Hasso-Plattner-Institut für Softwaresystemtechnik GmbH
Prof.-Dr.-Helmert-Str. 2-3, 14482 Potsdam
Fachgebiet für Systemanalyse und Modellierung

Traceability and Model Management with Executable and Dynamic Hierarchical Megamodels

**Dissertation zur Erlangung des akademischen Grades
"doctor rerum naturalium" (Dr. rer. nat.)
in der Wissenschaftsdisziplin "Systemanalyse und Modellierung"**

eingereicht an der
Mathematisch-Naturwissenschaftlichen Fakultät
der Universität Potsdam

von
Andreas Seibel

Potsdam, den 27.07.2012

Published online at the
Institutional Repository of the University of Potsdam:
URL http://opus.kobv.de/ubp/volltexte/2013/6422/
URN urn:nbn:de:kobv:517-opus-64222
http://nbn-resolving.de/urn:nbn:de:kobv:517-opus-64222

Abstract

Nowadays, model-driven engineering (MDE) promises to ease software development by decreasing the inherent complexity of classical software development. In order to deliver on this promise, MDE increases the level of abstraction and automation, through a consideration of domain-specific models (DSMs) and model operations (e.g. model transformations or code generations). DSMs conform to domain-specific modeling languages (DSMLs), which increase the level of abstraction, and model operations are first-class entities of software development because they increase the level of automation.

Nevertheless, MDE has to deal with at least two new dimensions of complexity, which are basically caused by the increased linguistic and technological heterogeneity.

The first dimension of complexity is setting up an MDE environment, an activity comprised of the implementation or selection of DSMLs and model operations. Setting up an MDE environment is both time-consuming and error-prone because of the implementation or adaptation of model operations. The second dimension of complexity is concerned with applying MDE for actual software development. Applying MDE is challenging because a collection of DSMs, which conform to potentially heterogeneous DSMLs, are required to completely specify a complex software system. A single DSML can only be used to describe a specific aspect of a software system at a certain level of abstraction and from a certain perspective. Additionally, DSMs are usually not independent but instead have inherent interdependencies, reflecting (partial) similar aspects of a software system at different levels of abstraction or from different perspectives. A subset of these dependencies are applications of various model operations, which are necessary to keep the degree of automation high. This becomes even worse when addressing the first dimension of complexity. Due to continuous changes, all kinds of dependencies, including the applications of model operations, must also be managed continuously. This comprises maintaining the existence of these dependencies and the appropriate (re-)application of model operations.

The contribution of this thesis is an approach that combines traceability and model management to address the aforementioned challenges of configuring and applying MDE for software development. The approach is considered as a traceability approach because it supports capturing and automatically maintaining dependencies between DSMs. The approach is considered as a model management approach because it supports managing the automated (re-)application of heterogeneous model operations.

In addition, the approach is considered as a comprehensive model management. Since the decomposition of model operations is encouraged to alleviate the first dimension of complexity, the subsequent composition of model operations is required to counteract their fragmentation. A significant portion of this thesis concerns itself with providing a method for the specification of decoupled yet still highly cohesive complex compositions of heterogeneous model operations. The approach supports two different kinds of compositions - data-flow compositions and context compositions. Data-flow composition is used to define a network of heterogeneous model operations coupled by sharing input and output DSMs alone. Context composition is related to a concept used in declarative model transformation approaches to compose individual model transformation rules (units) at any level of detail. In this thesis, context composition provides the ability to use a collection of dependencies as context for the composition of other dependencies, including model operations. In addition, the actual implementation of model operations, which are going to be composed, do not need to implement any composition concerns.

The approach is realized by means of a formalism called an executable and dynamic hierarchical megamodel, based on the original idea of megamodels. This formalism supports specifying compositions of dependencies (traceability and model operations). On top of this formalism, traceability is realized by means of a localization concept, and model management by means of an execution concept.

Zusammenfassung

Die modellgetriebene Softwareentwicklung (MDE) verspricht heutzutage, durch das Verringern der inhärenten Komplexität der klassischen Softwareentwicklung, das Entwickeln von Software zu vereinfachen. Um dies zu erreichen, erhöht MDE das Abstraktions- und Automationsniveau durch die Einbindung domänenspezifischer Modelle (DSMs) und Modelloperationen (z.B. Modelltransformationen oder Codegenerierungen). DSMs sind konform zu domänenspezifischen Modellierungssprachen (DSMLs), die dazu dienen das Abstraktionsniveau der Softwareentwicklung zu erhöhen. Modelloperationen sind essentiell für die Softwareentwicklung da diese den Grad der Automatisierung erhöhen. Dennoch muss MDE mit Komplexitätsdimensionen umgehen die sich grundsätzlich aus der erhöhten sprachlichen und technologischen Heterogenität ergeben.

Die erste Komplexitätsdimension ist das Konfigurieren einer Umgebung für MDE. Diese Aktivität setzt sich aus der Implementierung und Selektion von DSMLs sowie Modelloperationen zusammen. Eine solche Aktivität ist gerade durch die Implementierung und Anpassung von Modelloperationen zeitintensiv sowie fehleranfällig. Die zweite Komplexitätsdimension hängt mit der Anwendung von MDE für die eigentliche Softwareentwicklung zusammen. Das Anwenden von MDE ist eine Herausforderung weil eine Menge von heterogenen DSMs, die unterschiedlichen DSMLs unterliegen, erforderlich sind um ein komplexes Softwaresystem zu spezifizieren. Individuelle DSMLs werden verwendet um spezifische Aspekte eines Softwaresystems auf bestimmten Abstraktionsniveaus und aus bestimmten Perspektiven zu beschreiben. Hinzu kommt, dass DSMs sowie DSMLs grundsätzlich nicht unabhängig sind, sondern inhärente Abhängigkeiten besitzen. Diese Abhängigkeiten reflektieren äquivalente Aspekte eines Softwaresystems. Eine Teilmenge dieser Abhängigkeiten reflektieren Anwendungen diverser Modelloperationen, die notwendig sind um den Grad der Automatisierung hoch zu halten. Dies wird erschwert wenn man die erste Komplexitätsdimension hinzuzieht. Aufgrund kontinuierlicher Änderungen der DSMs, müssen alle Arten von Abhängigkeiten, inklusive die Anwendung von Modelloperationen, kontinuierlich verwaltet werden. Dies beinhaltet die Wartung dieser Abhängigkeiten und das sachgerechte (wiederholte) Anwenden von Modelloperationen.

Der Beitrag dieser Arbeit ist ein Ansatz, der die Bereiche Traceability und Model Management vereint. Das Erfassen und die automatische Verwaltung von Abhängigkeiten zwischen DSMs unterstützt Traceability, während das (automatische) wiederholte Anwenden von heterogenen Modelloperationen Model Management ermöglicht. Dadurch werden die zuvor erwähnten Herausforderungen der Konfiguration und Anwendung von MDE überwunden.

Die negativen Auswirkungen der ersten Komplexitätsdimension können gelindert werden indem Modelloperationen in atomare Einheiten zerlegt werