operator (*[[op]]* $\in \mathcal{O}p$). The rule R_prop3 applies if the expression *exp* assigned to the attribute or to which the attribute is compared is a parameter (*[[exp]]* $\in \mathcal{P}a$). The rule R_prop4 applies if this expression is a value (*[[exp]]* $\in \mathcal{V}$). Two type aspects must be checked to ensure the type-safety of this pattern. On the one hand, whether the attribute *attr* really belongs to the class of which *o* is an instance. According to the rule R_prop2, we can ensure this by checking whether *type([[o]])*$\circ$*[[attr]]* can be evaluated to an element of the set of data types $\mathcal{D}t$. On the other hand, the type of the attribute and the type of the expression must be the same. Thus, *type([[o]])*$\circ$*[[attr]]* must be equal to the parameter's type as returned by the function Θ (Θ*([[exp]])*) in the case of the rule R_prop3, or to the value's type (*dType([[exp]])*) in the case of the rule R_prop4.

R_unbound1 and R_unbound2 provide type information about unbound pattern objects. Though, the link to a bound object may provide additional information if this bound object belongs to the set of parameters. This is expressed by the rules R_prop5 and R_prop6. The pattern is composed of a bound object *o1* and an unbound object *o2* whose class *C2* is directly specified in the case of R_prop5 resp. whose class is parameterized in the case of R_prop6 (*[[cName]]* $\in \mathcal{P}a$). The bound object *o1* in both pattern belongs to the set of parameter as semantically represented by *[[o1]]* $\in (\mathcal{O} \cap \mathcal{P}a)$. In addition, this rule applies if the type of the object parameter *o1* is restricted by a class *C1* (Θ*([[o1]])* $\leq$ *[[C1]]*).
The rule's pattern must be metamodel compliant. Thus the type of *o2* must be restricted by the type of the property *r2* owned by the class *C1* as expressed by *[[C2]]* $\leq$ *[[C1]]*$\circ$*[[r2]]* resp. Θ*([[cName]])* $\leq$ *[[C1]]*$\circ$*[[r2]]*. If the pattern is matched and the constraint is fulfilled, the type of *o2* can be restricted by the type of the property *r2* as owned by the type of *o1* (*type([[o2]])* $\leq$ *type([[o1]])*$\circ$*[[r2]]*).

The rule R_prop7 is more general and simply checked that the link between two object *o1* and *o2*, bound or not as indicated by the ellipsis in the syntactic pattern, is metamodel compliant. The type of *o1* resp. *o2* is restricted by the classes *C1* resp. *C2*. Then, *C2* must be a a subclass of the type of the property *r2* owned by the class *C1* (*[[C2]]* $\leq$ *[[C1]]*$\circ$*[[r2]]*).
The rule R_prop8 checks that the attribute *attr* in a pattern object *o*, bound or not as indicated by the ellipsis in the syntactic pattern, is semantically correct. In other words, this rule ensures that a property *[[attr]]* belongs to the class the object pattern is an instance of. This is expressed by *type([[o]])*$\circ$*[[attr]]* $\in \mathcal{D}t$.

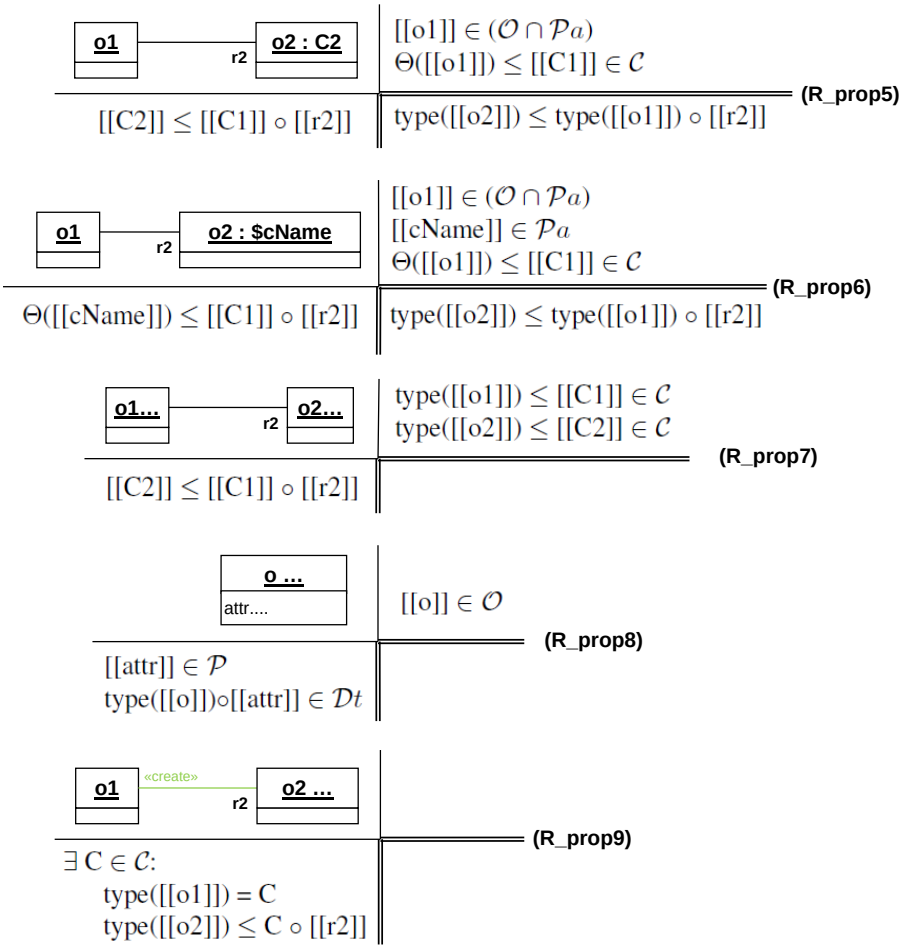


Figure 5.13: Rules Applicable on the Story Patterns 2/2

Finally, the rules `R_prop9` applies in the case of creation on a link between a bound object *o1* and a bound or unbound object *o2*. This rule looks like `R_prop5` or `R_prop6`, but is more restrictive. In the case the property *r2* is the redefinition of an association end, we must ensure that the type of property owned by the object *o1* is the redefined one. In other word, we need to know the type of *o1*, and ensure that the type of *o2* is restricted by the type of the property *r2* owned by *o1*, as expressed by the constraint:
$\exists\, C \in \mathcal{C}$:
type([[o1]]) = C
$[[C2]] \leq C \circ [[r2]]$
These constraints is more restrictive that the ones of `R_prop5` and `R_prop6`, where we only check that the type of *o2* is in the inheritance hierarchy of the property *r2* owned by *o1* ($[[C1]] \circ [[r2]]$, where $type([[o1]]) \leq [[C1]]$). Because the exact type of *o1* is generally not known until the method is called, the seman-

tic constraint of this rule of inference will produce a warning when analyzing the the story diagram, and must be checked again during the method call analysis (see algorithm description in Section 5.4.2). It will be illustrated by the examples in Sections 5.4.4 and 6.1.

5.3.4 Rules of Inference for the Method Calls

Fig. 5.14 depicts the rules which apply on method calls whose arguments must be inspected.

The syntactic pattern of the rule `R_call1` matches the i_{th} argument called *exp*. The semantic constraint ensures that the method possesses an i_{th} parameter the argument *exp* can correspond to. This rule allows for detecting an error if the number of arguments is higher than the number of parameter. For instance, if exp is the 3_{rd} argument whereas the method declaration only define 2 parameters, we can already conclude that the method call cannot be valid.

The rule `R_call2` applies when the corresponding i_{th} parameter consists of an object *[[o']]* of type *[[C']]*. The application of Θ on the i_{th} parameter returns type information, more precisely a class whose type is restricted by an element *[[C]]* from the set of classes $\mathcal{C}$. The semantic constraint ensures that the argument complies to the corresponding parameter. Thus, the argument o must be an object, as expressed by the semantic pattern *[[o]]*$\in \mathcal{O}$. The argument's type must respect the same type constraint as $\Theta([[m]]_i)$. Therefore, the type of the object o must be a subtype of the class *[[C]]* as expressed by *type([[o]])*$\leq$ *[[C]]*. Then, the object can be added to the semantic set of arguments (*[[o]]*$\in \mathcal{A}rg$). In addition, the argument is bound to the corresponding parameter by the function *arg* (*arg([[m]]$_i$) = [[o]]*).

Both `R_call3` and `R_call4` apply on a class argument C, and the i_{th} method parameter is an instance of MOFClass. In the context of `R_call4`, the type of the parameterized class is restricted by another class. In both cases, the argument C must belong to the set of classes ($C \in \mathcal{C}$). If the patterns resp. constraints of `R_call3` are matched resp. fulfilled, we can add C to the set of arguments (C $\in \mathcal{A}rg$) and connect the parameter to the argument (*arg([[m]]$_i$) = [[C]]*). In addition, we can infer that the type of C is restricted by the parameter type, which is semantically expressed by *[[C]]* $\leq \Theta$*([[m]]$_i$)*. In the case of `R_call4`, since the parameter is defined with a restriction, the argument not only belongs to the set of classes (*[[C]]*$\in \mathcal{C}$), but must also respect this restriction at runtime. Thus, the parameterized class C must be a subclass of the type returned by $\Theta([[m]]_i)$ (*[[C]]* $\leq \Theta$*([[m]]$_i$)*). Then, we can add C to the set of arguments (C $\in \mathcal{A}rg$) and connect the parameter to the argument (*arg([[m]]$_i$) = [[C]]*).

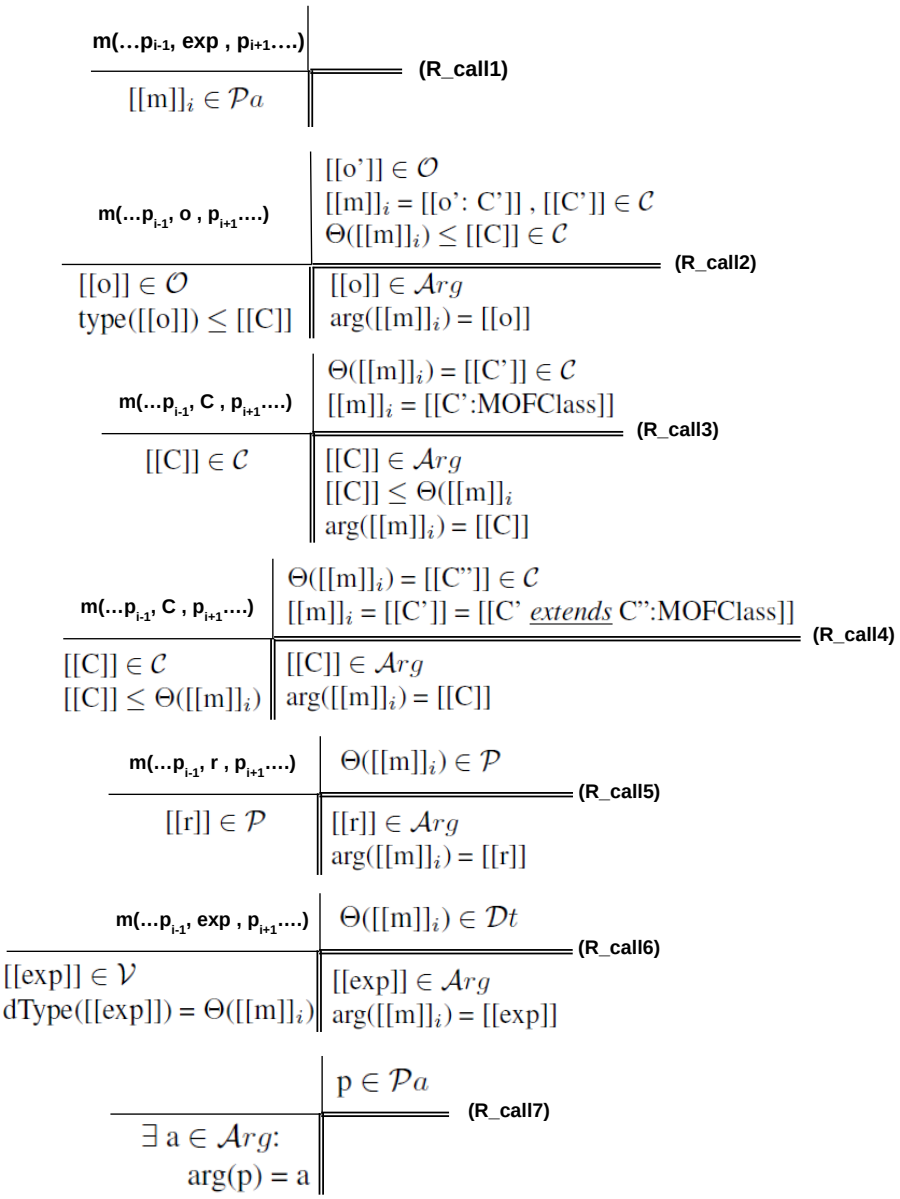


Figure 5.14: Rules applicable on the Method Calls 1/3

The rule `R_call5` applies when the corresponding i_{th} parameter is a property, i.e. that the type information extracted from the i_{th} parameter belongs to the set of properties ($\Theta([[m]]_i) \in \mathcal{P}$). The semantic constraint ensures that the object type complies to the method declaration, and thus that the argument *r* is a parameterized property ($[[r]] \in \mathcal{P}$). In this case, we can add *r* to the set of arguments ($C \in \mathcal{C}$) and connect the parameter to the argument ($arg([[m]]_i) = [[r]]$).

The rule `R_call6` applies when the corresponding i_{th} parameter is a data type, i.e. that the type information extracted from the i_{th} parameter belongs to the set of $\mathcal{D}t$. The semantic constraint ensures that the object type complies to the method declaration. Thus, the argument *exp* must be a value (*[[exp]]* $\in \mathcal{V}$). In addition, the type of the argument must comply to the corresponding parameter (*dType([[exp]])* = Θ*([[m]]$_i$)*). If theses constraints are fulfilled, we can add *exp* to the set of arguments (*[[exp]]* $\in \mathcal{A}rg$). In addition, we can connect the parameter to the argument (*arg([[m]]$_i$) = [[exp]]*).

Finally, the rule `R_call7` must ensure that the method call contains enough arguments. In other words, we must ensure that each method's parameter is connected to an argument. This is expressed by the semantic constraint $\exists\, a \in \mathcal{A}rg$*: arg(p) = a* , where *p* belongs to the set of parameter $\mathcal{P}a$

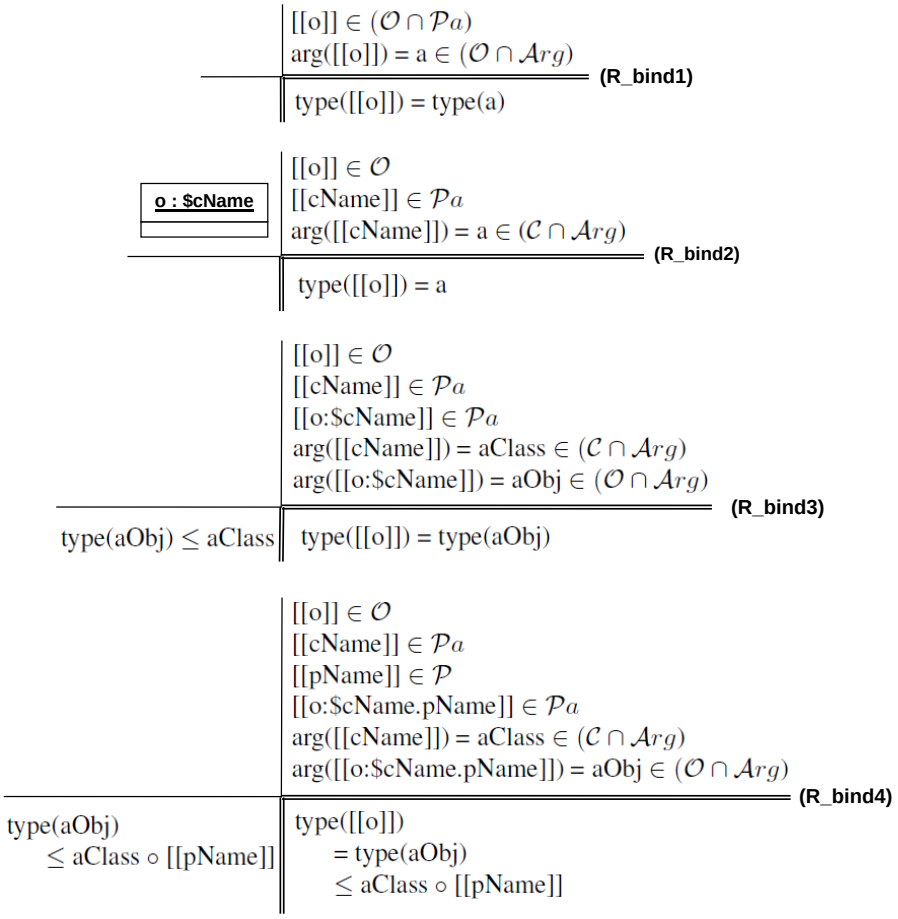


Figure 5.15: Rules applicable on the Method Calls 2/3

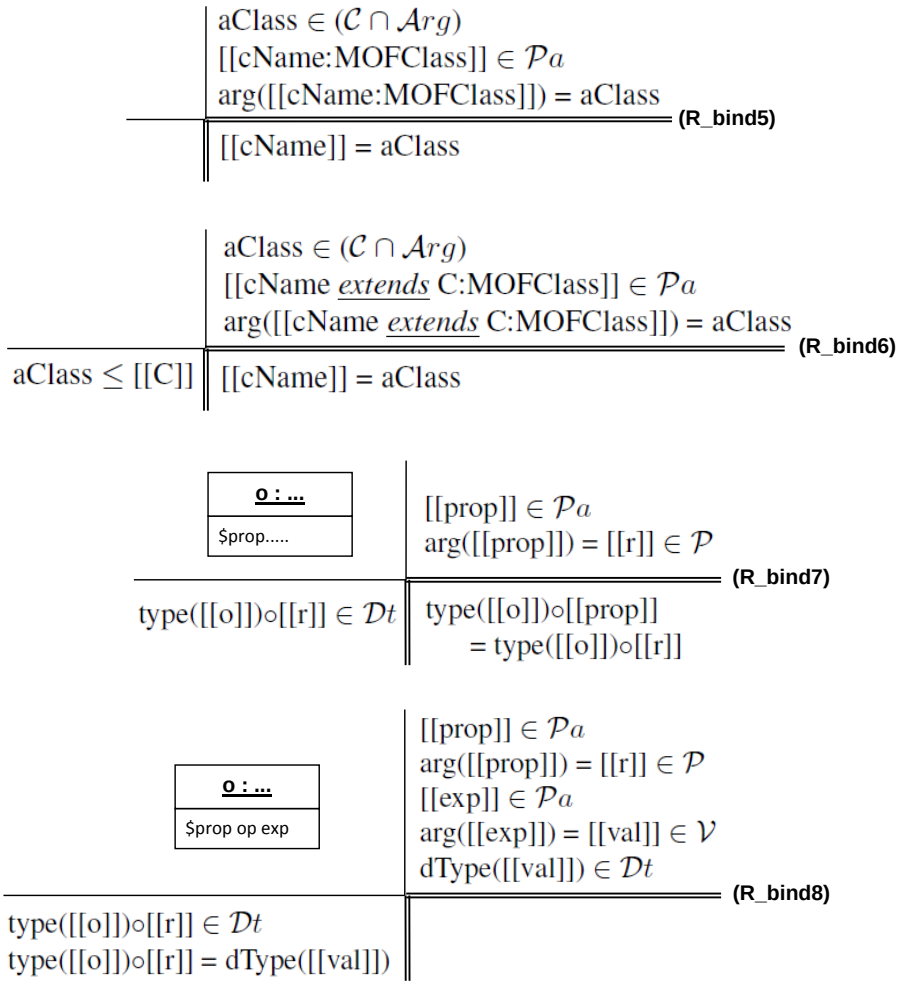


Figure 5.16: Rules applicable on the Method Calls 3/3

After having inspected the arguments by means of the rules of Fig. 5.14, it is possible to bind the argument to the pattern objects and unify their type. This is the role of the rules depicted in Fig. 5.15 and Fig. 5.16.

The rule `R_bind1` matches an object which belongs to the parameters (*[[o]]* $\in$ *($\mathcal{O} \cap \mathcal{P}a$)*). The object *a* is the argument which corresponds to the parameter *o* (*arg([[o]]) = a* $\in$ *($\mathcal{O} \cap \mathcal{A}rg$)*). Then, we can deduce that the type of the pattern object *o* is the same as the type of the argument *a* (*type([[o]]) = type(a)*).

The rule `R_bind2` applies on a pattern object whose type is parameterized (*[[o]]* $\in$ *$\mathcal{O}$, [[cName]]* $\in$ *$\mathcal{P}a$*). The class a is the argument which corresponds to the parameter cName (*arg([[cName]]) = a* $\in$ *($\mathcal{C} \cap \mathcal{A}rg$)*). Then, we can deduce that the type of the pattern object *o* is the argument *a* (*type([[o]]) = a*).

The rule R_bind3 matches an object which belongs to the parameters ([[o]] $\in$ $\mathcal{O}$, [[o:$cName]] $\in$ $\mathcal{P}a$) and whose type is parameterized (*[[cName]]* $\in$ $\mathcal{P}a$). The class *aClass* is the argument which corresponds to the parameterized type (*arg([[cName]])* = *aClass* $\in$ *(*$\mathcal{C} \cap \mathcal{A}rg$*)*). The object *aObj* is the argument corresponding to the parameterized object (*arg([[o:$cName]])* = *aObj* $\in$ *(*$\mathcal{O} \cap \mathcal{A}rg$*)*). Then, we must ensure that the type of the argument *aObj* belongs to the subclasses of the argument *aClass* so that the method call complies to the method declaration. If this condition is fulfilled, we can infer that the type of the pattern object *o* corresponds to the type of the argument *aObj*.

The following rule, R_bind4, is pretty similar to the rule R_bind3, except that the type of the object is composed of a parameterized class and a property (*[[cName]]* $\in$ $\mathcal{P}a$*, [[pName]]* $\in$ $\mathcal{P}$*, [[o:$cName.pName]]* $\in$ $\mathcal{P}a$). The class *aClass* is the argument which corresponds to the parameterized type (*arg([[cName]])* = *aClass* $\in$ *(*$\mathcal{C} \cap \mathcal{A}rg$*)*). The object *aObj* is the argument corresponding to the parameterized object (*arg([[o:$cName.pName]])* = *aObj* $\in$ *(*$\mathcal{O} \cap \mathcal{A}rg$*)*). Then, we must ensure that the method call complies to the method declaration. That means that the type of the argument *aObj* must be restricted by the type of the property *pName* as owned by the class *aClass* (*type(aObj)* $\leq$ *aClass* $\circ$ *[[pName]]*). If this condition is fulfilled, we can infer that the type of the pattern object *o* corresponds to the type of the argument *aObj* and belong to the subclasses of *aClass* $\circ$ *[[pName]]*.

The rule R_bind5 applies on a class named aClass which is used as argument (*aClass* $\in$ *(*$\mathcal{C} \cap \mathcal{A}rg$*)*. This argument is bound to a parameterized class (*[[cName: MOFCLass]]*$\in$ $\mathcal{P}a$) by the function *arg*. Then, we can connect the parameter with the class *aClass*. This is expressed by the conclusion *[[cName]]* = *aClass*
The next rule, R_bind6, is similar to the rule R_bind5. The only difference is that the parameter is a parameterized class whose type is restricted by another class (*[[cName* <u>*extends*</u> *C:MOFCLass]]*$\in$ $\mathcal{P}a$). Therefore we add a semantic constraint to ensure that the argument respects this condition (*aClass* $\leq$ *[[C]]*). If the constraint is fulfilled, we can connect the parameter with the class *aClass* similarly to the conclusion of the rule R_bind5.

Both last rules, R_bind7 and R_bind8, match a pattern object with an attribute checking/assignment.
The semantic pattern of R_bind7 is composed of the parameterized property *prop* and the corresponding argument *r* (*arg([[prop]])* = *[[r]]* $\in$ $\mathcal{P}$). The semantic constraint ensures that the property *r* given as argument of the method call belongs to the pattern object's class (*type([[o]])*$\circ$*[[r]]* $\in$ $\mathcal{D}t$). If this constraint is fulfilled, we can bind the parameterized attribute in the story pattern with the corresponding argument (*type([[o]])*$\circ$*[[prop]]* = *type([[o]])*$\circ$*[[r]]*).
The semantic pattern of R_bind8 is composed of the parameterized property *prop* and the corresponding argument *r* (*arg([[prop]])* = *[[r]]* $\in$ $\mathcal{P}$). The attribute's

value is defined in the method specification as parameter *exp* whose corresponding argument is *val* (*arg([[exp]]) = [[val]]* $\in \mathcal{V}$, *dType([[val]])* $\mathcal{D}t$). The semantic constraint ensures that the property *r* given as argument of the method call belongs to the pattern object's class (*type([[o]])*$\circ$*[[r]]* $\in \mathcal{D}t$), and that the value given as argument correspond to the attribute (*type([[o]])*$\circ$*[[r]] = dType([[val]])*).

5.4 Type Checking and Error Detection

Inference engines are programs which try to derive answers from a knowledge base. They are considered to be a special case of reasoning engines, which can use more general methods of reasoning like natural deduction. Because our approach is based on deductive systems and rules of inference, our application algorithm is inspired by the mechanisms of these engines.
In this section, we first present the inference engines before proposing our type checking algorithm. Finally, we illustrate our approach by an application example.

5.4.1 Inference Engines

Expert systems are computer systems that emulate the decision-making ability of a human expert. They generally consist of three parts: a knowledge base, an inference engine, and a user interface. The inference engines, which are the "brain" of the expert systems, can be described as a form of finite state machine with a cycle that entails three action states:

1. **Matching rules:**
   The inference engine finds all rules that are satisfied by the current contents of the knowledge base. The set of selected rules is called the *conflict set* or *matching set*. A pair composed of a rule and a subset of matching data items is called an *instantiation* of the rule. The same rule may appear in several instantiations if its premises match different subsets of the knowledge base.

2. **Choosing rules:**
   If the matching set contains several elements, the inference engine applies some selection strategy to determine which rules will actually be executed. There are many different conflict resolution strategies, such as Refraction (= select an instantiation only once), *First-in-First-Serve* (= the first rule that is matched), *Last-in-First-Serve* (= the last rule that is matched), *Prioritization* (= based on priorities set on rules), *Specificity* (= the most specific rule, or the rule that matches the most facts), *Recency* (= the rule that matches the most recently derived facts) [LvdG91] [J.94].

3. **Executing rules:**
   The inference engine applies the selected rules on the instantiation's second entry, i.e. the matching data items, as parameters.

The inference engine then cycles back to the first state and is ready to start over again. This control mechanism is referred to as the *recognize-act cycle*. The process terminates either after a given number of cycles, when the problem is solved or if no rule can be applied anymore.
The step of "Choosing rules" is the most time-consuming task of the recognize-act cycle. In many applications, when performance time considerations are critical or in the case of large amount of data and rules, this process becomes really complex. Then, it requires the use of efficient algorithms, for instance the Rete algorithm [For82] or its improved version called TREAT [Mir87].

There are two main methods of reasoning: forward-chaining and backward-chaining. Each one corresponds to a direction of search. Forward-chaining refers to data-driven search, whereas backward-chaining refers to goal-driven search.
Forward-chaining starts with the available data and uses inference rules to extract more data until a goal is reached. The inference engine searches the inference rules until it finds one whose premises are known to be true. When found it can infer the consequent, resulting in the addition of new information to the knowledge base.
On the contrary, backward-chaining starts with a list of goals and works backwards from the consequent to the antecedent to see if there is data available that will support any of these consequents. The inference engine searches the inference rules until it finds one whose conclusion matches a desired goal. If the premises of that rule are not known to be true, it is added to the list of goals. Else, the goal is confirmed as reachable. Inference engines with backward-chaining usually employ depth-first search strategy.
In most cases, forward and backward chaining can solve the same problems. Nevertheless, one approach will be preferred to the other one depending on the situation [Hin11]. For instance, backward-chaining is well suited if you already know what you are looking for whereas forward-chaining does not necessarily requires to know the final state of the solution. One of the advantages of forward-chaining is that the reception of new data can trigger new inferences. Thus, an inference engine with forward chaining is better suited to dynamic situations in which conditions are likely to change.

5.4.2 Rule Application Algorithm

Because our approach is based on deductive systems and rules of inference, our application algorithm is inspired by the mechanism of these engines. Here, the knowledge base is composed of the metamodel, the model transformations, the method calls, and the derived semantic informations. A naive rule application algorithm would consist of checking each rule against the known facts in the knowledge base, firing that rule if necessary, then moving on to the next rule (and looping back to the first rule when finished). Nevertheless, such an approach performs far too slowly, even for small knowledge bases.
We refine this application algorithm for a more rigorous and efficient execution.

When considering the rules, it appears clearly that the application of some rules requires some semantic information as premises which can only be obtained after the application of other rules. For instance, the rules which apply on the story diagrams require information from the metamodel, information which are available only after the application of the rules as described in Section 5.3.1. Thus, we define a rule classification to determine a more logical and efficient application strategy.
First, we classify the rules according to the elements on which they apply: the metamodel, the specification (signature and story pattern) and the calls of methods. Then, priorities are defined inside these sets to refine the order in which the rules are inspected. Fig. 5.17 summarizes the definition of these sets of rules and priorities assignment.

Fig. 5.19 illustrates the inference algorithm of our approach. The knowledge base consists of the syntactic knowledge base (i.e. the metamodel, the specification and the call of methods) and the semantic knowledge base (i.e. the semantic information obtained after the application of the rules of inference). Our algorithm consists of a rule application cycle which contains a recognize-act cycle performing successively on the metamodel, on each method specification, and on each method call. We will describe our algorithm later in this chapter.
For the sake of efficiency, each cycle does not apply on the complete knowledge base and does not require the complete set of rules, but only on a restricted syntactic knowledge base and a restricted set of rules. For instance, the analysis of a method call only needs:

- the syntactic information for this method and this method call, not for another method or another method call.
- the information extracted from the analysis of the specification for this method and not for another method.
- the set of rules restricted to the subset dedicated to the method call analysis.

Restricted Knowledge Bases and Restricted Sets of Rules

Let us describe each restricted syntactic knowledge base and the corresponding restricted set of rules as depicted by Fig. 5.17.

First, we collect information from the metamodel since all graph transformations are based on it. This concerns the following rules: `R_sem1`, `R_sem2`, `R_sem3`, `R_prop1`, `R_prop2`, `R_inherit1`, `R_inherit2`, `R_inherit3`, as well as `R_redefineProp`. By means of `R_sem1`, `R_sem2`, and `R_sem3`, syntactic elements (classes, properties and data types) are mapped to their semantical counterparts. The semantics of the property relationships are derived by application of the rules `R_prop1` and `R_prop2`. Then, the application of `R_inherit1` allows for establishing the inheritance hierarchy between classes, whereas `R_inherit2` and

R_inherit3 concern the inherited properties. Finally, the redefinition of properties are translated into the semantic domain by application of R_redefineProp.

Then, the model transformations' specification is examined, starting by the method signature followed by the diagram itself. The set of rules applicable on a method signature deals with two aspects. On the one hand, the rules R_sem4, R_sem5, R_sem6, R_sem7, R_sem8, and R_sem9 translate the syntactical parameters into their semantical counterparts. On the other hand, the rules R_theta1, R_theta2, R_theta3, R_theta4, and R_theta5 extract type information from the parameters. The set of rules applicable on the story diagram deals with four groups. First, the rule R_bound applies on bound objects. Afterwards, the unbound objects are analyzed by application of the rules R_unbound1 ("normal" objects), R_unbound2 (generic objects) and R_unbound3 (object creation). The rules add the unbound objects to the set of pattern objects and determine a type restriction. The rules R_prop8, R_prop3 and R_prop4 ensure that the properties (attributes) are semantically correct, i.e. that they belong to the objects' classes. Then, the rules R_prop5, R_prop6, R_prop7 and R_prop9 check that the links complies to the corresponding associations in the metamodel.

After having collected metamodel information and ensured the correctness of the graph transformation specifications, the method calls and their execution can be analyzed. A method call is analyzed by application of the rules R_call1, R_call2, R_call3, R_call4, R_call5, R_call6 andR_call7. These rules binds a parameter to the corresponding argument, and ensure that this argument complies to the method's signature. The analysis of the signature provide type information on the pattern objects, but the actual type value is fixed at the execution time, i.e. with the method call. The application of the rules R_bind1, R_bind2, R_bind3, R_bind4, R_bind5, R_bind6, R_bind7, R_bind8 and R_bind9 allows for determining the type of this object by assigning the corresponding argument's value.

The rules, which apply only on semantic patterns or on textual expressions, and hence are not specific to a syntactic domain, can be used as "auxiliary" rules for all the restricted set of rules. These rules are R_reflexive, R_transitive, R_antisymmetric, R_compos, R_substitute1, R_substitute2, R_generic, R_inherit2, R_inherit3, R_inheritSingle.

The semantic knowledge base is restricted as follows. When analyzing the metamodel, the semantic knowledge base first is empty. The analysis of a method signature requires the semantic knowledge derived from the metamodel analysis, and the analysis of the story diagram needs the information from the metamodel and from the signature. Finally, each method call can be inspected by means of the information derived from the analysis of the metamodel and of the corresponding method signature and story diagram.

	Set number	priority	Rule name
Metamodel	1 (class diagram)	1	R_sem1 R_sem2 R_sem3
		2	R_prop1 R_prop2
		3	R_inherit1 R_inherit2 R_inherit3
		4	R_redefineProp
Method specification analysis	2 (method signature)	1	R_sem4
		2	R_sem5 R_sem6 R_sem7 R_sem8 R_sem9
		3	R_theta1 R_theta2 R_theta3 R_theta4 R_theta5
	3 (story pattern)	1	R_bound
		2	R_unbound1 R_unbound2 R_unbound3
		3	R_prop8 R_prop3 R_prop4
		4	R_prop5 R_prop6 R_prop7 R_prop9
Method call analysis	4 (check argument)	1	R_call1 R_call2 R_call3 R_call4 R_call5 R_call6 R_call7
	5 (bind type)	1	R_bind1 R_bind2 R_bind3 R_bind4 R_bind5 R_bind6 R_bind7 R_bind8

Figure 5.17: Rule Classification

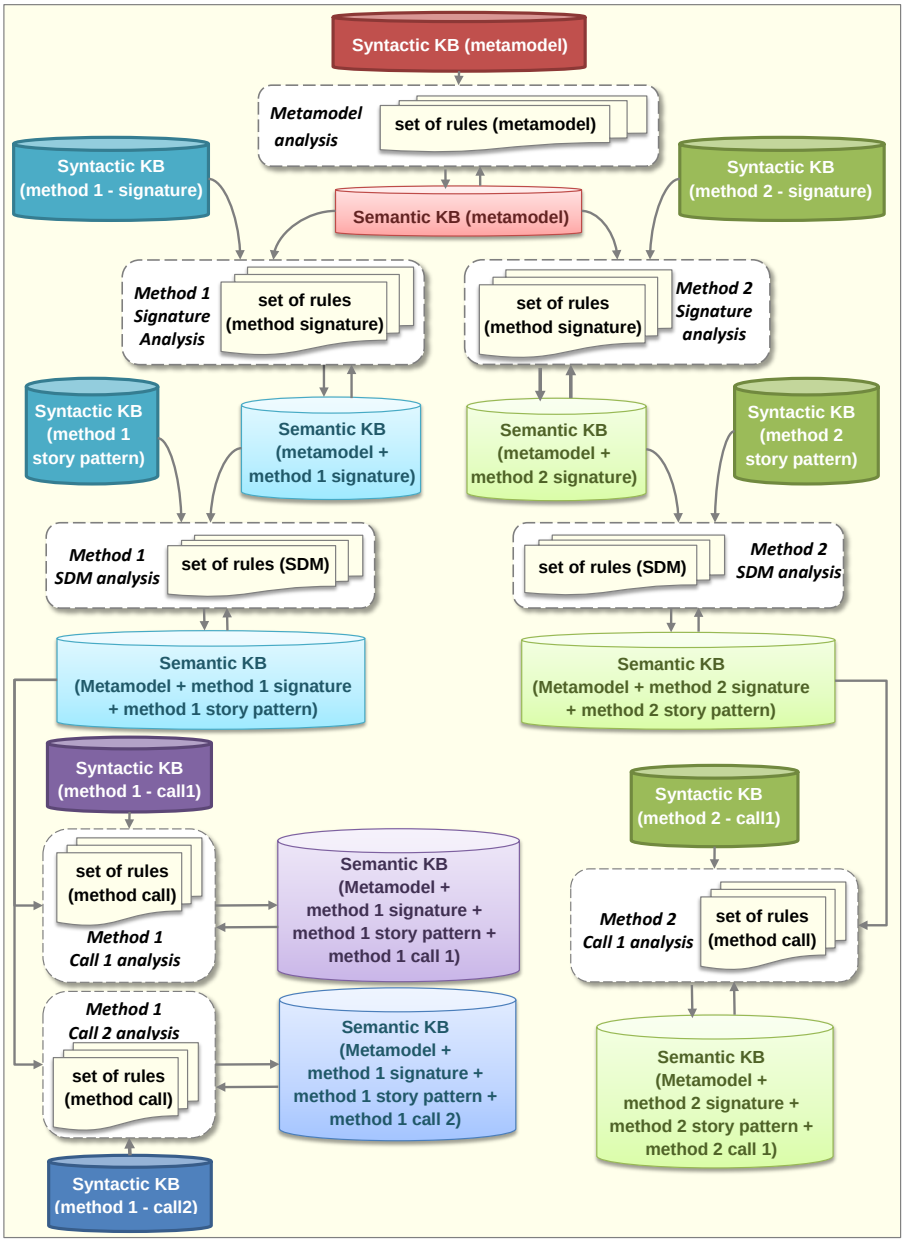


Figure 5.18: Execution example

Resume:

1. **Syntactic knowledge base:** restricted to the metamodel
   Semantic knowledge base: empty
   Rules: R_sem1, R_sem2, R_sem3, R_prop1, R_prop2, R_inherit1, R_inherit2, R_inherit3, R_redefineProp, R_reflexive, R_transitive, R_compos,R_generic, R_inheritSingle, R_antisymmetric, R_substitute1, R_substitute2.

2. **Syntactic knowledge base:** restricted to the method signature
   Semantic knowledge base: semantic facts derived from the analysis of the metamodel.
   Rules: R_sem4, R_sem5, R_sem6, R_sem7, R_sem8, R_sem9, R_theta1, R_theta2, R_theta3, R_theta4, R_theta5, R_reflexive, R_transitive, R_compos, R_generic, R_inherit2, R_inherit3, R_inheritSingle, R_antisymmetric, R_substitute1, R_substitute2.

3. **Syntactic knowledge base:** restricted to the method specification (story diagram).
   Semantic knowledge base: semantic facts derived from the analysis of the metamodel and the corresponding method signature.
   Rules: R_bound, R_unbound1, R_unbound2, R_unbound3, R_prop5, R_prop6, R_prop7, R_prop8, R_prop9, R_prop3, R_prop4, R_transitive, R_generic, R_inherit2, R_inherit3, R_reflexive, R_compos,R_inheritSingle, R_antisymmetric, R_substitute1, R_substitute2.

4. **Syntactic knowledge base:** restricted to the method call:
   Semantic knowledge base: semantic facts derived from the analysis of the metamodel and of the corresponding method signature and story diagram.
   Rules: R_call1, R_call2, R_call3, R_call4, R_call5, R_call6, R_call7,R_bind1, R_bind2, R_bind3, R_bind4, R_bind5, R_bind6, R_bind7, R_bind8, R_bind9, R_reflexive, R_transitive, R_generic, R_inherit2, R_inherit3, R_antisymmetric, R_compos, R_inheritSingle, R_substitute1, R_substitute2.

Fig. 5.18 illustrates how the rules and how the syntactic and semantic know-ledge bases (KB) are used when analyzing a metamodel, two method specifications and their calls (two calls of the first method and one call of the second method). The semantic knowledge base can be considered as an artifact for each analysis step. Because information derived by a rule can be an input for another rule, the semantic knowledge base evolves during the analysis process. The semantic knowledge base obtained after the metamodel analysis (*Semantic KB (metamodel)*) is used for the analysis of any method specified on this metamodel. Then, after each method specification's analysis, we obtain a different semantic knowledge base composed

of semantic information from the metamodel and from the given method specification (*Semantic KB (metamodel + method 1 signature + method 1 story pattern)* and *Semantic KB (metamodel + method 2 signature + method 2 story pattern)*). In the same way, the artifact obtained after the analysis of a method specification is used as knowledge base for the analysis of each call of this method. Here again, we obtain different knowledge bases from every method call analysis: for the *method 1*, a semantic knowledge base for the *method 1 call 1* and another for the *method 1 call 2*, and for the method 2 another semantic knowledge base for the *method 2 call 1*. The figure Fig. 5.18 also illustrates the use of restricted sets of rules and restricted syntactic knowledge bases for every analysis step.

Application Algorithm

Whereas Fig. 5.18 illustrates the rules' application in a particular case, Fig. 5.19 describes the application algorithm for our type checking approach by means of an activity diagram. The upper part of Fig. 5.19 shows the complete syntactic and semantic knowledge bases as well as the complete set of rules. Though, as explained above, we only need a part of these informations depending on the kind of elements we inspect: metamodel, story diagram, method call. The lower part of Fig. 5.19 shows the application algorithm when inspecting the metamodel, or a story diagram, or a method call. As depicted in this figure, the start point consists of a restricted syntactic knowledge base and a restricted set of rules.

The first and main part of the application rule cycle consists of a recognize-act cycle. Based on the semantic knowledge base, the restricted syntactic knowledge base and the restricted set of rules, pattern matching allows for determining a set of rule instantiations (*matching set*). Each rule instantiation is a pair consisting of a rule and a model's subgraph and/or semantic information which match the rule's premises.
Then, a rule instantiation must be selected. This selection is based on priority and refraction, and on the checking of the rule constraints. According to the concept of refraction, a rule instantiation can be selected only once. It avoids to try to apply unnecessarily the same rule on the same pattern several times. The priorities which contribute to the rule selection are the ones depicted in Fig. 5.17. Finally, only a rule instantiation whose constraint is fulfilled, can be selected.

If a rule can be chosen, it is fired from the set of rules' instantiations. The rule's constraint is added to the constraint database and labeled as checked and fulfilled. The semantic information resulting of the rule's application is then added to the semantic knowledge base, and the recognize-act cycle loops back. The algorithm takes account of the additional semantic information, and if this information allows for matching other rules' premises, the set of rules' instantiations is completed.
Because the semantic knowledge bases, the syntactic knowledge base and the set of rules contain a finite number of elements, the matching set is finite too. As a consequence and because the selection of the rule instantiation is based on the con-

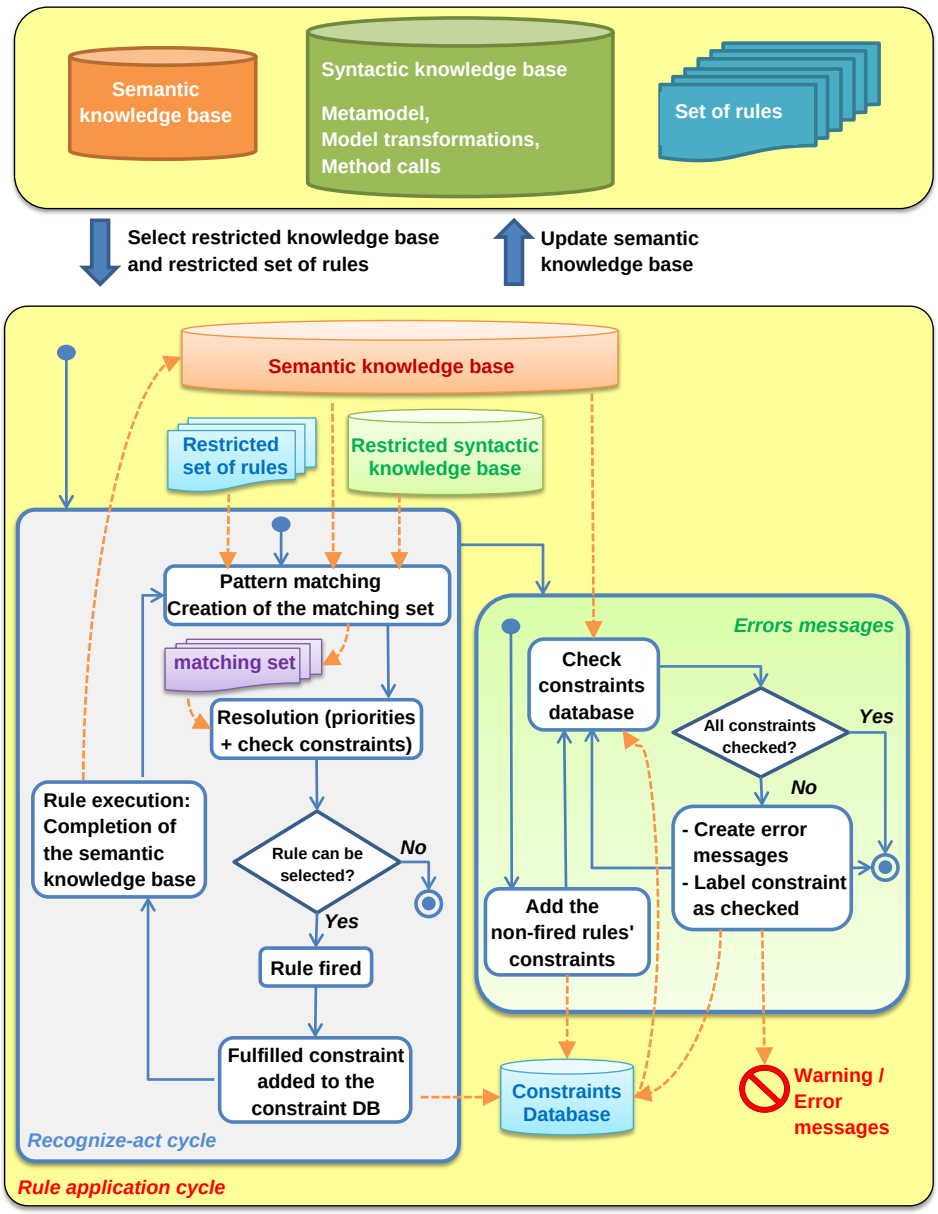


Figure 5.19: Rule Application Algorithm

cept of refraction (each rule instantiations is selected only once), it comes to the point where no rule instantiation can be chosen and the cycle terminates.

The recognize-act cycle terminates when the matching set is empty (all rule instantiations could be chosen and applied) or contains only rule instantiations whose semantic constraint cannot be fulfilled.
If the matching set is not empty, this means that it contains only rule instantiations whose syntactic pattern match a model's subgraph and/or whose semantic premises are satisfied by the semantic knowledge base, but whose semantic constraint cannot be fulfilled. Since all matching rules whose semantic constraint is fulfilled have be executed, no other semantic information can be derived from the remaining and the semantic knowledge base is complete when leaving the recognize-act cycle. It ensures that the remaining semantic constraints cannot be satisfied by the information contained in the semantic knowledge base. These semantic constraints are added to the constraint database and labeled as *non-checked* and *non-fulfilled*.

We then inspect the constraint data base with help of the semantic knowledge base. The constraints which can be fulfilled are labeled as *checked* and *fulfilled*. The constraints labeled as *checked* (i.e. from a previous analysis) and *non-fulfilled* are checked again by means of the semantic knowledge base. In the case the semantic knowledge base contains enough information, these constraints can be labeled as *checked* and *fulfilled*. As long as the constraint database contains constraint which are labeled as *non-checked*, we create for each constraint a warning and label the corresponding constraint in the constraint database as *checked*. When all constraints in the constraint database are labeled as *checked*, the algorithm terminates, and we can start again with the next syntactic knowledge base.
Please note that some constraints may be unfulfilled after having inspected the story diagram, but may be fulfilled after having inspecting the method call, i.e. when the value of elements have been replaced by the corresponding arguments' value. Whether the constraints can be fulfilled or not depends of course on the arguments' value in a given method call. That is why we will first only create a warning message. In the case the constraint database still contains constraints labeled as *checked* and *non-fulfilled* after having inspected a method call, we can then create the corresponding error messages.

5.4.3 Error Detection

The semantic constraints as defined in the set of rules which is applied on metamodels allows for detecting metamodeling errors such as incorrectly specified association redefinitions (rule `R_redefineProp`) or multiple inheritance in a context of single inheritance (rule `R_inheritSingle`). Similarly, the semantic constraints in the rules of inference applied to the story diagrams ensure for instance that a pattern object depicted as bound really belongs to the set of bound objects (rule `R_bound`).

The semantic constraints also aim at ensuring that a method specification is well-defined. For instance, a pattern object in a story diagram must match the instance of a metamodel's class. Semantically, it means that the type of a pattern object must belong to the set of class $\mathcal{C}$ (e.g. in `R_unbound1`). In the same way, the link between two pattern objects must correspond to the instance of an association from the metamodel (e.g. in `R_prop5`). In the case of attribute checking/assignment, the specified attribute of a pattern object must comply to the metamodel: the attribute must belong to a class corresponding to the pattern object's type, and the attribute's type as defined in the metamodel must correspond to the value's type in the story diagram (rule `R_prop4`).
Similarly, we can check that a method call respects the corresponding method signature by means of the rules' semantic constraints. These constraints ensure that an argument corresponds to a parameter (e.g in `R_call1`) or that an argument's type is conform to the corresponding parameter's type (e.g in `R_call2`).

The semantic knowledge base obtained by application of the rules of inference allows for defining type and relationship constraints between the elements (link, pattern objects, etc.). Depending on the argument values of the method calls, errors may occur or not. By means of the rules for the method calls (cf. Fig. 5.15 and Fig. 5.16) , the pattern objects' type can be determined. Then, rules such as `R_inheritSingle` allow for detecting errors caused by argument values.
For instance, if we consider the example of Fig. 5.2 with the method call *createConnector(outPort, vertex)* (*outPort* instance of the class *Outport* and *vertex* instance of the class *Vertex*). Assigning the type *Outport* to the pattern object *src* and the type *Vertex* to the pattern object *trg* results in a type conflict for the pattern object *srcToTrg* whose type should be a subclass of *Line* as well as *Transition*. Because there is no inheritance relationship between *Line* and *Transition*, the error can be detected by means of the rule `R_inheritSingle`. The rule application and error detection are illustrated by the examples in the next sections as well as in the appendix C.

5.4.4 Application

We will now illustrate our approach by application on an example. Because we concentrate in this chapter on type checking, we will consider the non-generic model transformation *createConnector(srcToTrg : Connector, src : ConnectableElement , trg : ConnectableElement)* depicted by Fig. 5.21. An application combining SDM extensions and type checking will be presented in the next chapter.
As method call, we will inspect *createConnector(ln, srcOutport, trgVertex)*, with *ln* instance of the class *Line*, *srcOutport* instance of the class *Outport* and *trgVertex* instance of the class *Vertex*. This method call tries to connect an *Outport* and a *Vertex* with a *Line*. This is incorrect because a *Line* can only connect a *Outport* with an *Inport*, not a *Vertex*. The type checking executed in this section will show how the error can be detected.

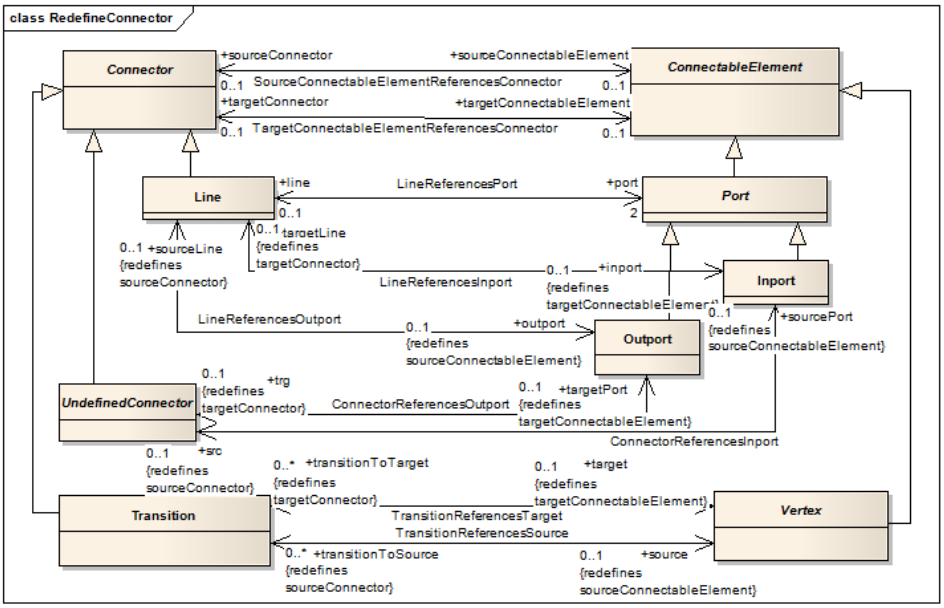


Figure 5.20: Metamodel Excerpt

1. Step: On the Metamodel

First, we have to examine and extract information from the metamodel. Here, the metamodel excerpt in Fig. 5.20 is sufficient and is the syntactic knowledge base in this step.

By applying the rules `R_sem1` and `R_sem3`, we can add to the semantic knowledge base a set $\mathcal{C}$ composed of the metamodel's classes and a set $\mathcal{P}$ composed of the metamodel's properties.
$\mathcal{C}$: { *[[Connector]], [[UndefinedConnector]], [[ConnectableElement]], [[Line]], [[Port]], [[Inport]], [[Outport]], [[Transition]], [[Vertex]]* }
$\mathcal{P}$: { *[[sourceConnector]], [[targetConnector]], [[sourceConnectable]], [[targetConnectableElement]], [[line]], [[port]], [[outport]], [[inport]], [[sourceLine]], [[targetLine]], [[transitionToTarget]], [[transitionToSource]], [[target]], [[source]], [[trg]], [[src]], [[sourcePort]], [[targetPort]]* }

Then, the application of the rules `R_inherit1` and `R_inherit2` allows for completing the semantic knowledge base with subtype relationships ($\leq$), more precisely inheritance relationships between classes and inherited properties.

[[UndefinedConnector]] $\leq$ *[[Connector]]*
[[Line]] $\leq$ *[[Connector]]*
[[Transition]] $\leq$ *[[Connector]]*
[[Vertex]] $\leq$ *[[ConnectableElement]]*

$[[Outport]] \leq [[Port]] \leq [[ConnectableElement]]$
$[[Inport]] \leq [[Port]] \leq [[ConnectableElement]]$
$[[Line]] \circ [[sourceConnectableElement]] \leq [[ConnectableElement]]$
$[[Line]] \circ [[targetConnectableElement]] \leq [[ConnectableElement]]$
$[[Port]] \circ [[sourceConnector]] \leq [[Connector]]$
$[[Port]] \circ [[targetConnector]] \leq [[Connector]]$
$[[Outport]] \circ [[sourceConnector]] \leq [[Connector]]$
$[[Inport]] \circ [[targetConnector]] \leq [[Connector]]$
$[[UndefinedConnector]] \circ [[sourceConnectableElement]] \leq [[ConnectableElement]]$
$[[UndefinedConnector]] \circ [[targetConnectableElement]] \leq [[ConnectableElement]]$
$[[Transition]] \circ [[sourceConnectableElement]] \leq [[ConnectableElement]]$
$[[Transition]] \circ [[targetConnectableElement]] \leq [[ConnectableElement]]$
$[[Vertex]] \circ [[sourceConnector]] \leq [[Connector]]$
$[[Vertex]] \circ [[targetConnector]] \leq [[Connector]]$

Finally, the rules `R_prop1` and `R_redefineProp` provide property relationships to the semantic knowledge base.

$[[Connector]] \circ [[sourceConnectableElement]] = [[ConnectableElement]]$
$[[Connector]] \circ [[targetConnectableElement]] = [[ConnectableElement]]$
$[[ConnectableElement]] \circ [[sourceConnector]] = [[Connector]]$
$[[ConnectableElement]] \circ [[targetConnector]] = [[Connector]]$
$[[Line]] \circ [[port]] = [[Port]]$
$[[Port]] \circ [[line]] = [[Line]]$
$[[Line]] \circ [[sourceConnectableElement]] = [[Line]] \circ [[outport]] = [[Outport]]$
$[[Line]] \circ [[targetConnectableElement]] = [[Line]] \circ [[inport]] = [[Inport]]$
$[[Outport]] \circ [[sourceConnector]] = [[Outport]] \circ [[sourceLine]] = [[Line]]$
$[[Outport]] \circ [[targetConnector]] = [[Outport]] \circ [[trg]] = [[UndefinedConnector]]$
$[[Inport]] \circ [[sourceConnector]] = [[Inport]] \circ [[src]] = [[UndefinedConnector]]$
$[[Inport]] \circ [[targetConnector]] = [[Inport]] \circ [[targetLine]] = [[Line]]$
$[[UndefinedConnector]] \circ [[sourceConnectableElement]]$
$= [[UndefinedConnector]] \circ [[sourcePort]] = [[Inport]]$
$[[UndefinedConnector]] \circ [[targetConnectableElement]]$
$= [[UndefinedConnector]] \circ [[targetPort]] = [[Outport]]$
$[[Transition]] \circ [[targetConnectableElement]]$
$= [[Transition]] \circ [[target]] = [[Vertex]]$
$[[Transition]] \circ [[sourceConnectableElement]]$
$= [[Transition]] \circ [[source]] = [[Vertex]]$
$[[Vertex]] \circ [[targetConnector]]$
$= [[Vertex]] \circ [[transitionToTarget]] = [[Transition]]$
$[[Vertex]] \circ [[sourceConnector]]$
$= [[Vertex]] \circ [[transitionToSource]] = [[Transition]]$

2. Step: On the Method Signature

Once the metamodel excerpt has been analyzed, we can inspect the method declaration (Fig. 5.21) starting by the method signature:
Analyzer :: createConnector(srcToTrg: Connector, src : ConnectableElement, trg : ConnectableElement) : Void
This signature represents the syntactic knowledge base for the signature analysis. The semantic knowledge base is the one obtained after the metamodel analysis.

By means of the rules `R_sem4` and `R_sem5`, we can complete the semantic knowledge base with a set $\mathcal{P}a$ composed of the parameters:
$\mathcal{P}a$: { *[[createConnector]]*$_1$, *[[createConnector]]*$_2$, *[[createConnector]]*$_3$ }
= { *[[srcToTrg:Connector]], [[src:ConnectableElement]], [[trg:ConnectableElement]]* }
= { *[[srcToTrg]], [[src]], [[trg]]* }

We can then create a set of bound pattern objects $\mathcal{O}$ containing the objects *[[srcToTrg]], [[src]]* and *[[trg]]*.
$\mathcal{O}$: { *[[srcToTrg]], [[src]], [[trg]]* }

In addition, we can extract type information by application of the Θ-operator on the parameters (rule `R_theta1`):
type([[srcToTrg]]) $\leq$ *[[Connector]]* = Θ*([[srcToTrg]])*
type([[src]]) $\leq$ *[[ConnectableElement]]* = Θ*([[src]])*
type([[trg]]) $\leq$ *[[ConnectableElement]]* = Θ*([[trg]])*

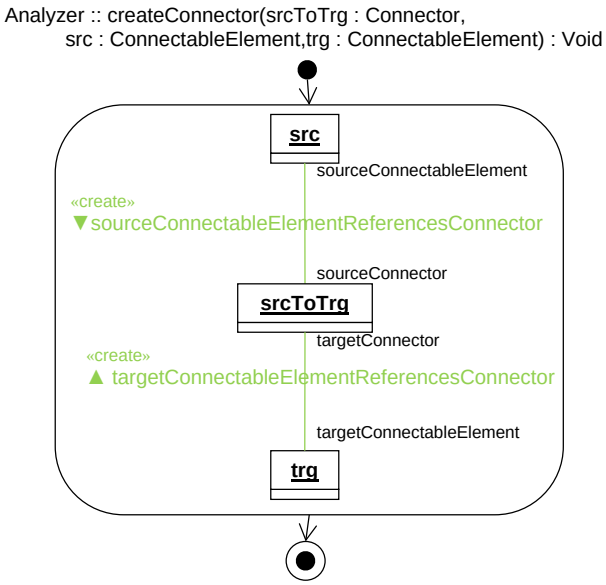


Figure 5.21: Method *createConnector*

3. Step: On the Story Diagram

After having checked and extracted information from the signature, we can analyze the story diagram (Fig. 5.21) which is the syntactic knowledge base for this new analysis step. The semantic knowledge base consists of the semantic information extracted from the metamodel and from the signature.

Since the story pattern contains the pattern objects *src*, *srcToTrg* and *trg* which are depicted as bound objects, we apply the rule `R_bound`. The rule's semantic constraint is fulfilled since the semantic knowledge base contains as information that the objects' semantic counterparts *[[src]]*, *[[srcToTrg]]* and *[[trg]]* belong to the set of bound pattern objects $\mathcal{O}$.

The bound pattern objects *src* and *srcToTrg* are connected together with the properties *sourceConnectableElement* resp. *sourceConnector*. In addition, both parameter types are restricted by a class (Θ*([[src]]) = [[ConnectableElement]]* and Θ*([[srcToTrg]]) = [[Connector]]*). Thus,the rule `R_prop7` matches in both direction, from *src* to *srcToTrg* and from *srcToTrg* to *src*. We then check whether the rule's semantic constraint is fulfilled for both rule's instanciations. The class *ConnectableElement* possesses the property *sourceConnector* which is of type *Connector*, and the class *Connector* possesses the property *sourceConnectableElement* which is of type *ConnectableElement*. Thanks to the reflexive property of the $\leq$-operator, the semantic constraint is fulfilled for both rule's instanciations:
$[[ConnectableElement]] \leq [[Connector]] \circ [[sourceConnectableElement]] : true$
$[[Connector]] \leq [[ConnectableElement]] \circ [[sourceConnector]] : true$.

Similarly, the same rule matches the pattern composed of the objects *trg* and *srcToTrg*, and its semantic constraints are fulfilled:
$[[ConnectableElement]] \leq [[Connector]] \circ [[targetConnectableElement]] : true$
$[[Connector]] \leq [[ConnectableElement]] \circ [[targetConnector]] : true$.

Finally, because the links are created between the bound objects *src* and *srcToTrg*, and between the bound objects *trg* and *srcToTrg*, the rule `R_prop9` is considered. Though, the constraints cannot be fulfilled yet. Let us consider the semantic constraint:

$\exists\, C \in \mathcal{C}$:
$type([[src]]) = C$
$type([[srcToTrg]]) \leq C \circ [[sourceConnector]]$

The type of *[[src]]* is restricted by *[[ConnectableElement]]*. Though, we don't know the exact type of *[[src]]* yet, we cannot assign a concrete type until the method is called. That is why the constraint cannot be fulfilled yet, and it must be checked again when inspecting the method call.

Similarly, these constraints cannot be fulfilled yet:
$\exists\, C \in \mathcal{C}$:
type([[srcToTrg]]) = C
type([[src]]) ≤ C ∘ [[sourceConnectableElement]]
$\exists\, C \in \mathcal{C}$:
type([[trg]]) = C
type([[srcToTrg]]) ≤ C ∘ [[targetConnector]]
$\exists\, C \in \mathcal{C}$:
type([[srcToTrg]]) = C
type([[trg]]) ≤ C ∘ [[targetConnectableElement]]

4. Step: On the Method Call

Because no error has been detected by the analysis of the method declaration, we can inspect the method calls. As semantic knowledge base, we use the knowledge base obtained after the analysis of the metamodel and the method declaration. Let us consider the erroneous method call *createConnector(ln, srcOutport, trgVertex)*, with *ln* instance of the class *Line*, *srcOutport* instance of the class *Outport* and *trgVertex* instance of the class *Vertex*.

These arguments are semantically equivalent to:
[[ln]] ∈ $\mathcal{O}$, type([[ln]]) = [[Line]]
[[srcOutport]] ∈ $\mathcal{O}$, type([[srcOutport]]) = [[Outport]]
[[trgVertex]] ∈ $\mathcal{O}$, type([[trgVertex]]) = [[Vertex]]

We first check the arguments by means of the rules `R_call1` and `R_call2`.
The rule `R_call1` applies on each arguments and its semantic constraint is fulfilled every time since the signature contains 3 parameters.
The semantic pattern of `R_call2` matches the method's first parameter because $[[createConnector]]_1$ is *[[srcToTrg:Connector]]*, where *[[srcToTrg]]* belongs to the set of objects $\mathcal{O}$. In addition, *[[Connector]]* is returned by the application of Θ on this parameter and belongs to the set of classes $\mathcal{C}$. The rule's semantic constraint is fulfilled: the argument *[[ln]]* is an element of the set $\mathcal{O}$, and its type *[[Line]]* is restricted by *[[Connector]]*. Thus, we can add *[[ln]]* to the set of arguments $\mathcal{A}rg$. In addition, we can bind this argument to the first parameter of the method *createConnector* via the operation *arg*.
We can apply similarly `R_call2` on the second argument *srcOutport* and the third argument *trgVertex*. Then, we can add *[[srcOutport]]* and *[[trgVertex]]* to the set of arguments $\mathcal{A}rg$. We can also connect these arguments to the corresponding parameter via the operation *arg*.
Thus, the semantic knowledge base contains this additional information:
$\mathcal{A}rg$ *: { [[ln]], [[srcOutport]], [[trgVertex]] }*
arg($[[createConnector]]_1$) = arg([[srcToTrg]])= [[ln]]

arg([[createConnector]]$_2$) = arg([[src]])= [[srcOutport]]
arg([[createConnector]]$_3$) = arg([[trg]])= [[trgVertex]]

After having checked the arguments, we bind each argument to the corresponding pattern object by means of the rule `R_bind1`. The rule can apply on the pattern object *srcToTrg* and the argument *ln* because *[[srcToTrg]]* is a parameter ($\in \mathcal{P}a$) which also belongs to the set of objects ($\in \mathcal{O}$). In addition, this parameter is connected to the argument *ln* which is an object too (*arg([[srcToTrg]]) = [[ln]]* $\in$ *($\mathcal{O} \cap \mathcal{A}rg$)*). Thus, we can infer that *srcToTrg* and *ln* have the same type which semantically means:
type([[srcToTrg]]) = type([[ln]]) = [[Line]]

We can apply this rule similarly on the pattern objects *src* and *trg* and the arguments *srcOutport* and *trgVertex*, and add this semantic information to the semantic knowledge base:
type([[src]]) = type([[srcOutport]]) = [[Outport]]
type([[trg]]) = type([[trgVertex]]) = [[Vertex]]

All matching rules with a semantic conclusion and whose semantic constraints are fulfilled have been applied. Thus, the semantic knowledge base is complete. Let us now consider the remaining matching rule `R_substitute2` .
It matches the pattern objects *[[src]]* and *[[srcToTrg]]*, and the property *[[sourceConnector]]*. The semantic knowledge base contains namely the information
type([[srcToTrg]]) $\leq$ *type([[src]])* $\circ$ *[[sourceConnector]]*
In addition, after the argument values have been bound to the corresponding parameter, the semantic knowledge base contains this additional information:
type([[srcToTrg]]) = type([[ln]]) $\leq$ *[[Line]]*
type([[src]]) = type([[srcOutport]]) = [[Outport]]
The property *sourceConnector* of the type of *src* and the class *Line* must have an inheritance relationship for the semantic constraint of `R_substitute2` to be fulfilled. This is the case since we have the relationship:
type([[src]])$\circ$*[[sourceConnector]]*
= *[[Outport]]*$\circ$*[[sourceConnector]]*
= *[[Line]]*

Similarly, the pattern of the rule `R_substitute2` matches the objects *[[src]]* and *[[srcToTrg]]*, and the property *[[sourceConnectableElement]]*. The property *sourceConnectableElement* of the type of *srcToTrg* and the class *Outport* must have an inheritance relationship for the semantic constraint to be fulfilled, which is the case. This is the case since we have the relationship:
type([[srcToTrg]])$\circ$*[[sourceConnectableElement]]*
= *[[Line]]*$\circ$*[[sourceConnectableElement]]*
= *[[Outport]]*

This rule matches the objects *[[trg]]* and *[[srcToTrg]]*, and the property *[[targetConnector]]* too. The semantic knowledge base contains namely the information
$type([[srcToTrg]]) \leq type([[trg]]) \circ [[targetConnector]]$
In addition, after the argument values have been bound to the corresponding parameter, the semantic knowledge base contains these additional information:
$type([[srcToTrg]]) = type([[ln]]) \leq [[Line]]$
$type([[trg]]) = type([[trgVertex]]) = [[Vertex]]$
The property *targetConnector* of the type of *trg* and the class *Line* must have an inheritance relationship for the semantic constraint of `R_substitute2` to be fulfilled. This is not the case. The *targetConnector* of the instance *trgVertex* of *Vertex* is a *Transition*:
$type([[trg]]) \circ [[targetConnector]]$
$= [[Vertex]] \circ [[targetConnector]]$
$= [[Transition]]$
There is no inheritance relationship between *Transition* and *Line*. This unfulfilled constraint is semantically expressed by:
$[[Transition]] \leq [[Line]] \vee [[Line]] \leq [[Transition]] : false$
The constraint of the rule `R_substitute2` must ensure that the classes of two objects connected by a link are connected by an association in the metamodel. In other words, this rule checks whether this link between both objects may exist. There is no association between *Outport* and *Line* in the metamodel. Thus, no link can be created. This error is detected by the unfulfilled constraint.

Similarly, the semantic constraint of the rule `R_substitute2` on the objects *[[trg]]* and *[[srcToTrg]]*, and the property *[[targetConnectableElement]]*. Here again, the semantic constraint is not fulfilled. The property *targetConnectableElement* of the type of *srcToTrg* and the class *Vertex* must have an inheritance relationship for the semantic constraint of `R_substitute2` to be fulfilled, which is not fulfilled. The *targetConnectableElement* of the instance *srcToTrg* of *Line* is a *Inport*:
$type([[srcToTrg]]) \circ [[targetConnectableElement]]$
$= [[Line]] \circ [[targetConnectableElement]]$
$= [[Inport]]$
There is no inheritance relationship between *Transition* and *Line*. This unfulfilled constraint is semantically expressed by:
$[[Vertex]] \leq [[Inport]] \vee [[Inport]] \leq [[Vertex]] : false$

Finally, let us check the remaining constraints from the story diagram analysis. The type of *[[src]]* is *[[Outport]]*, the type of *[[trg]]* is *[[Vertex]]*, and the type of *[[srcToTrg]]* is *[[Line]]*.
As a consequence, the constraints related to the creation of the link between *src* and *srcToTrg* can be fulfilled:
$type([[srcToTrg]]) \leq type([[src]]) \circ [[sourceConnector]]$
$\Rightarrow [[Line]] \leq [[Outport]] \circ [[sourceConnector]] : true$

type([[src]]) ≤ type([[srcToTrg]])∘[[sourceConnectableElement]]
⇒ [[Outport]] ≤ [[Line]]∘[[sourceConnectableElement]] : true
Though, the constraints related to the creation of the link between *trg* and *srcToTrg* cannot be fulfilled:
type([[srcToTrg]]) ≤ type([[trg]])∘[[targetConnector]]
⇒ [[Line]] ≤ [[Vertex]]∘[[targetConnector]] : false
type([[trg]]) ≤ type([[srcToTrg]])∘[[targetConnectableElement]]
⇒ [[Vertex]] ≤ [[Line]]∘[[targetConnectableElement]] : false

All matching rules have been applied. It remains unfulfilled constraints, thus, type errors are detected.
This example shows how errors can be detected by means of our approach. The same example with a correct method call (*createConnector(Transition, srcVertex, trgVertex)* which connect an instance of *Transition* to two instances of *Vertex*), i.e. where the rules' application does not cause any error, is depicted in the appendix (Section C.3).

5.5 Related Work

Domain-specific visual languages and their graph transformations are more and more popular as a response to the specific modeling needs of the users and due to their quite intuitive notation. Although the area of graph transformations has been extensively explored in the last 30 years, there are still limitations in the evaluation of correctness of model transformations. Contrary to other languages, there are still few tools and methods providing an efficient support for analyzing visual model transformations.

A classical approach for analyzing visual model transformations consists of a semantic mapping. Mapping is executed between a graph transformation based specification of a domain-specific language and another language supported by powerful tools and well-established analysis approaches, e.g. Maude [RGdLV08] or OCL [CCGdL08]. Thus, this possibility allows for using the techniques specific to the target semantic domain in order to analyze the source models. [CCGdL08] proposes an analysis of graph transformation rules based on an intermediate OCL representation, and, consequently, the analysis principally consists of the execution of already existing OCL constraints solver.
It must be noticed that the application of semantic mapping implies that the target language supports all the concepts of the source language and, thus, that a mapping of any graph transformation is possible. A semantic mapping can be executed by means of a *transformation model*, a concept proposed in [BBG+06]. This transformation model consists of an ordinary model which abstracts a model transformation and must not be mistaken with the eventual metamodel of a model transformation, and any model transformation has to satisfy this transformation

model. For instance, if the transformation model is an artifact for semantic mapping to OCL, it can be a UML/MOF class diagram together with OCL constraints. [CCGdL10] illustrates such an analysis based on transformation models automatically derived from model-to-model transformations and executed by means of a constraint solver. According to the authors of [BBG+06], one of the main benefits of transformation models are powerful possibilities for validation and verification provided by well-established methods and tools.
Although our approach can be considered as a mapping, we map syntactic elements to a semantic domain in order to extract type information whereas the above presented approach maps a language to another different modeling language.

Another classical approach in the area of "model transformations checking" is testing a transformation instead of analyzing them. In fact, testing methods are very common in the field of traditional software engineering. Therefore, application of testing methods for model transformations is an approach which can be found in several publications. For instance, in [LZG05], the authors presents a framework for checking model transformations by means of execution-based testing. [KGZ09] describes the testing of a transformation chain, more precisely the generation of test cases for transformation chains and each individual transformation. Though, contrary to our approach, model transformation testing focuses on whether the result of the transformation is the expected one, and not on ensuring that the transformation can be executed error free. In addition, a model transformation is not like a deterministic program, and there may be several different sequences of applicable rules. Consequently, testing or comparing the output of transformation programs (usually models) means checking all the possible sequences by means of a model checker, which can be very time consuming without ensuring a detection of all the errors.
In the same way, the author of [Sch10] does not consider the testing as the best-suited approach for the verification of model transformations. He points out that, for instance, concepts like coverage cannot be immediately applied on model-transformations, especially if those are rules-based or declarative. [Sch10] proposes a formal verification of soundness conditions based on the use of interactive theorem prover instead of using usual testing methods. Contrary to our type-checking approach, it aims at verifying the well-formedness of the models constructed via transformation, e.g. ensuring that no relevant elements of the source model are absent in the target model. In addition, the application of theorem provers needs a mathematical formalization of the program. Finally, the specification of transformations in [Sch10] is not based on graphical, rule-based descriptions, but uses a textual description based on a relational, declarative calculus.
There are other validation techniques consisting in checking model transformation properties. For instance, [LBA10] describes a symbolic model checker built to guarantee transformation properties. This model checker computes an equivalence class for each possible execution of a transformation, and validates that transformation by checking if the transformation property holds for every computed equiv-

alence class. [BW07] presents a verification technique for partner graph grammars, a formalism to model dynamic communication systems. This formalism uses a a restricted form of negative application conditions, called partner constraints which reflects the partner principle, i.e. that an object's behavior is determined by its state and by the state of its communication partners. The approach described in [BW07] is based on an abstraction called partner abstraction whose implementation allows for verifying the preservation of topology properties. Although these techniques propose the checking of model transformation, they aim at checking the preservation of properties, and not at ensuring type safety as described with our approach.

To ensure type safety, we based our approach on the use of rules of inference in order to obtain type information and check their consistency. In the context of type checking, type inference refers to the automatic deduction of the type of an expression in a programming language. The use of inference to determine types of elements in a language is not a new concept and is even pretty usual. Languages that include type inference are for instance Standard ML [MTM97], OCaml [OCa], Haskell [Has], Scala [Sca], D [DLa], Opa [Opa].
In type theory, Hindley-Milner [HHS02] is a classical type inference method with parametric polymorphism for the lambda calculus [BAG+92]. The lambda calculus is a formal system in mathematical logic for expressing computation by way of variable binding and substitution. Hindley-Milner was first described by J. Roger Hindley [Hin69], and was later rediscovered by Robin Milner [Mil78]. The Hindley-Milner approach consists 1) of a description of what types an expression can have and 2) of an algorithm (called algorithm W) behind the logic. How expressions and types fit to each other is described by means of a deductive system which is based on the use of rule of inference. The algorithm W is defined as a step-by-step procedure which aims at validating the deduction system with respect to the rules.
The Hindley-Milner approach was an inspiration for our work since we define a deductive system by application of rule of inference. Nevertheless, our approach is different. The main difference is of course the new notation we defined, adding a field for visual syntax pattern in the premises that must match for the rule to be considered, and semantic constraint that must be fulfilled for the conclusion to be derived. An other difference is that the Hindley-Milner approach applies on functional languages and uses type variables. Finally, we do not only use substitution (Cf. rules of the Fig. 5.15), but also extract a set of constraints and relationships in the semantic domain.

The language PROGRES is one of the rare graphical transformation languages which has a well-defined static type concept [MSW98]. A type system has been considered in [Sch91] and [Mue02]. Though, the type system as defined in [Sch91] does not include polymophism. [Mue02] includes parametric polymophism, but does not define a formal type system as presented in this work. The definition of constraints (global and local, active and passive) allows for the type checking of

PROGRES as follows. Global graph consistency conditions, i.e. global constraints, are defined as graph patterns which have to be present in a hostgraph at any time or which may never appear. Constraint attributes, i.e. local constraints, are constraints, textually or graphically specified, which are bound to node classes. Active constraints are passive constraints extended by an repair action which is triggered if a the condition does not hold: this is pretty similar to our use of SDM to check guideline in the MATE project, except that the constraints concentrate on the type checking.
The textual and graphical definition of constraints is close to the definition of our rules' premises. Though, the approach is quite different, since ours is based on the mapping to semantic domains, creation of sets of types and logic.

As far as we know, the approach which is the closest to our approach is the type checking approach for VIATRA2 described in [UHV11], [UHV09b] and [UHV09a]. Whereas most analysis approaches aim at ensuring the semantic of the model transformations, i.e. whether the transformations specify the required behavior or whether properties are preserved, [UHV11] focuses on detecting typing errors. It describes a static type checking approach for early detection of typing errors, considering type safety as constraint satisfaction problems. The concept is pretty close to our own concept, since errors are detected when type constraints cannot be satisfied. More precisely, as in our approach, the model transformations are mapped to an abstraction and type constraints, which must be satisfied, are extracted. Some relations defined for the needs of this approach are quite similar. For instance, the *type equality* resp. the *substitutability relations* are the counterparts of our = resp. $\leq$ Though, the mapping to the abstract interpretation of the transformation program is not the same. The VIATRA2 type checker first creates a representation of the type system of the transformation program as a graph (Transformation Program Model) which will be traversed to extract constraints, whereas our approach consists in the mapping to the semantic domain composed of different sets (set of classes, set of properties,...) In addition, the VIATRA2 languages does not provide the redefinition of association ends, which simplifies considerably the type checking for VIATRA2 compared to the one of our approach.

Compared to the other approaches, the main contribution is probably the definition of a new notation for rules of inference which allows for combining syntactic and semantic pattern matching as premises, constraint checking and inference of type information in a single rule. The second main contribution is the support of type analysis in the case of the redefinition of association ends, too. As far as we know, there are no equivalent approach in the area of the type checking for visual model transformations.

Chapter 6

Evaluation and Outlook

We developed in this work on the one hand new features for the SDM languages and on the other hand a type checking approach for SDM. We describe in a first section an example which combines both aspects of this work, i.e. the application of our type checking approach on a story diagram containing the new SDM features.
The generic and reflective features for SDM have been defined in order to improve the reusability and expressiveness of story diagrams. These features are not the only properties which are desirable for SDM. Therefore we describe in a second section possible additional features for SDM. Since the MATE/MAJA projects were the motivation for our work, we also present in this second section an evaluation of the impact of the new features on the MAAB guidelines.
We explained in the Section 2.5 various concepts and classification criteria in the area "type system" in order to provide the reader a better understanding. We can determine here a classification of our type system now that we have presented our approach in the previous chapter. We then present a way to test the type system, restricted to the case of attribute checking, before we evaluate and present an outlook of our approach.

6.1 Type Checking of Generic Graph Transformations

We have applied the generic and reflective feature in an example in Section 4.1.2 and showed a simple application of the static type checking in Section 5.4.4. In this section, we will now present a complete illustration of our work by using our generic feature in a graph transformation. We will apply our type checking approach on this transformation. Once the metamodel excerpt has been analyzed, we can analyze the method declaration starting by the method signature followed by the story diagram. After the method specification analysis, we can check each method call, using the semantic knowledge base derived from the metamodel excerpt and the story diagram. We consider various method calls, and show how our approach allows for distinguishing between use and misuse cases.

1. Step: On the Metamodel

This application example only requires the analysis of the metamodel excerpt of Fig.6.1. It is the same metamodel excerpt as the example in Section 5.4.4. Therefore we can use the same semantic knowledge base obtained by the rule's application on this metamodel excerpt, i.e.:

- The set of classes $\mathcal{C}$ obtained by application of the rule `R_sem1`:
  { *[[Connector]], [[UndefinedConnector]], [[ConnectableElement]], [[Line]], [[Port]], [[Inport]], [[Outport]], [[Transition]], [[Vertex]]* }
- The set of properties $\mathcal{P}$ obtained by application of the rule `R_sem3`:
  { *[[sourceConnector]], [[targetConnector]], [[sourceConnectable]], [[targetConnectableElement]], [[line]], [[port]], [[outport]], [[inport]], [[sourceLine]], [[targetLine]], [[transitionToTarget]], [[transitionToSource]], [[target]], [[source]], [[src]], [[trg]], [[sourcePort]], [[targetPort]]* }
- The inheritance relationships between classes and inherited properties obtained by application of the rules `R_inherit1` and `R_inherit2`.
- The property relationships obtained by application of the rules `R_prop1` and `R_redefineProp`

The detailed semantic knowledge base obtained from the analysis of the metamodel excerpt can be found in the first pages of Section 5.4.4.

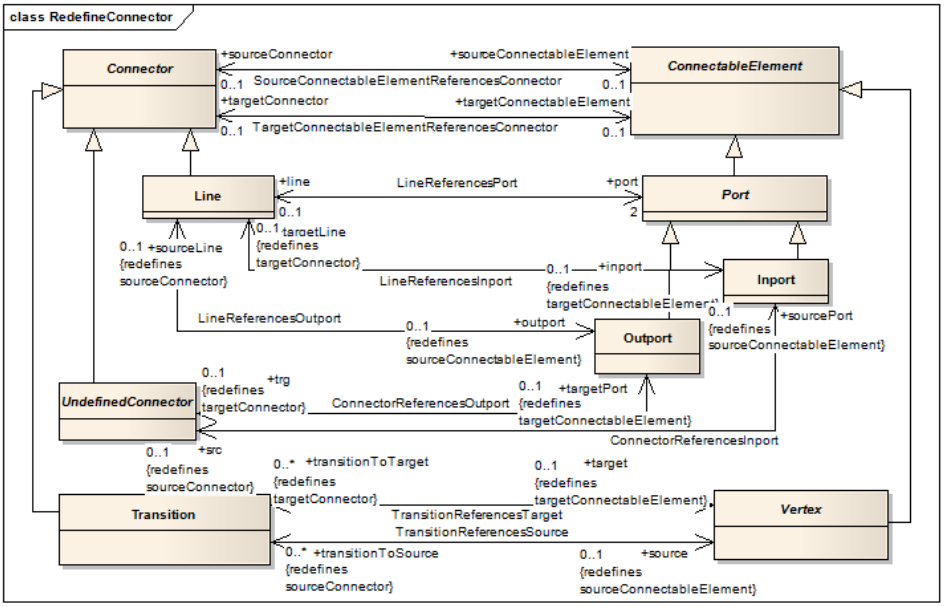


Figure 6.1: Metamodel Excerpt

2. Step: On the Method Signature

The method specification in Fig. 6.2 uses our generic feature as parameterized type. This method's application aims at creating an instance of type *Connector* between two instances of type *ConnectableElement*.
The parameter called *connect* is a subclass of *Connector*, which is expressed by the signature element *connect extends Connector : MOFClass*. This parameterized class will define the type of the object *srcToTrg* to be created. The parameter called *connected* is a class, and more precisely must be a subclass of *ConnectableElement*, which is expressed by the signature element *connected extends ConnectableElement : MOFClass*. This parameterized class is used in the method signature as type of the parameters *src* and *trg*, which is represented by the signature elements *src:$connected* and *trg:$connected*.

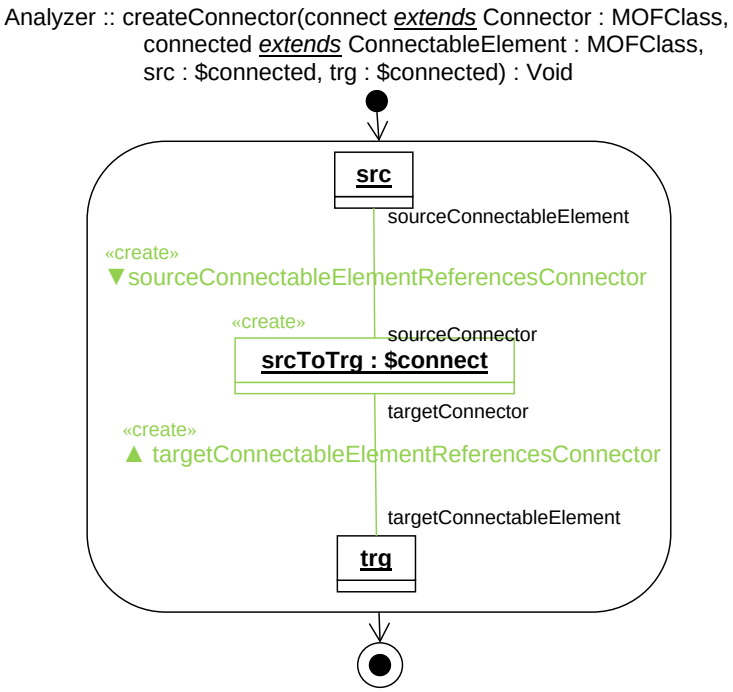


Figure 6.2: Method *createConnector* with parameterized type

We will first start with the analysis of the method's signature:

Analyzer :: createConnector(connect extends Connector : MOFClass,
connected extends ConnectableElement : MOFClass,
src : $connected, trg : $connected) : Void

This signature represents the syntactic knowledge base for the signature analysis and the semantic knowledge base is the one obtained after the metamodel analysis. By means of the rules `R_sem4`, `R_sem5`, `R_sem7` and `R_generic`, we can complete the semantic knowledge base with the set of parameters $\mathcal{P}a$.

$\mathcal{P}a$ = { *[[createConnector]]*$_1$, *[[createConnector]]*$_2$, *[[createConnector]]*$_3$,
[[createConnector]]$_4$ }
= { *[[connect extends Connector:MOFClass]],*
[[connected extends ConnectableElement:MOFClass]],
[[src:$connected]], [[trg:$connected]] }
= { *[[connect]], [[connected]], [[src]], [[trg]]* }

We can then create a set of bound pattern objects $\mathcal{O}$ containing the objects *[[src]]* and *[[trg]]* and extract type information by application of the Θ-operator on the parameters (rules `R_theta1`, `R_theta4`, `R_generic`).
$\mathcal{O}$: { *[[src]], [[trg]]* }
Θ(*[[CreateConnector]]*$_1$) = *[[Connector]]* $\in \mathcal{C}$
Θ(*[[CreateConnector]]*$_2$) = *[[ConnectableElement]]* $\in \mathcal{C}$
Θ(*[[CreateConnector]]*$_3$) = Θ(*[[CreateConnector]]*$_4$) = *[[connected]]* $\in \mathcal{C}$
type([[src]]) $\leq$ *[[connected]]*
type([[trg]]) $\leq$ *[[connected]]*

In addition, the application of these rules allows for completing the set of classes $\mathcal{C}$ with *[[connect]]* whose type is restricted by *[[Connector]]*, and with *[[connected]]* whose type is restricted by *[[ConnectableElement]]*:
$\mathcal{C}$: { *[[Connector]], [[ConnectableElement]], [[Line]], [[Port]], [[Inport]],*
[[Outport]], [[Transition]], [[Vertex]], [[connect]], [[connected]] }
[[connect]] $\leq$ *[[Connector]]*.
[[connected]] $\leq$ *[[ConnectableElement]]*.

3. Step: On the Story Diagram

After having checked and extracted information from the signature, we can analyze the story diagram which is the syntactic knowledge base for this new analysis step. The semantic knowledge base then consists of the semantic information extracted from the metamodel and from the signature.

Because the story pattern contains two pattern objects *src* and *trg* which are depicted as bound objects, we apply the rule `R_bound`. The rule's semantic constraint is fulfilled because the semantic knowledge base contains as information that the objects' semantic counterparts *[[src]]* and *[[trg]]* belong to the set of bound pattern objects $\mathcal{O}$.

The syntactic pattern of the rule `R_unbound2` matches the unbound pattern object *srcToTrg*. Its type *$connect* is parameterized (*[[connect]]* $\in \mathcal{P}a$). The type information returned by the application of Θ on this parameter returns *[[Connector]]* which belongs to the set of classes $\mathcal{C}$. The semantic constraint is fulfilled, and we can add the semantic object *[[srcToTrg]]* to the set of pattern objects $\mathcal{O}$:
$\mathcal{O}$: { *[[src]], [[trg]], [[srcToTrg]]* }

In addition, we can complete the semantic knowledge base with this information:
$type([[srcToTrg]]) \leq [[Connector]]$

The unbound pattern object *srcToTrg* whose type is parameterized is connected to the bound pattern object *src* which also belongs to the set of parameters. In addition, the parameter type is restricted by a class. This is semantically expressed by: $\Theta([[src]]) = [[connected]] \leq [[ConnectableElement]]$
Thus,the rule `R_prop6` matches. We first check that the link complies to the metamodel by means of the rule's semantic constraint. The class *ConnectableElement* possesses the property *sourceConnector* which is of type *Connector*. Thanks to the reflexive property of the $\leq$-operator, the semantic constraint is fulfilled:
$[[Connector]] \leq [[ConnectableElement]] \circ [[sourceConnector]] : true$
We can consequently complete the semantic knowledge base with the additional information on the type of *srcToTrg*:
$type([[srcToTrg]]) \leq type([[src]]) \circ [[sourceConnector]]$

Similarly, we can apply the same rule on the pattern composed of the objects *trg* and *srcToTrg* and deduce this information:
$type([[srcToTrg]]) \leq type([[trg]]) \circ [[targetConnector]]$

Finally, because the links are created between the bound object *src* and the unbound object *srcToTrg*, and between the bound object *trg* and the unbound object *srcToTrg*, the rule `R_prop9` is considered. Though, the constraints cannot be fulfilled yet. Let us consider the semantic constraint:

$\exists\, C \in \mathcal{C}$:
$type([[src]]) = C$
$type([[srcToTrg]]) \leq C \circ [[sourceConnector]]$

The type of *[[src]]* is restricted by *[[connected]]* whose value is only defined when calling the method. That is why the constraint cannot be fulfilled yet, and it must be checked again when inspecting the method call.
Similarly, this constraints cannot be fulfilled yet:

$\exists\, C \in \mathcal{C}$:
$type([[trg]]) = C$
$type([[srcToTrg]]) \leq C \circ [[targetConnector]]$

We will now check the method calls. The semantic knowledge base consists of the semantic information obtained by the application of the rules of inference on the metamodel excerpt and the graph transformation.

6.1.1 Use case

Let us consider this method call:
createConnector(Transition, Vertex, srcVertex, trgVertex)

This use case creates an instance of *Connector*, more precisely an instance of *Transition*, between two instances of the class *Vertex*. Let us inspect it with help of our type system. As semantic knowledge base, we use the semantic knowledge base obtained from the analysis of the metamodel and of the method's specification. The syntactic knowledge base is composed of the method call.

The arguments are semantically equivalent to:
[[Transition]] $\in \mathcal{C}$
[[Vertex]] $\in \mathcal{C}$
[[srcVertex]] $\in \mathcal{O}$, *type([[srcVertex]])* = *[[Vertex]]*
[[trgVertex]] $\in \mathcal{O}$, *type([[trgVertex]])* = *[[Vertex]]*

The premises of the rules `R_call1` and `R_call4` match the first and second arguments.
The semantic constraint of `R_call1` is fulfilled because there are a first and a second parameter corresponding to a first and a second argument.
The application of Θ on the first parameter returns *[[Connector]]*, i.e. an element belonging to the set $\mathcal{C}$, and the semantic pattern of the rule `R_call4` matches the parameter *[[connect extends Connector:MOFClass]]*. The semantic constraint is fulfilled: *[[Transition]]* belongs to the set $\mathcal{C}$ and is a subclass of *[[Connector]]*,i.e. by the return value of Θ applied on the first parameter. Consequently, we can derive the conclusions of `R_call4`. The argument *[[Transition]]* is added to the set of arguments $\mathcal{A}rg$. In addition, this argument is connected to its parameter by the application *arg*.

We can apply similarly `R_call4` on the second argument, and add the argument *[[Vertex]]* to the set of arguments $\mathcal{A}rg$. In addition, this argument is connected to its parameter by the application *arg*.

The premises of the rules `R_call1` and `R_call2` matches the third argument. The semantic constraint of `R_call1` is fulfilled since there is a third parameter $[[CreateConnector]]_3$ corresponding to the third argument.
$[[createConnector]]_3$ is *[[src : $connected]]*, where *[[src]]* belongs to the set of objects $\mathcal{O}$. In addition, *[[connected]]* is returned by the application of Θ on this parameter and belongs to the set of classes $\mathcal{C}$. Accordingly to the semantic knowledge, *[[connected]]* is a subtype of the class *[[ConnectableElement]]*. Thus, the semantic pattern of `R_call2` matches the method's third parameter $[[createConnector]]_3$, and this rule instantiation belongs to the matching set. The rule's semantic constraint is respected: the argument *[[srcVertex]]* is an element of

the set $\mathcal{O}$, and its type *[[Vertex]]* is restricted by *[[ConnectableElement]]*. Thus, we can add *[[srcVertex]]* to the set of arguments $\mathcal{A}rg$. In addition, we can bind this argument to the second parameter of the method *createConnector* via the operation *arg*.

We can apply similarly `R_call2` on the last argument *trgVertex*. Then, we can add *[[trgVertex]]* to the set of arguments $\mathcal{A}rg$, and we can connect this argument to the third parameter via the operation *arg*.

$\mathcal{A}rg$ *: { [[Transition]], [[Vertex]], [[srcVertex]], [[trgVertex]] }*
arg([[CreateConnector]]$_1$*) = [[Transition]].*
arg([[CreateConnector]]$_2$*) = [[Vertex]].*
arg([[createConnector]]$_3$*) = arg([[src]])= [[srcVertex]].*
arg([[createConnector]]$_4$*) = arg([[trg]])= [[trgVertex]].*

After having checked whether the arguments comply to the method signature, we will bind the arguments to the corresponding elements from the method declaration (parameters, pattern objects).

[[Transition]] belongs to the set of classes $\mathcal{C}$ and to the set of arguments $\mathcal{A}rg$ The function *arg* connects the parameter *[[connect extends Connector: MOFClass]]* to this argument. Thus, the premises of the rule `R_bind6` match the first argument. In addition, it respects the semantic constraint:
[[Transition]] $\leq$ *[[Connector]] : true*
Therefore, we can substitute the argument *[[Transition]]* for the parameter *[[connect]]*.
Similarly, we can apply the rule `R_bind6` on the second parameter and its corresponding argument. As a result, we can substitute the argument *[[Vertex]]* for the parameter *[[connected]]*.

[[src]] belongs to the set of objects $\mathcal{O}$. There is a parameter *[[src:$connected]]*, where *[[connected]]* belongs to the set of parameters $\mathcal{P}a$ too. *[[Vertex]]* belongs to $\mathcal{C}$ as well as to the set of arguments $\mathcal{A}rg$. The parameter related to this argument is the second parameter *[[CreateConnector]]*$_2$, i.e. *[[connected]]*. The parameter related to the argument *[[srcVertex]]* is the parameter [[src:$connected]], which is expressed by *arg([[src:$connected]]) = [[srcVertex]]* $\in$ *(*$\mathcal{O} \cap \mathcal{A}rg$*)*. Thus, the premises of the rule `R_bind3` match the third argument. In addition, the object *[[srcVertex]]* is instance of a subclass of *[[Vertex]]*. Thus, the semantic constraint is fulfilled. This is semantically expressed by:
type([[srcVertex]]) $\leq$ *[[Vertex]] : true*
We can deduce the rule's conclusion and substitute the type of the object *[[srcVertex]]* (from the method call) for the type of the object *[[src]]* (from the method declaration), i.e.
type ([[src]]) = type([[srcVertex]])

In the same way, we can apply the rule `R_bind3` on the pattern object *[[trg]]* and the argument *[[trgVertex]]*. By applying this rule, we can substitute the type of the object *[[trgVertex]]* (from the method call) for the type of the object *[[trg]]* (from the method declaration) i.e.
type ([[trg]]) = type([[trgVertex]]).

The unbound pattern object *[[srcToTrg]]* has a parameterized type *[[connect]]* which corresponds to the first parameter $[[CreateConnector]]_1$. This parameter is connected to the argument *[[Transition]]* which is a class:
$arg([[connect]]) = [[Transition]] \in (\mathcal{C} \cap \mathcal{A}rg)$
Thus, we can infer that *Transition* is the type of *srcToTrg* which semantically means: *type([[srcToTrg]]) = [[Transition]]*

All matching rules with a semantic conclusion and whose semantic constraints are fulfilled have been applied. Thus, the semantic knowledge base is complete. Let us now consider the remaining matching rule `R_substitute2` .
It matches the pattern objects *[[src]]* and *[[srcToTrg]]*, and the property *[[sourceConnector]]*. The semantic knowledge base contains namely the information
$type([[srcToTrg]]) \leq type([[src]]) \circ [[sourceConnector]]$
In addition, after the argument values have been bound to the corresponding parameter, the semantic knowledge base contains these additional information:
type([[srcToTrg]]) = [[Transition]]
type([[src]]) = type([[srcVertex]]) = [[Vertex]]
The property *sourceConnector* of the type of *src* and the class *Transition* must have an inheritance relationship for the semantic constraint of `R_substitute2` to be fulfilled. This is the case since we have the relationship:
$type([[src]]) \circ [[sourceConnector]]$
$= [[srcVertex]] \circ [[sourceConnector]]$
$= [[Transition]]$

The rule `R_substitute2` can be applied on the objects *[[src]]* and *[[srcToTrg]]*, and the property *[[sourceConnectableElement]]* too. Its pattern matches, and the semantic constraints are fulfilled.
Similarly, this rule `R_substitute2` can be applied successfully on the objects *[[trg]]* and *[[srcToTrg]]*, with the property *[[targetConnectableElement]]*, as well as with the property *[[targetConnector]]*.

Finally, let us check the remaining constraints from the story diagram analysis. The type of *[[src]]* is *[[Vertex]]*, the type of *[[trg]]* is *[[Vertex]]*, and the type of *[[srcToTrg]]* is *[[Transition]]*.
As a consequence, the remaining constraints can be fulfilled:
$type([[srcToTrg]]) \leq type([[src]]) \circ [[sourceConnector]]$
$\Rightarrow [[Transition]] \leq [[Vertex]] \circ [[sourceConnector]] : true$

type([[srcToTrg]]) ≤ type([[trg]])∘[[targetConnector]]
⇒ [[Transition]] ≤ [[Vertex]]∘[[targetConnector]] : true

Conclusion:
All matching rules have been applied, and no error has been detected.

6.1.2 Misuse case 1

Let us consider this method call:
createConnector(Transition, Port, srcOutport, trgInport)

In this example, both *ConnectableElement* are instances of *Port*. More precisely, this method call tries to create a *Connector* from an instance of *Outport* to an instance of *Inport*. Though, the first argument is erroneous because no *Transition* can be created between two *Port*. Let us show how this error can be detected by our type checking approach. As semantic knowledge base, we use the semantic knowledge base obtained from the analysis of the metamodel and of the method's specification. The syntactic knowledge base is composed of the method call.

The arguments are semantically equivalent to:
[[Transition]] $\in \mathcal{C}$
[[Port]] $\in \mathcal{C}$
[[srcOutport]] $\in \mathcal{O}$, *type([[srcOutport]])* = *[[Outport]]*
[[trgInport]] $\in \mathcal{O}$, *type([[trgInport]])* = *[[Inport]]*

The premises of the rules `R_call1` and `R_call4` match the first and second arguments. The semantic constraint of `R_call1` is fulfilled since there is a first and a second parameter corresponding to a first and a second argument. The application of Θ on the first parameter *[[connect extends Connector:MOFClass]]* returns *[[Connector]]*, i.e. an element belonging to the set $\mathcal{C}$, and the semantic pattern of the rule `R_call4` matches. The semantic constraint is fulfilled: *[[Transition]]* belongs to the set $\mathcal{C}$ and is a subclass of *[[Connector]]*,i.e. by the return value of Θ applied on the first parameter. Consequently, we can derive the conclusions of `R_call4`. The argument *[[Transition]]* is added to the set of arguments $\mathcal{A}rg$. In addition, this argument is connected to its parameter by the application *arg*. We can apply similarly `R_call4` on the second argument, and add the argument *[[Port]]* to the set of arguments $\mathcal{A}rg$. In addition, this argument is connected to its parameter by the application *arg*.

The premises of the rules `R_call1` and `R_call2` match the third argument. The semantic constraint of `R_call1` is fulfilled because there is a third parameter corresponding to the third argument. $[[createConnector]]_3$ is *[[src : $connected]]*, where *[[src]]* belongs to the set of objects $\mathcal{O}$. In addition, *[[connected]]* is returned by the application of Θ on this parameter and belongs to the set of classes.

Accordingly to the semantic knowledge, *[[connected]]* is a subtype of the class *[[ConnectableElement]]*. Thus, the semantic pattern of `R_call2` matches the method's third parameter $[[createConnector]]_3$, and this rule instantiation belongs to the matching set. The rule's semantic constraint is respected: the argument *[[srcOuport]]* is an element of the set $\mathcal{O}$, and its type *[[Outport]]* is restricted by *[[ConnectableElement]]*. Thus, we can add *[[srcOuport]]* to the set of arguments $\mathcal{A}rg$. In addition, we can bind this argument to the third parameter of the method *createConnector* via the operation *arg*.

We can similarly apply `R_call1` and `R_call2` on the last argument *trgInport*. Then, we can add *[[trgInport]]* to the set of arguments $\mathcal{A}rg$, and we can connect this argument to the last parameter via the operation *arg*.

$\mathcal{A}rg$: { *[[Transition]], [[Port]], [[srcOutport]], [[trgInport]]* }
arg([[CreateConnector]]$_1$*) = [[Transition]].*
arg([[CreateConnector]]$_2$*) = [[Port]].*
arg([[createConnector]]$_3$*) = arg([[src]])= [[srcOutport]].*
arg([[createConnector]]$_4$*) = arg([[trg]])= [[trgInport]].*

After having checked whether the arguments comply to the method signature, we will bind the arguments to the corresponding elements from the method declaration (parameters, pattern objects).

[[Transition]] belongs to the set of classes $\mathcal{C}$ and to the set of arguments $\mathcal{A}rg$. The function *arg* connects the parameter *[[connect extends Connector: MOFClass]]* to this argument. Thus, the premises of the rule `R_bind6` match the first argument. In addition, it respects the semantic constraint, i.e.
$[[Transition]] \leq [[Connector]] : true$
Therefore, we can substitute the argument *[[Transition]]* for the first parameter *[[connect]]*.

Similarly, we can apply the rule `R_bind6` on the second parameter and its corresponding argument. As a result, we can substitute the argument *[[Port]]* for the parameter *[[connected]]*.

[[src]] belongs to the set of objects $\mathcal{O}$. There is a parameter *[[src:$connected]]*, where *[[connected]]* belongs to the set of parameters $\mathcal{P}a$ too. *[[Port]]* belongs to $\mathcal{C}$ as well as to the set of arguments $\mathcal{A}rg$. The parameter related to this argument is the second parameter $[[CreateConnector]]_2$, i.e. *[[connected]]*. The parameter related to the argument *[[srcOutport]]* is the parameter *[[src:$connected]]*:
$arg([[src:\$connected]]) = [[srcOutport]] \in (\mathcal{O} \cap \mathcal{A}rg)$
Thus, the premises of the rule `R_bind3` match the third argument. In addition, the object *[[srcOutport]]* is instance of a subclass of *[[Port]]*. Thus, the semantic constraint is fulfilled: $type([[srcOutport]]) \leq [[Port]] : true$

We can apply the rule and deduce the rule's conclusion, i.e. substitute the type of the object *[[srcOutport]]* (from the method call) for the type of the object *[[src]]* (from the method declaration):
type ([[src]]) = type([[srcOutport]])

In the same way, we can apply the rule `R_bind3` on the pattern object *[[trg]]* and the argument *[[trgInport]]*. By applying this rule, we can substitute the type of the object *[[trgInport]]* (from the method call) for the type of the object *[[trg]]* (from the method declaration):
type ([[trg]]) = type([[trgInport]])

The unbound pattern object *[[srcToTrg]]* has a parameterized type *[[connect]]* which corresponds to the first parameter $[[CreateConnector]]_1$. This parameter is connected to the argument *[[Transition]]* which is a class:
$arg([[connect]]) = [[Transition]] \in (\mathcal{C} \cap \mathcal{A}rg)$
Thus, we can apply the rule `R_bind2` and infer that *Transition* is the type of *srcToTrg* which semantically means:
type([[srcToTrg]]) = [[Transition]]

All matching rules with a semantic conclusion and whose semantic constraints are fulfilled have been applied. Thus, the semantic knowledge base is complete. Let us now consider the remaining matching rule `R_substitute2`.
It matches the pattern objects *[[src]]* and *[[srcToTrg]]*, and the property *[[sourceConnector]]*. The semantic knowledge base contains namely the information
$type([[srcToTrg]]) \leq type([[src]]) \circ [[sourceConnector]]$
In addition, after the argument values have been bound to the corresponding parameter, the semantic knowledge base contains these additional information:
type([[srcToTrg]]) = [[Transition]]
type([[src]]) = type([[srcOutport]]) = [[Outport]]
The property *sourceConnector* of the type of *src* and the class *Transition* must have an inheritance relationship for the semantic constraint of `R_substitute2` to be fulfilled. This is not the case. The *sourceConnector* of the instance *srcOutport* of *Outport* is a *Line*:
$type([[src]]) \circ [[sourceConnector]]$
$= [[Outport]] \circ [[sourceConnector]]$
$= [[Line]]$
There is no inheritance relationship between *Transition* and *Line*. This unfulfilled constraint is semantically expressed by:
$[[Transition]] \leq [[Line]] \vee [[Line]] \leq [[Transition]] : false$

The rule `R_substitute2` matches the objects *[[src]]* and *[[srcToTrg]]*, and the property *[[sourceConnectableElement]]* too. The property *sourceConnectableElement* of the type of *srcToTrg* and the class *Outport* must have an inheritance relationship for the semantic constraint of `R_substitute2` to be fulfilled. This is

not the case. The *sourceConnectableElement* of *srcToTrg* (which is of type *Transition*), is a *Vertex*:
$type([[srcToTrg]]) \circ [[sourceConnectableElement]]$
$= [[Transition]] \circ [[sourceConnectableElement]]$
$= [[Vertex]]$
There is no inheritance relationship between *Vertex* and *Outport*. This unfulfilled constraint is semantically expressed by:
$[[Vertex]] \leq [[Outport]] \vee [[Outport]] \leq [[Vertex]] : false$

Similarly, the rule `R_substitute2` matches the objects *[[trg]]* and *[[srcToTrg]]*, with the property *[[targetConnectableElement]]*, as well as with the property *[[targetConnector]]*. In both cases, the semantic constraint cannot be fulfilled.

Finally, let us check the remaining constraints from the story diagram analysis. The type of *[[src]]* is *[[Outport]]*, the type of *[[trg]]* is *[[Inport]]*, and the type of *[[srcToTrg]]* is *[[Transition]]*.
The consequence is that the remaining constraint cannot be fulfilled:
$type([[srcToTrg]]) \leq type([[src]]) \circ [[sourceConnector]]$
$\Rightarrow [[Transition]] \leq [[Outport]] \circ [[sourceConnector]] : false$
$type([[srcToTrg]]) \leq type([[trg]]) \circ [[targetConnector]]$
$\Rightarrow [[Transition]] \leq [[Inport]] \circ [[targetConnector]] : false$

Conclusion:
All matching rules have been applied, and it remains unfulfilled constraints: type error detected.

6.1.3 Misuse case 2

Let us consider this method call:
createConnector(Line, Port, srcInport, trgOutport)

This method call seems to be correct at the first glance, creating a *Line* between two *Port*, but it is not correct. Due to the properties *sourceConnectableElement* and *targetConnectableElement* and their redefinition in the metamodel, the third argument should be the instance of *Outport*, and the last argument should be the instance of *Inport*. Let us show how we can detect this error statically by means of our approach.

The arguments are semantically equivalent to:
$[[Line]] \in \mathcal{C}$
$[[Port]] \in \mathcal{C}$
$[[srcInport]] \in \mathcal{O}$, $type([[srcInport]]) = [[Inport]]$
$[[trgOutport]] \in \mathcal{O}$, $type([[trgOutport]]) = [[Outport]]$

The premises of the rules R_call1 and R_call4 match both first arguments. The semantic constraint of R_call1 is fulfilled since there is a first and a second parameter corresponding to a first and a second argument. The application of Θ on the first parameter returns *[[Connector]]*, i.e. an element belonging to the set $\mathcal{C}$, and the semantic pattern of the rule R_call4 matches:
[[connect extends Connector:MOFClass]]
Thus, this rule instantiation belongs to the matching set. The semantic constraint is fulfilled: *[[Line]]* belongs to the set $\mathcal{C}$ and is a subclass of *[[Connector]]*,i.e. by the return value of Θ applied on the first parameter. Consequently, we can derive the conclusions of R_call4. The argument *[[Line]]* is added to the set of arguments $\mathcal{A}rg$. In addition, this argument is connected to its parameter by the application *arg*. We can apply similarly R_call4 on the second argument, and add the argument *[[Port]]* to the set of arguments $\mathcal{A}rg$. In addition, this argument is connected to its parameter by the application *arg*.

The premises of the rules R_call1 and R_call2 matches the third argument. The semantic constraint of R_call1 is fulfilled because there is a third parameter corresponding to the third argument. $[[createConnector]]_3$ is *[[src : $connected]]*, where *[[src]]* belongs to the set of objects $\mathcal{O}$. In addition, *[[connected]]* is returned by the application of Θ on this parameter and belongs to the set of classes $\mathcal{C}$. Accordingly to the semantic knowledge, *[[connected]]* is a subtype of the class *[[ConnectableElement]]*. Thus, the semantic pattern of R_call2 matches the method's third parameter $[[createConnector]]_3$, and this rule instantiation belongs to the matching set. The rule's semantic constraint is respected: the argument *[[srcInport]]* is an element of the set $\mathcal{O}$, and its type *[[Inport]]* is restricted by *[[ConnectableElement]]*. Thus, we can add *[[srcInport]]* to the set of arguments $\mathcal{A}rg$. In addition, we can bind this argument to the third parameter of the method *createConnector* via the operation *arg*.

We can similarly apply R_call1 and R_call2 on the last argument *trgOutport*. Then, we can add *[[trgOutport]]* to the set of arguments $\mathcal{A}rg$, and we can connect this argument to the last parameter via the operation *arg*.

$\mathcal{A}rg$: { *[[Transition]], [[Port]], [[srcInport]], [[trgOutport]]* }
$arg([[CreateConnector]]_1)$ = *[[Transition]]*.
$arg([[CreateConnector]]_2)$ = *[[Port]]*.
$arg([[createConnector]]_3)$ = *arg([[src]])= [[srcInport]]*.
$arg([[createConnector]]_4)$ = *arg([[trg]])= [[trgOutport]]*.

After having checked whether the arguments comply to the method signature, we will bind the arguments to the corresponding elements from the method declaration (parameters, pattern objects).

[[Line]] belongs to the set of classes $\mathcal{C}$ and to the set of arguments $\mathcal{A}rg$ The func-

tion *arg* connects the parameter *[[connect extends Connector: MOFClass]]* to this argument. Thus, the premises of the rule `R_bind6` match the first argument. In addition, it respects the semantic constraint:
[[Line]] $\leq$ *[[Connector]] : true*
Therefore, we can substitute the argument *[[Line]]* for the parameter *[[connect]]*. Similarly, we can apply the rule `R_bind6` on the second parameter and its corresponding argument. As a result, we can substitute the argument *[[Port]]* for the parameter *[[connected]]*.

[[src]] belongs to the set of objects $\mathcal{O}$. There is a parameter *[[src:$connected]]*, where *[[connected]]* belongs to the set of parameters $\mathcal{P}a$ too. *[[Port]]* belongs to $\mathcal{C}$ as well as to the set of arguments $\mathcal{A}rg$. The parameter related to this argument is the second parameter $[[CreateConnector]]_2$, i.e. *[[connected]]*. The parameter related to the argument *[[srcInport]]* is the parameter *[[src:$connected]]*:
arg([[src:$connected]]) = *[[srcInport]]* $\in (\mathcal{O} \cap \mathcal{A}rg)$
The premises of the rule `R_bind3` match the third argument. In addition, the object *[[srcInport]]* is instance of a subclass of *[[Port]]*. Thus, the semantic constraint is fulfilled: *type([[srcInport]])* $\leq$ *[[Port]] : true*
The rule can be applied, and we can deduce the rule's conclusion, i.e. substitute the type of the object *[[srcInport]]* (from the method call) for the type of the object *[[src]]* (from the method declaration):
type ([[src]]) = *type([[srcInport]])*

In the same way, we can apply the rule `R_bind3` on the pattern object *[[trg]]* and the argument *[[trgOutport]]*. By applying this rule, we can substitute the type of the object *[[trgOutport]]* (from the method call) for the type of the object *[[trg]]* (from the method declaration):
type ([[trg]]) = *type([[trgOutport]])*

The unbound pattern object *[[srcToTrg]]* has a parameterized type *[[connect]]* which corresponds to the first parameter $[[CreateConnector]]_1$. This parameter is connected to the argument *[[Line]]* which is a class:
arg([[connect]]) = *[[Line]]* $\in (\mathcal{C} \cap \mathcal{A}rg)$
Thus, we can infer that *Line* is the type of *srcToTrg* which semantically means:
type([[srcToTrg]]) = *[[Line]]*

All matching rules with a semantic conclusion and whose semantic constraints are fulfilled have been applied. Thus, the semantic knowledge base is complete. Let us now consider the remaining matching rule `R_substitute2` . It matches the pattern objects *[[src]]* and *[[srcToTrg]]*, and the property *[[sourceConnector]]*. The semantic knowledge base contains namely the information
type([[srcToTrg]]) $\leq$ *type([[src]])* $\circ$ *[[sourceConnector]]*
In addition, after the argument values have been bound to the corresponding parameter, the semantic knowledge base contains these additional information:

type([[srcToTrg]]) = [[Line]]
type([[src]]) = type([[srcInport]]) = [[Inport]]
The property *sourceConnector* of the type of *src* and the class *Line* must have an inheritance relationship for the semantic constraint of `R_substitute2` to be fulfilled. This is not the case. The *sourceConnector* of the instance *srcInport* of *Inport* is an *UndefinedConnector*:
type([[src]]) ∘ [[sourceConnector]]
= [[Inport]] ∘ [[sourceConnector]] = [[UndefinedConnector]]
There is no inheritance relationship between *UndefinedConnector* and *Line*. This unfulfilled constraint is semantically expressed by:
[[UndefinedConnector]]≤[[Line]] ∨ [[Line]]≤[[UndefinedConnector]] : false
The rule `R_substitute2` matches the objects *[[src]]* and *[[srcToTrg]]*, and the property *[[sourceConnectableElement]]* too. The property *sourceConnectableElement* of the type of *srcToTrg* and the class *Inport* must have an inheritance relationship for the semantic constraint of `R_substitute2` to be fulfilled. This is not the case. The *sourceConnectableElement* of *srcToTrg* (which is of type *Line*), is a *Outport*:
type([[srcToTrg]]) ∘ [[sourceConnectableElement]]
= [[Line]] ∘ [[sourceConnectableElement]] = [[Outport]]
There is no inheritance relationship between *Inport* and *Outport*. This unfulfilled constraint is semantically expressed by:
[[Inport]]≤[[Outport]] ∨ [[Outport]]≤[[Inport]] : false
Similarly, the rule `R_substitute2` matches the objects *[[trg]]* and *[[srcToTrg]]*, with the property *[[targetConnectableElement]]*, as well as with the property *[[targetConnector]]*. In both cases, the semantic constraint cannot be fulfilled.

Finally, let us check the remaining constraints from the story diagram analysis. The type of *[[src]]* is *[[Inport]]*, the type of *[[trg]]* is *[[Outport]]*, and the type of *[[srcToTrg]]* is *[[Line]]*. The consequence is that the remaining constraint cannot be fulfilled:
type([[srcToTrg]]) ≤ type([[src]])∘[[sourceConnector]]
⇒ [[Line]] ≤ [[Inport]]∘[[sourceConnector]] : false
type([[srcToTrg]]) ≤ type([[trg]])∘[[targetConnector]]
⇒ [[Line]] ≤ [[Outport]]∘[[targetConnector]] : false

Conclusion:
All matching rules have been applied, and it remains unfulfilled constraints: type error detected.

This example of graph transformation and the detection of the use and misuse cases illustrate the combination of both contributions of our works: the extension of the SDM language combined with the type checking supported by our type system. We present in the next sections an evaluation and an outlook of both aspects of our work.

6.2 Extension of SDM: Evaluation and Outlook

We presented a generic feature for SDM consisting in the parameterization of elements to allow for the reusability of story diagrams. The other extension for SDM presented in this word is the reflective feature which aims at improving the expressiveness of story diagrams.
We propose in this section an outlook on additional features which also could beneficially extend the current SDM syntax: embedding of regular expressions, iteration over typed collections, etc. We evaluate the impact of the extended SDM-syntax on our application case, the MAAB guidelines, and consider the benefits and limitations of these additional SDM-features.

6.2.1 Additional SDM Features

The generic and reflective features presented in this work improve the graph transformations' reusability and expressiveness. These are not the only possible extensions for SDM. We present here additional features which could be useful for the specification of story diagrams.

Regular Expressions

As explained in Section 3.3.3, the checking of guidelines for naming conventions requires a mechanism which can analyze thoroughly strings, differentiate upper and lower case letters, etc. Regular expressions are the most suitable way to express rules such as the naming guidelines.

Let us consider the example of the guideline jc_0201 (Cf. Fig. 3.12). The regular expression representing the allowed pattern for a subsystem name according the guideline is: *[a-zA-Z]([_]?[a-zA-Z0-9]+)**.
Fig. 6.3 shows the implementation of this guideline with the integration of regular expression in the story diagram. The part a of Fig. 6.3 is the main method of the implementation of jc_0201, and has the model as start point. All the subsystem blocks in the root system of the model are iterated. Each matched subsystem block (*subsystemBlock*) is given as argument to the method *jc_0201(subSystemBlock : SubSystemBlock)* whose story diagram is represented in the part b of Fig. 6.3. The first activity shows the use of regular expression to check the name of *subSystemBlock*. The rest of the story diagram aims at pursuing the exploration of the model, and checking the pattern of the subsystem blocks recursively.
The integration of the regular expressions in SDM requires above all an adaptation of the generated code. MOFLON generates Java code, and Java provides a package called `java.util.regex` to support regular expressions. The classes *Pattern*, and *Matcher* belong to this package. A regular expression is compiled into an instance of the class *Pattern*. This pattern can then be used to create an instance of *Matcher* which performs a match operation on a String by interpreting the pattern.

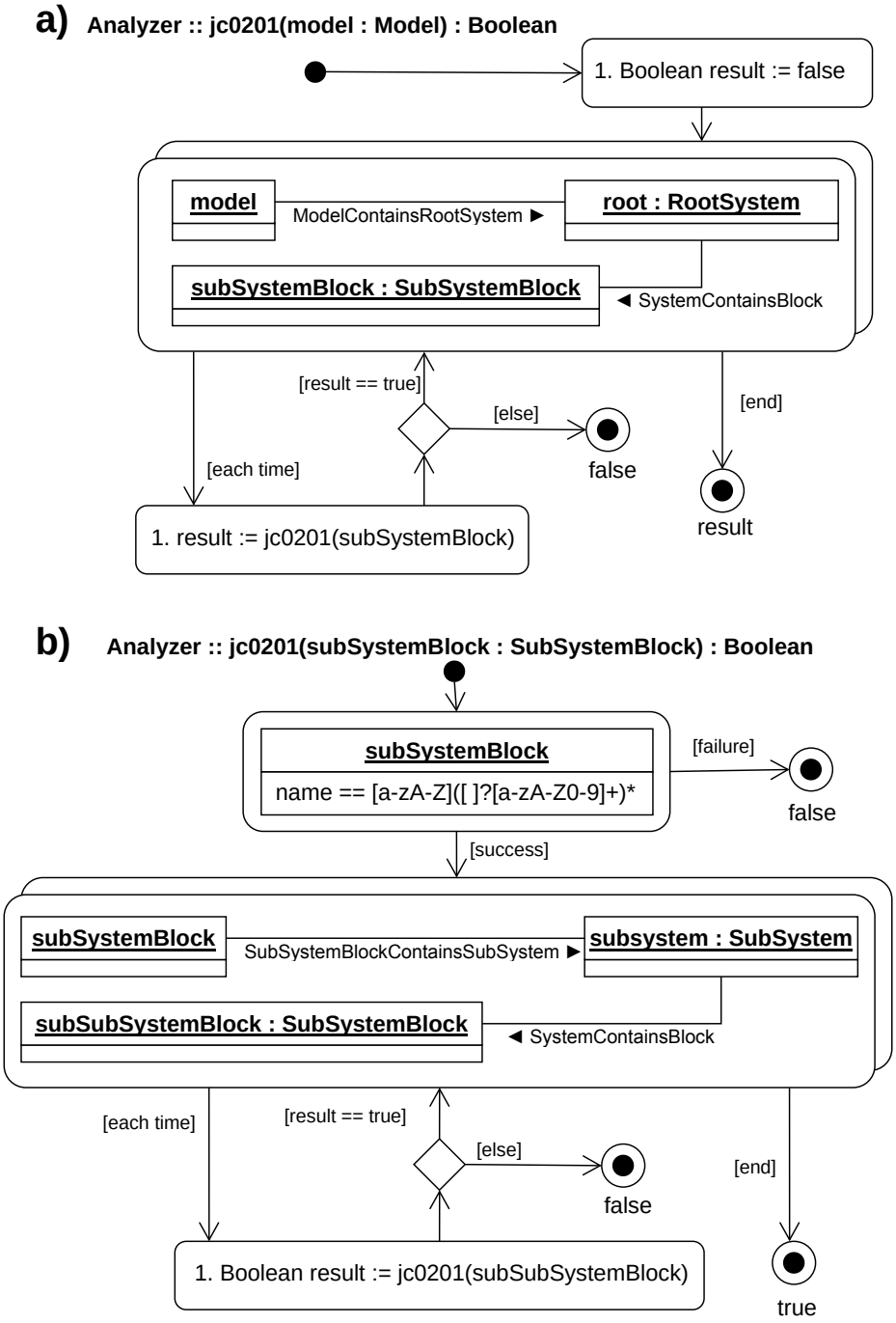


Figure 6.3: Guideline jc_0201 - Regular expression in SDM

A typical invocation sequence of the regular expression *a*b* on the String *aaaaab* is thus:

```
Pattern p = Pattern.compile("a*b");
Matcher m = p.matcher("aaaaab");
Boolean b = m.matches();
```

Or, in a condensed form:

```
Boolean b = Pattern.compile("a*b")
              .matcher("aaaaab").matches();
```

The integration of regular expressions in SDM would principally consists in adding an option to indicate that the value to be checked is a regular expression, and in adapting the Velocity templates to integrate the adequate Java code. Then, the code generated for the first activity of the story diagram of Fig. 6.3.b would be this code:

```
// attribute condition
JavaSDM.ensure(Pattern.compile("[a-zA-Z]([_]?[a-zA-Z0-9]+)*")
   .matcher(subSystemBlock.getName()).matches());
```

Iteration Activity over all Instances of a Class

Fig. 6.3 points out an aspect of SDM that should be improved. Although SDM is pretty well suited for the navigation through a model, the definition of story diagrams becomes more complex as soon as elements such as subsystems are nested. In the context of our application case, numerous guidelines apply on a given kind of element, and, hence, require the checking of all instances of a given class in a story diagram. A generic solution should provide a mechanism to express an iteration over all objects of a given class or type.

We introduce an additional feature in the form of a new kind of activity. This activity is parameterized with a class name, representing the collection of all instances of this type, and a user-defined variable as running variable for the iteration. We propose two variants of this new iteration activity: a variant labeled with "***for*** *iterVariable* ***ofClass*** *ClassName*", and a variant labeled with "***for*** *iterVariable* ***ofType*** *ClassName*", where *iterVariable* represents the iteration variable, and *ClassName* the type of the iterated object set.
As indicated in Section 3.3.3, SDM is understandable and easy to learn. A good way to preserve this plus consists of avoiding the definition of new keywords or formalisms, and trying to be similar to a standard or a well-known language. Thus, we chose the form of both iteration statements in reference to the *for each*-construct introduced in Java 5 [Jav]: *"**for** (typeName iterVariable **:** collName)"*, where *iterVariable* represents the iteration variable, and *collName* the iterated collection.
The iteration feature relates to two methods belonging to the JMI interface *Ref-*

Class: the method *refAllOfClass()*, and the method *refAllOfType()*. Both operations return the set of all instances of the class. The only difference between them is the following: *refAllOfType()* returns all the objects of the given class or one of its subclasses, whereas *refAllOfClass()* returns only the objects whose type is exactly the one of the given class. Thus, the keyword *ofClass* refers to the JMI method *refAllOfClass()*, and the keyword *ofType* refers to the JMI method *refAllOfType()*.

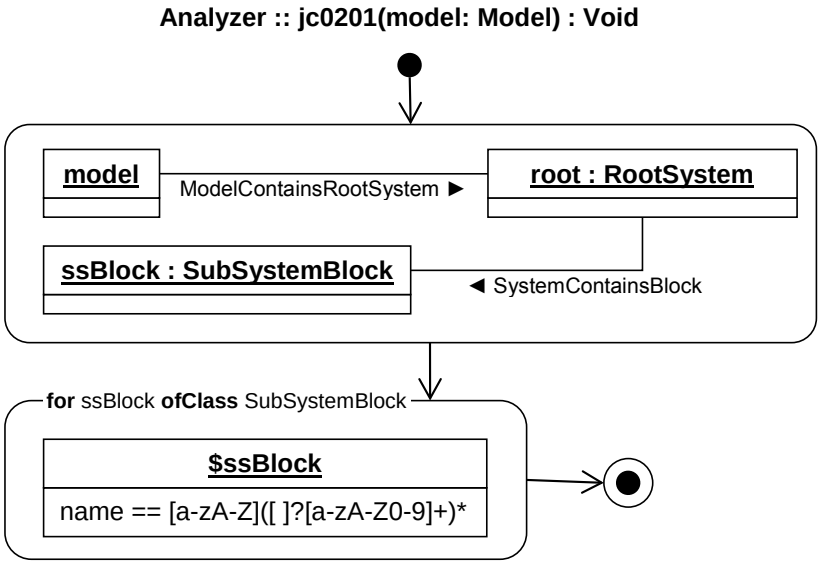


Figure 6.4: Guideline jc_0201 - Iteration over objects in SDM

Fig. 6.4 is another specification of the guideline jc_0201, such as in Fig. 6.3, but here using the iteration feature with the *ofClass*-variant. The first activity is a simple story activity, verifying is the Simulink model (parameter of the method *jc0201(model:Model)*) contains at least one subsystem block *ssBlock*. The second activity is labeled with "***for*** *ssBlock* ***ofClass*** *SubSystemBlock*". This activity contains a bound object named *ssBlock* whose attribute name is compared to the regular expression which represents the correct name pattern for a subsystem block. All instances of *SubSystemBlock* are iterated, each element of this set is bound to *ssBlock* and, then, may be used as such in the activity. This means that the activity is executed as many time as the number of instances of *SubSystemBlock*. In other words, the name of all subsystem blocks of the model may be checked in the single activity without caring about where they are in the model and how they are nested. Though, only one condition must be fulfilled in order to use the iteration over a class's instances: an instance of this class must be bound, with the same name as the iteration variable. Similarly to the Java *for each*-statement where the iterated collection must be a bound variable, we need a bound object to obtain the object set to be iterated.

The generated code for this activity would be:

```
// iterate ofClass activity
Collection<SubSystemBlock> ssBlockColl = ssBlock.refClass()
                                          .refAllOfClass();
for(SubSystemBlock ssBlock :  SubSystemBlock){
   // attribute condition
   ......................
}
```

Iteration Activity over Typed Collection

Not only the iteration over the instance of a class would be useful for an efficient and expressive specification of model transformation with SDM, but more generally the iteration over typed collections. The generic specification of the guideline na_0004 depicted in Fig. 4.7 could be improved by means of an iteration over typed collections. For instance, instead of calling a method to check and correct each Boolean property, we could define a collection of properties to be checked and call the method only once.

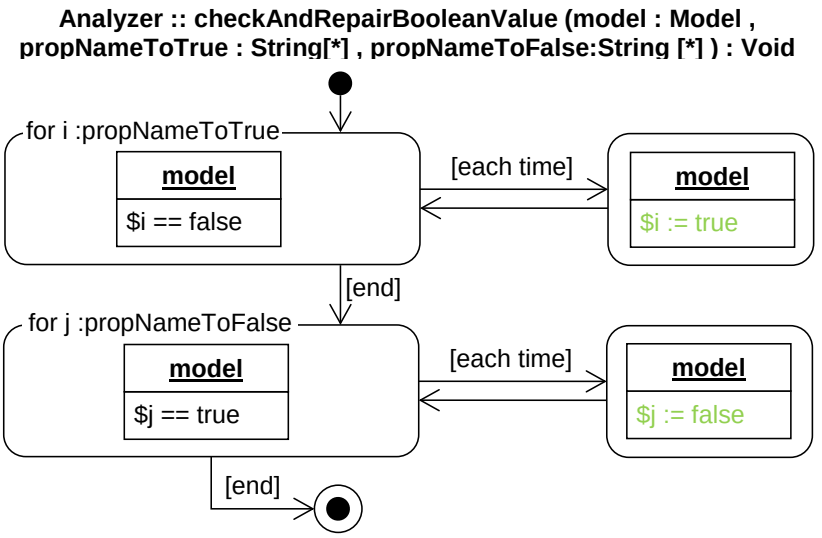


Figure 6.5: Generic Method with Iteration over Collections in SDM

This concept is illustrated by the Fig.6.5.
The parameter *propNameToTrue* is a String collection containing the names of the properties whose value must be checked and, if necessary, set to *true*. Similarly, the parameter *propNameToFalse* is a String collection containing the name of the properties whose value must be checked and, if necessary, set to *false*. The new method *checkAndRepairBooleanValue* in Fig. 6.5 aims at replacing the multiple calls of the same method in Fig. 4.7. Thus, the String collection *propNameToTrue*

contains *('statusBar', 'toolBar', 'showTestPointIcons', 'showLinearizationAnnotation', 'showViewerIcons', 'wideLines')*. The other parameter *propNameToFalse* is a String collection containing *('sortedOrder', 'modelBrowserVisibility', 'executionContextIcon', 'showModelReferenceBlockIO', 'sampleTimeColors', 'showPortDataTypes', 'showLineDimensions', 'showStorageClass')*.
In the first activity, the bound object *model* is matched. We introduce here a new kind of activity which is equipped with the typed collection as parameter and a user-defined variable as running variable for the iteration of the collection. At the top of the activity, a running variable *i* is defined as part of the expression *for i : propNameToTrue*. This means that the collection after the colon is iterated, and the actual object of the iteration (here: the property's name) is bound to the variable. The related activity is executed for each single iteration. Thus, for each property name in the collection, we check whether this property's value is set to *false*. If the pattern is matched, the transition *[each time]* is traversed and the repair action, i.e. setting the value to *true*, is executed. Then, we go back to the previous activity which is executed with the next property name of the collection *propNameToTrue*. When all the collection's elements have been iterated, the transition *[end]* is traversed. The next activity which is equipped with the collection *propNameToFalse* and the running variable *j* is executed in the same way.

As illustrated by this example, this new kind of activity can reduce the effort in the diagram specification since we do not have to define a new call for each property name. In addition, it makes easier the guidelines' maintenance. For instance, if the guideline na_0004 is updated and the list of the properties is modified, we only have to change the values in the collection instead of searching and edit each method call.

Iteration Activity combined with Foreach-Activity

Both kinds of iteration that we introduced (iteration over all instances of a class and iteration over a typed class) can be combined with each other as well as with the already existing foreach-activity of the SDM syntax. We introduce an extension of the SDM syntax in order to an expressive and compact notation which combines these iterations.

Fig 6.6 and Fig 6.7 illustrate this concept by means of four simple examples and their interpretation. Although the four SDM diagrams in Fig 6.6 look pretty similar, they do not have the same semantics.
The first SDM diagram of Fig 6.6.a shows the combination of a foreach-activity with the iteration over a Integer collection called *numbers*. The second SDM diagram is the equi-valent of the first diagram. We first iterate over the collection *numbers*. Then, for each integer element from the collection *numbers* that is bound to the running variable *i*, not only one but all matches of the graph pattern are processed, which is represented by the *foreach*-activity containing the pattern.

The first SDM diagram of Fig 6.6.b shows the combination of a foreach-activity with the iteration over all instances of the class *Block*. The second SDM diagram is the equivalent of the first diagram, composed of the iteration over the instance of *Block* and a for each-activity containing a pattern. Similarly to the part a of Fig 6.6, for each instance of *Block*, we don't only want to process a single but all matches of the pattern.

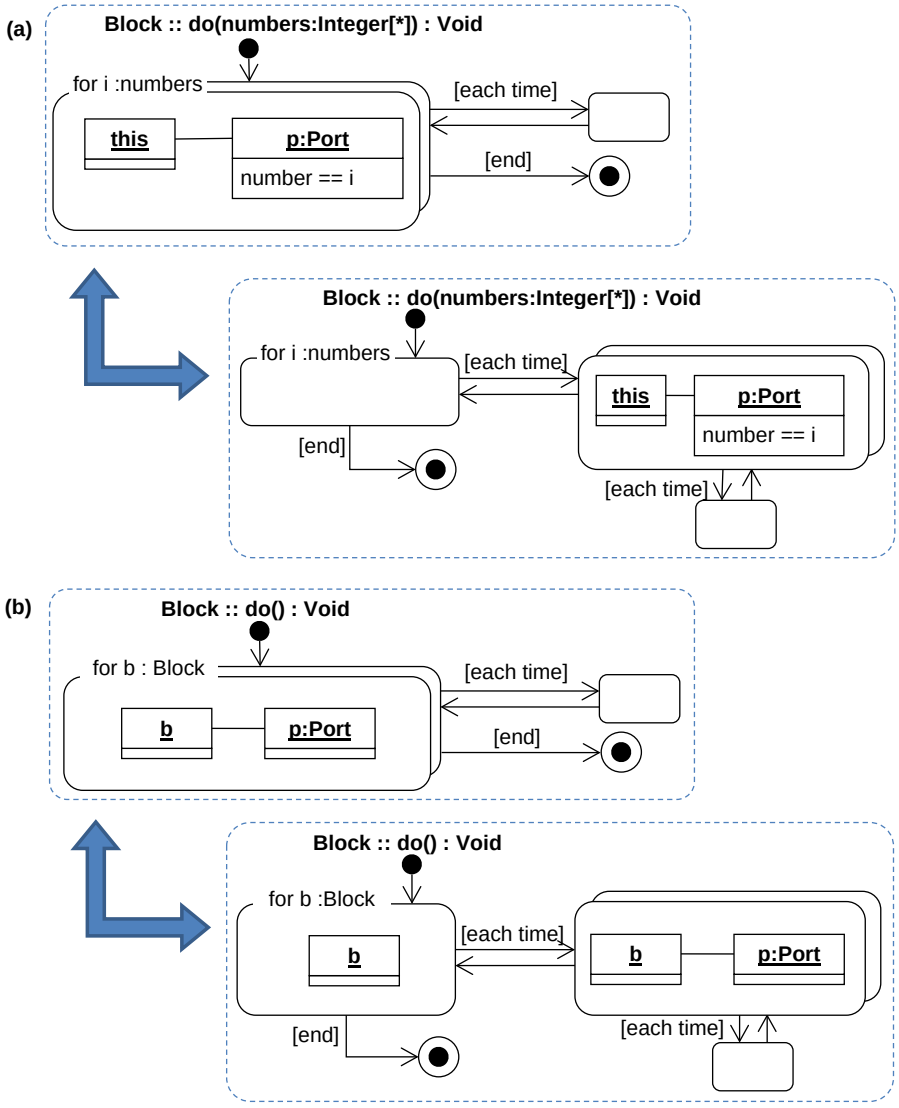


Figure 6.6: Iteration Activity combined with *foreach*-Activity 1/2

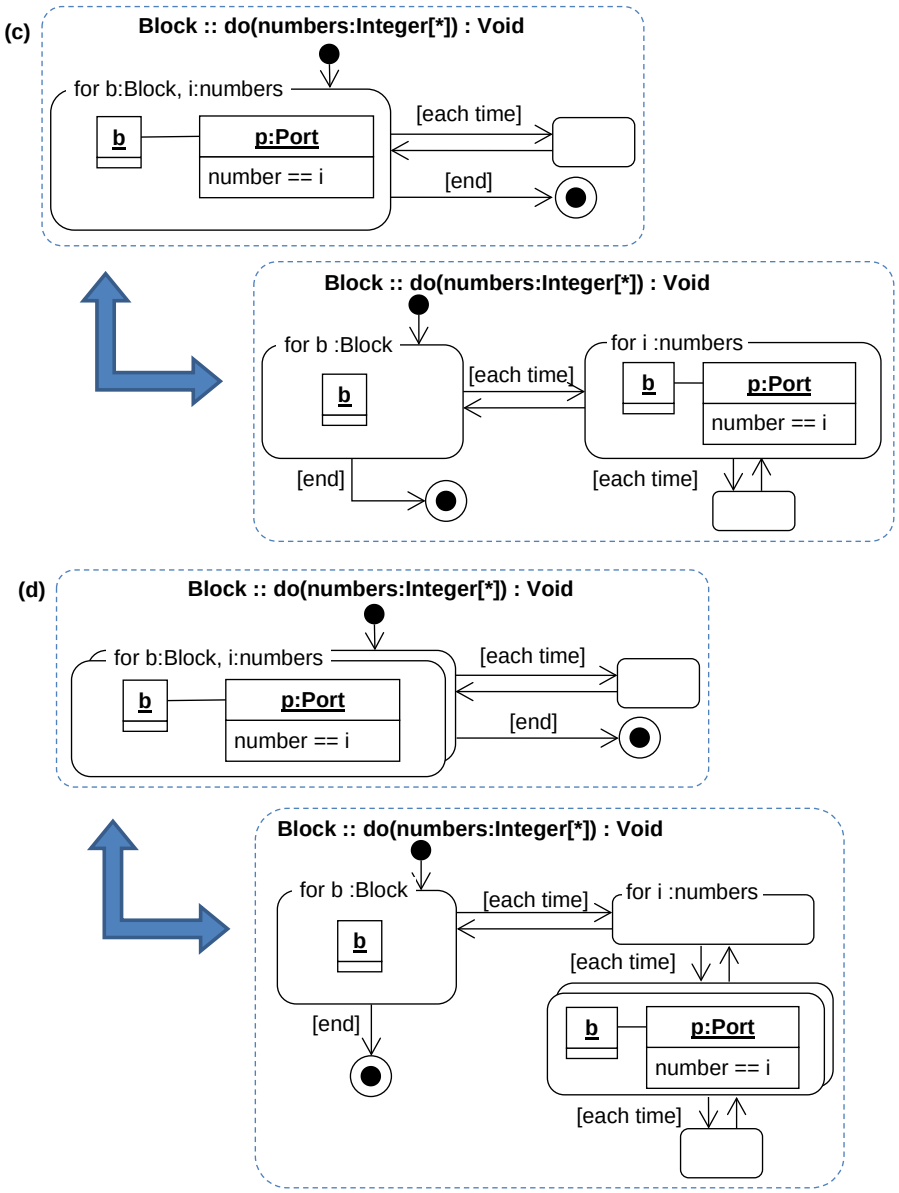


Figure 6.7: Iteration Activity combined with *foreach*-Activity 2/2

The notation in the first SDM diagram of Fig 6.7.c is a short-hand for an iteration with two nested loops: an iteration over the instances of the class *Block* and an iteration over the Integer collection called *numbers*. The second SDM diagram illustrates the execution of theses loops. The external loop consists in the iteration over the instances of *Block*. For each instance of *Block*, we iterate over the collection *numbers* to match the pattern contained in the activity.

Finally, the first SDM diagram in Fig 6.7.d is a combination of a *foreach*-activity and two nested loops: iteration over the class *Block* and iteration over the collection *numbers*. The interpretation of this compact notation is depicted in the second part of Fig 6.7.d . The external loop consists in the iteration over the instances of *Block*. For each instance of *Block*, we execute the internal loop, i.e. over the collection *numbers*, to process all pattern matches. In other words, for each integer element from the collection *numbers* that is bound to the running variable *i*, a *foreach*-activity containing the pattern to be matched is executed.

The main drawback of visual graph transformation is the effort in drawing the diagrams and managing the layouts. As shown by each second diagram in Fig 6.6 and Fig 6.7, these new kinds of activities allow for the specification of compact and, yet, expressive graph transformations.

6.2.2 Evaluation

We introduced the MATE/MAJA project in Section 3.1 and the checking of the MAAB guidelines as application for our SDM extension. We present here an evaluation of the extended SDM language and their impact on the MAAB guidelines, before we consider the benefits and limitations of the extended SDM-syntax.

Application on the MAAB Guidelines

Most of guidelines apply on a complete category of MATLAB elements. For instance, the guideline db_0081 requires that we inspect all inports and outports to ensure that none is unconnected and, if necessary, fix it. When considering the MATLAB metamodel, this means that it is necessary to explore the metamodel, and to apply the guideline on each systems and nested systems.

Fig.6.8 shows how a MATLAB model can be explored using the old SDM syntax. The part a of Fig.6.8 is the first method with an instance of the class *Model* as parameter (*model: MODEL*) and, if necessary, additional parameters (*additionalParam*) required by the guideline specification. A boolean value called *result* is set to *true*. This variable indicates whether a guideline is respected or not: in the case of guideline violation, it is set to *false*. This method matches the model's root system *root*. Then, it calls the second method depicted by the part b of Fig.6.8. The parameters of this method are an instance of *System*, a Boolean corresponding to the value *result* and, if there are, the *additionalParam*) from the first method. The first story pattern concerns the guideline's specification itself. The following story patterns of Fig.6.8.b aim at pursuing the model exploration in order to match other systems and to apply the guideline's specification. This is specified by means of a *for-each* activity which matches all contained subsystem blocks *ssb* and the corresponding subsystems *ss*. The second method is applied recursively for each subsystem which is matched.

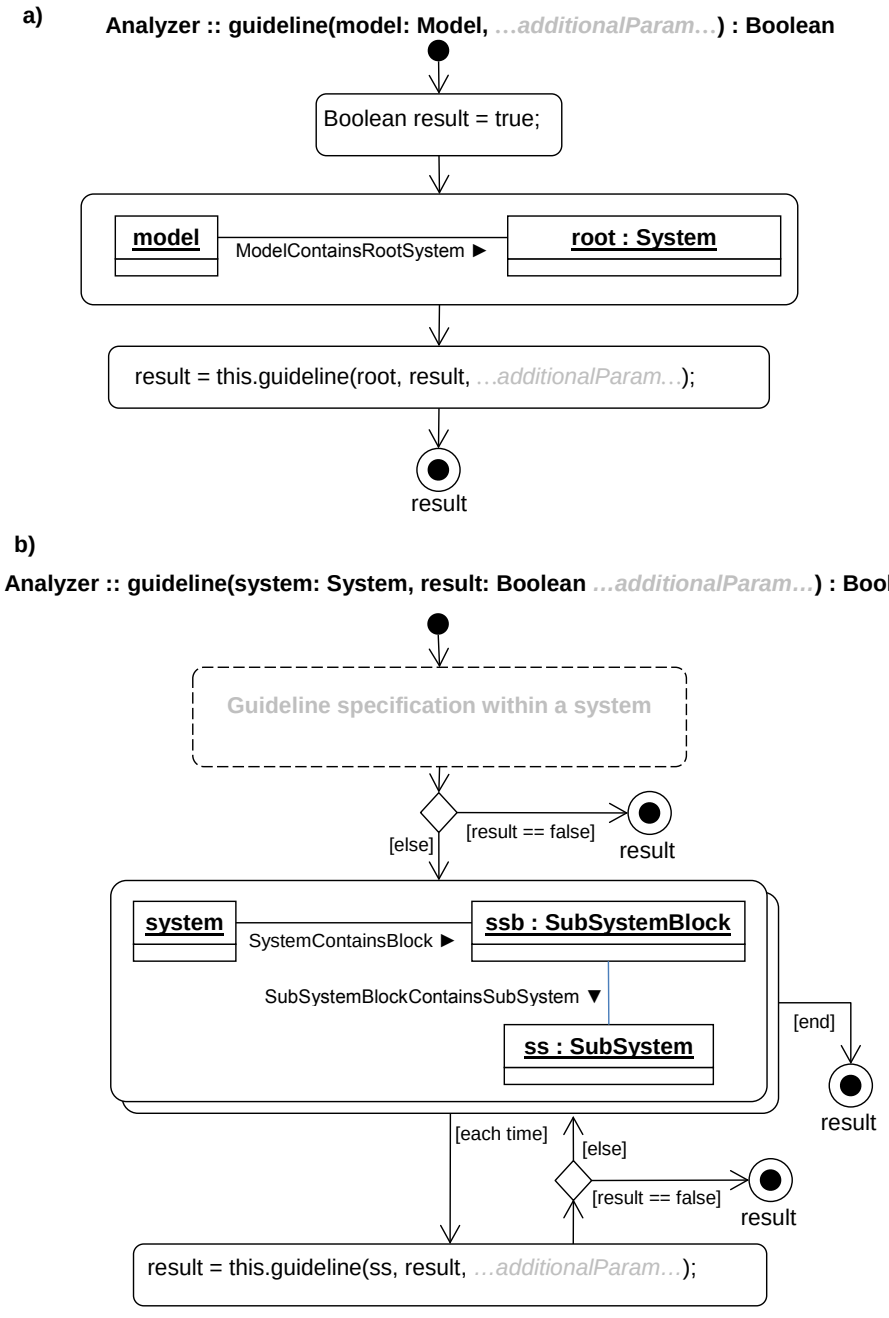


Figure 6.8: Model exploration - Old SDM Syntax

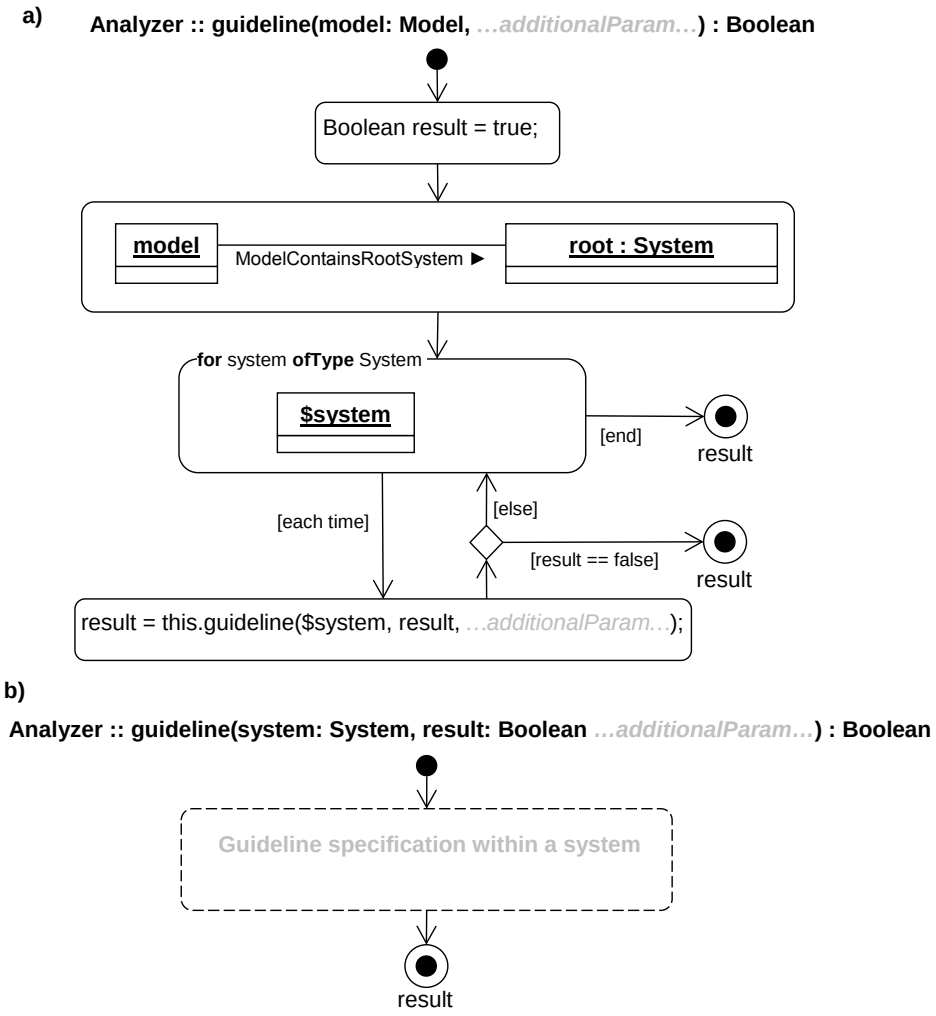


Figure 6.9: Model exploration - New SDM Syntax

Fig.6.9 is the counterpart of Fig.6.8. It is defined by means of the extended SDM syntax, more precisely by means of the iteration over all instances of the class *System* as depicted in the part a of Fig.6.9. The reader must notice that we are using *ofType System* instead of *ofClass System* so that we can match not only the instances of the classe *System*, but also of its subclasses (*RootSystem*, *SubSystem*). Fig.6.9.b depicts the method called for each matched instance *$system*, and dedicated to the guideline's specification.

This comparison shows an advantage of the extended SDM syntax for the analysis of MATLAB models according to the MAAB guideline. An extension of the SDM language, the iteration over the instance of a given type, allows for the sep-

aration of concerns: on the one hand, the model's exploration (via the first story diagram), and on the other hand, the actual guideline's specification (via the second story diagram). This modularization provides an effort reduction in drawing the diagrams as well as an easier maintenance.
In the case of Fig.6.8, each guideline does not only require the guideline's specification itself, but also the diagram's part dedicated to the model's exploration. The new version depicted by Fig.6.9 reduces this effort. Since the first story diagram is dedicated to the model's exploration and the second one to the actual guideline's specification, we can use the first story diagram for several guidelines. In this case, we only have to add the corresponding method calls and complete the parameters' list. Similarly, we only have to update the list of parameters and to adapt the guideline's specification in the case of guideline modification. In the case of guideline creation/deletion, we can add/remove a method call in the first story diagram, update the parameters' list and create/remove a story diagram accordingly.

We could use the first method to call several guideline analysis at once. Though, we will not do it to avoid a too long list of parameter, difficult to maintain in case of modification of the guidelines and/or of the story diagrams. We must namely find a balance between reducing the effort in drawing diagrams and reducing the effort in maintaining the method calls' consistency.

The MAAB guidelines consists in the following guidelines categories: 1) Software environment, 2) Naming convention, 3) Model Architecture, 4) J-MAAB Model Architecture Decomposition, 5) Model Configuration Options and 6) Simulink. This corresponds to 66 guidelines.
Unfortunately, some guidelines cannot be specified with help of the MATLAB metamodel presented in this work because they concerns aspects which are not defined in this metamodel. For instance, guidelines such as *db_0042*, *jm_0002* or *db_0032* cannot be specified because they imply aspects which are not part of our MATLAB metamodel: layout informations, element's size and relative position, etc. Similarly, guidelines such as the guidelines *na_0026*, *ar_0001* or *ar_0002* cannot be specified either because they require informations about file names, software version, etc.
Let us consider the guidelines which can be specified with the MATLAB metamodel presented in this work. The SDM language is Turing-complete, i.e. any computable function can be specified using SDM. Though, some functions require the use of the Java statements in order to support specific operations, e.g. the use of regular expressions. This is the case of the guidelines about naming convention. Other functions can be specified using story patterns, but require a too long and too complex story diagram (or numerous smaller story diagrams).

Fig.6.10 shows the part *guideline specification* of Fig.6.8, i.e. using the old SDM syntax, in the case of the guideline *jc_0281*. The guidelines *jc_0281* concerns the naming of trigger or enable blocks. More precisely, the block name should match

the name of the signal triggering the subsystem which contains the trigger or enable block. Please note these story diagram only specify the model analysis, not the model correction. This guideline does not require any additional parameter, except the instance of System *system* and the Boolean *result*. Each subsystem block *ssblock* contained in *system* inspected. This guideline specification is composed of two very similar patterns: the instances of *EnableBlock* and *TriggerBlock* are the only difference between both parts of the story diagram.

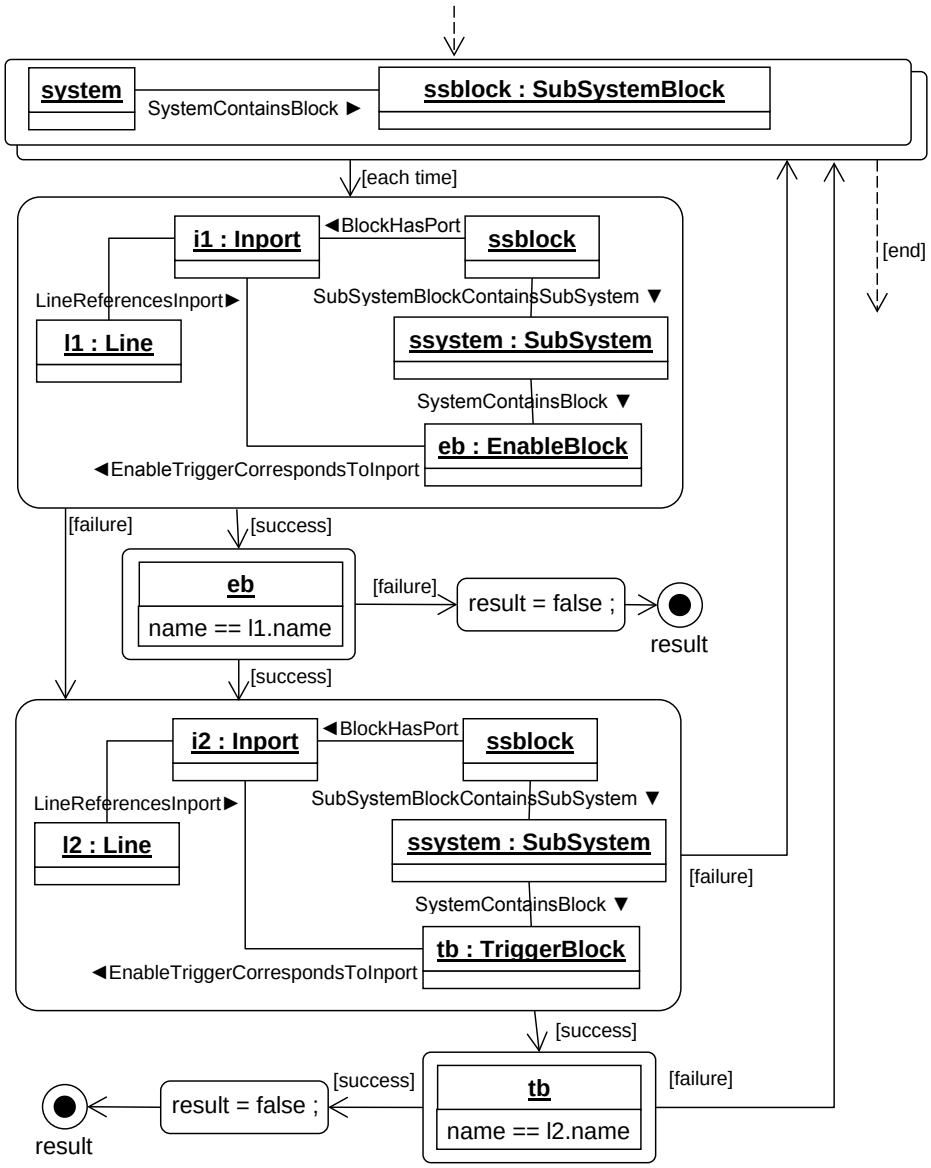


Figure 6.10: MAAB Guidelines jc_0281 - Old SDM syntax

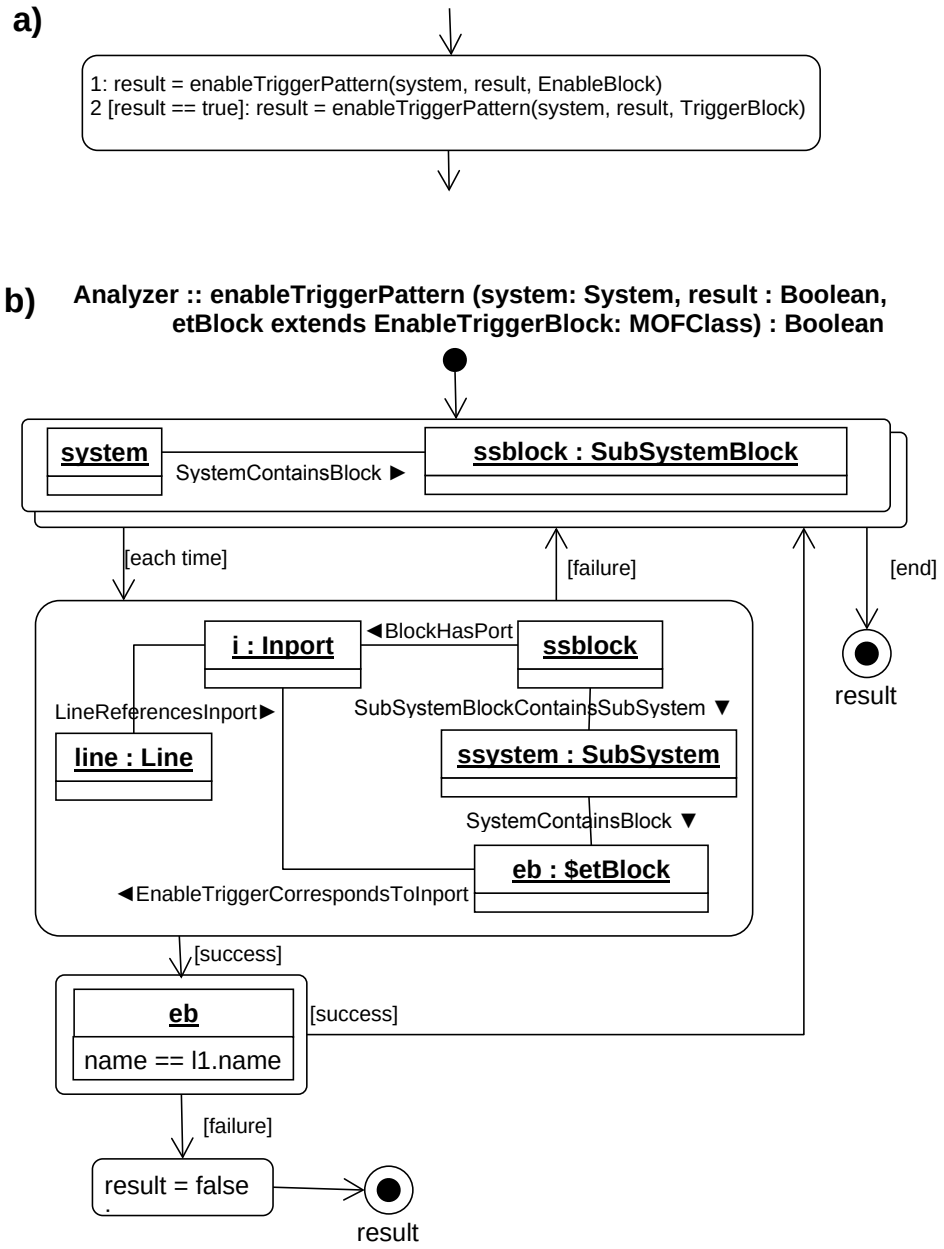


Figure 6.11: MAAB Guidelines jc_0281 - New SDM syntax

Consequently, the use of the generic feature would make sense for the specification of this guideline. This is illustrated by Fig.6.11. The part a of Fig.6.11 consists of the calls of the method *enableTriggerPattern(system:System, result:Boolean, etBlock:MOFClass)* which is depicted by Fig.6.11.b. This method corresponds to the pattern we want to reuse. The parameter *etBlock* allows for choosing whether we

check the name of an enable block or of a trigger block. We call this method twice: first with the class *EnableBlock*, then with the classe *TriggerBlock*. This avoid to draw twice almost the same diagram, and, thus, reduces the effort in drawing diagrams.

Fig.6.12 illustrates another benefit of the extended SDM syntax for the MAAB guidelines. Two extensions allows for a compact and simple specification of the guideline *jc_0231*. This guideline defines a name convention which applies for all blocks, except the subsystem blocks. Fig.6.12 is the corresponding story diagram. The use of regular expression allows for checking the name convention. The regular expression is defined here are String parameter so that it can be easily changed in the case of modification of the guideline *jc_0231*. The reflective feature allows for ensuring that the matched block b is not an instance of *SubSystemBlock* and, thus, that the guideline can apply. This is realized by checking the name attribute of the MOFClass the matched block is instance of.

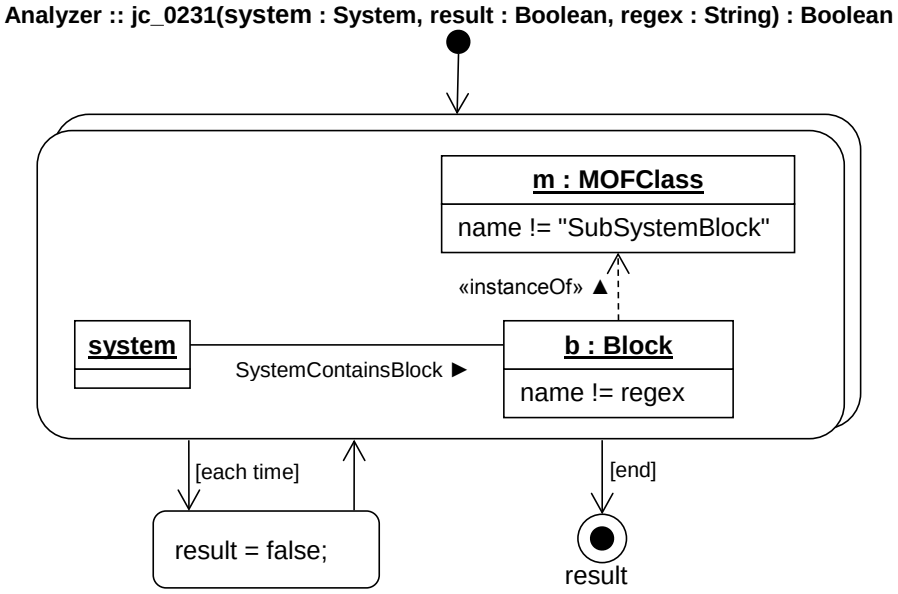


Figure 6.12: MAAB Guidelines - jc_0231

Numerous guidelines such as the guideline *na_0004* consists in checking setting values. Fig.6.13 illustrates the benefits of the extended SDM syntax for the MAAB guidelines. The guideline *na_0004* requires to check the value of 13 model settings, i.e. the value of 13 attributes of the class *Model*.

The part a of Fig.6.13 shows the guideline specification with the old SDM syntax (more precisely, only 5 settings for a question of space). If we want to correct invalid value, we need 2 story pattern for each setting: one story pattern to match

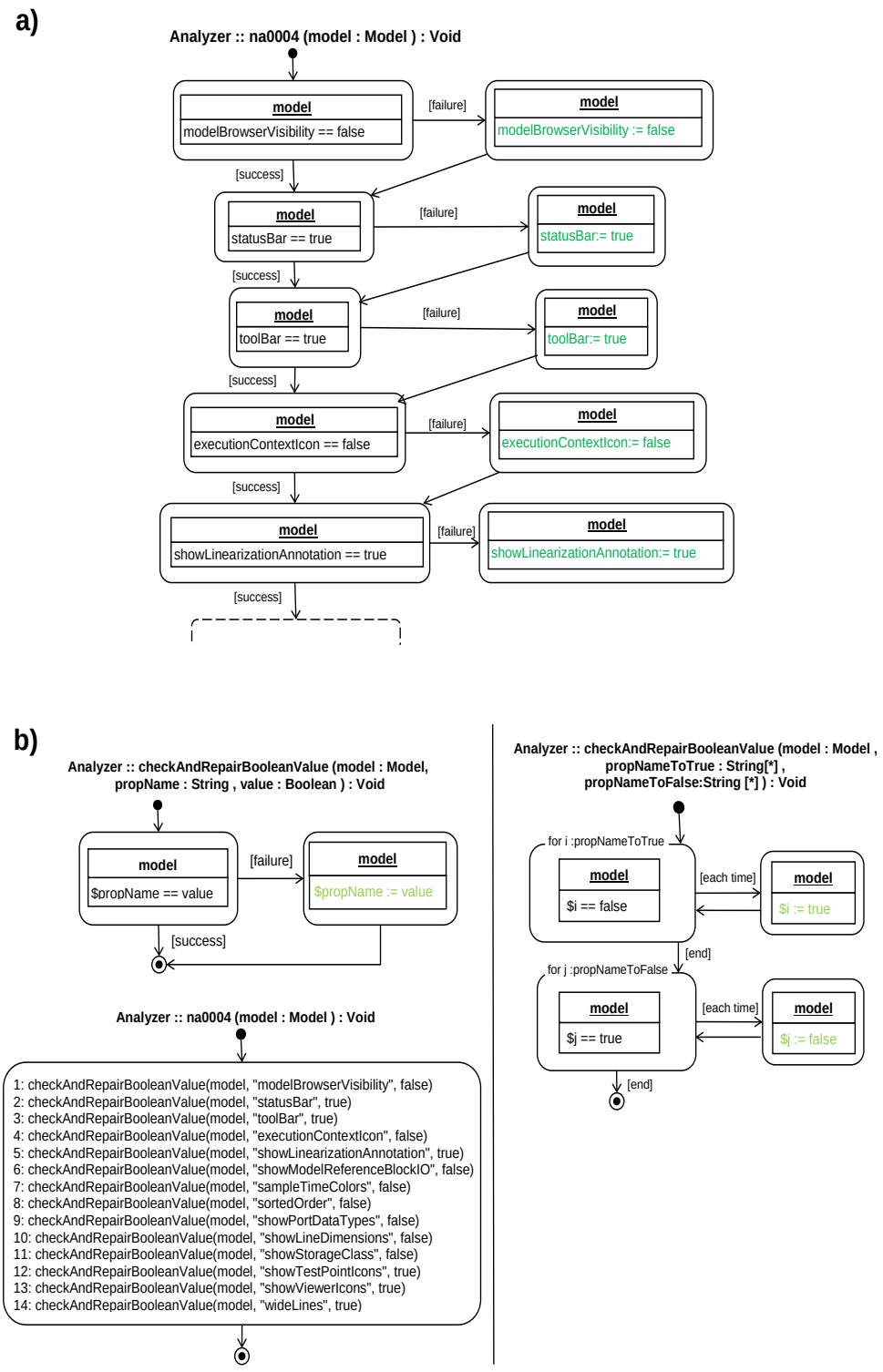


Figure 6.13: Guideline na_0004 - Generic versions

and check the value and, if necessary, one story pattern to correct the seting's value. As shown by the left part of Fig.6.13.b, the generic SDM feature allows for a more compact specification by parameterizing the attribute of the class *Model* as String parameter and reusing the pattern for each setting to be checked. If we use the iteration over a collection, we can make the guideline more compact as depicted by the right side of Fig.6.13.b: we do not need to call the story diagram for each setting,but we can give the list of settings to be checked as parameter (*propNameToTrue* for the settings whose value must be *true* and *propNameToFalse* for the settings whose value must be *false*)
The guideline specification is not only more compact using the extended SDM, it is also easier to maintain. If the guideline is modified after a new release of the MAAB catalog (additional settings, or settings which had to be *true* and now must be *false*), we only have to modify the method call, without having to modify the story diagram itself. Thus, the user does not need to check the method calls and to find which ones must be deleted/added, but only to update the parameter collections.

Benefits and Limitations

We did not modify but only extend the existing SDM language with our generic and reflective feature. This extension preserves the "flavour" of SDM. For instance, the generic pattern objects and the non-generic pattern objects differ only in the use of the $-symbol. The format of the parameterized classes or properties is the same as the other parameters: *"parameterName : parameterType"*.
Finally, the notation for type restriction of parameterized classes has been designed to be as intuitive as possible for the user. The keyword *"extends"* refers to the Java programming language which is well-known in the domain of model-driven engineering since it is an object-oriented language. The reserved word *"extends"* in Java is used in a class declaration in order to set the relationship between a parent class and a child class. For instance, when declaring a class *ClassB* which must specialize a class *ClassA*, the class declaration of *ClassB* starts whith "`public class ClassB extends ClassA`"
As shown by the example, the generic features allows for the reuse of story diagram, and, hence, reduces the effort in drawing diagrams and improving diagram layout. In addition, this improvement is reflected in a size reduction of the generated code. A block of code is generated for each story pattern. The mapping between reflective and tailored interfaces (Cf. Fig.4.1) results in a slight difference between the code generated from a generic story pattern and the code generated from a non-generic story pattern. The reuse of graph patterns due to the specification of generic model transformations results in a decreased number of story patterns, and, hence, in a reduced number of generated blocks of code.
We present in this work an extension for SDM. This approach can be adapted to other visual languages as long as the code generated from the model transformation provides reflective interfaces (similarly to the JMI reflective interfaces)

Our approach has some limitations despite its benefits and possible language extensions. Even if the extended syntax is quite understandable, the specification of story diagrams using the new features in order to optimize the transformations is not always intuitive and requires more reflexion from the user. Another limitation is the risk of type errors which can occur when using the generic features. Though, the risk of type errors was precisely the motivation for the specification of a type system for SDM.

6.3 Type System: Evaluation and Outlook

We defined in this work a type system for SDM which allows for type checking of method specifications and method calls.
We first propose in this section a classification of our type system according to the criteria described in Section 2.5. Then, we describe how we can evaluate our type system, illustrated by the example of the type checking of attributes. Finally, we describe the pros and cons of our type system as well as an outlook.

6.3.1 Classification

According to the definitions presented in Section 2.5, we can qualify our approach as follows. MOSL is a typed language which supports universal polymorphism, i.e. subtypes and parametric polymorphism. The types must be declared, thus, the typing of MOSL is manifest and consists in type reconstruction (instead of type inference as in the case of latent typing). Our type system defines a strong and nominative typing. It does not allow for type conversion or add-hoc polymorphism. On the contrary, it determines typing rules and type constraints.
As described in Section 5.4, our type checking approach is executed as follows: we first check and extract information from the metamodel, then we check the methods' signature and the story diagrams, and complete the semantic knowledge base with the derived information. Finally, the rules of inferences defined on the method calls (Section 5.3.4) associate type information from the metamodel and the methods' signature and story diagram to the method call's arguments. This is static typing.

6.3.2 Evaluation by means of Use and Misuse Cases

The SDM specification of many guidelines such as the guidelines for name convention (e.g. `jc_0201`, `jc_0211`) or model settings (e.g. `jc_0021`) requires the checking of attributes. Type checking of attributes, i.e. properties owned by classes, is therefore part of the type system presented in this work.
Let us show that our type system allows for ensuring the correct specification of the checking of attributes, and,thus, for detecting semantically incorrect specifications.

In order to evaluate our type system for the attributes, we generate use and misuse cases, and apply our approach on these cases. We limit the context to the simple metamodel excerpt of Fig. 6.14 composed of the class *Block* and its subclass *SubSystemBlock*, the class *System* and its subclasses *SubSystem* and *RootSystem*, as well as the datatypes *Boolean* and *String*. The class *Block* has the attribute *showName* which is of type *Boolean*, and the class *System* has the attribute *screenColor* which is of type *String*. This metamodel excerpt allows to test the type system for the attributes. It contains two different datatypes as well as inherited properties.

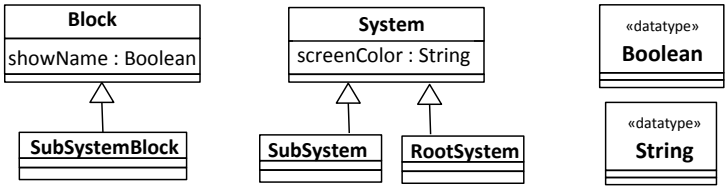


Figure 6.14: Metamodel Excerpt

Type errors can occur in the specification of the story diagram or in the method call, when an attribute does not belong to a class and/or when the type of the attribute does not correspond to the type of the value. Therefore, we consider error-free as well as erroneous SDM specifications. The specification is pretty simple: a story pattern composed of an object and the checking (operator =) of the attribute's value. We consider the different features and their combinations: bound or unbound object, parameterized or non-parameterized class, parameterized or non-parameterized property. Fig.6.15 shows the error-free specifications and the combination of features.

An attribute's value can be specified in different ways: as concrete value, parameterized value, composition (e.g. attribute of another object). Though, only the value's datatype, and not the way the value is computed, is relevant here. Therefore, the attribute's datatype and its concrete value are parameterized.
In the case of error-free SDM specifications, we consider use and misuse cases, i.e. method calls which cause errors or not when executed, and show how the errors can be detected by means of the type system's rules of inference.
In the context of attribute checking within a story pattern, a type error may have two causes: the attribute does not belong to the class the object is an instance of, or the attribute's type does not correspond to the attribute's value. There can be two causes for a misuse case as method call: either the arguments do not respect the method's signature, or the arguments cause a type error as described above. This second case occurs if the class and/or the property are parameterized.

Fig. 6.15 shows the variants of the error-free SDM specifications. The parts a, b and c of Fig. 6.15 illustrate the case of a non-parameterized property (*show-*

Name). Fig. 6.15.a corresponds to the specification composed of a bound object (*obj*) with the signature *m(exp:Boolean, obj:Block)*. Fig. 6.15.b depicts the specification using an unbound object of type *Block* (*obj:Block*) with the signature *m(exp:Boolean)*. Fig. 6.15.c depicts the specification composed of an unbound object whose type is a parameterized class (*obj:$cname*) with the signature *m(exp:Boolean, cname extends Block : MOFClass)*. Fig.6.15.d and e correspond to the case of a parameterized property (*$pname*). The former corresponds to the specification using an unbound object of type *Block* (*obj:Block*) with the signature *m(exp:Boolean, pname:MOFProperty)*. The latter corresponds to the specification composed of an unbound object whose type is a parameterized class (*obj:$cname*) with the signature *m(cname extends Block : MOFClass, exp:Boolean, pname:MOFProperty)*.

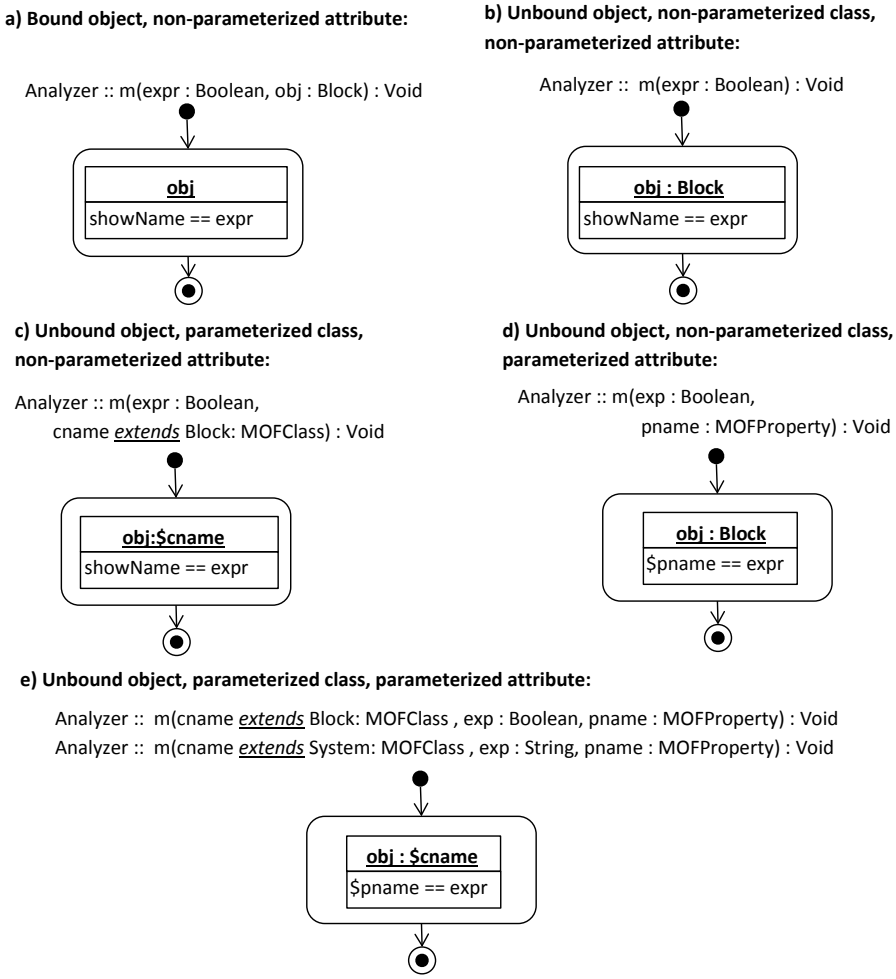


Figure 6.15: Checking Attributes - Story Diagrams

Fig.6.15.a, b and c have in each case 2 erroneous counterparts: on the one hand, the case where the attribute does not belong to the class *Block*, and on the other hand, the case where the value's datatype does not correspond to the attribute's datatype. In the case of the error-free specifications depicted by Fig.6.15.d and e, we cannot check whether the attribute belong to the class or whether the value's datatype correspond to the attribute's datatype since this property is parameterized. Consequently, an error can be detected only when the method is called, i.e. when the property is known.
The type error caused by the attribute not belonging to the class *Block* can be detected by means of the rule of inference `R_prop8`. Fig.6.16.a illustrates this kind of type error. When applying `R_prop8`, we obtain the constraint *type([[obj]]) ∘ [[screenColor]]*. This constraint cannot be fulfilled according to the semantic knowledge base because *[[screenColor]]* does not belong to the type of *[[obj]]*, i.e. the class *Block*.
The type error caused by the value's datatype not corresponding to the attribute's datatype can be detected by the rule of inference`R_prop4`. Fig.6.16.b illustrates such a case. When applying `R_prop4`, we obtain the constraint *type([[obj]]) ∘ [[showName]] = dType([[exp]])*. This constraint cannot be fulfilled according to the semantic knowledge base because *type([[obj]]) ∘ [[showName]]* equals *[[Boolean]]* whereas *dType([[exp]])* equals *[[String]]*.
The details of the type checking can be found in the Appendix D.

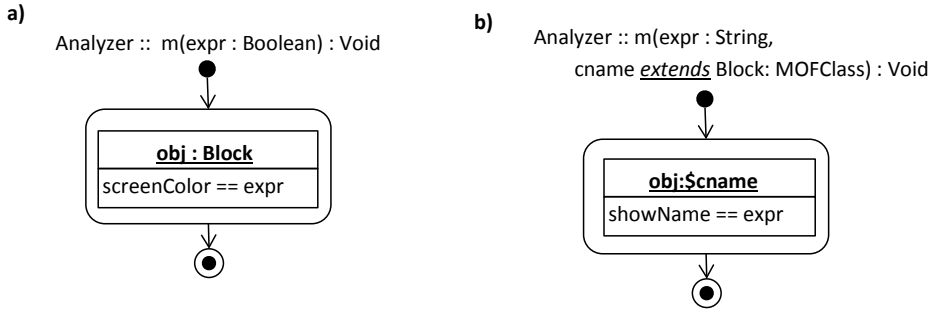


Figure 6.16: Checking Attributes - Erroneous Specifications

For each error-free specification, we consider the use and misuse cases. In the case of the error-free specifications depicted by Fig.6.15.a, b and c (i.e. without parameterized property), a misuse-case corresponds to a method call which does not respect the method's signature. In the case of the error-free specifications depicted by Fig.6.15.d and 2, there are two kinds of misuse-case. On the one hand, a method call which does not respect the method's signature. On the other hand, a method call with a property as argument which does not belong to the object's class or whose type does not comply to the value's data type, and, consequently, causes a type error when the method call is executed.

A misuse-case where the method call does not respect the method's signature can be detected by the rule of inference R_callx (x = 1 to 6). For instance, *m(true, systemObj)* where *type([[systemObj]]) ≤ [[System]]* is such a misuse case for the specification depicted by Fig.6.15.a. The erroneous argument is *systemObj*, and this error is detected by R_call2 whose semantic constraint *type([[systemObj]]) ≤ [[Block]]* cannot be fulfilled.
A misuse-case where the property given as argument does not belong to the object's class can be detected by means of the rule of inference R_bind7. For instance, *m(true, screenColor)* is such a misuse case for the specification depicted by Fig.6.15.d. The erroneous argument is *screenColor*, and this error is detected by R_bind7 whose semantic constraint *type([[obj]])∘[[screenColor]]* cannot be fulfilled.
A misuse-case where the attribute's datatype does not correspond to the value's datatype can be detected by means of the rule of inference R_bind8. *m('red', showName)* is such a misuse case for the specification depicted by Fig.6.15.d. The erroneous argument is *'red'*, and this error is detected by R_bind8 whose semantic constraint *type([[obj]]) ∘ [[showName]] = dType([['red']])* cannot be fulfilled. The details of the type checking can be found in the Appendix D.

This example of evaluation shows that our type system and the associated type checking are able to ensure the semantically correct specification of attribute checking. The rest of the type system can be evaluated in the same way. This means 1) by defining a relevant metamodel excerpt, 2) by generating error-free and erroneous specifications as well as use and misuse cases, and 3) by applying our type checking on these specifications and (mis)use cases.

6.3.3 Evaluation and Outlook

The type checking approach presented in this work is based on well-known concepts, i.e. rules of inference. Consequently, it is possible to define an algorithm by adapting well-known algorithms such as the recognize-act cycle to our approach. In addition, the new notation which completes the syntactic and semantic patterns with the definition of constraints allows for a definition of error messages and, thus, makes the correction of the graph transformations easier to the user.

Even if some features are very useful for an expressive and efficient metamodel and graph transformation specification, they can be error-prone. For instance, redefinitions of association ends can cause errors if they are not specified and used correctly. That is why we included this concept in our type system, e.g. by means of the rule R_redefineProp whose constraint ensures a correct specification of association end redefinition and derives semantic informations from the matched patterns.
In the same way, parametric polymorphism allows for the reuse of graph transformations which is a desirable property as shown in the context of the MATE and

MAJA projects. Though, this feature can be error-prone if the graph transformation is not specified in a safe way or if incorrect arguments are used in the method calls. This was the main motivation for defining a type system as explained in Section 5.1. The rules `R_sem6`, `R_sem7`, `R_sem9`, `R_theta3`, `R_theta4` and `R_theta5` allow for checking parameterized classes and properties and for deriving semantic information which can be used for the static type checking. The rules `R_unbound2` and `R_prop6` match subgraphs composed of objects whose type is a parameterized class and allows for deriving semantic information after having checked that these subgraphs are metamodel-compliant by means of the rule's semantic constraint. Because parametric polymorphism is especially error-prone, we defined in the type system rules of inference to check arguments corresponding to parameterized classes and properties (`R_call3`, `R_call4`, `R_call5`). They are completed by additional rules of inference which bind these arguments with the pattern objects.

It must also be noticed that our rules do not generate any conflict. A conflict occurs when two or more rules match, but the application of one rule makes the application of the other rule impossible because the premises do not match anymore or the constraint cannot be fulfilled anymore.
The rules are defined in such a way that most rules have disjunctive premises so that only few rules of inference have common premises. Rules with similar premises and constraints are for instance the rules which can be applied to method signatures, e.g. the rules `R_sem7` and `R_theta4`. Both rules have the same premises and constraints: the premises are a signature with a parameterized class whose type is restricted by a given class *C2*, and the constraint consists in checking that *C2* actually belongs to the set of classes ($[[C2]] \in \mathcal{C}$). The conclusions of both rules do not modify the premises and the constraints so that we can apply the rules of inference in any order. As explained in Section 5.4.2, we improve the efficiency in selecting the rules of inference by means of priorities as defined in Fig.5.17 .

According to our approach, we first inspect the metamodel in order to create a semantic knowledge base which is then used and completed while checking graph transformations and method calls. Though, as illustrated by the running example of Chapter 5, even a small metamodel excerpt produces a large semantic knowledge base. The same example shows that it is not necessary to inspect the complete metamodel in order to collect enough information for the type checking of a given graph transformation. Thus, it appears clearly that we could improve our approach by determining the smallest metamodel excerpt which is necessary to inspect the desired graph transformation(s).
Determing the metamodel excerpt would add a preparatory step in the execution of type checking. Nevertheless, it would result in a more efficient metamodel and graph transformation inspection by reducing the size of the semantic knowledge base.

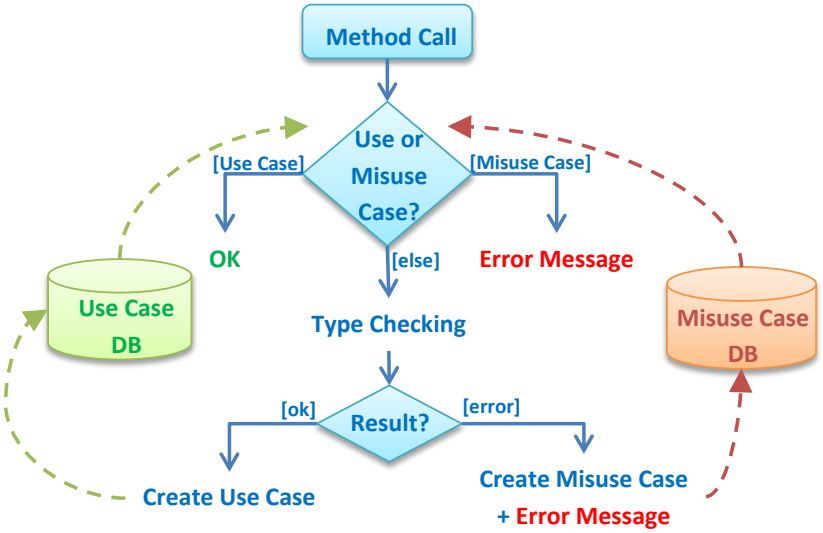


Figure 6.17: Use and Misuse Cases

As shown by the examples in this work, even if the analysis of the signature and the story diagram of a graph transformation does not detect any error, an error can occur depending on the arguments of the method calls. That is why we inspect statically the method calls, too. The approach described in Chapter 5 consists in checking each method call. If the same graph transformation is called repeatedly, checking each method call can become time-consuming and even repetitive in the case of similar method calls. In such a case, we can consider use cases and misuse cases in order to avoid the repetitive inspection of similar method calls. A SDM graph transformation's *use case* consists of a method and a list of arguments and their type that lead to a correct execution of the graph transformation. A *misuse case* is then a method and a list of arguments and their type that lead to an error when the corresponding method call is executed.
Let us consider the running example of Section 5.4.4 and 6.1 with the method *createConnector(connected extends ConnectableElement : MOFClass, src : $connected, trg : $connected)*. If we can classify a method call which has been inspected into use case or misuse case, we can reuse these information for similar calls of the graph transformation. A use case corresponds e.g. to the method calls with the list of arguments and their type $\{Vertex, srcVertex, trgVertex\}$ (where *srcVertex* and *trgVertex* are instances of the class *Vertex*). Then, we can deduce that the method call *createConnector(Vertex, srcVertex2, trgVertex2)* (where *srcVertex2* and *trgVertex2* are instances of *Vertex*) will not cause any type error. A misuse case corresponds e.g. to the method calls with the list of arguments $\{Inport, srcInport, trgInport\}$ (where *srcInport* and *trgInport* are instances of the class *Inport*). Then, we can deduce that the method call *createConnector(Inport, srcInport2, trgInport2)* (where *srcInport2* and *trgInport2* are instances of *Inport*) will produce an error when executed.

We can separate the use cases from the misuse cases for method calls with help of the type checking presented in this work. Fig. 6.17 illustrates this idea. Before starting the analysis of any new method call, we can check whether it corresponds to a use case or a misuse case we have already determined. If it is the case, we do not need to inspect this new method call and can determine whether it is erroneous or not. Else, the inspection of this new method call will allow for creating a new use case resp. misuse case which is then added to a use case database resp. misuse case database. As described above, if we already have determined that the list of arguments $\{Vertex, srcVertex, trgVertex\}$ is a use case for this method, we consequently do not have to execute the type checking for each individual method call using a similar set of arguments. In the same way, since $\{Inport, srcInport, trgInport\}$ is classified a misuse case for this method, we can reject all method calls with such a set of arguments without having to analyze each call individually. Thus, such an approach would allow for a more efficient type checking of method calls.

As illustrated by the MATE/MAJA projects, we can use graph transformation to repair guideline violations of domain-specific languages. Similarly, we could use the graph transformation language SDM to repair the MOSL specifications. This requires the definition of a metamodel for MOSL so that we can define the graph transformation on it. For instance, if the semantic constraint of rule `R_prop2` is violated, this means that the type *T* of the attribute *attr* belonging to the class *C* has not been defined in the metamodel. This error can be repaired either by specifying the data type *T* in the metamodel, or by changing the data type of *attr*. If the semantic constraint of `R_unbound1` is not fulfilled, this means that the class *C* is not part of the metamodel. This error can be repaired by correcting the object's type in the story diagram. We can note that most of these repair actions requires the user's decision. Fig. 6.18 completes the schema presented in the introduction (Fig. 1.1) and gives an overview of this proposal. The MATLAB SL/SF metamodel as well as the guidelines' specification (by means of SDM) are instances of the MOSL metamodel. This is a closed architecture, i.e. the MOSL metamodel defines itself. Repair actions are defined as SDM graph transformations on the MOSL metamodel. If type errors can be detected by means of our type checking approach, we can apply these graph transformations in order to repair the type errors.

The type system described in this work is a type system for the MOSL language completed by the generic feature. Though, we did not include rules of inferences for the reflective features, this could be part of a future work. We described in Section 6.2.1 additional SDM features such as the iteration over typed collections. If we introduce new SDM elements, we have to extend the type system in order to support the type checking for these new elements. The rules' premises of the type system described in this work cannot match these additional elements. Thus, checking graph transformations expressed with these additional features would not

be possible, and we would need additional rules in order to inspect them. In the same way, we can consider our type system approach not only for the extended MOSL language, but more generally for other languages. By means of the semantic domains and the rules of inference, we translate syntactic and semantic patterns to a tool-independent abstract representation of the type information contained in the metamodel and the transformations. Thus, this approach, in particular our new notation, for defining a type system could be adapted for defining type systems of other visual languages, especially MOF-compliant languages.

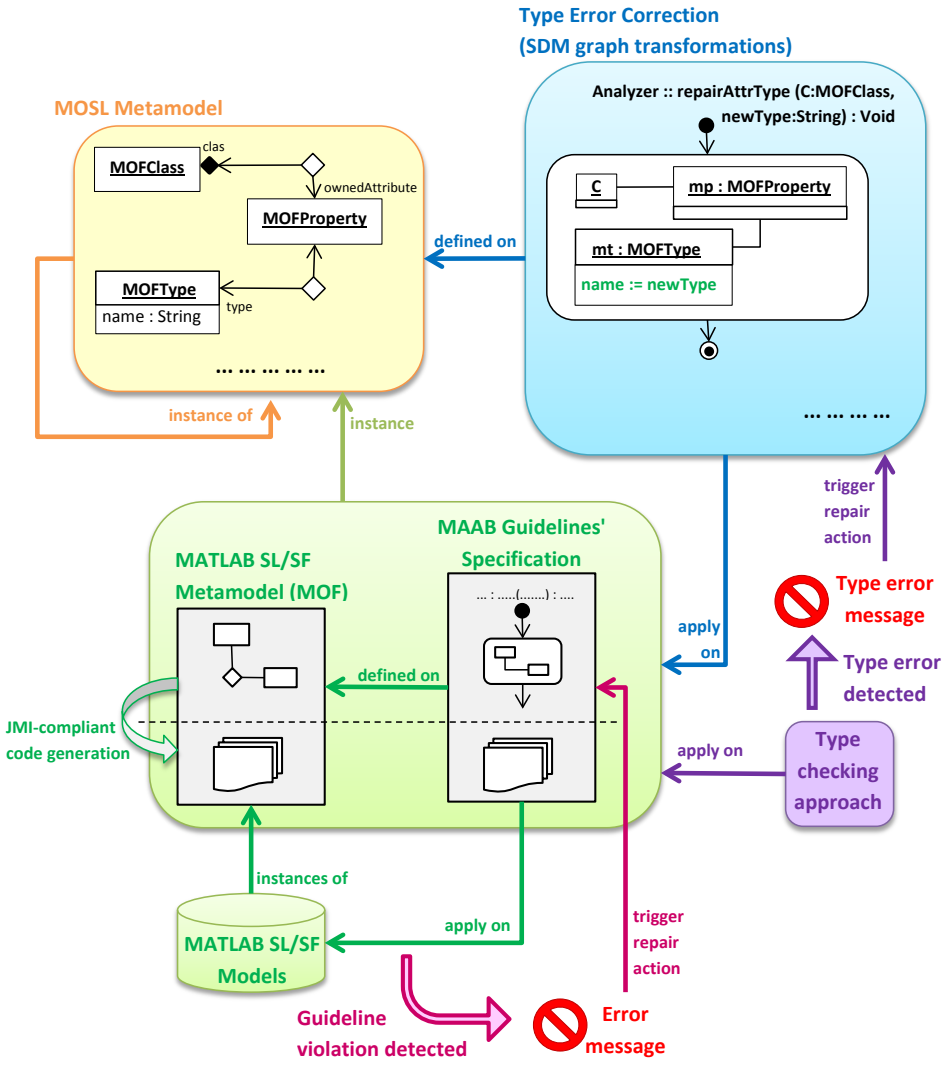


Figure 6.18: Correction of type errors

Chapter 7

Conclusion

7.1 Summary

This thesis starts with an explanation of the relevant concepts for this work in Chapter 2 as well as a description of the context of this work in Chapter 3 in order to provide the reader with a better understanding of our approach.
In the context of model-driven architecture, models are not simply sketches, but artifacts integrated in the different phases of the development process. Model transformations allow for ensuring the traceability from model to model as well as the consistency when manipulating a model. There are different kinds of model transformations. We concentrate in this thesis on graph transformations such as SDM (Story Driven Modeling). SDM is a transformation language which is integrated in the MOFLON Specification Language (MOSL). MOSL is composed of MOF as schema language, OCL as constraint language, and SDM as transformation language. The code obtained from a specification (metamodel + graph transformation) is JMI-compliant. JMI is the standard mapping between MOF and Java. In addition to tailored interfaces, JMI provides reflective interfaces which are the same for all metamodels. Thus, these reflective interfaces support the manipulation of a metamodel even if its tailored interfaces are unknown.
MOSL is the language used in the context of the MATE/MAJA projects which are described in Chapter 3. These projects have been motivated by the urgent need for the automation of MATLAB Simulink/Stateflow model analysis according to modeling guidelines. Due to the increasing complexity of the developed systems, the models have become so intricate and huge that an automated model analysis and correction is necessary. The approach of the MATE/MAJA projects is based on the specification of the MATLAB metamodel by means of MOF and the specification of the guidelines as SDM model transformations.
Our work has two main aspects: on the one hand, the extension of SDM with generic and reflective features as described in Chapter 4 , and on the other hand the definition of a type system and a type checking approach for MOSL as described in Chapter 5.

The specification of modeling guidelines by means of SDM in the context of the MATE/MAJA projects shows that this language is pretty well-suited for this task. For instance, patterns to be matched can be specified more easily in this visual way than in a textual way. Though, the specification of guidelines for the MATE/MAJA projects underlines some desirable improvements, too. Graph drawing requires time and efforts. Therefore the reusability of story diagrams needs to be improved by allowing for the specification of generic methods. The code generated from the SDM specifications is JMI-conform. As explained in Chapter 2, JMI provides reflective interfaces which support the manipulation of a metamodel even if its tailored interfaces are unknown. Metamodel-related information for which the JMI interfaces provide reflective access are attribute names, class names and association names. Since tailored and reflective interfaces provide the same functionalities, the parameterization of all metamodel-related information can be passed to the description of model transformations without any loss of functionality. Thus, it is possible to generate generic code from the graph transformation. Nevertheless, syntax elements are missing in the SDM language in order to define the corresponding generic transformations. Therefore we propose an extension for SDM which provides visual counterparts in the graph schemata to these JMI facilities. The extension uses the $-symbol. The expression following this symbol is a metamodel-related information which is evaluated when executing the graph transformation. Thus, the expression following the $-symbol corresponds to the information the JMI interfaces provide reflective access to, i.e. attribute names, class names and association names. It can be used in the method's signatures as well as in the story diagrams. We also introduce the keyword *extends* in the method's signatures which allows for defining a type restriction when parameterizing a class.
The JMI reflective interface *RefBaseObject* at the top of the interface hierarchy provides the common reflective method *refMetaObject()* which returns the metaobject of the calling object. In other words, JMI provides a mechanism for the discovery of a metamodel's metadata. We propose an extension of the SDM syntax with a connector represented by a dashed arrow with the label *instance of* between an object and its metaobject. The matching of those metalevel-spanning structures can be very beneficially integrated into the existing matching of intra-metalevel structures. It provides access to additional information by means of a simple representation. This information was not available, or could be accessed only by means of a complicated diagram. Thus, this feature completes SDM, and improves its expressiveness.

As described in Chapter 5, we define a type system for MOSL and a type checking approach in order to inspect SDM graph transformations. A type system is the part of a typed language which keeps track of the type of variables and, in general, of the types of all expressions in a program. We noticed that the adoption of genericity can introduce type errors when the graph transformations are executed because the type of the parameterized elements is only known at runtime. Therefore, we propose a formal method to statically check the type-safety of graph transformations.

As explained in Section 2.5, type inference systems allow for inferring types from a program specification, with or without some type annotations. This approach is based on the application of so-called type rules which are specified as rules of inference. A collection of type rules form a formal type system. Our approach described in Chapter 5 builds on the usual type inference systems, but is slightly different. We do not only deduct type information from premises composed themselves of type information, but extract type information from the syntactic elements and/or from other derived type information, and define constraints in the rules of inference which allows for detecting type errors.
We first define sets called *semantic domains* which represent elements of the static semantics (e.g. classes, properties, parameter) as well as functions applicable on the elements of these semantic domains. Then, we introduce an operator *[[]]* which maps an element from the (visual) syntactical domain to the corresponding element from the (textual) semantical domain. In addition, we modify the usual notation of the rules of inference by dividing in half the field over the horizontal line, reserving the left side to the syntactic pattern and the right side to the semantic premises. We also divide the field under the horizontal line in half, the left side containing semantic constraints and the right side containing the rule's conclusions. This new kind of rules of inference is interpreted as follows. The syntactic pattern and the semantic premises must match for the rule to be considered. If they match, the semantic constraints can be checked. Unfulfilled semantic constraints mean that the matched pattern is potentially not type-safe (only potentially, since this constraint may be fulfilled anyway after having derived enough type information from the application of other rules). Finally, if the premises match and the constraints are fulfilled, the rule can apply and the conclusion can be derived. We define rules of inference which apply on the metamodel (i.e. syntactic knowledge base for the graph transformations), on the signature and story diagrams (i.e. to check the specification's safety) and on the methods' call (i.e. to evaluate the transformation with the concrete values).
We also propose a basic algorithm for type checking by means of the rules' application. Fig. 5.19 in Section 5.4.2 illustrates this rule application algorithm. A rule application cycle applies on the metamodel, then on the graph transformations (for each, first the signature, then the story diagram), and finally on each method call, as illustrated by Fig. 5.18. Each rule application cycle starts with a recognize-act cycle where we create a matching set containing all possible rules instantiations, i.e. pairs consisting of a rule and a subset of data items matching the rule's syntactic and/or semantic premises. If the semantic constraint of a rule's instantiation from the matching set is fulfilled, the rule can be applied and semantic information derived. These information complete the semantic knowledge base. Then, the cycle restarts by checking whether the new information in the semantic knowledge base allows for finding new rule instantiations and for completing the matching set. The cycle terminates when the matching set is empty (i.e. all rule instantiations could be applied and no new rule instantiation can be added) or if no rule instantiation can be selected and executed (i.e. the semantic constraints of the rule instantiations

cannot be fulfilled). If the matching set is empty when leaving the recognize-act, no type error is detected. On the contrary, if the matching set is not empty, it means that the semantic constraints of the remaining rule instantiation cannot be fulfilled, and this corresponds to a type error. We then create an error message for each unfulfilled semantic constraint.
We illustrate our type checking approach by applying it to a simple example. Then, we describe in Chapter 6 another application example which combines both aspects of our work, i.e. the type checking of a graph transformation using the generic features.

The extension of the SDM syntax with generic and reflective features was motivated by the MATE/MAJA projects in order to specify the analysis and correction of modeling guidelines more efficiently. The generic and reflective features allow for improving the reusability of patterns and, thus, reducing the effort in drawing diagrams. Another purpose of the extended SDM syntax is the improvement of this language's expressiveness.
In order to make the extended SDM as user-friendly as possible, we preserve the "flavor" of SDM. Therefore, generic and non-generic pattern objects differ only in the use of the $-symbol. In the same way, the keyword "*extends*" for type restriction of parameterized classes refers to the Java programming language which is well-known in the domain of model-driven engineering. Another advantage of the extended SDM and a consequence of the reuse of graph patterns is the reduced number of the blocks of code generated. Though, even if the extended syntax is user-friendly, the optimization of story diagrams by means of the new features can become nontrivial and require more reflexion from the user. Another limitation is the possible occurrence of type errors when using the generic features.
The occurrence of type errors when using the generic features was precisely the motivation for the specification of a strong and nominative type checking approach for SDM. The application of the rules of inference on the metamodel, the method signatures, the story diagrams and the method calls allows for a static typing. It must also be noticed that our rules of inference do not generate any conflict.
The type system presented in this work is inspired by the usual concept of rules of inference. As a consequence, it has the advantage that we can adapt well-known algorithms to our approach in order to inspect a given specification. The new notation for rules of inference allows for combining syntactic and semantic information in order to derive new semantic information. This new notation which completes the syntactic and semantic patterns with the definition of semantic constraints allows for the detection of type errors too. In addition, it provides a more precise definition of error messages which makes the correction of the graph transformations easier to the user.

7.2 Future Developments

The generic and reflective features presented in Chapter 4 are a real improvement for the reusability and expressiveness of story diagrams. Nevertheless, we can improve the SDM language further by means of additional features as described in Section 6.2.1.
For instance, modeling guidelines about naming conventions illustrate the benefit of adding regular expressions in SDM. The extension of SDM with regular expressions would allow for inspecting thoroughly String attributes, differentiating upper and lower case letters, etc. The integration of the regular expressions in SDM requires mainly an adaptation of the generated code. Since MOFLON generates Java code, and Java provides a package called `java.util.regex` to support the parsing of regular expressions, this makes the integration of regular expressions in SDM possible.
The idea behind the generic feature consists in using the reflective interfaces of JMI. The generic feature builds upon the JMI methods which provide reflective access to attribute names, class names and association names. There are other methods in the JMI reflective interfaces which could be useful, namely methods returning the set of all instances of a class. We can introduce a new iteration feature in the form of a new kind of activity which is parameterized with a class name, representing the collection of all instances of this type, and a user-defined variable as running variable for the iteration. Similarly, we could introduce another iteration feature as an activity which is equipped with a typed collection as parameter and a user-defined variable as running variable for the iteration of the collection. Both kinds of iteration (iteration over all instances of a class and iteration over a typed class) can be combined with each other as well as with the already existing *foreach*-activity of the SDM syntax as depicted by Fig. 6.6.
The set of rules of inference presented in this work has been defined for SDM extended with the generic and reflective features of Chapter 4. If we introduce additional features as proposed in this outlook, we would have to extend the set of typing rules since we need rules whose syntactic pattern matches the new features.

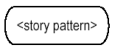
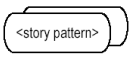
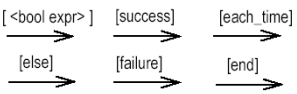
Not only the SDM syntax can be extended with additional features. We can optimize our type checking approach too. According to our approach, we first inspect the metamodel in order to create a semantic knowledge base. This knowledge base is then used and completed while checking graph transformations and method calls. Though, as illustrated by the application example in this thesis, even a small metamodel excerpt can produce a large semantic knowledge base. In addition, this example proves that an inspection of the complete metamodel is not necessary in order to collect enough information. We could improve our approach by determining the smallest metamodel excerpt which is necessary to inspect the desired graph transformation. It would add a preparatory step in the execution of our type checking approach, but would result in a more efficient metamodel and graph transformation inspection by reducing the size of the semantic knowledge base.

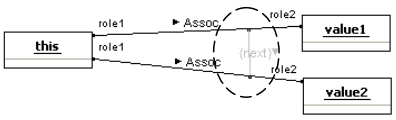
Another possible improvement of our approach would consist in the generation of use- and misuse-cases. A SDM graph transformation's *use case* consists of a method and a list of arguments and their type that lead to a correct execution of the graph transformation. A *misuse case* is then a method and a list of arguments and their type that lead to an error when the corresponding method call is executed. We can separate the use cases from the misuse cases for method calls with help of the type checking approach presented in this work. Fig. 6.17 in Section 6.3.3 illustrates this idea. Before starting the analysis of any new method call, we can check whether it corresponds to a use case or a misuse case we have already determined. If it is the case, we do not need to inspect this new method call and can determine whether it is erroneous or not.
Finally, as illustrated by the MATE/MAJA projects, we can use graph transformation to repair guideline violations of domain-specific languages. Thus, we could use the graph transformation language SDM to repair the errors detected by our type checking approach in MOSL specifications. This approach implies the definition of a MOSL metamodel as well as the definition of SDM graph transformation on it. Fig. 6.18 in Section 6.3.3 illustrates this possible extension of our work.

The result of our approach on the application examples shows how promising our work is. We improved the reusability and expressiveness of graph transformation, and were able to detect type errors. Furthermore, the extensions and improvement proposed show that our approach can be the starting point of further research works. We believe that it would be worthwhile to pursue our work, especially our approach for type checking. We are convinced that our approach for type checking with our new notation can be applied not only to SDM, but more generally to other (visual) graph transformation approaches, too.

Appendix A

SDM Syntax

Syntax element		Representation
Activity	Start point	
	End point	
	Branching	
	Story Pattern	
	Story Pattern "for each"	
	Statement Activity	
Boolean Constraints		{ *<boolExpr>* }
Collaboration Statements		1: m (...) 3 [<bool expr>]: ... 3.1 [i = 1..100]: ... 1': m (...) 2: x = m () 4 [while <bool expr>]: ...
Transition	Simple Transition (*guard*: "*none*")	
	Transition with *guard*	

Syntax element		Representation					
Node		Unbound	Bound	Negative	«create»	«destroy»	Negative - «create»
	Normal Node	v : Class	v	v : Class	«create» v : Class	«destroy» v : Class «destroy» v	«create» v : Class
	Optional Node	v : Class	v	//////////	«create» v : Class «create» v	«destroy» v : Class «destroy» v	//////////
	Multi-Object Node	v : Class	v	//////////	//////////	«destroy» v : Class «destroy» v	//////////
Attribut-Condition	Pre-Condition (Comparison)	==, !=, <, >, <=, >=					
	Post-Condition (Assignment)	:=					
Link	Normal Link	assoc	a	a «create»	a «destroy»	a «create»	
	Optional Link	//////////	//////////	«create»	«destroy»	//////////	
Multilink	"*first*"-Multilink	{first}					
	"*last*"-Multilink	{last}					
	"*direct*"-, "*index*"-, "*indirect*"-Multilink: Arc between 2 Links						
	"*direct*"-Multilink	{next}					
	"*index*"-Multilink	{next[3]}					
	"*indirect*"-Multilink	{...}					

Appendix B

Type System

B.1 Type Sets, Operations and Notation

- Type Sets:
  - $\mathcal{C}$: set of classes contained in the metamodel.
  - $\mathcal{D}t$: set of data types (e.g. `String`, `Boolean`, `Integer`)
  - $\mathcal{O}p$: set of operations : {:=, ==, $<$, $>$, $\leq$, $\geq$}
  - $\mathcal{P}$: set of properties.
  - $\mathcal{O}$: set of bound objects (more precisely, bound pattern objects)
  - $\mathcal{V}$: set of concrete values (e.g. `true`, `false`, `0.01`, `''Hello World''`).
  - $\mathcal{P}a$: set of parameters.
  - $\mathcal{A}rg$: set of a method's arguments.
- Operations:
  - *type:* $\mathcal{O} \rightarrow \mathcal{C}$:
    returns the class of an object.
  - *dType:* $\mathcal{V} \rightarrow \mathcal{D}t$:
    returns the data type of a value.
  - $\leq$: $\mathcal{C} \times \mathcal{C} \rightarrow$ { *true, false* }:
    represents the class inheritance
  - $\circ : \mathcal{C} \times \mathcal{P} \rightarrow \mathcal{C} \bigcup \mathcal{D}t$:
    returns the type of an owned property
  - $[[m]]_i \in \mathcal{P}a$ *with* $i \in \mathbb{N}$:
    represents the i^{th} parameter of the method *m*
  - $\Theta : \mathcal{P}a \rightarrow \mathcal{C} \bigcup \mathcal{P} \bigcup \mathcal{D}t$:
    returns type information about the parameter it is applied on

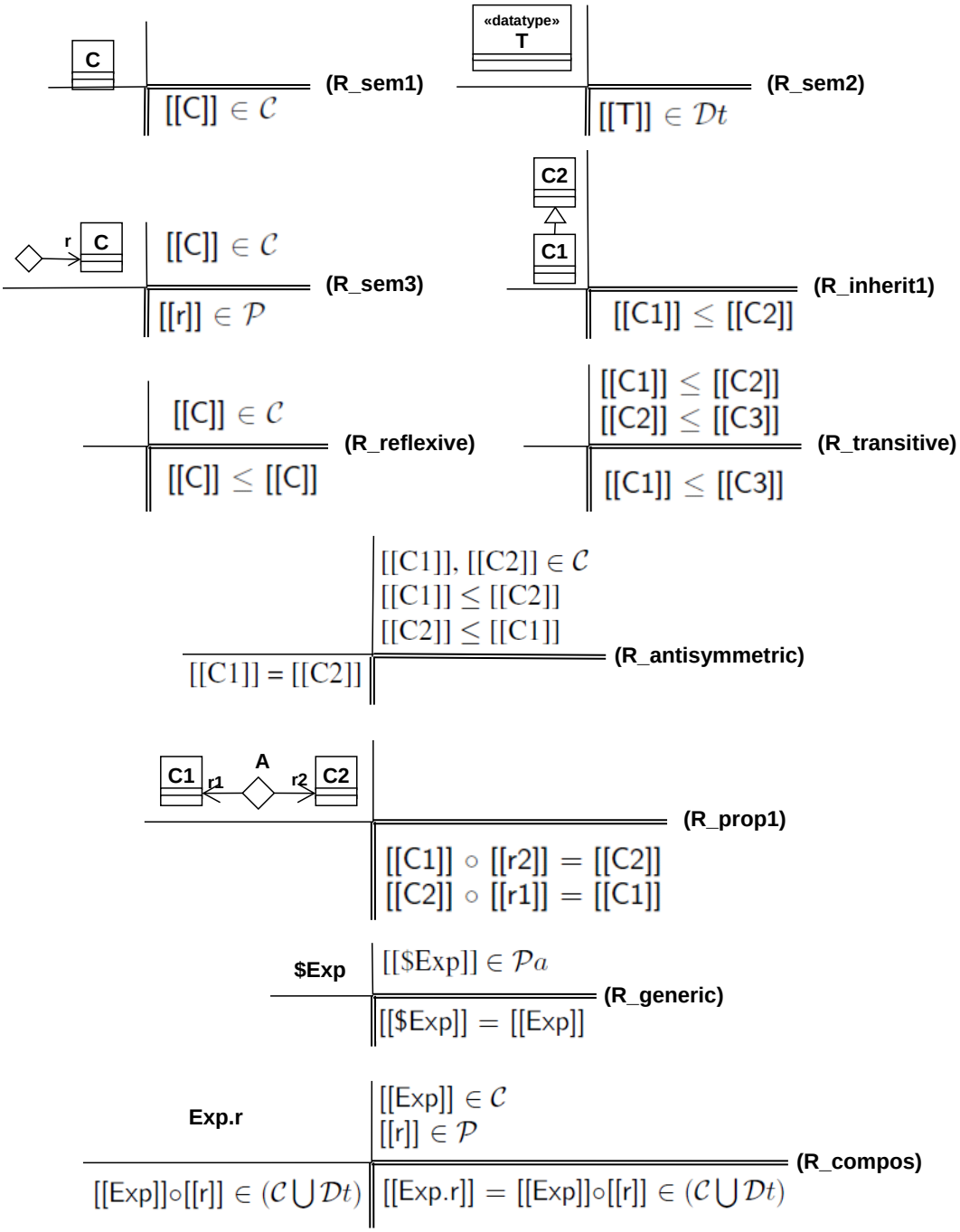
- *arg:* $\mathcal{P}a \rightarrow \mathcal{A}rg$:
  returns the argument to which a method's parameter corresponds (in the context of a method call)

- New Notation of the rules of inference:

Syntactic pattern	**Semantic premises**
Semantic constraints	**Conclusion**

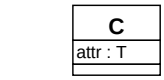
Rule notation and example

B.2 Rules of Inference

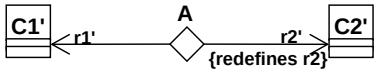


$$\frac{[[o]] \in \mathcal{O},\ [[r]] \in \mathcal{P} \quad \text{type}([[o]]) \circ [[r]] \in (\mathcal{C} \cup \mathcal{D}t) \quad \text{type}([[o]]) = [[C]] \in \mathcal{C}}{[[C]] \circ [[r]] \in (\mathcal{C} \cup \mathcal{D}t)} \text{ (R\_substitute1)}$$

$$\frac{[[o1]], [[o2]] \in \mathcal{O}\ ,\ [[r1]] \in \mathcal{P} \quad \text{type}([[o1]]) \leq \text{type}([[o2]]) \circ [[r1]] \in \mathcal{C} \quad \text{type}([[o1]]) \leq [[C1]] \in \mathcal{C} \quad \text{type}([[o2]]) = [[C2]] \in \mathcal{C}}{[[C2]] \circ [[r1]] \in \mathcal{C} \quad ([[C2]] \circ [[r1]] \leq [[C1]]) \vee ([[C1]] \leq [[C2]] \circ [[r1]])} \text{ (R\_substitute2)}$$



$$\frac{}{[[T]] \in \mathcal{D}t \quad [[attr]] \in \mathcal{P} \quad [[C]] \circ [[attr]] = [[T]]} \text{ (R\_prop2)}$$



$$\frac{}{[[r2]] \in \mathcal{P} \quad \exists\, C \in \mathcal{C}\text{: } C \circ [[r2]] \in \mathcal{C} \wedge ([[C1']] \leq C) \wedge ([[C2']] \leq C \circ [[r2]]) \quad [[C1']] \circ [[r2]] = [[C1']] \circ [[r2']] = [[C2']]} \text{ (R\_redefineProp)}$$

$$\frac{[[C1]],\ [[C2]] \in \mathcal{C},\ [[r]] \in \mathcal{P} \quad [[C1]] \leq [[C2]] \quad [[C2]] \circ [[r]] \in \mathcal{C}}{[[C1]] \circ [[r]] \in \mathcal{C} \quad [[C1]] \circ [[r]] \leq [[C2]] \circ [[r]]} \text{ (R\_inherit2)}$$

$$\frac{[[C1]],\ [[C2]] \in \mathcal{C},\ [[r]] \in \mathcal{P} \quad [[C1]] \leq [[C2]] \quad [[C2]] \circ [[r]] \in \mathcal{D}t}{[[C1]] \circ [[r]] \in \mathcal{D}t \quad [[C1]] \circ [[r]] = [[C2]] \circ [[r]]} \text{(R\_inherit3)}$$

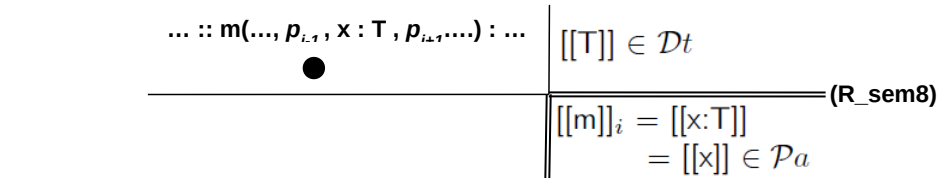
$$\frac{[[C1]],\ [[C2]],\ [[C3]] \in \mathcal{C} \quad [[C3]] \leq [[C1]] \quad [[C3]] \leq [[C2]]}{([[C1]] \leq [[C2]]) \vee ([[C2]] \leq [[C1]])} \text{(R\_inheritSingle)}$$

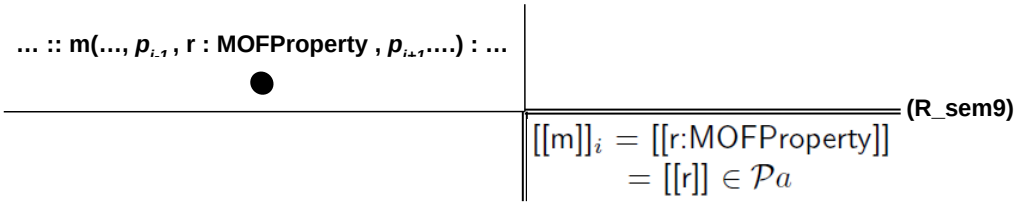
$$\frac{\text{... :: m(..., } p_{i-1}\text{ , } p_i\text{, } p_{i+1}\text{....) : ...}}{[[p_i]] = [[m]]_i \in \mathcal{P}a} \text{(R\_sem4)}$$

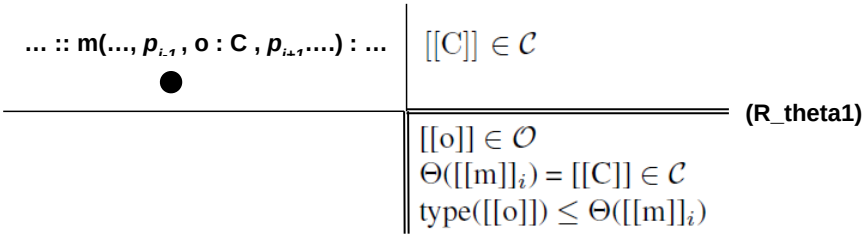
$$\frac{\text{... :: m(..., } p_{i-1}\text{ , o : C , } p_{i+1}\text{....) : ...} \quad [[C]] \in \mathcal{C}}{[[m]]_i = [[o{:}C]] = [[o]] \in \mathcal{P}a} \text{(R\_sem5)}$$

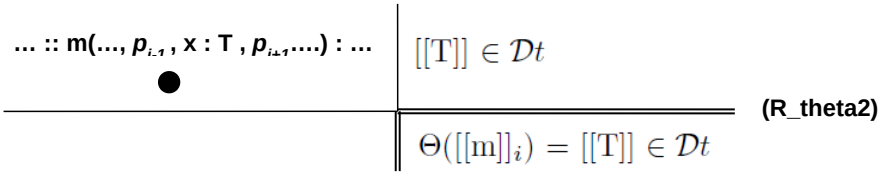
$$\frac{\text{... :: m(..., } p_{i-1}\text{ , C : MOFClass , } p_{i+1}\text{....) : ...}}{[[m]]_i = [[C{:}MOFClass]] = [[C]] \in \mathcal{P}a} \text{(R\_sem6)}$$

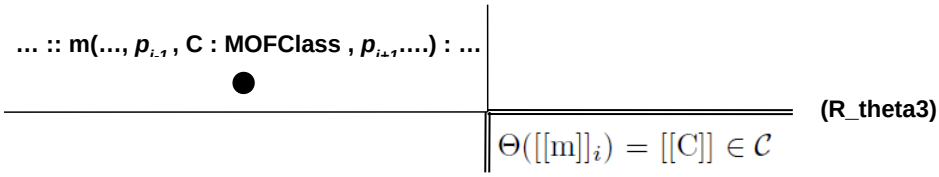
$$\frac{\text{... :: m(..., } p_{i-1}\text{ , C1 } \underline{\textit{extends}}\text{ C2: MOFClass , } p_{i+1}\text{....) :} \quad [[C2]] \in \mathcal{C}}{[[m]]_i = [[C1\ \underline{\textit{extends}}\ C2{:}MOFClass]] = [[C1]] \in \mathcal{P}a} \text{(R\_sem7)}$$

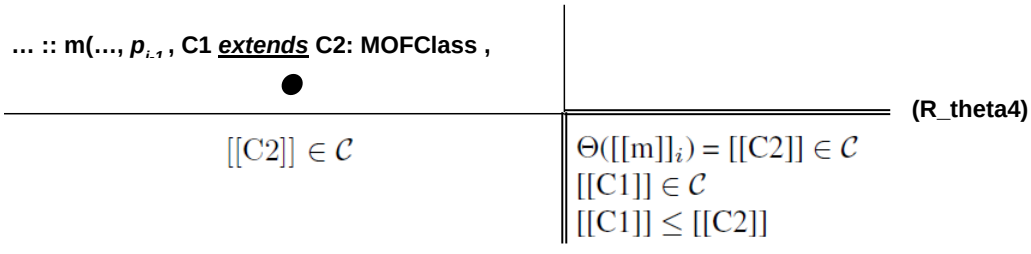

... :: m(..., p_{i-1} , x : T , p_{i+1},....) : ...
[[T]] ∈ $\mathcal{D}t$
(R_sem8)
$[[m]]_i$ = [[x:T]]
= [[x]] ∈ $\mathcal{P}a$

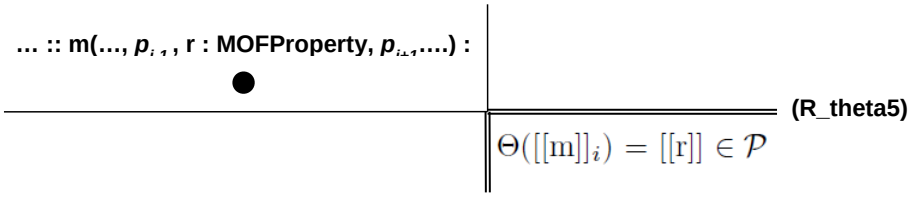

... :: m(..., p_{i-1} , r : MOFProperty , p_{i+1},....) : ...
(R_sem9)
$[[m]]_i$ = [[r:MOFProperty]]
= [[r]] ∈ $\mathcal{P}a$

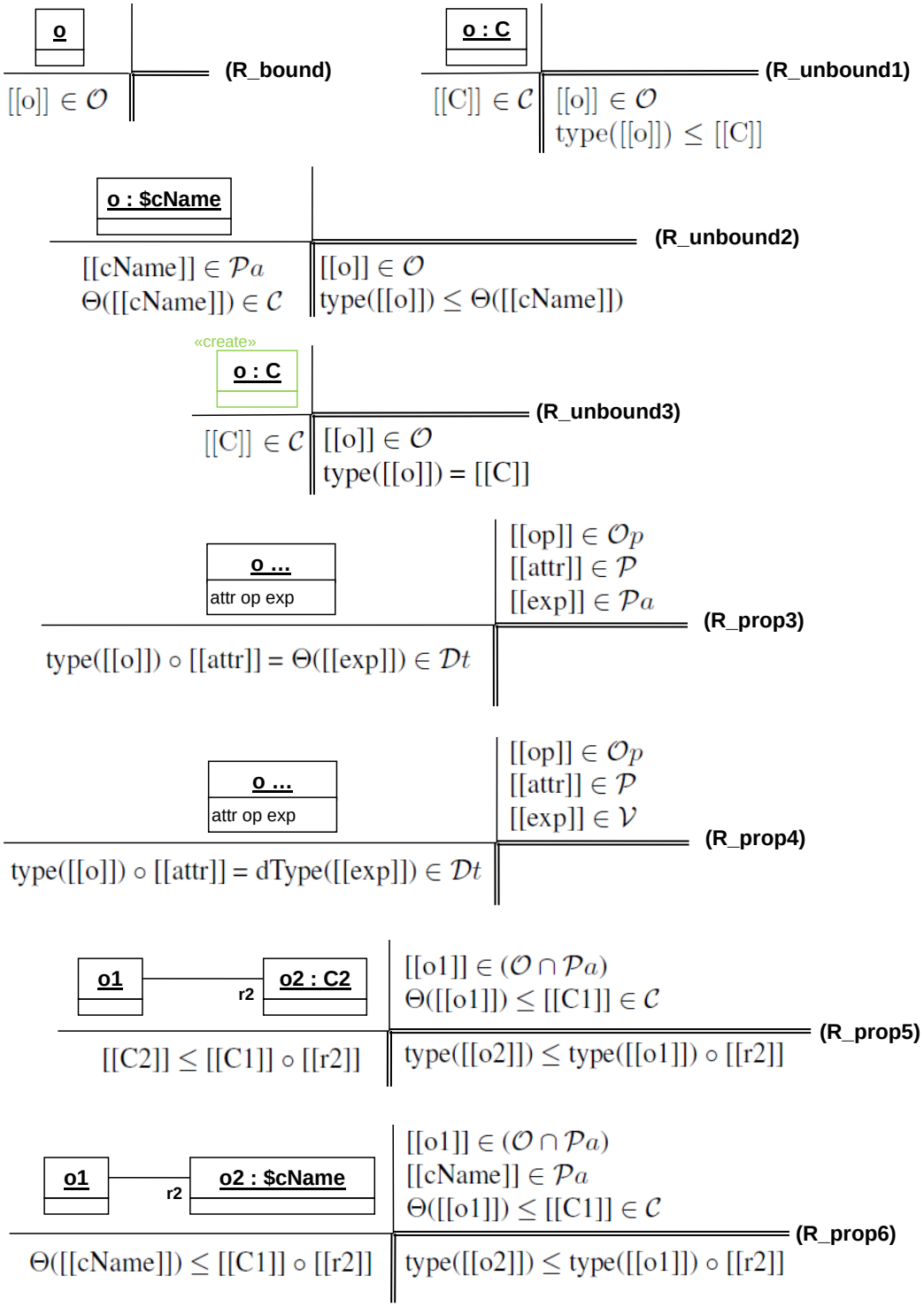

... :: m(..., p_{i-1} , o : C , p_{i+1},....) : ...
[[C]] ∈ $\mathcal{C}$
(R_theta1)
[[o]] ∈ $\mathcal{O}$
Θ($[[m]]_i$) = [[C]] ∈ $\mathcal{C}$
type([[o]]) ≤ Θ($[[m]]_i$)

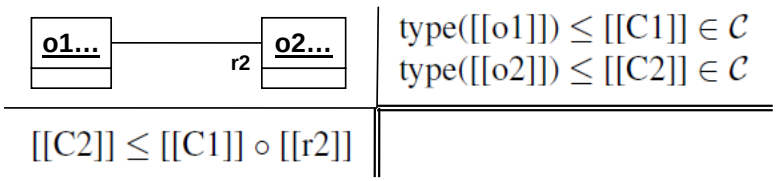

... :: m(..., p_{i-1} , x : T , p_{i+1},....) : ...
[[T]] ∈ $\mathcal{D}t$
(R_theta2)
Θ($[[m]]_i$) = [[T]] ∈ $\mathcal{D}t$


... :: m(..., p_{i-1} , C : MOFClass , p_{i+1},....) : ...
(R_theta3)
Θ($[[m]]_i$) = [[C]] ∈ $\mathcal{C}$

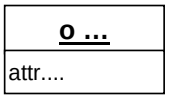

... :: m(..., p_{i-1} , C1 extends C2: MOFClass ,
[[C2]] ∈ $\mathcal{C}$
(R_theta4)
Θ($[[m]]_i$) = [[C2]] ∈ $\mathcal{C}$
[[C1]] ∈ $\mathcal{C}$
[[C1]] ≤ [[C2]]


... :: m(..., p_{i-1} , r : MOFProperty, p_{i+1},....) :
(R_theta5)
Θ($[[m]]_i$) = [[r]] ∈ $\mathcal{P}$


o
[[o]] ∈ 𝒪
(R_bound)
o : C
[[C]] ∈ 𝒞
[[o]] ∈ 𝒪
type([[o]]) ≤ [[C]]
(R_unbound1)
o : $cName
[[cName]] ∈ 𝒫a
Θ([[cName]]) ∈ 𝒞
[[o]] ∈ 𝒪
type([[o]]) ≤ Θ([[cName]])
(R_unbound2)
«create»
o : C
[[C]] ∈ 𝒞
[[o]] ∈ 𝒪
type([[o]]) = [[C]]
(R_unbound3)
o ...
attr op exp
[[op]] ∈ 𝒪p
[[attr]] ∈ 𝒫
[[exp]] ∈ 𝒫a
type([[o]]) ∘ [[attr]] = Θ([[exp]]) ∈ 𝒟t
(R_prop3)
o ...
attr op exp
[[op]] ∈ 𝒪p
[[attr]] ∈ 𝒫
[[exp]] ∈ 𝒱
type([[o]]) ∘ [[attr]] = dType([[exp]]) ∈ 𝒟t
(R_prop4)
o1
r2
o2 : C2
[[o1]] ∈ (𝒪 ∩ 𝒫a)
Θ([[o1]]) ≤ [[C1]] ∈ 𝒞
[[C2]] ≤ [[C1]] ∘ [[r2]]
type([[o2]]) ≤ type([[o1]]) ∘ [[r2]]
(R_prop5)
o1
r2
o2 : $cName
[[o1]] ∈ (𝒪 ∩ 𝒫a)
[[cName]] ∈ 𝒫a
Θ([[o1]]) ≤ [[C1]] ∈ 𝒞
Θ([[cName]]) ≤ [[C1]] ∘ [[r2]]
type([[o2]]) ≤ type([[o1]]) ∘ [[r2]]
(R_prop6)



$$\frac{\text{type}([[o1]]) \leq [[C1]] \in \mathcal{C} \quad \text{type}([[o2]]) \leq [[C2]] \in \mathcal{C}}{[[C2]] \leq [[C1]] \circ [[r2]]} \text{ (R\_prop7)}$$



$$\frac{[[o]] \in \mathcal{O}}{[[attr]] \in \mathcal{P} \quad \text{type}([[o]]) \circ [[attr]] \in \mathcal{D}t} \text{ (R\_prop8)}$$

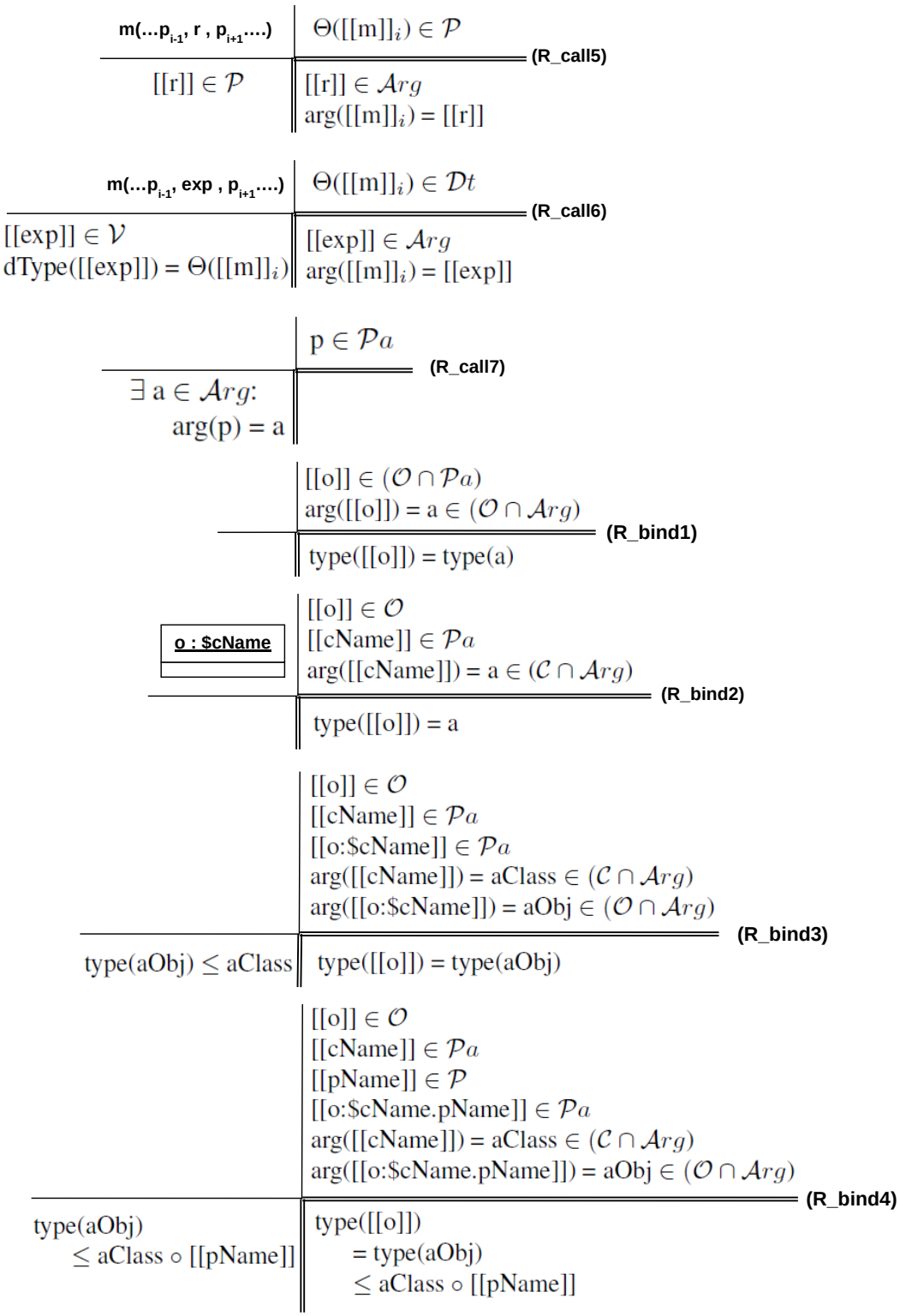
$$\frac{}{\exists\, C \in \mathcal{C}: \quad \text{type}([[o1]]) = C \quad \text{type}([[o2]]) \leq C \circ [[r2]]} \text{ (R\_prop9)}$$

$$\frac{\text{m(...p}_{i-1}\text{, exp , p}_{i+1}\text{....)}}{[[m]]_i \in \mathcal{P}a} \text{ (R\_call1)}$$

$$\frac{\text{m(...p}_{i-1}\text{, o , p}_{i+1}\text{....)}}{[[o]] \in \mathcal{O} \quad \text{type}([[o]]) \leq [[C]]} \quad \frac{[[o']] \in \mathcal{O} \quad [[m]]_i = [[o': C']] \,,\, [[C']] \in \mathcal{C} \quad \Theta([[m]]_i) \leq [[C]] \in \mathcal{C}}{[[o]] \in \mathcal{A}rg \quad \text{arg}([[m]]_i) = [[o]]} \text{ (R\_call2)}$$

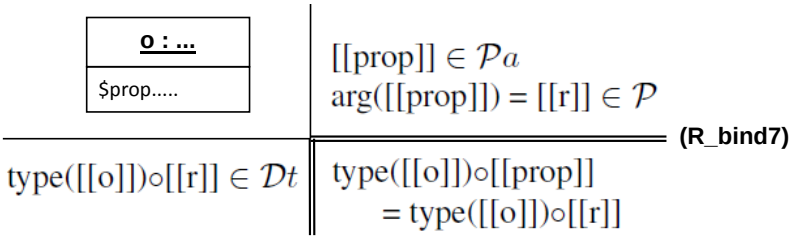
$$\frac{\text{m(...p}_{i-1}\text{, C , p}_{i+1}\text{....)}}{[[C]] \in \mathcal{C}} \quad \frac{\Theta([[m]]_i) = [[C']] \in \mathcal{C} \quad [[m]]_i = [[C':\text{MOFClass}]]}{[[C]] \in \mathcal{A}rg \quad [[C]] \leq \Theta([[m]]_i \quad \text{arg}([[m]]_i) = [[C]]} \text{ (R\_call3)}$$

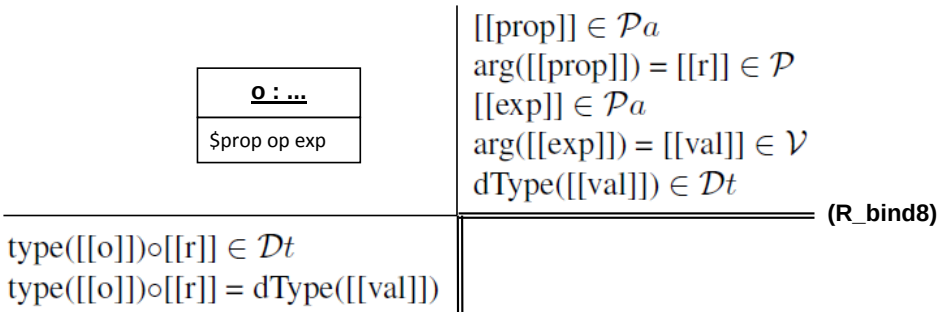
$$\frac{\text{m(...p}_{i-1}\text{, C , p}_{i+1}\text{....)}}{[[C]] \in \mathcal{C} \quad [[C]] \leq \Theta([[m]]_i)} \quad \frac{\Theta([[m]]_i) = [[C'']] \in \mathcal{C} \quad [[m]]_i = [[C']] = [[C' \ \underline{extends}\ C'':\text{MOFClass}]]}{[[C]] \in \mathcal{A}rg \quad \text{arg}([[m]]_i) = [[C]]} \text{ (R\_call4)}$$


m(...p_{i-1}, r , p_{i+1}....)
Θ([[m]]_i) ∈ 𝒫
(R_call5)
[[r]] ∈ 𝒫
[[r]] ∈ 𝒜rg
arg([[m]]_i) = [[r]]
m(...p_{i-1}, exp , p_{i+1}....)
Θ([[m]]_i) ∈ 𝒟t
(R_call6)
[[exp]] ∈ 𝒱
dType([[exp]]) = Θ([[m]]_i)
[[exp]] ∈ 𝒜rg
arg([[m]]_i) = [[exp]]
p ∈ 𝒫a
(R_call7)
∃ a ∈ 𝒜rg:
arg(p) = a
[[o]] ∈ (𝒪 ∩ 𝒫a)
arg([[o]]) = a ∈ (𝒪 ∩ 𝒜rg)
(R_bind1)
type([[o]]) = type(a)
o : $cName
[[o]] ∈ 𝒪
[[cName]] ∈ 𝒫a
arg([[cName]]) = a ∈ (𝒞 ∩ 𝒜rg)
(R_bind2)
type([[o]]) = a
[[o]] ∈ 𝒪
[[cName]] ∈ 𝒫a
[[o:$cName]] ∈ 𝒫a
arg([[cName]]) = aClass ∈ (𝒞 ∩ 𝒜rg)
arg([[o:$cName]]) = aObj ∈ (𝒪 ∩ 𝒜rg)
(R_bind3)
type(aObj) ≤ aClass
type([[o]]) = type(aObj)
[[o]] ∈ 𝒪
[[cName]] ∈ 𝒫a
[[pName]] ∈ 𝒫
[[o:$cName.pName]] ∈ 𝒫a
arg([[cName]]) = aClass ∈ (𝒞 ∩ 𝒜rg)
arg([[o:$cName.pName]]) = aObj ∈ (𝒪 ∩ 𝒜rg)
(R_bind4)
type(aObj)
≤ aClass ∘ [[pName]]
type([[o]])
= type(aObj)
≤ aClass ∘ [[pName]]

$$\frac{\begin{array}{l}\text{aClass} \in (\mathcal{C} \cap \mathcal{A}rg)\\ [[\text{cName:MOFClass}]] \in \mathcal{P}a\\ \text{arg}([[\text{cName:MOFClass}]]) = \text{aClass}\end{array}}{[[\text{cName}]] = \text{aClass}}\ \text{(R\_bind5)}$$

$$\frac{\begin{array}{l}\text{aClass} \in (\mathcal{C} \cap \mathcal{A}rg)\\ [[\text{cName } \underline{\textit{extends}} \text{ C:MOFClass}]] \in \mathcal{P}a\\ \text{arg}([[\text{cName } \underline{\textit{extends}} \text{ C:MOFClass}]]) = \text{aClass}\end{array}}{\text{aClass} \leq [[\text{C}]] \quad [[\text{cName}]] = \text{aClass}}\ \text{(R\_bind6)}$$





Appendix C

Rule Application

C.1 Application Example

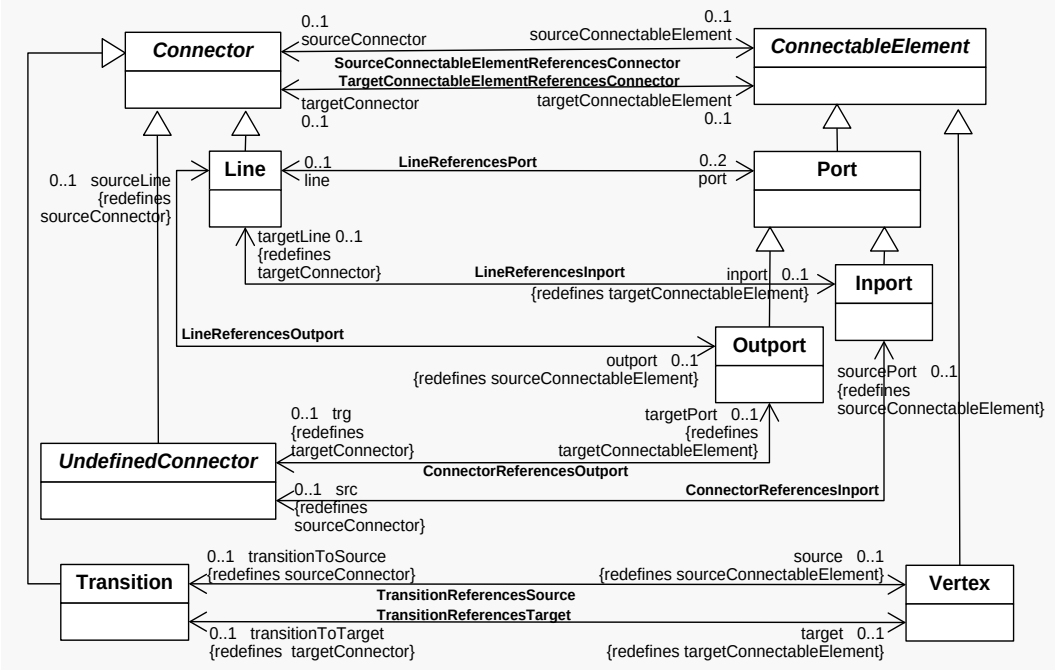


Figure C.1: Redefine *Connector* and *ConnectableElement*

C.2 Metamodel Analysis

Sets of classes (rule `R_sem1`) and properties (rule `R_sem3`):

$\mathcal{C}$: { [[Connector]], [[UndefinedConnector]], [[ConnectableElement]], [[Line]], [[Port]], [[Inport]], [[Outport]], [[Transition]], [[Vertex]] }

$\mathcal{P}$: { [[sourceConnector]], [[targetConnector]], [[sourceConnectable]], [[target-ConnectableElement]], [[line]], [[port]], [[outport]], [[inport]], [[sourceLine]], [[tar-getLine]], [[transitionToTarget]], [[transitionToSource]], [[target]], [[source]], [[trg]], [[src]], [[sourcePort]], [[targetPort]] }

Inheritance relationships (rule `R_inherit1`) and inherited properties (rule `R_inherit2`):

[[UndefinedConnector]] $\leq$ [[Connector]]
[[Line]] $\leq$ [[Connector]]
[[Transition]] $\leq$ [[Connector]]
[[Vertex]] $\leq$ [[ConnectableElement]]
[[Outport]] $\leq$ [[Port]] $\leq$ [[ConnectableElement]]
[[Inport]] $\leq$ [[Port]] $\leq$ [[ConnectableElement]]

[[Line]]$\circ$[[sourceConnectableElement]] $\leq$ [[ConnectableElement]]
[[Line]]$\circ$[[targetConnectableElement]] $\leq$ [[ConnectableElement]]
[[Port]]$\circ$[[sourceConnector]] $\leq$ [[Connector]]
[[Port]]$\circ$[[targetConnector]] $\leq$ [[Connector]]
[[Outport]]$\circ$[[sourceConnector]] $\leq$ [[Connector]]
[[Outport]]$\circ$[[targetConnector]] $\leq$ [[Connector]]
[[Inport]]$\circ$[[sourceConnector]] $\leq$ [[Connector]]
[[Inport]]$\circ$[[targetConnector]] $\leq$ [[Connector]]
[[UndefinedConnector]]$\circ$[[sourceConnectableElement]] $\leq$ [[ConnectableElement]]
[[UndefinedConnector]]$\circ$[[targetConnectableElement]] $\leq$ [[ConnectableElement]]
[[Transition]]$\circ$[[sourceConnectableElement]] $\leq$ [[ConnectableElement]]
[[Transition]]$\circ$[[targetConnectableElement]] $\leq$ [[ConnectableElement]]
[[Vertex]]$\circ$[[sourceConnector]] $\leq$ [[Connector]]
[[Vertex]]$\circ$[[targetConnector]] $\leq$ [[Connector]]

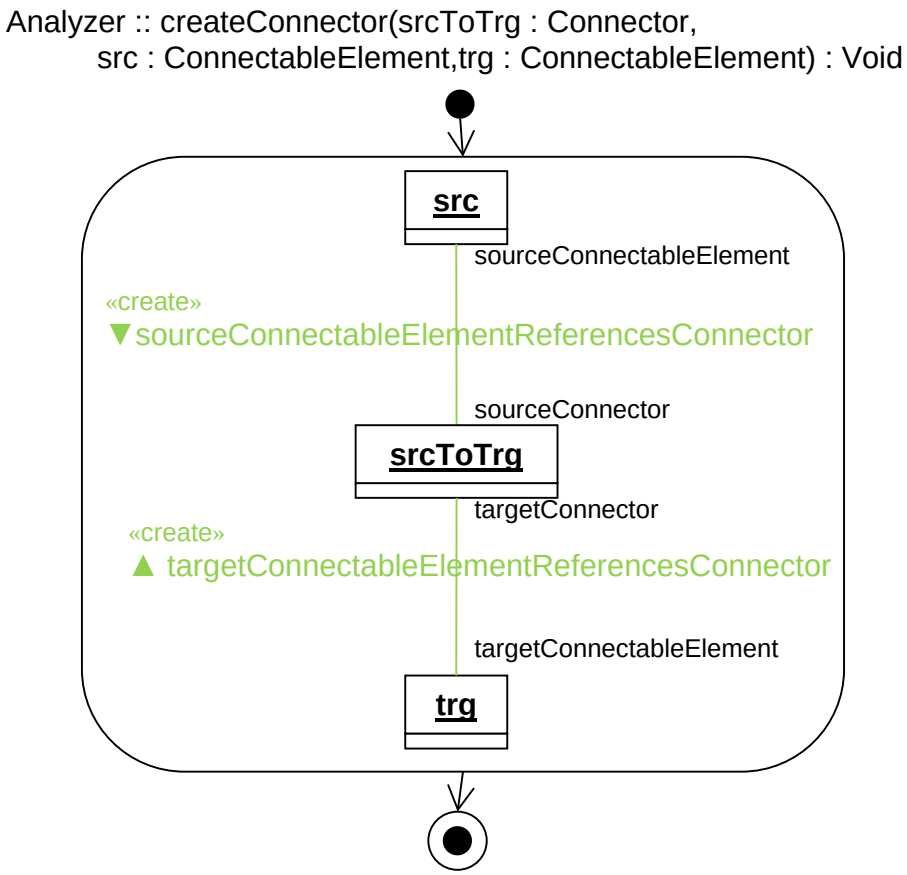
Property relationships (rule `R_prop1`) and redefined property relationships (rule `R_redefineProp`):

[[Connector]]$\circ$[[sourceConnectableElement]] = [[ConnectableElement]]
[[Connector]]$\circ$[[targetConnectableElement]] = [[ConnectableElement]]
[[ConnectableElement]]$\circ$[[sourceConnector]] = [[Connector]]
[[ConnectableElement]]$\circ$[[targetConnector]] = [[Connector]]

[[Line]]∘[[port]] = [[Port]]
[[Port]]∘[[line]] = [[Line]]

[[Line]]∘[[sourceConnectableElement]] = [[Line]]∘[[outport]] = [[Outport]]
[[Line]]∘[[targetConnectableElement]] = [[Line]]∘[[inport]] = [[Inport]]
[[Outport]]∘[[sourceConnector]] = [[Outport]]∘[[sourceLine]] = [[Line]]
[[Outport]]∘[[targetConnector]] = [[Outport]]∘[[trg]] = [[UndefinedConnector]]
[[Inport]]∘[[sourceConnector]] = [[Inport]]∘[[src]] = [[UndefinedConnector]]
[[Inport]]∘[[targetConnector]] = [[Inport]]∘[[targetLine]] = [[Line]]
[[UndefinedConnector]]∘[[sourceConnectableElement]]
= [[UndefinedConnector]]∘[[sourcePort]] = [[Inport]]
[[UndefinedConnector]]∘[[targetConnectableElement]]
= [[UndefinedConnector]]∘[[targetPort]] = [[Outport]]
[[Transition]]∘[[targetConnectableElement]] = [[Transition]]∘[[target]] = [[Vertex]]
[[Transition]]∘[[sourceConnectableElement]] = [[Transition]]∘[[source]] = [[Vertex]]
[[Vertex]]∘[[targetConnector]] = [[Vertex]]∘[[transitionToTarget]] = [[Transition]]
[[Vertex]]∘[[sourceConnector]] = [[Vertex]]∘[[transitionToSource]] = [[Transition]]

C.3 Method without Generic Feature



C.3.1 Method Specification Analysis

1. Signature Analysis

Parameters and Objects (rules `R_sem4` and `R_sem5`):

$\mathcal{P}a$: { [[createConnector]]$_1$, [[createConnector]]$_2$, [[createConnector]]$_3$ }
= { [[srcToTrg: Connector]], [[src:ConnectableElement]],
[[trg:ConnectableElement]] }
= { [[srcToTrg]], [[src]], [[trg]] }

Application of the Θ-operator (rules `R_theta1`):

$\mathcal{O}$: { [[srcToTrg]], [[src]], [[trg]] }

$\Theta([[\text{createConnector}]]_1) = [[\text{Connector}]] \in \mathcal{C}$
$\text{type}([[\text{srcToTrg}]]) \leq \Theta([[\text{createConnector}]]_1)$

$\Theta([[\text{createConnector}]]_2) = [[\text{ConnectableElement}]] \in \mathcal{C}$
$\text{type}([[\text{src}]]) \leq \Theta([[\text{createConnector}]]_2)$

$\Theta([[\text{createConnector}]]_3) = [[\text{ConnectableElement}]] \in \mathcal{C}$
$\text{type}([[\text{trg}]]) \leq \Theta([[\text{createConnector}]]_3)$

2. Story Diagram Analysis

Bound objects (Rule `R_bound`):

[[srcToTrg]] $\in \mathcal{O}$: *true*
[[src]] $\in \mathcal{O}$: *true*
[[trg]] $\in \mathcal{O}$: *true*

Properties (Rules `R_prop7` and `R_reflexive`):

[[Connector]] $\leq$ [[ConnectableElement]]$\circ$[[sourceConnector]] : *true*
[[Connector]] $\leq$ [[ConnectableElement]]$\circ$[[targetConnector]] : *true*
[[ConnectableElement]] $\leq$ [[Connector]]$\circ$[[sourceConnectableElement]] : *true*
[[ConnectableElement]] $\leq$ [[Connector]]$\circ$[[targetConnectableElement]] : *true*

Properties (Rules `R_prop9`:

type([[src]]), type([[trg]]), type([[srcToTrg]]): *???*
type([[srcToTrg]]) $\leq$ type([[src]]) $\circ$ [[sourceConnector]] : *???*
type([[srcToTrg]]) $\leq$ type([[trg]]) $\circ$ [[targetConnector]] : *???*
type([[src]])
$\leq$ type([[srcToTrg]]) $\circ$ [[sourceConnectableElement]] : *???*
type([[trg]])
$\leq$ type([[srcToTrg]]) $\circ$ [[targetConnectableElement]] : *???*

3. Resume of Type Information

$\mathcal{P}a$: { [[createConnector]]$_1$, [[createConnector]]$_2$, [[createConnector]]$_3$ }
= { [[srcToTrg:Connector]], [[src:ConnectableElement]],
[[trg:ConnectableElement]] }
= { [[srcToTrg]], [[src]], [[trg]] }

Θ([[CreateConnector]]$_1$) = [[Connector]] $\in \mathcal{C}$
Θ([[CreateConnector]]$_2$) = [[ConnectableElement]] $\in \mathcal{C}$
Θ([[CreateConnector]]$_3$) = [[ConnectableElement]] $\in \mathcal{C}$

$\mathcal{O}$: { [[src]], [[trg]], [[srcToTrg]] }

type([[src]]) $\leq$ [[ConnectableElement]]
type([[trg]]) $\leq$ [[ConnectableElement]]
type([[srcToTrg]]) $\leq$ [[Connector]]

Checked but non-fulfilled:
type([[srcToTrg]]) $\leq$ type([[src]]) $\circ$ [[sourceConnector]]
type([[srcToTrg]]) $\leq$ type([[trg]]) $\circ$ [[targetConnector]]
type([[src]]) $\leq$ type([[srcToTrg]]) $\circ$ [[sourceConnectableElement]]
type([[trg]]) $\leq$ type([[srcToTrg]]) $\circ$ [[targetConnectableElement]]

C.3.2 Method Call Analysis - 1st Example

createConnector(trans, srcVertex, trgVertex)

[[trans]] $\in \mathcal{O}$ type([[trans]]) = [[Transition]]
[[srcVertex]] $\in \mathcal{O}$ type([[srcVertex]]) = [[Vertex]]
[[trgVertex]] $\in \mathcal{O}$ type([[trgVertex]]) = [[Vertex]]

1. Check Arguments (Rules `R_call1` and `R_call2`):

[[createConnector]]$_1 \in \mathcal{P}a$: *true*

[[srcToTrg]] $\in \mathcal{O}$
Θ([[createConnector]]$_1$) = [[Connector]] $\in \mathcal{C}$
[[CreateConnector]]$_1$ = [[srcToTrg:Connector]]
[[trans]] $\in \mathcal{O}$: *true*
type([[trans]]) = [[Transition]] $\leq$ [[Connector]] : *true*
$\Rightarrow$ [[trans]] $\in \mathcal{A}rg$
arg([[createConnector]]$_1$) = [[trans]]

[[createConnector]]$_2 \in \mathcal{P}a$: *true*

[[src]] $\in \mathcal{O}$
Θ([[createConnector]]$_2$) = [[ConnectableElement]] $\in \mathcal{C}$
[[CreateConnector]]$_2$ = [[src:ConnectableElement]]
[[srcVertex]] $\in \mathcal{O}$: *true*
type([[srcVertex]]) = [[Vertex]] $\leq$ [[ConnectableElement]] : *true*
$\Rightarrow$ [[srcVertex]] $\in \mathcal{A}rg$
arg([[createConnector]]$_1$) = [[srcVertex]]

[[createConnector]]$_3 \in \mathcal{P}a$: *true*

[[trg]] $\in \mathcal{O}$
Θ([[createConnector]]$_3$) = [[ConnectableElement]] $\in \mathcal{C}$
[[CreateConnector]]$_3$ = [[trg:ConnectableElement]]
[[trgVertex]] $\in \mathcal{O}$: *true*
type([[trgVertex]]) = [[Vertex]] $\leq$ [[ConnectableElement]] : *true*
$\Rightarrow$ [[trgVertex]] $\in \mathcal{A}rg$
arg([[createConnector]]$_2$) = [[trgVertex]]

2. Bind method declaration and method call (Rule `R_bind1`)

[[srcToTrg]] $\in (\mathcal{O} \cap \mathcal{P}a)$
arg([[srcToTrg]]) = arg([[createConnector]]$_1$) = [[trans]] $\in (\mathcal{O} \cap \mathcal{A}rg)$
$\Rightarrow$ type([[srcToTrg]]) = type([[trans]])

[[src]] $\in (\mathcal{O} \cap \mathcal{P}a)$
arg([[src]]) = arg([[createConnector]]$_2$) = [[srcVertex]] $\in (\mathcal{O} \cap \mathcal{A}rg)$
$\Rightarrow$ type([[src]]) = type([[srcVertex]])

[[trg]] $\in (\mathcal{O} \cap \mathcal{P}a)$
arg([[trg]]) = arg([[createConnector]]$_3$) = [[trgVertex]] $\in (\mathcal{O} \cap \mathcal{A}rg)$
$\Rightarrow$ type([[trg]]) = type([[trgVertex]])

3. Other matching rules (Rule `R_substitute2`, Rules `R_reflexive`)

type([[srcToTrg]]) $\leq$ type([[src]]) $\circ$ [[sourceConnector]]
type([[srcToTrg]]) = type([[trans]]) $\leq$ [[Transition]]
type([[src]]) = type([[srcVertex]]) = [[Vertex]]
type([[src]]) $\circ$ [[sourceConnector]] = [[Vertex]] $\circ$ [[sourceConnector]]
= [[Transition]] $\in \mathcal{C}$: *true*
[[Transition]] $\leq$ [[Transition]] : *true*

type([[src]]) $\leq$ type([[srcToTrg]]) $\circ$ [[sourceConnectableElement]]
type([[src]]) = type([[srcVertex]]) $\leq$ [[Vertex]]
type([[srcToTrg]]) = type([[trans]]) = [[Transition]]
type([[srcToTrg]])$\circ$[[sourceConnectableElement]]=[[Transition]]$\circ$[[sourceConnectableElement]]
= [[Vertex]] $\in \mathcal{C}$: *true*
[[Vertex]] $\leq$ [[Vertex]] : *true*

type([[srcToTrg]]) $\leq$ type([[trg]]) $\circ$ [[targetConnector]]
type([[srcToTrg]]) = type([[trans]]) $\leq$ [[Transition]]
type([[trg]]) = type([[trgVertex]]) = [[Vertex]]
type([[trg]]) $\circ$ [[targetConnector]] = [[Vertex]] $\circ$ [[targetConnector]]
= [[Transition]] $\in \mathcal{C}$: *true*
[[Transition]] $\leq$ [[Transition]] : *true*

type([[trg]]) $\leq$ type([[srcToTrg]]) $\circ$ [[targetConnectableElement]]
type([[trg]]) = type([[trgVertex]]) $\leq$ [[Vertex]]
type([[srcToTrg]]) = type([[trans]]) = [[Transition]]
type([[srcToTrg]])$\circ$[[targetConnectableElement]]=[[Transition]]$\circ$[[targetConnectableElement]]
= [[Vertex]] $\in \mathcal{C}$: *true*
[[Vertex]] $\leq$ [[Vertex]] : *true*

Remaining constraints (Rules `R_prop9` , Rules `R_reflexive`)

type([[src]]) = [[Vertex]]
type([[trg]]) = [[Vertex]]
type([[srcToTrg]]) = [[Transition]]

type([[srcToTrg]]) $\leq$ type([[src]]) $\circ$ [[sourceConnector]]
$\Rightarrow$ [[Transition]] $\leq$ [[Vertex]] $\circ$ [[sourceConnector]] : *true*
type([[srcToTrg]]) $\leq$ type([[trg]]) $\circ$ [[targetConnector]]
$\Rightarrow$ [[Transition]] $\leq$ [[Vertex]] $\circ$ [[targetConnector]] : *true*
type([[src]]) $\leq$ type([[srcToTrg]]) $\circ$ [[sourceConnectableElement]]
$\Rightarrow$ [[Vertex]] $\leq$ [[Transition]] $\circ$ [[sourceConnectableElement]] : *true*
type([[trg]]) $\leq$ type([[srcToTrg]]) $\circ$ [[targetConnectableElement]]
$\Rightarrow$ [[Vertex]] $\leq$ [[Transition]] $\circ$ [[targetConnectableElement]] : *true*

$\Rightarrow$ ***No type error detected***

C.3.3 Method Call Analysis - 2nd Example

createConnector(ln, srcOutport, trgVertex)

[[ln]] $\in \mathcal{O}$ type([[ln]]) = [[Line]]
[[srcOutport]] $\in \mathcal{O}$ type([[srcOutport]]) = [[Outport]]
[[trgVertex]] $\in \mathcal{O}$ type([[trgVertex]]) = [[Vertex]]

1. Check Arguments (Rules `R_call1` and `R_call2`):

[[createConnector]]$_1 \in \mathcal{P}a$: *true*

[[srcToTrg]] $\in \mathcal{O}$
Θ([[CreateConnector]]$_1$) = [[Connector]] $\in \mathcal{C}$
[[CreateConnector]]$_1$ = [[srcToTrg:Connector]]
[[ln]] $\in \mathcal{O}$: *true*
type([[ln]]) = [[Line]] $\leq$ [[Connector]] : *true*
$\Rightarrow$ [[ln]] $\in \mathcal{A}rg$
arg([[createConnector]]$_1$) = [[ln]]

[[createConnector]]$_2 \in \mathcal{P}a$: *true*

[[src]] $\in \mathcal{O}$
Θ([[CreateConnector]]$_2$) = [[ConnectableElement]] $\in \mathcal{C}$
[[CreateConnector]]$_2$ = [[src:ConnectableElement]]
[[srcOutport]] $\in \mathcal{O}$: *true*
type([[srcOutport]]) = [[Outport]] $\leq$ [[ConnectableElement]] : *true*
$\Rightarrow$ [[srcOutport]] $\in \mathcal{A}rg$
arg([[createConnector]]$_2$) = [[srcOutport]]

[[createConnector]]$_3 \in \mathcal{P}a$: *true*

[[trg]] $\in \mathcal{O}$
Θ([[CreateConnector]]$_3$) = [[ConnectableElement]] $\in \mathcal{C}$
[[CreateConnector]]$_3$ = [[trg:ConnectableElement]]
[[trgVertex]] $\in \mathcal{O}$: *true*
type([[trgVertex]]) = [[Vertex]] $\leq$ [[ConnectableElement]] : *true*
$\Rightarrow$ [[trgVertex]] $\in \mathcal{A}rg$
arg([[createConnector]]$_3$) = [[trgVertex]]

2. Bind method declaration and method call (Rules `R_bind1`)

[[srcToTrg]] $\in (\mathcal{O} \cap \mathcal{P}a)$
arg([[srcToTrg]]) = arg([[createConnector]]$_1$) = [[ln]] $\in (\mathcal{O} \cap \mathcal{A}rg)$
$\Rightarrow$ type([[srcToTrg]]) = type([[ln]])

[[src]] $\in (\mathcal{O} \cap \mathcal{P}a)$
arg([[src]]) = arg([[createConnector]]$_2$) = [[srcOutport]] $\in (\mathcal{O} \cap \mathcal{A}rg)$
$\Rightarrow$ type([[src]]) = type([[srcOutport]])

[[trg]] $\in (\mathcal{O} \cap \mathcal{P}a)$
arg([[trg]]) = arg([[createConnector]]$_3$) = [[trgVertex]] $\in (\mathcal{O} \cap \mathcal{A}rg)$
$\Rightarrow$ type([[trg]]) = type([[trgVertex]])

3. Other matching rules (Rules `R_substitute2`, Rules `R_reflexive`)

type([[srcToTrg]]) $\leq$ type([[src]]) $\circ$ [[sourceConnector]]
type([[srcToTrg]]) = type([[ln]]) $\leq$ [[Line]]
type([[src]]) = type([[srcOutport]]) = [[Outport]]
type([[src]]) $\circ$ [[sourceConnector]] = [[Outport]] $\circ$ [[sourceConnector]]
= [[Line]] $\in \mathcal{C}$: *true*
[[Line]] $\leq$ [[Line]] : *true*

type([[src]]) $\leq$ type([[srcToTrg]]) $\circ$ [[sourceConnectableElement]]
type([[src]]) = type([[srcOutport]]) $\leq$ [[Outport]]
type([[srcToTrg]]) = type([[ln]]) = [[Line]]
type([[srcToTrg]])$\circ$[[sourceConnectableElement]]=[[Line]]$\circ$[[sourceConnectableElement]]
= [[Outport]] $\in \mathcal{C}$: *true*
[[Outport]] $\leq$ [[Outport]] : *true*

type([[srcToTrg]]) $\leq$ type([[trg]]) $\circ$ [[targetConnector]]
type([[srcToTrg]]) = type([[ln]]) $\leq$ [[Line]]
type([[trg]]) = type([[trgVertex]]) = [[Vertex]]
type([[trg]]) $\circ$ [[targetConnector]] = [[Vertex]] $\circ$ [[targetConnector]]
= [[Transition]] $\in \mathcal{C}$: *true*
([[Transition]] $\leq$ [[Line]]) $\vee$ ([[Line]] $\leq$ [[Transition]]) : *false*
$\Rightarrow$ ***Constraint unfulfilled***

type([[trg]]) $\leq$ type([[srcToTrg]]) $\circ$ [[targetConnectableElement]]
type([[trg]]) = type([[trgVertex]]) $\leq$ [[Vertex]]
type([[srcToTrg]]) = type([[ln]]) = [[Line]]
type([[srcToTrg]])$\circ$[[targetConnectableElement]]=[[Line]]$\circ$[[targetConnectableElement]]
= [[Inport]] $\in \mathcal{C}$: *true*
([[Inport]] $\leq$ [[Vertex]]) $\vee$ ([[Vertex]] $\leq$ [[Inport]]) : *false*
$\Rightarrow$ ***Constraint unfulfilled***

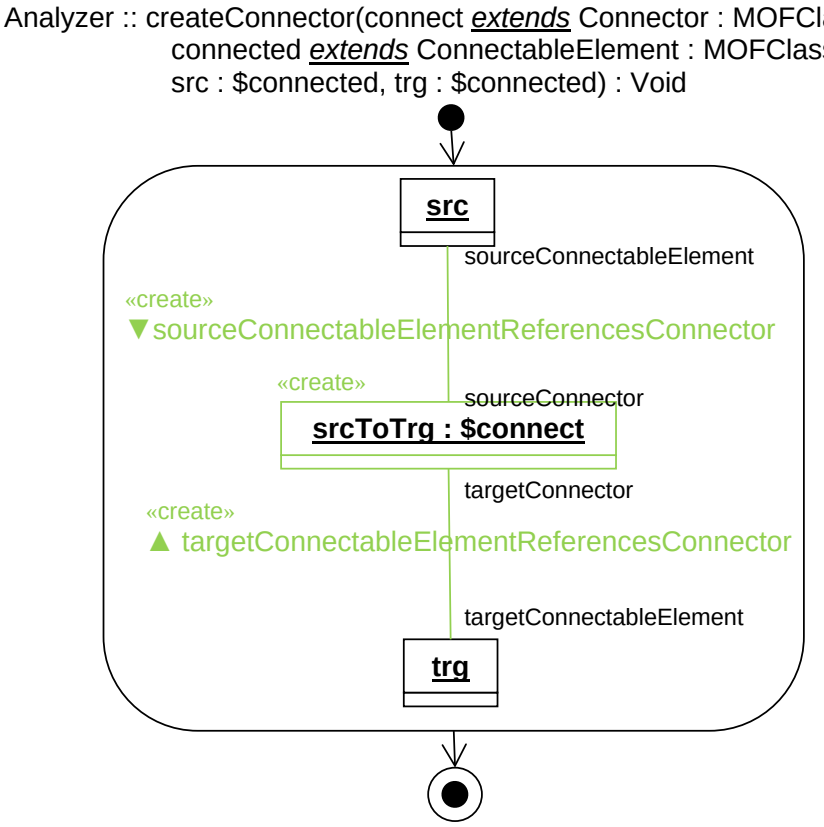
Remaining constraints (Rules `R_prop9` , Rules `R_reflexive`)

type([[src]]) = [[Outport]]
type([[trg]]) = [[Vertex]]
type([[srcToTrg]]) = [[Line]]

type([[srcToTrg]]) $\leq$ type([[src]]) $\circ$ [[sourceConnector]]
$\Rightarrow$ [[Line]] $\leq$ [[Outport]] $\circ$ [[sourceConnector]] : *true*
type([[srcToTrg]]) $\leq$ type([[trg]]) $\circ$ [[targetConnector]]
$\Rightarrow$ [[Line]] $\leq$ [[Vertex]] $\circ$ [[targetConnector]] : *false*
$\Rightarrow$ ***Constraint unfulfilled***
type([[src]]) $\leq$ type([[srcToTrg]]) $\circ$ [[sourceConnectableElement]]
$\Rightarrow$ [[Outport]] $\leq$ [[Line]] $\circ$ [[sourceConnectableElement]] : *true*
type([[trg]]) $\leq$ type([[srcToTrg]]) $\circ$ [[targetConnectableElement]]
$\Rightarrow$ [[Vertex]] $\leq$ [[Line]] $\circ$ [[targetConnectableElement]] : *false*
$\Rightarrow$ ***Constraint unfulfilled***

$\Rightarrow$ ***Type error***

C.4 Method with Generic Feature - 1



C.4.1 Method Specification Analysis

1. Signature Analysis

Parameters and Objects (rules `R_sem4`, `R_sem5`, `R_sem7` and `R_generic`):

$\mathcal{P}a$: { [[createConnector]]$_1$, [[createConnector]]$_2$, [[createConnector]]$_3$,
[[createConnector]]$_4$ }
= { [[connect *extends* Connector:MOFClass]],
[[connected *extends* ConnectableElement:MOFClass]],
[[src:$connected]], [[trg:$connected]] }
= { [[connect]], [[connected]], [[src]], [[trg]] }

Application of the Θ-operator (rules `R_theta1`, `R_theta4`, `R_generic`):

[[Connector]] $\in \mathcal{C}$: *true*
$\Rightarrow \Theta([[\text{createConnector}]]_1)$ = [[Connector]]
$\Rightarrow$ [[connect]] $\in \mathcal{C}$
$\Rightarrow$ [[connect]] $\leq$ [[Connector]]

[[ConnectableElement]] $\in \mathcal{C}$: *true*
$\Rightarrow \Theta([[\text{createConnector}]]_2)$ = [[ConnectableElement]]
$\Rightarrow$ [[connected]] $\in \mathcal{C}$
$\Rightarrow$ [[connected]] $\leq$ [[ConnectableElement]]

[[$connected]] = [[connected]] $\in \mathcal{C}$
$\Rightarrow$ [[src]] $\in \mathcal{O}$
$\Rightarrow \Theta([[\text{createConnector}]]_3)$ = [[connected]] $\in \mathcal{C}$
$\Rightarrow$ type([[src]]) $\leq \Theta([[\text{createConnector}]]_2)$
$\Rightarrow$ type([[src]]) $\leq$ [[connected]]

[[$connected]] = [[connected]] $\in \mathcal{C}$
$\Rightarrow$ [[trg]] $\in \mathcal{O}$
$\Rightarrow \Theta([[\text{createConnector}]]_4)$ = [[connected]] $\in \mathcal{C}$
$\Rightarrow$ type([[trg]]) $\leq \Theta([[\text{createConnector}]]_3)$
$\Rightarrow$ type([[trg]]) $\leq$ [[connected]]

2. Story Diagram Analysis

Bound objects (Rule `R_bound`):

[[src]] $\in \mathcal{O}$: *true*
[[trg]] $\in \mathcal{O}$: *true*

Unbound objects (Rules `R_unbound2` and `R_prop6`):

[[connect]] $\in \mathcal{P}a$: *true*
Θ([[connect]]) = [[Connector]] $\in \mathcal{C}$: *true*
$\Rightarrow$ [[srcToTrg]] $\in \mathcal{O}$
$\Rightarrow$ type([[srcToTrg]]) $\leq$ [[Connector]]

[[src]] $\in (\mathcal{O} \cap \mathcal{P}a)$
[[connect]] $\in \mathcal{P}a$
Θ([[src]]) = [[connected]] $\leq$ [[ConnectableElement]]
[[ConnectableElement]] $\circ$ [[sourceConnector]] = [[Connector]]
[[Connector]] $\leq$ [[ConnectableElement]] $\circ$ [[sourceConnector]] : *true*
$\Rightarrow$ type([[srcToTrg]]) $\leq$ type([[src]]) $\circ$ [[sourceConnector]]

[[trg]] $\in (\mathcal{O} \cap \mathcal{P}a)$
[[connect]] $\in \mathcal{P}a$
Θ([[trg]]) = [[connected]] $\leq$ [[ConnectableElement]]
[[ConnectableElement]] $\circ$ [[targetConnector]] = [[Connector]]
[[Connector]] $\leq$ [[ConnectableElement]] $\circ$ [[targetConnector]] : *true*
$\Rightarrow$ type([[srcToTrg]]) $\leq$ type([[trg]]) $\circ$ [[sourceConnector]]

Properties (Rules `R_prop9`:

type([[src]]), type([[trg]]), type([[srcToTrg]]): *???*
type([[srcToTrg]]) $\leq$ type([[src]]) $\circ$ [[sourceConnector]] : *???*
type([[srcToTrg]]) $\leq$ type([[trg]]) $\circ$ [[targetConnector]] : *???*

3. Resume of Type Information

$\mathcal{P}a$: { [[createConnector]]$_1$, [[createConnector]]$_2$, [[createConnector]]$_3$,
[[createConnector]]$_4$ }
= { [[connect extends Connector:MOFClass]],
[[connected extends ConnectableElement:MOFClass]],
[[src:$connected]], [[trg:$connected]] }
= { [[connect]], [[connected]], [[src]], [[trg]] }

Θ([[CreateConnector]]$_1$) = [[Connector]] $\in \mathcal{C}$
[[connect]] $\in \mathcal{C}$, [[connect]] $\leq$ [[Connector]]
Θ([[CreateConnector]]$_2$) = [[ConnectableElement]] $\in \mathcal{C}$
[[connected]] $\in \mathcal{C}$, [[connected]] $\leq$ [[ConnectableElement]]
Θ([[CreateConnector]]$_3$) = [[connected]] $\in \mathcal{C}$
Θ([[CreateConnector]]$_4$) = [[connected]] $\in \mathcal{C}$

$\mathcal{O}$: { [[src]], [[trg]], [[srcToTrg]] }

type([[src]]) $\leq$ Θ([[createConnector]]$_2$) $\leq$ [[ConnectableElement]]
type([[trg]]) $\leq$ Θ([[createConnector]]$_3$) $\leq$ [[ConnectableElement]]
type([[srcToTrg]]) $\leq$ [[Connector]]
type([[srcToTrg]]) $\leq$ type([[src]])$\circ$[[sourceConnector]]
type([[srcToTrg]]) $\leq$ type([[trg]])$\circ$[[targetConnector]]

Checked but non-fulfilled:
type([[srcToTrg]]) $\leq$ type([[src]]) $\circ$ [[sourceConnector]]
type([[srcToTrg]]) $\leq$ type([[trg]]) $\circ$ [[targetConnector]]

C.4.2 Method Call Analysis - 1st Example

createConnector(Transition, Vertex, srcVertex, trgVertex)

[[srcVertex]] $\in \mathcal{O}$ type([[srcVertex]]) = [[Vertex]]
[[trgVertex]] $\in \mathcal{O}$ type([[trgVertex]]) = [[Vertex]]

1st and 2nd argument:

Check Arguments (Rules `R_call1` and `R_call4`):

[[createConnector]]$_1 \in \mathcal{P}a$: *true*

Θ([[CreateConnector]]$_1$) = [[Connector]] $\in \mathcal{C}$: *true*
[[CreateConnector]]$_1$ = [[connect *extends* Connector:MOFClass]]
[[Transition]] $\in \mathcal{C}$: *true*
[[Transition]] $\leq$ [[Connector]] : *true*
$\Rightarrow$ [[Transition]] $\in \mathcal{A}rg$
$\Rightarrow$ arg([[CreateConnector]]$_1$) = [[Transition]]

[[createConnector]]$_2 \in \mathcal{P}a$: *true*

Θ([[CreateConnector]]$_2$) = [[ConnectableElement]] $\in \mathcal{C}$: *true*
[[CreateConnector]]$_2$ = [[connected *extends* ConnectableElement:MOFClass]]
[[Vertex]] $\in \mathcal{C}$: *true*
[[Vertex]] $\leq$ [[ConnectableElement]] : *true*
$\Rightarrow$ [[Vertex]] $\in \mathcal{A}rg$
$\Rightarrow$ arg([[CreateConnector]]$_2$) = [[Vertex]]

3rd and 4th argument:

Check Arguments (Rules `R_call1` and `R_call2`):

[[createConnector]]$_3 \in \mathcal{P}a$: *true*

[[src]] $\in \mathcal{O}$
Θ([[CreateConnector]]$_3$) = [[connected]] $\in \mathcal{C}$
[[connected]] $\leq$ [[ConnectableElement]] $\in \mathcal{C}$
[[createConnector]]$_3$ = [[src:$connected]]
[[srcVertex]] $\in \mathcal{O}$: *true*
type([[srcVertex]]) = [[Vertex]] $\leq$ [[ConnectableElement]] : *true*
$\Rightarrow$ [[srcVertex]] $\in \mathcal{A}rg$
arg([[createConnector]]$_3$) = [[srcVertex]]

[[createConnector]]$_4 \in \mathcal{P}a$: *true*

[[trg]] $\in \mathcal{O}$
Θ([[CreateConnector]]$_4$) = [[connected]] $\in \mathcal{C}$
[[connected]] $\leq$ [[ConnectableElement]] $\in \mathcal{C}$
[[createConnector]]$_4$ = [[trg:$connected]]
[[trgVertex]] $\in \mathcal{O}$: *true*
type([[trgVertex]]) = [[Vertex]] $\leq$ [[ConnectableElement]] : *true*
$\Rightarrow$ [[trgVertex]] $\in \mathcal{A}rg$
arg([[createConnector]]$_4$) = [[trgVertex]]

Bind method declaration and method call (Rules `R_bind6`, `R_bind3`):

[[Transition]] $\in (\mathcal{C} \cap \mathcal{A}rg)$
[[connect *extends* Connector:MOFClass]] $\in \mathcal{P}a$
arg([[connect *extends* Connector:MOFClass]]) = [[Transition]]
[[Transition]] $\leq$ [[Connector]] : *true*
$\Rightarrow$ [[connect]] = [[Transition]]

[[Vertex]] $\in (\mathcal{C} \cap \mathcal{A}rg)$
[[connected *extends* ConnectableElement:MOFClass]] $\in \mathcal{P}a$
arg([[connected *extends* ConnectableElement:MOFClass]]) = [[Vertex]]
[[Vertex]] $\leq$ [[ConnectableElement]] : *true*
$\Rightarrow$ [[connected]] = [[Vertex]]

[[src]], [[trg]] $\in \mathcal{O}$
[[connected]] $\in \mathcal{P}a$
[[src: $connected]], [[trg: $connected]] $\in \mathcal{P}a$
arg([[connected]]) = [[Vertex]] $\in (\mathcal{C} \cap \mathcal{A}rg)$
arg([[src: $connected]]) = [[srcVertex]] $\in (\mathcal{O} \cap \mathcal{A}rg)$
arg([[trg: $connected]]) = [[trgVertex]] $\in (\mathcal{O} \cap \mathcal{A}rg)$
type([[srcVertex]]) $\leq$ [[Vertex]] : *true*
$\Rightarrow$ type([[src]]) = type([[srcVertex]]) = [[Vertex]]
type([[trgVertex]]) $\leq$ [[Vertex]] : *true*
$\Rightarrow$ type([[trg]]) = type([[trgVertex]]) = [[Vertex]]

3. Other matching rules (Rules `R_bind2`, `R_substitute2`, `R_reflexive`)

[[srcToTrg]] $\in \mathcal{O}$
[[connect]] $\in \mathcal{P}a$
arg([[connect]]) = [[Transition]] ($\mathcal{C} \cap \mathcal{A}rg$)
$\Rightarrow$ type([[srcToTrg]]) = [[Transition]]

type([[srcToTrg]]) $\leq$ type([[src]]) $\circ$ [[sourceConnector]]
type([[srcToTrg]]) = [[Transition]]
type([[src]]) = type([[srcVertex]]) = [[Vertex]]
type([[src]]) $\circ$ [[sourceConnector]] = [[Vertex]] $\circ$ [[sourceConnector]]
= [[Transition]] $\in \mathcal{C}$: *true*
[[Transition]] $\leq$ [[Transition]] : *true*

type([[src]]) $\leq$ type([[srcToTrg]]) $\circ$ [[sourceConnectableElement]]
type([[src]]) = type([[srcVertex]]) $\leq$ [[Vertex]]
type([[srcToTrg]]) = [[Transition]]
type([[srcToTrg]]) $\circ$ [[sourceConnectableElement]] = [[Transition]] $\circ$ [[sourceConnectableElement]]
= [[Vertex]] $\in \mathcal{C}$: *true*
[[Vertex]] $\leq$ [[Vertex]] : *true*

type([[srcToTrg]]) $\leq$ type([[trg]]) $\circ$ [[targetConnector]]
type([[srcToTrg]]) = [[Transition]]
type([[trg]]) = type([[trgVertex]]) = [[Vertex]]
type([[trg]]) $\circ$ [[targetConnector]] = [[Vertex]] $\circ$ [[targetConnector]]
= [[Transition]] $\in \mathcal{C}$: *true*
[[Transition]] $\leq$ [[Transition]] : *true*

type([[trg]]) $\leq$ type([[srcToTrg]]) $\circ$ [[targetConnectableElement]]
type([[trg]]) = type([[trgVertex]]) $\leq$ [[Vertex]]
type([[srcToTrg]]) = [[Transition]]
type([[srcToTrg]]) $\circ$ [[targetConnectableElement]] = [[Transition]] $\circ$ [[targetConnectableElement]]
= [[Vertex]] $\in \mathcal{C}$: *true*
[[Vertex]] $\leq$ [[Vertex]] : *true*

Remaining constraints (Rules `R_prop9` , Rules `R_reflexive`)

type([[src]]) = [[Vertex]]
type([[trg]]) = [[Vertex]]
type([[srcToTrg]]) = [[Transition]]

type([[srcToTrg]]) $\leq$ type([[src]]) $\circ$ [[sourceConnector]]
$\Rightarrow$ [[Transition]] $\leq$ [[Vertex]] $\circ$ [[sourceConnector]] : *true*
type([[srcToTrg]]) $\leq$ type([[trg]]) $\circ$ [[targetConnector]]
$\Rightarrow$ [[Transition]] $\leq$ [[Vertex]] $\circ$ [[targetConnector]] : *true*

$\Rightarrow$ ***No type error detected***

C.4.3 Method Call Analysis - 2nd Example

createConnector(Transition, Port, srcOutport, trgInport)

[[srcOutport]] $\in \mathcal{O}$ type([[srcOutport]]) = [[Outport]]
[[trgInport]] $\in \mathcal{O}$ type([[trgInport]]) = [[Inport]]

1st and 2nd argument:

Check Arguments (Rules `R_call1` and `R_call4`):

[[createConnector]]$_1 \in \mathcal{P}a$: *true*

Θ([[CreateConnector]]$_1$) = [[Connector]] $\in \mathcal{C}$: *true*
[[CreateConnector]]$_1$ = [[connect *extends* Connector:MOFClass]]
[[Transition]] $\in \mathcal{C}$: *true*
[[Transition]] $\leq$ [[Connector]] : *true*
$\Rightarrow$ [[Transition]] $\in \mathcal{A}rg$
$\Rightarrow$ arg([[CreateConnector]]$_1$) = [[Transition]]

[[createConnector]]$_2 \in \mathcal{P}a$: *true*

Θ([[CreateConnector]]$_2$) = [[ConnectableElement]] $\in \mathcal{C}$: *true*
[[CreateConnector]]$_2$ = [[connected *extends* ConnectableElement:MOFClass]]
[[Port]] $\in \mathcal{C}$: *true*
[[Port]] $\leq$ [[ConnectableElement]] : *true*
$\Rightarrow$ [[Port]] $\in \mathcal{A}rg$
$\Rightarrow$ arg([[CreateConnector]]$_2$) = [[Port]]

3rd and 4th argument:

Check Arguments (Rules `R_call1` and `R_call2`):

[[createConnector]]$_3 \in \mathcal{P}a$: *true*

[[src]] $\in \mathcal{O}$
Θ([[CreateConnector]]$_3$) = [[connected]] $\in \mathcal{C}$
[[connected]] $\leq$ [[ConnectableElement]] $\in \mathcal{C}$
[[CreateConnector]]$_3$ = [[src:$connected]]
[[srcOutport]] $\in \mathcal{O}$: *true*
type([[srcOutport]]) = [[Outport]] $\leq$ [[ConnectableElement]] : *true*
$\Rightarrow$ [[srcOutport]] $\in \mathcal{A}rg$
$\Rightarrow$ arg([[CreateConnector]]$_3$) = [[srcOutport]]

[[createConnector]]$_4 \in \mathcal{P}a$: *true*

[[trg]] $\in \mathcal{O}$
Θ([[CreateConnector]]$_4$) = [[connected]] $\in \mathcal{C}$
[[connected]] $\leq$ [[ConnectableElement]] $\in \mathcal{C}$
[[CreateConnector]]$_4$ = [[trg:$connected]]
[[trgInport]] $\in \mathcal{O}$: *true*
type([[trgInport]]) = [[Inport]] $\leq$ [[ConnectableElement]] : *true*
$\Rightarrow$ arg([[CreateConnector]]$_4$) = [[trgInport]]

Bind method declaration and method call (Rules `R_bind6`, `R_bind3`):

[[Transition]] $\in (\mathcal{C} \cap \mathcal{A}rg)$
[[connect *extends* Connector:MOFClass]] $\in \mathcal{P}a$
arg([[connect *extends* Connector:MOFClass]]) = [[Transition]]
[[Transition]] $\leq$ [[Connector]] : *true*
$\Rightarrow$ [[connect]] = [[Transition]]

[[Port]] $\in (\mathcal{C} \cap \mathcal{A}rg)$
[[connected *extends* ConnectableElement:MOFClass]] $\in \mathcal{P}a$
arg([[connected *extends* ConnectableElement:MOFClass]]) = [[Port]]
[[Port]] $\leq$ [[ConnectableElement]] : *true*
$\Rightarrow$ [[connected]] = [[Port]]

[[src]], [[trg]] $\in \mathcal{O}$
[[connected]] $\in \mathcal{P}a$
[[src: $connected]], [[trg: $connected]] $\in \mathcal{P}a$
arg([[connected]]) = [[Port]] $\in (\mathcal{C} \cap \mathcal{A}rg)$
arg([[src: $connected]]) = [[srcOutport]] $\in (\mathcal{O} \cap \mathcal{A}rg)$
arg([[trg: $connected]]) = [[trgInport]] $\in (\mathcal{O} \cap \mathcal{A}rg)$
type([[srcOutport]]) = [[Outport]] $\leq$ [[Port]] : *true*
$\Rightarrow$ type([[src]]) = type([[srcOutport]]) = [[Outport]]
type([[trgInport]]) = [[Inport]] $\leq$ [[Port]] : *true*
$\Rightarrow$ type([[trg]]) = type([[trgInport]]) = [[Inport]]

3. Other matching rules (Rules `R_bind2`, `R_substitute2`)

[[srcToTrg]] $\in \mathcal{O}$
[[connect]] $\in \mathcal{P}a$
arg([[connect]]) = [[Transition]] ($\mathcal{C} \cap \mathcal{A}rg$)
$\Rightarrow$ type([[srcToTrg]]) = [[Transition]]

type([[src]]) $\leq$ type([[srcToTrg]]) $\circ$ [[sourceConnectableElement]]
type([[src]]) $\leq$ [[Outport]]
type([[srcToTrg]]) = [[Transition]]
type([[srcToTrg]])$\circ$[[sourceConnectableElement]]=[[Transition]]$\circ$[[sourceConnectableElement]]
= [[Vertex]] $\in \mathcal{C}$: *true*
([[Outport]] $\leq$ [[Vertex]]) $\vee$ ([[Vertex]] $\leq$ [[Outport]]) : *false*
$\Rightarrow$ ***Constraint unfulfilled***

type([[srcToTrg]]) $\leq$ type([[src]]) $\circ$ [[sourceConnector]]
type([[srcToTrg]]) $\leq$ [[Transition]]
type([[src]]) = type([[srcOutport]]) = [[Outport]]
type([[src]]) $\circ$ [[sourceConnector]] = [[Outport]] $\circ$ [[sourceConnector]]
= [[Line]] $\in \mathcal{C}$: *true*
([[Transition]] $\leq$ [[Line]]) $\vee$ ([[Line]] $\leq$ [[Transition]]) : *false*
$\Rightarrow$ ***Constraint unfulfilled***

type([[trg]]) $\leq$ type([[srcToTrg]]) $\circ$ [[targetConnectableElement]]
type([[trg]]) $\leq$ [[Inport]]
type([[srcToTrg]]) = [[Transition]]
type([[srcToTrg]])$\circ$[[targetConnectableElement]]=[[Transition]]$\circ$[[targetConnectableElement]]
= [[Vertex]] $\in \mathcal{C}$: *true*
([[Inport]] $\leq$ [[Vertex]]) $\vee$ ([[Vertex]] $\leq$ [[Inport]]) : *false*
$\Rightarrow$ ***Constraint unfulfilled***

type([[srcToTrg]]) $\leq$ type([[trg]]) $\circ$ [[targetConnector]]
type([[srcToTrg]]) $\leq$ [[Transition]]
type([[trg]]) = type([[trgInport]]) = [[Inport]]
type([[trg]]) $\circ$ [[targetConnector]] = [[Inport]] $\circ$ [[targetConnector]]
= [[Line]] $\in \mathcal{C}$: *true*
([[Transition]] $\leq$ [[Line]]) $\vee$ ([[Line]] $\leq$ [[Transition]]) : *false*
$\Rightarrow$ ***Constraint unfulfilled***

Remaining constraints (Rules `R_prop9` , Rules `R_reflexive`)

type([[src]]) = [[Outport]]
type([[trg]]) = [[Inport]]
type([[srcToTrg]]) = [[Transition]]

type([[srcToTrg]]) $\leq$ type([[src]]) $\circ$ [[sourceConnector]]
$\Rightarrow$ [[Transition]] $\leq$ [[Outport]] $\circ$ [[sourceConnector]] : *false*
$\Rightarrow$ ***Constraint unfulfilled***
type([[srcToTrg]]) $\leq$ type([[trg]]) $\circ$ [[targetConnector]]
$\Rightarrow$ [[Transition]] $\leq$ [[Inport]] $\circ$ [[targetConnector]] : *false*
$\Rightarrow$ ***Constraint unfulfilled***

$\Rightarrow$ ***Type error***

C.4.4 Method Call Analysis - 3rd Example

createConnector(Line, Port, srcInport, trgOutport)

[[srcOutport]] $\in \mathcal{O}$ type([[srcOutport]]) = [[Outport]]
[[trgOutport]] $\in \mathcal{O}$ type([[trgOutport]]) = [[Outport]]

1st and 2nd argument:

Check Arguments (Rules `R_call1` and `R_call4`):

[[createConnector]]$_1 \in \mathcal{P}a$: *true*

Θ([[CreateConnector]]$_1$) = [[Connector]] $\in \mathcal{C}$: *true*
[[CreateConnector]]$_1$ = [[connect *extends* Connector:MOFClass]]
[[Line]] $\in \mathcal{C}$: *true*
[[Line]] $\leq$ [[Connector]] : *true*
$\Rightarrow$ [[Line]] $\in \mathcal{A}rg$
$\Rightarrow$ arg([[CreateConnector]]$_1$) = [[Line]]

[[createConnector]]$_2 \in \mathcal{P}a$: *true*

Θ([[CreateConnector]]$_2$) = [[ConnectableElement]] $\in \mathcal{C}$: *true*
[[CreateConnector]]$_2$ = [[connected *extends* ConnectableElement:MOFClass]]
[[Port]] $\in \mathcal{C}$: *true*
[[Port]] $\leq$ [[ConnectableElement]] : *true*
$\Rightarrow$ [[Port]] $\in \mathcal{A}rg$
$\Rightarrow$ arg([[CreateConnector]]$_2$) = [[Port]]

3rd and 4th argument:

Check Arguments (Rules `R_call1` and `R_call2`):

[[createConnector]]$_3 \in \mathcal{P}a$: *true*

[[src]] $\in \mathcal{O}$
Θ([[CreateConnector]]$_3$) = [[connected]] $\in \mathcal{C}$
[[connected]] $\leq$ [[ConnectableElement]] $\in \mathcal{C}$
[[CreateConnector]]$_3$ = [[src:$connected]]
[[srcInport]] $\in \mathcal{O}$: *true*
type([[srcInport]]) = [[Inport]] $\leq$ [[ConnectableElement]] : *true*
$\Rightarrow$ [[srcInport]] $\in \mathcal{A}rg$
$\Rightarrow$ arg([[CreateConnector]]$_3$) = [[srcInport]]

[[createConnector]]$_4 \in \mathcal{P}a$: *true*

[[trg]] $\in \mathcal{O}$
Θ([[CreateConnector]]$_4$) = [[connected]] $\in \mathcal{C}$
[[connected]] $\leq$ [[ConnectableElement]] $\in \mathcal{C}$
[[CreateConnector]]$_4$ = [[trg:$connected]]
[[trgOutport]] $\in \mathcal{O}$: *true*
type([[trgOutport]]) = [[Outport]] $\leq$ [[ConnectableElement]] : *true*
$\Rightarrow$ [[trgOutport]] $\in \mathcal{A}rg$
$\Rightarrow$ arg([[CreateConnector]]$_4$) = [[trgOutport]]

Bind method declaration and method call (Rules `R_bind6`, `R_bind3`):

[[Line]] $\in (\mathcal{C} \cap \mathcal{A}rg)$
[[connect *extends* Connector:MOFClass]] $\in \mathcal{P}a$
arg([[connect *extends* Connector:MOFClass]]) = [[Line]]
[[Line]] $\leq$ [[Connector]] : *true*
$\Rightarrow$ [[connect]] = [[Line]]

[[Port]] $\in (\mathcal{C} \cap \mathcal{A}rg)$
[[connected *extends* ConnectableElement:MOFClass]] $\in \mathcal{P}a$
arg([[connected *extends* ConnectableElement:MOFClass]]) = [[Port]]
[[Port]] $\leq$ [[ConnectableElement]] : *true*
$\Rightarrow$ [[connected]] = [[Port]]

[[src]], [[trg]] $\in \mathcal{O}$, [[connected]] $\in \mathcal{P}a$
[[src: $connected]], [[trg: $connected]] $\in \mathcal{P}a$
arg([[connected]]) = [[Port]] $\in (\mathcal{C} \cap \mathcal{A}rg)$
arg([[src: $connected]]) = [[srcInport]] $\in (\mathcal{O} \cap \mathcal{A}rg)$
arg([[trg: $connected]]) = [[trgOutport]] $\in (\mathcal{O} \cap \mathcal{A}rg)$
type([[srcInport]]) = [[Inport]] $\leq$ [[Port]] : *true*
$\Rightarrow$ type([[src]]) = type([[srcInport]]) = [[Inport]]
type([[trgOutport]]) = [[Outport]] $\leq$ [[Port]] : *true*
$\Rightarrow$ type([[trg]]) = type([[trgOutport]]) = [[Outport]]

3. Other matching rules (Rules `R_bind2`, `R_substitute2`)

[[srcToTrg]] $\in \mathcal{O}$
[[connect]] $\in \mathcal{P}a$
arg([[connect]]) = [[Line]] ($\mathcal{C} \cap \mathcal{A}rg$)
$\Rightarrow$ type([[srcToTrg]]) = [[Line]]

type([[src]]) $\leq$ type([[srcToTrg]]) $\circ$ [[sourceConnectableElement]]
type([[src]]) $\leq$ [[Inport]]
type([[srcToTrg]]) = [[Line]]
type([[srcToTrg]])$\circ$[[sourceConnectableElement]]=[[Line]]$\circ$[[sourceConnectableElement]]
= [[Outport]] $\in \mathcal{C}$: *true*
([[Outport]] $\leq$ [[Inport]]) $\vee$ ([[Inport]] $\leq$ [[Outport]]) : *false*
$\Rightarrow$ ***Constraint unfulfilled***

type([[srcToTrg]]) $\leq$ type([[src]]) $\circ$ [[sourceConnector]]
type([[srcToTrg]]) $\leq$ [[Line]]
type([[src]]) = type([[srcInport]]) = [[Inport]]
type([[src]]) $\circ$ [[sourceConnector]] = [[Inport]] $\circ$ [[sourceConnector]]
= [[UndefinedConnector]] $\in \mathcal{C}$: *true*
([[UndefinedConnector]] $\leq$ [[Line]]) $\vee$ ([[Line]] $\leq$ [[UndefinedConnector]]) : *false*
$\Rightarrow$ ***Constraint unfulfilled***

type([[trg]]) $\leq$ type([[srcToTrg]]) $\circ$ [[targetConnectableElement]]
type([[trg]]) $\leq$ [[Outport]]
type([[srcToTrg]]) = [[Line]]
type([[srcToTrg]])$\circ$[[targetConnectableElement]]=[[Line]]$\circ$[[targetConnectableElement]]
= [[Inport]] $\in \mathcal{C}$: *true*
([[Inport]] $\leq$ [[Outport]]) $\vee$ ([[Outport]] $\leq$ [[Inport]]) : *false*
$\Rightarrow$ ***Constraint unfulfilled***

type([[srcToTrg]]) $\leq$ type([[trg]]) $\circ$ [[targetConnector]]
type([[srcToTrg]]) $\leq$ [[Line]]
type([[trg]]) = type([[trgOutport]]) = [[Outport]]
type([[trg]]) $\circ$ [[targetConnector]] = [[Outport]] $\circ$ [[targetConnector]]
= [[UndefinedConnector]] $\in \mathcal{C}$: *true*
([[UndefinedConnector]] $\leq$ [[Line]]) $\vee$ ([[Line]] $\leq$ [[UndefinedConnector]]) : *false*
$\Rightarrow$ ***Constraint unfulfilled***

Remaining constraints (Rules `R_prop9` , Rules `R_reflexive`)

type([[src]]) = [[Inport]]
type([[trg]]) = [[Outport]]
type([[srcToTrg]]) = [[Line]]

type([[srcToTrg]]) $\leq$ type([[src]]) $\circ$ [[sourceConnector]]
$\Rightarrow$ [[Line]] $\leq$ [[Inport]] $\circ$ [[sourceConnector]] : *false*
$\Rightarrow$ ***Constraint unfulfilled***
type([[srcToTrg]]) $\leq$ type([[trg]]) $\circ$ [[targetConnector]]
$\Rightarrow$ [[Line]] $\leq$ [[Outport]] $\circ$ [[targetConnector]] : *false*
$\Rightarrow$ ***Constraint unfulfilled***

$\Rightarrow$ ***Type error***

C.4.5 Method Call Analysis - 4^{th} Example

createConnector(Line, Outport, srcOutport, trgInport)

[[srcOutport]] $\in \mathcal{O}$ type([[srcOutport]]) = [[Outport]]
[[trgInport]] $\in \mathcal{O}$ type([[trgInport]]) = [[Inport]]

1^{st} and 2^{nd} argument:

Check Arguments (Rules `R_call1` and `R_call4`):

[[createConnector]]$_1 \in \mathcal{P}a$: *true*

Θ([[CreateConnector]]$_1$) = [[Connector]] $\in \mathcal{C}$: *true*
[[CreateConnector]]$_1$ = [[connect *extends* Connector:MOFClass]]
[[Line]] $\in \mathcal{C}$: *true*
[[Line]] $\leq$ [[Connector]] : *true*
$\Rightarrow$ [[Line]] $\in \mathcal{A}rg$
$\Rightarrow$ arg([[CreateConnector]]$_1$) = [[Line]]

[[createConnector]]$_2 \in \mathcal{P}a$: *true*

Θ([[CreateConnector]]$_2$) = [[ConnectableElement]] $\in \mathcal{C}$: *true*
[[CreateConnector]]$_2$ = [[connected *extends* ConnectableElement:MOFClass]]
[[Outport]] $\in \mathcal{C}$: *true*
[[Outport]] $\leq$ [[ConnectableElement]] : *true*
$\Rightarrow$ [[Outport]] $\in \mathcal{A}rg$
$\Rightarrow$ arg([[CreateConnector]]$_2$) = [[Outport]]

3^{nd} and 4^{rd} argument:

Check Arguments (Rules `R_call1` and `R_call2`):

[[createConnector]]$_3 \in \mathcal{P}a$: *true*

[[src]] $\in \mathcal{O}$
Θ([[CreateConnector]]$_3$) = [[connected]] $\in \mathcal{C}$
[[connected]] $\leq$ [[ConnectableElement]] $\in \mathcal{C}$
[[CreateConnector]]$_3$ = [[src:$connected]]
[[srcOutport]] $\in \mathcal{O}$: *true*
type([[srcOutport]]) = [[Outport]] $\leq$ [[ConnectableElement]] : *true*
$\Rightarrow$ [[srcOutport]] $\in \mathcal{A}rg$
$\Rightarrow$ arg([[CreateConnector]]$_3$) = [[srcOutport]]

[[createConnector]]$_4 \in \mathcal{P}a$: *true*

[[trg]] $\in \mathcal{O}$
Θ([[CreateConnector]]$_4$) = [[connected]] $\in \mathcal{C}$
[[connected]] $\leq$ [[ConnectableElement]] $\in \mathcal{C}$
[[CreateConnector]]$_4$ = [[trg:$connected]]
[[trgInport]] $\in \mathcal{O}$: *true*
type([[trgInport]]) = [[Inport]] $\leq$ [[ConnectableElement]] : *true*
$\Rightarrow$ [[trgInport]] $\in \mathcal{A}rg$
$\Rightarrow$ arg([[CreateConnector]]$_4$) = [[trgInport]]

Bind method declaration and method call (Rules `R_bind6`, `R_reflexive`, `R_bind3`):

[[Line]] $\in (\mathcal{C} \cap \mathcal{A}rg)$
[[connect *extends* Connector:MOFClass]] $\in \mathcal{P}a$
arg([[connect *extends* Connector:MOFClass]]) = [[Line]]
[[Line]] $\leq$ [[Connector]] : *true*
$\Rightarrow$ [[connect]] = [[Line]]

[[Outport]] $\in (\mathcal{C} \cap \mathcal{A}rg)$
[[connected *extends* ConnectableElement:MOFClass]] $\in \mathcal{P}a$
arg([[connected *extends* ConnectableElement:MOFClass]]) = [[Outport]]
[[Outport]] $\leq$ [[ConnectableElement]] : *true*
$\Rightarrow$ [[connected]] = [[Outport]]

[[src]], [[trg]] $\in \mathcal{O}$
[[connected]] $\in \mathcal{P}a$
[[src: $connected]], [[trg: $connected]] $\in \mathcal{P}a$
arg([[connected]]) = [[Outport]] $\in (\mathcal{C} \cap \mathcal{A}rg)$
arg([[src: $connected]]) = [[srcOutport]] $\in (\mathcal{O} \cap \mathcal{A}rg)$
arg([[trg: $connected]]) = [[trgInport]] $\in (\mathcal{O} \cap \mathcal{A}rg)$
type([[srcOutport]]) = [[Outport]] $\leq$ [[Outport]] : *true*
$\Rightarrow$ type([[src]]) = type([[srcOutport]]) = [[Outport]]
type([[trgInport]]) = [[Inport]] $\leq$ [[Outport]] : *false*
$\Rightarrow$ ***Constraint unfulfilled***

Remaining constraints (Rules `R_prop9` , Rules `R_reflexive`)

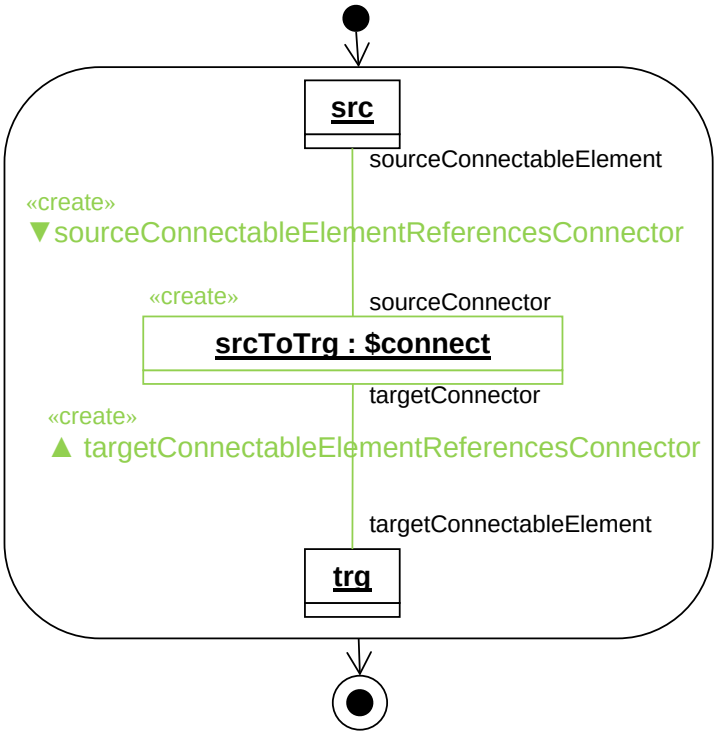
type([[src]]) = [[Outport]]
type([[trg]]) = [[Outport]]
type([[srcToTrg]]) = [[Line]]

type([[srcToTrg]]) $\leq$ type([[src]]) $\circ$ [[sourceConnector]]
$\Rightarrow$ [[Line]] $\leq$ [[Outport]] $\circ$ [[sourceConnector]] : *true*
type([[srcToTrg]]) $\leq$ type([[trg]]) $\circ$ [[targetConnector]]
$\Rightarrow$ [[Line]] $\leq$ [[Outport]] $\circ$ [[targetConnector]] : *false*
$\Rightarrow$ ***Constraint unfulfilled***

$\Rightarrow$ ***Type error***

C.5 Method with Generic Feature - 2

Analyzer :: createConnector(connect *extends* Connector : MOFClass,
src : $connect.sourceConnectableElement,
trg : $connect.targetConnectableElement) : Void



C.5.1 Method Specification Analysis

1. Signature Analysis

Parameters and Objects (rules `R_sem4`, `R_sem5`, `R_sem7` and `R_generic`):

$\mathcal{P}a$: { [[createConnector]]$_1$, [[createConnector]]$_2$, [[createConnector]]$_3$ }
= { [[connect *extends* Connector:MOFClass]], [[src:$connect.sourceConnectableElement]],
[[trg:$connect.targetConnectableElement]] }
= { [[connect]], [[src]], [[trg]] }

1st parameter (rules `R_theta4`):

[[Connector]] $\in \mathcal{C}$: *true*
$\Rightarrow \Theta([[\text{createConnector}]]_1)$ = [[Connector]]
$\Rightarrow$ [[connect]] $\in \mathcal{C}$
$\Rightarrow$ [[connect]] $\leq$ [[Connector]]

2nd parameter (rules `R_theta1`, `R_generic`, `R_inherit2`, `R_compos`):

[[$connect]] = [[connect]] $\leq$ [[Connector]] $\in \mathcal{C}$
[[sourceConnectableElement]] $\in \mathcal{P}$
[[Connector]]$\circ$[[sourceConnectableElement]] = [[ConnectableElement]] $\in \mathcal{C}$
$\Rightarrow$ [[connect]]$\circ$[[sourceConnectableElement]] $\in \mathcal{C}$
$\Rightarrow$ [[connect]]$\circ$[[sourceConnectableElement]] $\leq$ [[ConnectableElement]]

[[$connect]] = [[connect]] $\in \mathcal{C}$
[[sourceConnectableElement]] $\in \mathcal{P}$
[[connect]]$\circ$[[sourceConnectableElement]] $\in \mathcal{C}$: *true*
$\Rightarrow$ [[$connect.sourceConnectableElement]]
= [[connect]]$\circ$[[sourceConnectableElement]] $\in \mathcal{C}$

[[$connect.sourceConnectableElement]] $\in \mathcal{C}$
$\Rightarrow$ [[src]] $\in \mathcal{O}$
$\Rightarrow \Theta([[\text{createConnector}]]_2)$ = [[connect]]$\circ$[[sourceConnectableElement]]
$\Rightarrow$ type([[src]]) $\leq$ [[connect]]$\circ$[[sourceConnectableElement]]

3rd parameter (rules `R_theta1`, `R_generic`, `R_inherit2`, `R_compos`):

[[$connect]] = [[connect]] $\leq$ [[Connector]] $\in \mathcal{C}$
[[targetConnectableElement]] $\in \mathcal{P}$
[[Connector]]$\circ$[[targetConnectableElement]] = [[ConnectableElement]] $\in \mathcal{C}$
$\Rightarrow$ [[connect]]$\circ$[[targetConnectableElement]] $\in \mathcal{C}$
$\Rightarrow$ [[connect]]$\circ$[[targetConnectableElement]] $\leq$ [[ConnectableElement]]

[[$connect]] = [[connect]] $\in \mathcal{C}$
[[targetConnectableElement]] $\in \mathcal{P}$
[[connect]]$\circ$[[targetConnectableElement]] $\in \mathcal{C}$: *true*
$\Rightarrow$ [[$connect.targetConnectableElement]]
= [[connect]]$\circ$[[targetConnectableElement]] $\in \mathcal{C}$

[[$connect.sourceConnectableElement]] $\in \mathcal{C}$
$\Rightarrow$ [[trg]] $\in \mathcal{O}$
$\Rightarrow \Theta([[\text{createConnector}]]_2)$ = [[connect]]$\circ$[[targetConnectableElement]]
$\Rightarrow$ type([[trg]]) $\leq$ [[connect]]$\circ$[[targetConnectableElement]]

2. Story Diagram Analysis

Bound objects (Rule `R_bound`):

[[src]] $\in \mathcal{O}$: *true*
[[trg]] $\in \mathcal{O}$: *true*

Unbound objects (Rules `R_unbound2`, `R_prop6` and `R_reflexive`):

[[connect]] $\in \mathcal{P}a$
Θ([[connect]]) = [[Connector]] $\in \mathcal{C}$: *true*
$\Rightarrow$ [[srcToTrg]] $\in \mathcal{O}$
$\Rightarrow$ type([[srcToTrg]]) $\leq$ [[Connector]]

[[src]] $\in (\mathcal{O} \cap \mathcal{P}a)$
[[connect]] $\in \mathcal{P}a$
Θ([[src]]) = [[connect]]$\circ$[[sourceConnectableElement]]
$\leq$ [[ConnectableElement]]
Θ([[connect]]) = [[Connector]]
$\leq$ [[ConnectableElement]]$\circ$[[sourceConnector]] : *true*
$\Rightarrow$ type([[srcToTrg]]) $\leq$ type([[src]])$\circ$[[sourceConnector]]

[[trg]] $\in (\mathcal{O} \cap \mathcal{P}a)$
[[connect]] $\in \mathcal{P}a$
Θ([[trg]]) = [[connect]]$\circ$[[targetConnectableElement]]
$\leq$ [[ConnectableElement]]
Θ([[connect]]) = [[Connector]]
$\leq$ [[ConnectableElement]]$\circ$[[targetConnector]] : *true*
$\Rightarrow$ type([[srcToTrg]]) $\leq$ type([[trg]])$\circ$[[targetConnector]]

Properties (Rules `R_prop9`:

type([[src]]), type([[trg]]), type([[srcToTrg]]): *???*
type([[srcToTrg]]) $\leq$ type([[src]]) $\circ$ [[sourceConnector]] : *???*
type([[srcToTrg]]) $\leq$ type([[trg]]) $\circ$ [[targetConnector]] : *???*

3. Resume of Type Information

$\mathcal{P}a$: { [[createConnector]]$_1$, [[createConnector]]$_2$, [[createConnector]]$_3$ }
= { [[connect *extends* Connector:MOFClass]], [[src:$connect.sourceConnectableElement]],
[[trg:$connect.targetConnectableElement]] }
= { [[connect]], ([[src]]), ([[trg]]) }

Θ([[CreateConnector]]$_1$) = [[Connector]]
[[connect]] $\in \mathcal{C}$, [[connect]] $\leq$ [[Connector]]
Θ([[CreateConnector]]$_2$) = [[connect]]$\circ$[[sourceConnectableElement]] $\in \mathcal{C}$
Θ([[CreateConnector]]$_3$) = [[connect]]$\circ$[[targetConnectableElement]] $\in \mathcal{C}$

$\mathcal{O}$: { [[src]], [[trg]], [[srcToTrg]] }

type([[src]]) $\leq$ Θ([[createConnector]]$_2$) $\leq$ [[ConnectableElement]]
type([[trg]]) $\leq$ Θ([[createConnector]]$_3$) $\leq$ [[ConnectableElement]]
type([[srcToTrg]]) $\leq$ [[Connector]]
type([[srcToTrg]]) $\leq$ type([[src]])$\circ$[[sourceConnector]]
type([[srcToTrg]]) $\leq$ type([[trg]])$\circ$[[targetConnector]]

Checked but non-fulfilled:
type([[srcToTrg]]) $\leq$ type([[src]]) $\circ$ [[sourceConnector]]
type([[srcToTrg]]) $\leq$ type([[trg]]) $\circ$ [[targetConnector]]

C.5.2 Method Call Analysis - 1st Example

createConnector(Transition, srcVertex, trgVertex)

[[srcVertex]] $\in \mathcal{O}$ type([[srcVertex]]) = [[Vertex]]
[[trgVertex]] $\in \mathcal{O}$ type([[trgVertex]]) = [[Vertex]]

1st argument:

Check Arguments (Rules `R_call1` and `R_call4`):

[[createConnector]]$_1 \in \mathcal{P}a$: *true*

Θ([[createConnector]]$_1$) = [[Connector]] $\in \mathcal{C}$
[[createConnector]]$_1$ = [[connect *extends* Connector:MOFClass]]
[[Transition]] $\in \mathcal{C}$: *true*
[[Transition]] $\leq$ [[Connector]] : *true*
$\Rightarrow$ [[Transition]] $\in \mathcal{A}rg$
$\Rightarrow$ arg([[createConnector]]$_1$) = arg([[connect]]) = [[Transition]]

2nd and 3rd argument:

Check Arguments (Rules `R_call1`) and `R_call2`):

[[createConnector]]$_2 \in \mathcal{P}a$: *true*

[[src]] $\in \mathcal{O}$
Θ([[createConnector]]$_2$) = [[$connect.sourceConnectableElement]]
= [[connect]]$\circ$[[sourceConnectableElement]]
$\leq$ [[ConnectableElement]] $\in \mathcal{C}$
[[createConnector]]$_2$ = [[src:$connect.sourceConnectableElement]]
[[srcVertex]] $\in \mathcal{O}$: *true*
type([[srcVertex]]) = [[Vertex]] $\leq$ [[ConnectableElement]] : *true*
$\Rightarrow$ [[srcVertex]] $\in \mathcal{A}rg$
$\Rightarrow$ arg([[createConnector]]$_2$) = arg([[src]]) = [[srcVertex]]

[[createConnector]]$_3 \in \mathcal{P}a$: *true*

[[trg]] $\in \mathcal{O}$
Θ([[createConnector]]$_3$) = [[$connect.targetConnectableElement]]
= [[connect]]$\circ$[[targetConnectableElement]]
$\leq$ [[ConnectableElement]] $\in \mathcal{C}$
[[createConnector]]$_3$ = [[src:$connect.targetConnectableElement]]
[[trgVertex]] $\in \mathcal{O}$: *true*
type([[trgVertex]]) = [[Vertex]] $\leq$ [[ConnectableElement]] : *true*
$\Rightarrow$ [[trgVertex]] $\in \mathcal{A}rg$
$\Rightarrow$ arg([[createConnector]]$_3$) = arg([[trg]]) = [[trgVertex]]

Bind method declaration and method call (Rules `R_bind2`, `R_bind6`, `R_bind4`):

[[Transition]] $\in (\mathcal{C} \cap \mathcal{A}rg)$
[[connect *extends* Connector:MOFClass]] $\in \mathcal{P}a$
arg([[connect *extends* Connector:MOFClass]]) = arg([[connect]]) = [[Transition]]
[[Transition]] $\leq$ [[Connector]] : *true*
$\Rightarrow$ [[connect]] = [[Transition]]

[[srcToTrg]] $\in \mathcal{O}$
[[connect]] $\in \mathcal{P}a$
arg([[connect]]) = [[Transition]]
$\Rightarrow$ type([[srcToTrg]]) = [[Transition]]

[[src]] $\in \mathcal{O}$
[[connect]] $\in \mathcal{P}a$
[[sourceConnectableElement]] $\in \mathcal{P}$
[[src: $connect.sourceConnectableElement]] $\in \mathcal{P}a$
arg([[connect]]) = [[Transition]] $\in (\mathcal{C} \cap \mathcal{A}rg)$
arg([[src: $connect.sourceConnectableElement]]) = [[srcVertex]] $\in (\mathcal{O} \cap \mathcal{A}rg)$
type([[srcVertex]]) = [[Vertex]] $\leq$ [[Transition]]$\circ$[[sourceConnectableElement]] :
true
$\Rightarrow$ type([[src]]) = type([[srcVertex]]) = [[Vertex]]

[[trg]] $\in \mathcal{O}$
[[connect]] $\in \mathcal{P}a$
[[targetConnectableElement]] $\in \mathcal{P}$
[[trg: $connect.targetConnectableElement]] $\in \mathcal{P}a$
arg([[connect]]) = [[Transition]] $\in (\mathcal{C} \cap \mathcal{A}rg)$
arg([[trg: $connect.targetConnectableElement]]) = [[trgVertex]] $\in (\mathcal{O} \cap \mathcal{A}rg)$
type([[trgVertex]]) = [[Vertex]] $\leq$ [[Transition]]$\circ$[[targetConnectableElement]] :
true
$\Rightarrow$ type([[trg]]) = type([[trgVertex]]) = [[Vertex]]

3. Other matching rules (Rule `R_substitute2`)

type([[srcToTrg]]) $\leq$ type([[src]]) $\circ$ [[sourceConnector]]
type([[srcToTrg]]) $\leq$ [[Connector]]
type([[src]]) = type([[srcVertex]]) = [[Vertex]]
type([[src]]) $\circ$ [[sourceConnector]] = [[Vertex]] $\circ$ [[sourceConnector]]
= [[Transition]] $\in \mathcal{C}$: *true*
[[Transition]] $\leq$ [[Connector]] : *true*

type([[srcToTrg]]) $\leq$ type([[trg]]) $\circ$ [[targetConnector]]
type([[srcToTrg]]) $\leq$ [[Connector]]
type([[trg]]) = type([[trgVertex]]) = [[Vertex]]
type([[trg]]) $\circ$ [[targetConnector]] = [[Vertex]] $\circ$ [[targetConnector]]
= [[Transition]] $\in \mathcal{C}$: *true*
[[Transition]] $\leq$ [[Connector]] : *true*

Remaining constraints (Rules `R_prop9` , Rules `R_reflexive`)

type([[src]]) = [[Vertex]]
type([[trg]]) = [[Vertex]]
type([[srcToTrg]]) = [[Transition]]

type([[srcToTrg]]) $\leq$ type([[src]]) $\circ$ [[sourceConnector]]
$\Rightarrow$ [[Transition]] $\leq$ [[Vertex]] $\circ$ [[sourceConnector]] : *true*
type([[srcToTrg]]) $\leq$ type([[trg]]) $\circ$ [[targetConnector]]
$\Rightarrow$ [[Transition]] $\leq$ [[Vertex]] $\circ$ [[targetConnector]] : *true*

$\Rightarrow$ ***No type error detected***

C.5.3 Method Call Analysis - 2nd Example

createConnector(Line, srcOutport, trgOutport)

[[srcInport]] $\in \mathcal{O}$ type([[srcInport]]) = [[Outport]]
[[trgOutport]] $\in \mathcal{O}$ type([[trgOutport]]) = [[Outport]]

1st argument:

Check Arguments (Rules `R_call1` and `R_call4`):

[[createConnector]]$_1 \in \mathcal{P}a$: *true*

Θ([[CreateConnector]]$_1$) = [[Connector]] $\in \mathcal{C}$
[[CreateConnector]]$_1$ = [[connect *extends* Connector:MOFClass]]
[[Line]] $\in \mathcal{C}$: *true*
[[Line]] $\leq$ [[Connector]] : *true*
$\Rightarrow$ [[Line]] $\in \mathcal{A}rg$
$\Rightarrow$ arg([[CreateConnector]]$_1$) = arg([[connect]]) = [[Line]]

2nd and 3rd argument:

Check Arguments (Rules `R_call1` and `R_call2`):

[[createConnector]]$_2 \in \mathcal{P}a$: *true*

[[src]] $\in \mathcal{O}$
Θ([[createConnector]]$_2$) = [[$connect.sourceConnectableElement]]
= [[connect]]$\circ$[[sourceConnectableElement]]
$\leq$ [[ConnectableElement]] $\in \mathcal{C}$
[[createConnector]]$_2$ = [[src:$connect.sourceConnectableElement]]
[[srcOutport]] $\in \mathcal{O}$: *true*
type([[srcOutport]]) = [[Outport]] $\leq$ [[ConnectableElement]] : *true*
$\Rightarrow$ [[srcOutport]] $\in \mathcal{A}rg$
$\Rightarrow$ arg([[createConnector]]$_2$) = arg([[src]]) = [[srcOutport]]

[[createConnector]]$_3 \in \mathcal{P}a$: *true*

[[trg]] $\in \mathcal{O}$
Θ([[createConnector]]$_3$) = [[$connect.targetConnectableElement]]
= [[connect]]$\circ$[[targetConnectableElement]]
$\leq$ [[ConnectableElement]] $\in \mathcal{C}$
[[createConnector]]$_3$ = [[trg:$connect.targetConnectableElement]]
[[trgOutport]] $\in \mathcal{O}$: *true*
type([[trgOutport]]) = [[Outport]] $\leq$ [[ConnectableElement]] : *true*
$\Rightarrow$ [[trgOutport]] $\in \mathcal{A}rg$
$\Rightarrow$ arg([[createConnector]]$_3$) = arg([[trg]]) = [[trgOutport]]

Bind method declaration and method call (Rules `R_bind2`, `R_bind6`, `R_bind4`):

[[Line]] $\in (\mathcal{C} \cap \mathcal{A}rg)$
[[connect *extends* Connector:MOFClass]] $\in \mathcal{P}a$
arg([[connect *extends* Connector:MOFClass]]) = arg([[connect]]) = [[Line]]
[[Line]] $\leq$ [[Connector]] : *true*
$\Rightarrow$ [[connect]] = [[Line]]

[[srcToTrg]] $\in \mathcal{O}$
[[connect]] $\in \mathcal{P}a$
arg([[connect]]) = [[Line]]
$\Rightarrow$ type([[srcToTrg]]) = [[Line]]

[[src]] $\in \mathcal{O}$
[[connect]] $\in \mathcal{P}a$
[[sourceConnectableElement]] $\in \mathcal{P}$
[[src: $connect.sourceConnectableElement]] $\in \mathcal{P}a$
arg([[connect]]) = [[Line]] $\in (\mathcal{C} \cap \mathcal{A}rg)$
arg([[src: $connect.sourceConnectableElement]]) = [[srcOutport]] $\in (\mathcal{O} \cap \mathcal{A}rg)$
type([[srcOutport]]) = [[Outport]] $\leq$ [[Line]]$\circ$[[sourceConnectableElement]] : *true*
$\Rightarrow$ type([[src]]) = type([[srcOutport]]) = [[Outport]]

[[trg]] $\in \mathcal{O}$
[[connect]] $\in \mathcal{P}a$
[[targetConnectableElement]] $\in \mathcal{P}$
[[trg: $connect.targetConnectableElement]] $\in \mathcal{P}a$
arg([[connect]]) = [[Line]] $\in (\mathcal{C} \cap \mathcal{A}rg)$
arg([[trg: $connect.targetConnectableElement]]) = [[trgOutport]] $\in (\mathcal{O} \cap \mathcal{A}rg)$
type([[trgOutport]]) = [[Outport]]
[[Line]]$\circ$[[targetConnectableElement]] = [[Inport]]
([[Outport]] $\leq$ [[Inport]]) $\vee$ ([[Inport]] $\leq$ [[Outport]]) : *false*
$\Rightarrow$ ***Constraint unfulfilled***

Remaining constraints (Rules `R_prop9` , Rules `R_reflexive`)

type([[src]]) = [[Outport]]
type([[trg]]) = ??? *(could not be bound, see above)*
type([[srcToTrg]]) = [[Line]]

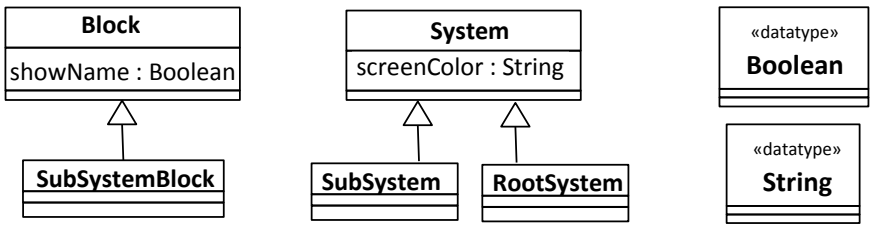
type([[srcToTrg]]) $\leq$ type([[src]]) $\circ$ [[sourceConnector]]
$\Rightarrow$ [[Transition]] $\leq$ [[Vertex]] $\circ$ [[sourceConnector]] : *true*
type([[srcToTrg]]) $\leq$ type([[trg]]) $\circ$ [[targetConnector]] : *false*
$\Rightarrow$ ***Constraint unfulfilled***

$\Rightarrow$ ***Type error***

Appendix D

Type System Evaluation

D.1 Metamodel excerpt:



R_sem1:
$\mathcal{C}$: { [[Block]], [[SubSystemBlock]], [[System]], [[SubSystem]], [[RootSystem]] }

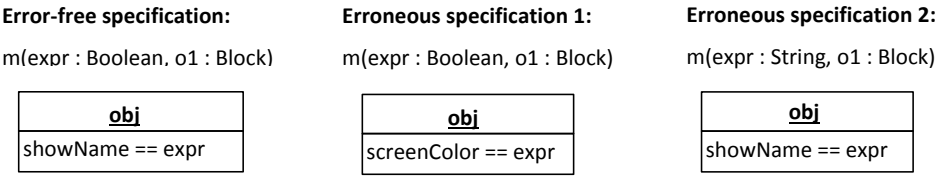
R_sem2:
$\mathcal{D}t$: { [[String]], [[Boolean]] }

R_inherit1:
[[SubSystemBlock]] $\leq$ [[Block]]
[[SubSystem]] $\leq$ [[System]]
[[RootSystem]] $\leq$ [[System]]

R_prop2:
$\mathcal{P}$: { [[showName]], [[screenColor]] }
[[Block]]$\circ$[[showName]] = [[Boolean]] $\in \mathcal{D}t$
[[System]]$\circ$[[screenColor]] = [[String]] $\in \mathcal{D}t$

R_inherit3:
[[SubSystemBlock]]$\circ$[[showName]] = [[Block]]$\circ$[[showName]] = [[Boolean]]
[[SubSystem]]$\circ$[[screenColor]] = [[System]]$\circ$[[screenColor]] = [[String]]
[[RootSystem]]$\circ$[[screenColor]] = [[System]]$\circ$[[screenColor]] = [[String]]

D.2 Bound object, non-parameterized attribute:



D.2.1 Error-free specification:

R_sem8: $[[m]]_1 = [[exp:Boolean]] = [[exp]] \in \mathcal{P}a$
R_theta2: $\Theta([[exp]]) = [[Boolean]] \in \mathcal{D}t$
R_sem5: $[[m]]_2 = [[obj:Block]] = [[obj]] \in \mathcal{O}$
R_theta1: $[[obj]] \in \mathcal{O}$ $\Theta([[obj]]) = [[Block]] \in \mathcal{C}$ $type([[obj]]) \leq [[Block]]$

R_bound: $[[obj]] \in \mathcal{O}$: *true*
R_prop8: $[[showName]] \in \mathcal{P}$ $type([[obj]]) \circ [[showName]] = [[Boolean]] \in \mathcal{D}t$
R_prop3: $type([[obj]]) \circ [[showName]] = \Theta([[exp]]) = [[Boolean]] \in \mathcal{D}t$: *true*

Use and Misuse Cases

Use case: m(true, blockObj)
where$[[true]] \in \mathcal{V}$, $dType([[true]]) = [[Boolean]]$,
$[[blockObj]] \in \mathcal{O}$, $type([[blockObj]]) \leq [[Block]]$

Misuse case: m('red', blockObj)
where$[['red']] \in \mathcal{V}$, $dType([['red']]) = [[String]]$,
$[[blockObj]] \in \mathcal{O}$, $type([[blockObj]]) \leq [[Block]]$

Error type: not compliant with the method signature (wrong argument).
Detected by R_call6:
$\Theta([[m]]_1) = [[Boolean]] \in \mathcal{D}t$
$[['red']] \in \mathcal{V}$: *true*
$dType([['red']]) = [[String]]$
$\Theta([[m]]_1) = dType([['red']])$: *false*

Misuse case 2: m(true, systemObj)
where[[true]] $\in \mathcal{V}$, dType([[true]]) = [[Boolean]],
[[systemObj]] $\in \mathcal{O}$, type([[systemObj]]) $\leq$ [[System]]

Error type: not compliant with the method signature (wrong argument).
Detected by `R_call2` and `R_inheritSingle`:
[[systemObj]] $\in \mathcal{O}$
$\Theta([[m]]_2)$ = [[Block]] $\in \mathcal{C}$
[[systemObj]] $\in \mathcal{O}$: *true*
type([[systemObj]]) $\leq$ [[Block]] : *false*

D.2.2 Erroneous specification 1:

`R_sem8`: $[[m]]_1$ = [[exp:Boolean]] = [[exp]] $\in \mathcal{P}a$
`R_theta2`: Θ([[exp]]) = [[Boolean]] $\in \mathcal{D}t$
`R_sem5`: $[[m]]_2$ = [[obj:Block]] = [[obj]] $\in \mathcal{O}$
`R_theta1`: [[obj]] $\in \mathcal{O}$
Θ([[obj]]) = [[Block]] $\in \mathcal{C}$
type([[obj]]) $\leq$ [[Block]]

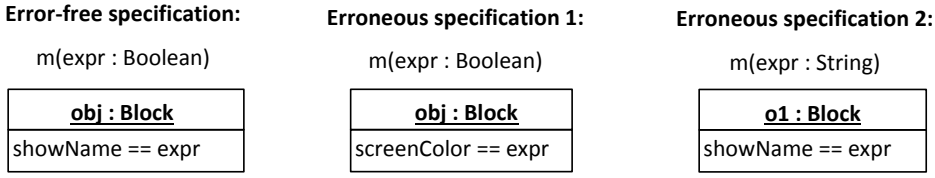
`R_bound`: [[obj]] $\in \mathcal{O}$: *true*
`R_prop8`: [[screenColor]] $\in \mathcal{P}$: *true*
type([[obj]])$\circ$[[screenColor]] $\in \mathcal{D}t$: *false*

D.2.3 Erroneous specification 2:

`R_sem8`: $[[m]]_1$ = [[exp:String]] = [[exp]] $\in \mathcal{P}a$
`R_theta2`: Θ([[exp]]) = [[String]] $\in \mathcal{D}t$
`R_sem5`: $[[m]]_2$ = [[obj:Block]] = [[obj]] $\in \mathcal{O}$
`R_theta1`: [[obj]] $\in \mathcal{O}$
Θ([[obj]]) = [[Block]] $\in \mathcal{C}$
type([[obj]]) $\leq$ [[Block]]

`R_bound`: [[obj]] $\in \mathcal{O}$: *true*
`R_prop8`: [[showName]] $\in \mathcal{P}$: *true*
type([[obj]])$\circ$[[showName]] $\in \mathcal{D}t$: *true*
`R_prop3`: type([[obj]])$\circ$[[showName]] = [[Boolean]]
Θ([[exp]]) = [[String]]
type([[obj]])$\circ$[[showName]] = Θ([[exp]]) : *false*

D.3 Non-parameterized Class, non-parameterized Attribute:



D.3.1 Error-free Specification:

R_sem8: $[[m]]_1 = [[exp:Boolean]] = [[exp]] \in \mathcal{P}a$
R_theta2: $\Theta([[exp]]) = [[Boolean]] \in \mathcal{D}t$

R_unbound1: $[[obj]] \in \mathcal{O}$
$type([[obj]]) \leq [[Block]]$
R_prop8: $[[showName]] \in \mathcal{P}$: *true*
$type([[obj]]) \circ [[showName]] = [[Boolean]] \in \mathcal{D}t$: *true*
R_prop3: $type([[obj]]) \circ [[showName]] = \Theta([[exp]]) = [[Boolean]] \in \mathcal{D}t$: *true*

Use and Misuse Cases

Use case: m(true)
where$[[true]] \in \mathcal{V}$, $dType([[true]]) = [[Boolean]]$,

Misuse case m('red')
where$[['red']] \in \mathcal{V}$, $dType([['red']]) = [[String]]$,

Error type: not compliant with the method signature (wrong argument).
Detected by R_call6:
$\Theta([[m]]_1) = [[Boolean]] \in \mathcal{D}t$
$[['red']] \in \mathcal{V}$: *true*
$dType([['red']]) = [[String]]$
$\Theta([[m]]_1) = dType([['red']])$: *false*

D.3.2 Erroneous Specification 1:

R_sem8: $[[m]]_1 = [[exp:Boolean]] = [[exp]] \in \mathcal{P}a$
R_theta2: $\Theta([[exp]]) = [[Boolean]] \in \mathcal{D}t$

R_unbound1: $[[obj]] \in \mathcal{O}$
$type([[obj]]) \leq [[Block]]$
R_prop8: $[[screenColor]] \in \mathcal{P}$: *true*
$type([[obj]]) \circ [[screenColor]] \in \mathcal{D}t$: *false*

D.3.3 Erroneous Specification 2:

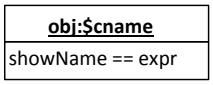
R_sem8: $[[m]]_1$ = [[exp:String]] = [[exp]] $\in \mathcal{P}a$
R_theta2: Θ([[exp]]) = [[String]] $\in \mathcal{D}t$

R_unbound1: [[obj]] $\in \mathcal{O}$
type([[obj]]) $\leq$ [[Block]]
R_prop8: [[showName]] $\in \mathcal{P}$: *true*
type([[obj]])$\circ$[[showName]] = [[Boolean]] $\in \mathcal{D}t$: *true*
R_prop3: type([[obj]])$\circ$[[showName]] = [[Boolean]]
Θ([[exp]]) = [[String]]
type([[obj]])$\circ$[[showName]] = Θ([[exp]]) : *false*

D.4 Parameterized Class, non-parameterized Attribute:

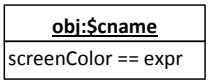
Error-free specification:

m(expr : Boolean, cname *extends* Block: MOFClass)



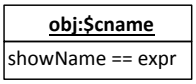
Erroneous specification 1:

m(expr : Boolean, cname *extends* Block: MOFClass)



Erroneous specification 2:

m(expr : String, cname *extends* Block: MOFClass)



D.4.1 Error-free Specification:

R_sem8: $[[m]]_1 = [[exp:Boolean]] = [[exp]] \in \mathcal{P}a$
R_theta2: $\Theta([[exp]]) = [[Boolean]] \in \mathcal{D}t$
R_sem7: $[[m]]_2 = [[cname\ \underline{extends}\ Block]] = [[cname]] \in \mathcal{P}a$
R_theta4: $\Theta([[m]]_2) = [[Block]] \in \mathcal{C}$
$[[cname]] \in \mathcal{C}$
$[[cname]] \leq [[Block]]$

R_unbound2: $\Theta([[cname]]) \in \mathcal{C}$: *true*
$[[obj]] \in \mathcal{O}$
$type([[obj]]) \leq [[Block]]$
R_prop8: $[[showName]] \in \mathcal{P}$: *true*
$type([[obj]]) \circ [[showName]] = [[Boolean]] \in \mathcal{D}t$: *true*
R_prop3: $type([[obj]]) \circ [[showName]] = \Theta([[exp]])$: *true*

Use and Misuse Cases

Use case: m(true, SubSystemBlock)
where$[[true]] \in \mathcal{V}$, dType($[[true]]$) = [[Boolean]],
$[[SubSystemBlock]] \in \mathcal{C}$, $[[SubSystemBlock]] \leq [[Block]]$

Misuse case 1: m('red', SubSystemBlock)
where[['red']] $\in \mathcal{V}$, dType([['red']]) = [[String]],
[[SubSystemBlock]] $\in \mathcal{C}$, [[SubSystemBlock]] $\leq$ [[Block]]

Error type: not compliant with the method signature (wrong argument).
Detected by `R_call6`:
$\Theta([[m]]_1) \in \mathcal{D}t$
[['red']]$\in \mathcal{V}$: *true*
dType([['red']]) = [[String]]
$\Theta([[m]]_1)$ = [[Boolean]]
dType([['red']]) = $\Theta([[m]]_1)$: *false*

Misuse case 2: m(true, SubSystem)
where[[true]] $\in \mathcal{V}$, dType([[true]]) = [[Boolean]],
[[SubSystem]] $\in \mathcal{C}$

Error type: not compliant with the method signature (wrong argument).
Detected by `R_call4`:
[[SubSystem]] $\in \mathcal{C}$: *true*
$\Theta([[m]]_2)$ = [[Block]]
[[SubSystem]] $\leq$ [[Block]] : *false*

D.4.2 Erroneous Specification 1:

`R_sem8`: $[[m]]_1$ = [[exp:Boolean]] = [[exp]] $\in \mathcal{P}a$
`R_theta2`: $\Theta([[exp]])$ = [[Boolean]] $\in \mathcal{D}t$
`R_sem7`: $[[m]]_2$ = [[cname *extends* Block]] = [[cname]] $\in \mathcal{P}a$
`R_theta4`: $\Theta([[m]]_2)$ = [[Block]] $\in \mathcal{C}$
[[cname]] $\in \mathcal{C}$
[[cname]] $\leq$ [[Block]]

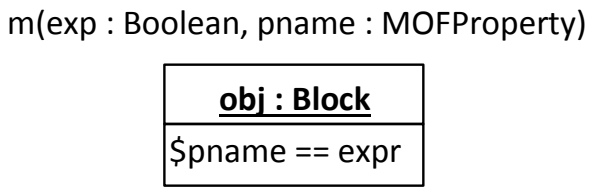
`R_unbound2`: $\Theta([[cname]]) \in \mathcal{C}$: *true*
[[obj]] $\in \mathcal{O}$
type([[obj]]) $\leq$ [[Block]]
`R_prop8`: [[screenColor]] $\in \mathcal{P}$: *true*
type([[obj]])$\circ$[[screenColor]] $\in \mathcal{D}t$: *false*

D.4.3 Erroneous Specification 2:

R_sem8: $[[m]]_1$ = [[exp:String]] = [[exp]] $\in \mathcal{P}a$
R_theta2: Θ([[exp]]) = [[String]] $\in \mathcal{D}t$
R_sem7: $[[m]]_2$ = [[cname *extends* Block]] = [[cname]] $\in \mathcal{P}a$
R_theta4: $\Theta([[m]]_2)$ = [[Block]] $\in \mathcal{C}$
    [[cname]] $\in \mathcal{C}$
    [[cname]] $\leq$ [[Block]]

R_unbound2: Θ([[cname]]) $\in \mathcal{C}$: *true*
    [[obj]] $\in \mathcal{O}$
    type([[obj]]) $\leq$ [[Block]]
R_prop8: [[showName]] $\in \mathcal{P}$: *true*
    type([[obj]])$\circ$[[showName]] = [[Boolean]] $\in \mathcal{D}t$: *true*
R_prop3: type([[obj]])$\circ$[[showName]] = [[Boolean]]
Θ([[exp]]) = [[String]]
type([[obj]])$\circ$[[showName]] = Θ([[exp]]) : *false*

D.5 Non-parameterized Class, parameterized Attribute:



R_sem8: $[[m]]_1 = [[exp:Boolean]] = [[exp]] \in \mathcal{P}a$
R_theta2: $\Theta([[m]]_1) = [[Boolean]] \in \mathcal{D}t$
R_sem9: $[[m]]_1 = [[pname:MOFProperty]] = [[pname]] \in \mathcal{P}a$
R_theta5: $\Theta([[m]]_2) = [[pname]] \in \mathcal{P}$

R_unbound1: $[[obj]] \in \mathcal{O}$
$type([[obj]]) \leq [[Block]]$

Use and Misuse Cases

Use case: m(true, showName)
where$[[true]] \in \mathcal{V}$, dType([[true]]) = [[Boolean]],
$[[showName]] \in \mathcal{P}$

R_call6: $[[true]] \in \mathcal{V}$: *true*
dType([[true]]) = $[[Boolean]] = \Theta([[m]]_1)$: *true*
$[[true]] \in \mathcal{A}rg$
$arg([[m]]_1) = [[true]]$
R_call5: $[[showName]] \in \mathcal{P}$: true
$[[showName]] \in \mathcal{A}rg$
$arg([[m]]_2) = [[showName]]$

R_bind7: $type([[obj]]) \circ [[showName]] = [[Boolean]] \in \mathcal{D}t$: *true*
$type([[obj]]) \circ [[pname]] = type([[obj]]) \circ [[showName]]$
R_bind8: $type([[obj]]) \circ [[showName]] \in \mathcal{D}t$: *true*
dType([[true]]) = [[Boolean]]
$type([[obj]]) \circ [[showName]] = dType([[true]])$: *true*

Misuse case 1: m('red', showName)
where[['red']] $\in \mathcal{V}$, dType([['red']]) = [[String]],
[[showName]] $\in \mathcal{P}$

Error type: value's datatype does not correspond to the attribute
Error detected by `R_bind7` and `R_bind8`:
`R_bind7`: type([[obj]])$\circ$[[showName]] = [[Boolean]] $\in \mathcal{D}t$: *true*
type([[obj]])$\circ$[[pname]] = type([[obj]])$\circ$[[showName]]
`R_bind8`: type([[obj]])$\circ$[[showName]] $\in \mathcal{D}t$: *true*
dType([['red']]) = [[String]]
type([[obj]])$\circ$[[showName]] = dType([['red']]) : *false*

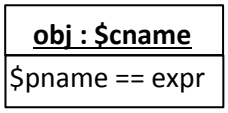
Misuse case 2: m(true, screenColor)
where[[true]] $\in \mathcal{V}$, dType([[true]]) = [[Boolean]],
[[screenColor]] $\in \mathcal{P}$

Error type: value's datatype does not correspond to the attribute
Error detected by `R_bind7`
`R_bind7`: type([[obj]])$\circ$[[screenColor]] $\in \mathcal{D}t$: *false*

D.6 Parameterized Class, parameterized Attribute:

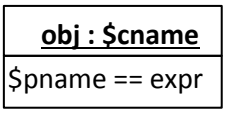
Error-free specification 1:

m(cname *extends* Block: MOFClass , exp : Boolean, pname : MOFProperty)



Error-free specification 2:

m(cname *extends* System: MOFClass , exp : String, pname : MOFProperty)



D.6.1 Error-free Specification 1:

R_sem7: $[[m]]_1 = [[\text{cname } \textit{extends} \text{ Block}]] = [[\text{cname}]] \in \mathcal{P}a$
R_theta4: $\Theta([[m]]_1) = [[\text{Block}]] \in \mathcal{C}$
R_sem8: $[[m]]_2 = [[\text{exp:Boolean}]] = [[\text{exp}]] \in \mathcal{P}a$
R_theta2: $\Theta([[m]]_2) = [[\text{Boolean}]] \in \mathcal{D}t$
R_sem9: $[[m]]_3 = [[\text{pname:MOFProperty}]] = [[\text{pname}]] \in \mathcal{P}a$
R_theta5: $\Theta([[m]]_3) = [[\text{pname}]] \in \mathcal{P}$

R_unbound2: $\Theta([[\text{cname}]]) \in \mathcal{C} : \textit{true}$
$[[\text{obj}]] \in \mathcal{O}$
$\text{type}([[\text{obj}]]) \leq [[\text{Block}]]$

Use and Misuse Cases

Use case: m(SubSystemBlock, true, showName)
where$[[\text{SubSystemBlock}]] \leq [[\text{Block}]] \in \mathcal{C}$,
$[[\text{true}]] \in \mathcal{V}$, dType($[[\text{true}]]$) = [[Boolean]]
$[[\text{showName}]] \in \mathcal{P}$

R_call4 : $[[\text{SubSystemBlock}]] \in \mathcal{C} : \textit{true}$
$\Theta([[m]]_1) = [[\text{Block}]]$
$[[\text{SubSystemBlock}]] \leq \Theta([[m]]_1) : \textit{true}$
$[[\text{SubSystemBlock}]] \in \mathcal{A}rg$
$\text{arg}([[m]]_1) = [[\text{SubSystemBlock}]]$

R_call6: [[true]] $\in \mathcal{V}$: *true*
dType([[true]]) = [[Boolean]] = $\Theta([[m]]_2)$: *true*
[[true]] $\in \mathcal{A}rg$
arg($[[m]]_2$) = [[true]]
R_call5: [[showName]] $\in \mathcal{P}$: true
[[showName]] $\in \mathcal{A}rg$
arg($[[m]]_3$) = [[showName]]
R_bind2 : type([[obj]]) = [[SubSystemBlock]]
R_bind7: type([[obj]])$\circ$[[showName]] = [[Boolean]] $\in \mathcal{D}t$: *true*
type([[obj]])$\circ$[[pname]] = type([[obj]])$\circ$[[showName]]
R_bind8: type([[obj]])$\circ$[[showName]] $\in \mathcal{D}t$: *true*
dType([[true]]) = [[Boolean]]
type([[obj]])$\circ$[[showName]] = dType([[true]]) : *true*

Misuse case 1: m(SubSystemBlock, **'red'**, showName)
where[[SubSystemBlock]] $\leq$ [[Block]] $\in \mathcal{C}$,
[['red']] $\in \mathcal{V}$, dType([['red']]) = [[String]]
[[showName]] $\in \mathcal{P}$

Error type: not compliant with the method signature (wrong argument).
Detected by R_call6 : [['red']] $\in \mathcal{V}$: *true*
dType([['red']]) = [[String]]
$\Theta([[m]]_2)$ = [[Boolean]]
dType([['red']]) = $\Theta([[m]]_2)$: *false*

Misuse case 2: m(SubSystemBlock, true, **screenColor**)
where[[SubSystemBlock]] $\leq$ [[Block]] $\in \mathcal{C}$,
[[true]] $\in \mathcal{V}$, dType([[true]]) = [[Boolean]]
[[screenColor]] $\in \mathcal{P}$

Error type: value's datatype does not correspond to the attribute
Error detected by R_bind7: type([[obj]])$\circ$[[screenColor]] $\in \mathcal{D}t$: *false*

Misuse case 3: m(**SubSystem**, true, showName)
where[[SubSystemBlock]] $\leq$ [[Block]] $\in \mathcal{C}$,
[[true]] $\in \mathcal{V}$, dType([[true]]) = [[Boolean]]
[[showName]] $\in \mathcal{P}$

Error type: not compliant with the method signature (wrong argument).
Detected by R_call4 : [[SubSystem]] $\in \mathcal{C}$: *true*
$\Theta([[m]]_1)$ = [[Block]]
[[SubSystem]] $\leq \Theta([[m]]_1)$: *false*

Misuse case 4+: Combination of the previous miscases

D.6.2 Error-free specification 2:

R_sem7: $[[m]]_1 = [[\text{cname } \underline{\textit{extends}} \text{ System}]] = [[\text{cname}]] \in \mathcal{P}a$
R_theta4: $\Theta([[m]]_1) = [[\text{System}]] \in \mathcal{C}$
R_sem8: $[[m]]_2 = [[\text{exp:String}]] = [[\text{exp}]] \in \mathcal{P}a$
R_theta2: $\Theta([[m]]_2) = [[\text{String}]] \in \mathcal{D}t$
R_sem9: $[[m]]_3 = [[\text{pname:MOFProperty}]] = [[\text{pname}]] \in \mathcal{P}a$
R_theta5: $\Theta([[m]]_3) = [[\text{pname}]] \in \mathcal{P}$

R_unbound2: $\Theta([[\text{cname}]]) \in \mathcal{C}$: *true*
$[[\text{obj}]] \in \mathcal{O}$
$\text{type}([[\text{obj}]]) \leq [[\text{System}]]$

Use and Misuse Cases

Use case: m(SubSystem,'red', screenColor)
where$[[\text{SubSystem}]] \leq [[\text{System}]] \in \mathcal{C}$
$[['\text{red}']] \in \mathcal{V}$, $\text{dType}([['\text{red}']]) = [[\text{String}]]$,
$[[\text{screenColor}]] \in \mathcal{P}$,

R_call4 : $[[\text{SubSystem}]] \in \mathcal{C}$: *true*
$\Theta([[m]]_1) = [[\text{System}]]$
$[[\text{SubSystem}]] \leq \Theta([[m]]_1)$: *true*
$[[\text{SubSystem}]] \in \mathcal{A}rg$
$\text{arg}([[m]]_1) = [[\text{SubSystem}]]$
R_call6: $[['\text{red}']] \in \mathcal{V}$: *true*
$\text{dType}([['\text{red}']]) = [[\text{String}]] = \Theta([[m]]_2)$: *true*
$[['\text{red}']] \in \mathcal{A}rg$
$\text{arg}([[m]]_2) = [['\text{red}']]$
R_call5: $[[\text{screenColor}]] \in \mathcal{P}$: true
$[[\text{screenColor}]] \in \mathcal{A}rg$
$\text{arg}([[m]]_3) = [[\text{screenColor}]]$

R_bind2 : $\text{type}([[\text{obj}]]) = [[\text{SubSystem}]]$
R_bind7: $\text{type}([[\text{obj}]]) \circ [[\text{screenColor}]] = [[\text{String}]] \in \mathcal{D}t$: *true*
$\text{type}([[\text{obj}]]) \circ [[\text{pname}]] = \text{type}([[\text{obj}]]) \circ [[\text{screenColor}]]$
R_bind8: $\text{type}([[\text{obj}]]) \circ [[\text{screenColor}]] \in \mathcal{D}t$: *true*
$\text{dType}([['\text{red}']]) = [[\text{String}]]$
$\text{type}([[\text{obj}]]) \circ [[\text{screenColor}]] = \text{dType}([[\text{true}]])$: *true*

Misuse case 1: m(SubSystem, **true**, screenColor)
where[[SubSystem]] $\leq$ [[System]] $\in \mathcal{C}$,
[[true]] $\in \mathcal{V}$, dType([[true]]) = [[Boolean]]
[[screenColor]] $\in \mathcal{P}$

Error type: not compliant with the method signature (wrong argument).
Detected by `R_call6` : [[true]] $\in \mathcal{V}$: *true*
dType([[true]]) = [[Boolean]]
$\Theta([[m]]_2)$ = [[String]]
dType([[true]]) = $\Theta([[m]]_2)$: *false*

Misuse case 2: m(SubSystem, 'red', **showName**)
where[[SubSystem]] $\leq$ [[System]] $\in \mathcal{C}$,
[['red']] $\in \mathcal{V}$, dType([['red']]) = [[String]]
[[showName]] $\in \mathcal{P}$

Error type: value's datatype does not correspond to the attribute
Error detected by `R_bind7`: type([[obj]])$\circ$[[showName]] $\in \mathcal{D}t$: *false*

Misuse case 3: m(**SubSystemBlock**, 'red', screenColor)
where[[SubSystemBlock]] $\leq$ [[Block]] $\in \mathcal{C}$,
[['red']] $\in \mathcal{V}$, dType([['red']]) = [[String]]
[[screenColor]] $\in \mathcal{P}$

Error type: not compliant with the method signature (wrong argument).
Detected by `R_call4` : [[SubSystemBlock]] $\in \mathcal{C}$: *true*
$\Theta([[m]]_1)$ = [[System]]
[[SubSystemBlock]] $\leq \Theta([[m]]_1)$: *false*

Misuse case 4+: Combination of the previous miscases

Bibliography

[AKRS06] C. Amelunxen, A. Königs, T. Rötschke, and A. Schürr. MOSL: Composing a Visual Language for a Metamodeling Framework. In J. Grundy and J. Howse, editors, IEEE Symposium on Visual Languages and Human-Centric Computing (VLHCC 2006), Los Alamitos, September 2006. IEEE Computer Society Press.

[ALPS11] A. Anjorin, M. Lauder, S. Patzina, and A. Schürr. eMoflon: Leveraging EMF and Professional CASE Tools. In 3. Workshop Methodische Entwicklung von Modellierungswerkzeugen (MEMWe2011), 2011.

[Ame09] C. Amelunxen. Metamodel-based Design Rule Checking and Enforcement. PhD thesis, Technische Universität Darmstadt, 2009.

[AR06] C. Amelunxen and T. Rötschke. Creating Well-Structured Specifications in MOFLON. In A. Zündorf and D. Varro, editors, Proceedings of the 3rd International Workshop on Graph Based Tools (GraBaTs 2006), volume 1 of Electronic Communications of EASST, September 2006.

[ATLa] ATL Homepage. `http://www.eclipse.org/atl/`.

[ATLb] ATL Use Case. `http://www.eclipse.org/m2m/atl/doc/ATLUseCase_Families2Persons.pdf`.

[ATLc] ATL Documentation. `http://wiki.eclipse.org/index.php/ATL`.

[Atl11] AtlanMode Homepage. `http://www.emn.fr/z-info/atlanmod`, 2011.

[AVK+05] Aditya Agrawal, Attila Vizhanyo, Zsolt Kalmar, Feng Shi, Anantha Narayanan, and Gabor Karsai. Reusable idioms and patterns in graph transformation languages. Electron. Notes Theor. Comput. Sci., 127(1):181–192, March 2005.

[BAG+92] Henk Barendregt, S. Abramsky, D. M. Gabbay, T. S. E. Maibaum, and H. P. Barendregt. Lambda calculi with types. In Handbook of Logic in Computer Science, pages 117–309. Oxford University Press, 1992.

[Bar05] J. Bartlett. The art of metaprogramming. `http://www.ibm.com/developerworks/linux/library/l-metaprog1.html`, 2005.

[BBG+06] Jean B'ezivin, Fabian B"uttner, Martin Gogolla, Fr'ed'eric Jouault, Ivan Kurtev, and Arne Lindow. Model Transformations? Transformation Models! pages 440–453, 2006.

[BE02] Jon Barwise and John Etchemendy. Language, Proof and Logic. Center for the Study of Language and Inf, 2002.

[Bic03] Lutz Bichler. A flexible code generator for mof-based modeling languages. In in 2nd OOPSLA Workshop on Generative Techniques in the context of Model Driven Architecture. 2003, 2003.

[BNvBK06] Daniel Balasubramanian, Anantha Narayanan, Christopher van Buskirk, and Gabor Karsai. The graph rewriting and transformation language: Great. Electronic Communications of the EASST, 1, 2006.

[BW07] Jörg Bauer and Reinhard Wilhelm. Static analysis of dynamic communication systems by partner abstraction. In Static Analysis, 14th International Symposium, SAS 2007, pages 249–264, 2007.

[Car97] Luca Cardelli. Type systems. In Allen B. Tucker, editor, The Computer Science and Engineering Handbook, pages 2208–2236. CRC Press, 1997.

[CC02] Peter Chen Computer and Peter P. Chen. Entity-relationship modeling: Historical events, future trends, and lessons learned. In In: Software Pioneers: Contributions to Software Engineering, pages 297–310. Springer, 2002.

[CCGdL08] Jordi Cabot, Robert Clarisó, Esther Guerra, and Juan de Lara. Analysing graph transformation rules through ocl. In Antonio Vallecillo, Jeff Gray, and Alfonso Pierantonio, editors, ICMT, volume 5063 of Lecture Notes in Computer Science, pages 229–244. Springer, 2008.

[CCGdL10] Jordi Cabot, Robert Clarisó, Esther Guerra, and Juan de Lara. Verification and validation of declarative model-to-model transformations through invariants. Journal of Systems and Software, 83(2):283–302, 2010.

[CG06] D. Costal and C. Gómez. On the use of association redefinition in uml class diagrams. In Conceptual Modeling - ER 2006, Lecture Notes in Computer Science, pages 513–527. Springer, 2006.

[CH03] Krzysztof Czarnecki and Simon Helsen. Classification of model transformation approaches. In OOPSLA'03 Workshop on Generative Techniques in the Context of Model-Driven Architecture, 2003.

[CH06] K. Czarnecki and S. Helsen. Feature-based survey of model transformation approaches. IBM Syst. J., 45:621–645, July 2006.

[Cho57] Noam Chomsky. Syntactic Structures. Mouton, The Hague, Netherlands, 1957.

[Chu36] Alonzo Church. An unsolvable problem of elementary number theory. American Journal of Mathematics, 58(2):345–363, April 1936.

[CM97] Manuel Clavel and José Meseguer. Reflection in Rewriting Logic and its Applications in the Maude Language. In Proceedings of IMSA-97, pages 128–139, Japan, 1997.

[CW85] Luca Cardelli and Peter Wegner. On understanding types, data abstraction, and polymorphism. ACM Computing Surveys, 17(4):471–522, 1985.

[DHJ+07] F. Drewes, B. Hoffmann, D. Janssens, M. Minas, and N. Van Eetvelde. Shaped Generic Graph Transformation. In A. Schürr, M. Nagl, and A. Zündorf, editors, Proceedings of the Third International Symposium on Applications of Graph Transformations with Industrial Relevance, pages 201–216, October 2007.

[Dia13] DiaGen/DiaMeta Homepage. `http://www.unibw.de/inf2/DiaGen/`, 2013.

[Die10] Reinhard Diestel. Graph Theory. Springer-Verlag, fourth edition, July 2010.

[Dir02] Ravi Dirckze. Java Metadata Interface Specification - Version 1.0. Unisys, June 2002.

[DLa] D Programming Language Homepage. `http://dlang.org`.

[DLC10] Samba Diaw, Redouane Lbath, and Bernard Coulette. Etat de l'art sur le développement logiciel basé sur les transformations de modèles. Technique et Science Informatiques, Ingénierie Dirigée par les Modèles, 29(4-5/2010):505–536, June 2010.

[dLG12] Juan de Lara and Esther Guerra. Reusable graph transformation templates. In Proceedings of the 4th international conference on Applications of Graph Transformations with Industrial Relevance, AGTIVE'11, pages 35–50. Springer-Verlag, 2012.

[Dre07] XSL Transformations (XSLT) Version 2.0 - W3C Recommendation 23 January 2007. http://www.w3.org/TR/xslt20/, January 2007.

[Dre11] Dresden OCL Toolkit Homepage. http://www.dresden-ocl.org/index.php/DresdenOCL, 2011.

[DRP99] Elfriede Dustin, Jeff Rashka, and John Paul. Automated software testing: introduction, management, and performance. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[EA11] Enterprise Architect - Product page. http://www.sparxsystems.com.au/, 2011.

[Ecl11] The Eclipse Foundation open source community website. http://www.eclipse.org/, 2011.

[EGdL+05a] Karsten Ehrig, Esther Guerra, Juan de Lara, Laszló Lengyel, Tihamér Levendovszky, Ulrike Prange, Gabriele Taentzer, Dániel Varró, and Szilvia Varró-Gyapay. Model transformation by graph transformation: A comparative study. In MTiP 2005, International Workshop on Model Transformations in Practice (Satellite Event of MoDELS 2005), Montego Bay, Jamaica, October 2005.

[EGdL+05b] Karsten Ehrig, Esther Guerra, Juan de Lara, Laszló Lengyel, Tihamér Levendovszky, Ulrike Prange, Gabriele Taentzer, Dániel Varró, and Szilvia Varró-Gyapay. Model transformation by graph transformation: A comparative study. In MTiP 2005, International Workshop on Model Transformations in Practice (Satellite Event of MoDELS 2005), 2005.

[EMF11] Eclipse Modeling Framework Project - Homepage. http://http://www.eclipse.org/modeling/emf//, 2011.

[FHR06] T. Farkas, C. Hein, and T. Ritter. Automatic Evaluation of Modeling Rules and Design Guidelines. In European Conference on Model Driven Architecture - Foundations and Applications (ECMDA2006), 2006.

[FMRS07] Christian Fuss, Christof Mosler, Ulrike Ranger, and Erhard Schultchen. The jury is still out: A comparison of agg, fujaba, and progres. ECEASST, 6, 2007.

[For82] Charles L. Forgy. Rete: A fast algorithm for the many pattern/many object pattern match problem. Artificial Intelligence, 19(1):17–37, 1982.

[Fow99] Martin Fowler. Refactoring: Improving the Design of Existing Code. Addison-Wesley, Boston, MA, USA, 1999.

[Fri02] Jeffrey E. F. Friedl. Mastering Regular Expressions. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2 edition, 2002.

[FUJ11] FUJABA Homepage. `http://www.fujaba.de/`, 2011.

[gen] Generic Java Homepage. `http://homepages.inf.ed.ac.uk/wadler/gj/`.

[GGKH03] Tracy Gardner, Catherine Griffin, Jana Koehler, and Rainer Hauser. A review of OMG MOF 2.0 Query / Views / Transformations Submissions and Recommendations towards the final Standard. `http://www.win.tue.nl/ipa/archive/springdays2005/gardner-etal.pdf`, July 2003.

[GJ06] P. Van Gorp and D. Janssens. Copying subgraphs within model repositories. In Fifth International Workshop on Graph Transformation and Visual Modeling Techniques, Electronic Notes in Theoretical Computer Science, pages 127–139. Elsevier, 2006.

[Gor08] Pieter Van Gorp. Model-driven Development of Model Transformations. Ph.D thesis, University of Antwerp, Dept. of Mathematics and Computer Science, 2008.

[GSR05] L. Geiger, C. Schneider, and C. Reckord. Template- and modelbased code generation for MDA-Tools. In 3rd International Fujaba Days, Paderborn, Germany, September 2005.

[GY06] Tracy Gardner and Larry Yusuf. Explore model-driven development (MDD) and related approaches: A closer look at model-driven development and other industry initiatives, March 2006.

[Has] Haskell Homepage. `http://www.haskell.org`.

[Hec06] Reiko Heckel. Graph transformation in a nutshell. Electronic Notes in Theoretical Computer Science, 148:187–198, 2006.

[HHS02] Bastiaan Heeren, Jurriaan Hage, and Doaitse Swierstra. Generalizing Hindley-Milner Type Inference Algorithms. Technical report, Department of Information and Computing Sciences, Utrecht University, 2002.

[Hin69] R. Hindley. The Principal Type-Scheme of an Object in Combinatory Logic. Trans. Amer. Math. Soc, 146:29–60, December 1969.

[Hin11] K. Hinkelmann. Forward chaining vs. backward chaining. http://www.hinkelmann.ch/knut/lectures/KE/KE-4_FC_vs_BC.pdf, 2011. Knowledge Engineering Courses, University of Applied Sciences Northwester Switzerland.

[IEC10] IEC and Functional Safety Homepage. http://www.iec.ch/functionalsafety/, 2010.

[ISO10] ISO Homepage. http://www.iso.org/iso/home.html, 2010.

[J.94] Durkin J. Expert Systems: Design and Development. Macmillan Publishing Company, 1994.

[Jav] Java 2 Platform Standard Edition 5.0 - Release Notes - New Features and Enhancements. http://download.oracle.com/javase/1,5.0/docs/relnotes/features.html.

[JGB05] Guy Steele James Gosling, Bill Joy and Gilad Bracha. The Java Language Specification - Third Edition. Sun Microsystems, 2005.

[KGZ09] Jochen Malte Küster, Thomas Gschwind, and Olaf Zimmermann. Incremental development of model transformation chains using automated testing. In MoDELS, pages 733–747, 2009.

[KM08] Pierre Kelsen and Qin Ma. A Lightweight Approach for Defining the Formal Semantics of a Modeling Language. In MoDELS 2008, LNCS 5301, pages 690–704, 2008.

[Kön05] Alexander Königs. Model transformation with triple graph grammars. In In Model Transformations in Practice Satellite Workshop of MODELS 2005, Montego, 2005.

[Kur05] Ivan Kurtev Kurtev. Adaptability of model transformations. PhD thesis, Enschede, the Netherlands, May 2005.

[Kur08] Ivan Kurtev. Application of reflection in model transformation languages. In ICMT '08: Proceedings of the 1st international conference on Theory and Practice of Model Transformations, pages 199–213, Berlin, Heidelberg, 2008. Springer-Verlag.

[Kur10] Ivan Kurtev. Application of reflection in a model transformation language. Software and System Modeling, 9(3):311–333, 2010.

[Lab05] Daniel Clemente Laboreo. Introduction to natural deduction. http://homepage.univie.ac.at/christian.damboeck/ps06/clemente_nat_ded.pdf, May 2005.

[LBA10] Levi Lucio, Bruno Barroca, and Vasco Amaral. A technique for automatic validation of model transformations. In Model Driven Engineering Languages and Systems, 13th International Conference, MODELS 2010, pages 136–150, 2010.

[LvdG91] Peter J. F. Lucas and Linda C. van der Gaag. Principles of expert systems. International computer science series. Addison-Wesley, 1991.

[LZ74] Barbara Liskov and Stephen Zilles. Programming with abstract data types. In Proceedings of the ACM SIGPLAN symposium on Very high level languages, pages 50–59, New York, NY, USA, 1974. ACM.

[LZG05] Yuehua Lin, Jing Zhang, and Jeff Gray. A Testing Framework for Model Transformations. pages 219–236. 2005.

[MAA] MathWorks Automotive Advisory Board Homepage. http://www.mathworks.com/industries/auto/maab.html.

[MAJ] MAJA Homepage.

[Mat00] MathWorks. Stateflow - User's Guide, Version 4, 2000.

[Mat07] MathWorks Automotive Advisory Board. Control Algorithm Modeling Guidelines using MATLAB,Simulink, and Stateflow - Version 2.1, July 2007.

[Mat10a] MathWorks. Simulink 7 - User's Guide, pages 380–398. 2010.

[Mat10b] MathWorks. Stateflow and Stateflow Coder 7 - User's Guide, 2010.

[MAT10c] MATLAB Homepage. http://www.mathworks.com/, 2010.

[MCG05] Tom Mens, Krzysztof Czarnecki, and Pieter Van Gorp. A taxonomy of model transformations. In Jean Bezivin and Reiko Heckel, editors, Language Engineering for Model-Driven Software Development, number 04101 in Dagstuhl Seminar Proceedings, Dagstuhl, Germany, 2005. Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany.

[MDA] MDA Specification. http://www.omg.org/mda/specs.htm.

[MGVK06] Tom Mens, Pieter Van Gorp, D'aniel Varr'o, and G'abor Karsai. Applying a model transformation taxonomy to graph transformation technology. Electronic Notes in Theoretical Computer Science, 152:143–159, March 2006.

[Mil78] Robin Milner. A Theory of Type Polymorphism in Programming. Journal of Computer and System Sciences, 17:348–375, 1978.

[Mir87] Daniel P. Miranker. Treat: A better match algorithm for ai production systems. Technical report, Austin, TX, USA, 1987.

[MOF11] MOFLON Homepage. `http://www.moflon.org/`, 2011.

[MSW98] M. Muench, A. Schuerr, and A. Winter. Integrity constraints in the multi-paradigm language progres. In Proceedings of the IEEE Symposium on Visual Languages, VL '98, Washington, DC, USA, 1998. IEEE Computer Society.

[MTM97] Robin Milner, Mads Tofte, and David Macqueen. The Definition of Standard ML (revised). MIT Press, Cambridge, MA, USA, 1997.

[Mue02] Manfred Muench. Generic Modelling with Graph Rewriting Systems. PhD thesis, Aachen University of Technology, Germany, 2002.

[Mur89] T. Murata. Petri nets: Properties, analysis and applications. Proceedings of the IEEE, (4):541–580, April 1989.

[MVM10] Frederic P. Miller, Agnes F. Vandome, and John McBrewster. Generic programming. Alphascript Publishing, 2010.

[MXA10] Model Engineering Solutions, Berlin - Model Examiner. `http://www.model-engineers.com/en/our-products/model-examiner.html`, 2010.

[Obj05a] Object Management Group. UML Infrastructure - Version 2.0, July 2005.

[Obj05b] Object Management Group. UML Superstructure - Version 2.0, July 2005.

[Obj05c] Object Management Group. XML Metadata Interchange - Version 2.1, September 2005.

[Obj06a] Object Management Group. Meta Object Facility (MOF) 2.0 Core Specification, January 2006.

[Obj06b] Object Management Group. Object Constraint Language - Version 2.0, May 2006.

[OCa] Caml Homepage. `http://caml.inria.fr/index.en.html`.

[OMGa] MOF Homepage. `http://www.omg.org/mof/`.

[OMGb] OMG Homepage. `http://www.omg.org/`.

[Opa] Opa Homepage. `http://opalang.org`.

[Pha12] Quyet Thang Pham. Model Transformation Reuse: A Graph-based Model Typing Approach. PhD thesis, INFO - Dépt. Informatique (Institut Mines-Télécom-Télécom Bretagne-UEB), december 2012.

[PJMS01] François Pennaneac'h, Jean-Marc Jézéquel, Jacques Malenfant, and Gerson Sunyé. Uml reflections. In Akinori Yonezawa and Satoshi Matsuoka, editors, Reflection, volume 2192 of Lecture Notes in Computer Science, pages 210–230. Springer, 2001.

[Pra71] Terrence W. Pratt. Pair grammars, graph languages and string-to-graph translations. J. Comput. Syst. Sci., 5:560–595, December 1971.

[PRO11] PROGRES Homepage. `http://www.se.rwth-aachen.de/tikiwiki/tiki-index.php%3Fpage=Research:+Progres.html`, 2011.

[PSW76] D. L. Parnas, John E. Shore, and David Weiss. Abstract types defined as classes of variables. In Proceedings of the 1976 conference on Data : Abstraction, definition and structure, pages 149–154, New York, NY, USA, 1976. ACM.

[Rat11] Rational Rose - Product page. `http://www-01.ibm.com/software/awdtools/developer/rose/`, 2011.

[RG97] Rozenberg and Grzegorz, editors. Handbook of graph grammars and computing by graph transformation: volume I. Foundations. World Scientific Publishing Co., Inc., River Edge, NJ, USA, 1997.

[RGdLV08] José Eduardo Rivera, Esther Guerra, Juan de Lara, and Antonio Vallecillo. Analyzing rule-based behavioral semantics of visual modeling languages with maude. In SLE, pages 54–73, 2008.

[Sca] Scala Homepage. `http://www.scala-lang.org`.

[SCFD06] Ingo Stürmer, Mirko Conrad, Ines Fey, and Heiko Dörr. Experiences with model and autocode reviews in model-based software development. In Proceedings of the 2006 international workshop on Software engineering for automotive systems, SEAS '06, pages 45–52, New York, NY, USA, 2006. ACM.

[Sch91] Andy Schürr. Operational Specification with PROgrammed Graph REwriting Systems. DUV Informatik. Deutscher Universitätsverlag, 1991. in German - original title: Operationales Spezifizieren mit programmierten Graphersetzungssystemen - formale Definitionen, Anwendungsbeispiele und Werkzeugunterstützung.

[Sch06] Douglas C. Schmidt. Model-driven engineering. IEEE Computer, 39(2), February 2006.

[Sch10] Bernhard Schätz. Verification of model transformations. ECEASST, 29, 2010.

[Sim10] MATLAB Simulink/Stateflow Homepage. `http://www.mathworks.com/products/`, 2010.

[SK95] Kenneth Slonneger and Barry Kurtz. Formal Syntax and Semantics of Programming Languages: A Laboratory Based Approach. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 1995.

[Smi82] Brian Cantwell Smith. Procedural Reflection in Programming Languages. PhD thesis, Massachusetts Institute of Technology, Laboratory for Computer Science, 1982.

[SMM+12] Sagar Sen, Naouel Moha, Vincent Mahé, Olivier Barais, Benoit Baudry, and Jean-Marc Jézéquel. Reusable model transformations. Software and System Modeling, 11(1):111–125, 2012.

[SSSL10] Andy Schürr, Wilhelm Sch"afer, Ingo St"urmer, and Elodie Legros. Mate - a model analysis and transformation environment for matlab simulink. In Model-Based Engineering of Embedded Real-Time Systems, volume 6100 of LNCS, pages 323–328. Giese, H. and Karsai, G. and Lee, E. and Rumpe, B. and Sch"atz, B., 2010.

[Str67] Christopher Strachey. Fundamental concepts in programming languages. In Lecture Notes for the International Summer School in Computer Programming, August 1967.

[Tan04] Éric Tanter. From Metaobject Protocols to Versatile Kernels for Aspect-Oriented Programming. PhD thesis, University of Nantes and University of Chile, nov 2004.

[TC10] Gabriel Tamura and Anthony Cleve. A Comparison of Taxonomies for Model Transformation Languages. Paradigma, 4(1), March 2010.

[TJF+09] Massimo Tisi, Frédéric Jouault, Piero Fraternali, Stefano Ceri, and Jean Bézivin. On the Use of Higher-Order Model Transformations. In ECMDA-FA, pages 18–33, 2009.

[UHV09a] Zoltán Ujhelyi, Ákos Horváth, and Dániel Varró. Generic static analysis of transformation programs. 2009.

[UHV09b] Zoltán Ujhelyi, Ákos Horváth, and Dániel Varró. Static type checking of model transformations by constraint satisfaction programming. 2009.

[UHV11] Zoltán Ujhelyi, Ákos Horváth, and Dániel Varró. Static type checking of model transformation programs. ECEASST, 38, 2011.

[Vel07] Apache Velocity - Velocity User Guide. `http://velocity.apache.org/engine/releases/velocity-1.5/user-guide.html`, March 2007.

[Via11a] VIATRA2 Homepage. `http://www.eclipse.org/gmt/VIATRA2/`, 2011.

[Via11b] VIATRA2 Documentation. `http://wiki.eclipse.org/VIATRA2`, 2011.

[VP03] Dániel Varró and András Pataricza. VPM: A visual, precise and multilevel metamodeling framework for describing mathematical domains and UML. Journal of Software and Systems Modeling, 2(3):187–210, October 2003.

[VP04] D. Varró and A. Pataricza. Generic and Meta-Transformations for Model Transformation Engineering. In T. Baar, A. Strohmeier, A. Moreira, and S. Mellor, editors, Proc. UML 2004: 7th International Conference on the Unified Modeling Language, volume 3273, pages 290–304. Springer, 2004.

[vWMP+] A. van Wijngaarden, B. J. Mailloux, J. E. L. Peck, C. H. A. Koster, M. Sintzoff, C. H. Lindsey, L. G. L. T. Meertens, and R. G. Fisker. Revised report on the algorithmic language ALGOL 68.

[WF94] Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. Inf. Comput., 115(1):38–94, 1994.

[Win12] Sabine Winetzhammer. Modgraph - generating executable emf models. ECEASST, 54, 2012.

[Zün01] A. Zündorf. Rigorous Object Oriented Software Development. University of Paderborn, 2001. Habilitation Thesis.

Wissenschaftlicher Werdegang

Datenschutz:
Der Lebenslauf ist in der Online-Version nicht enthalten.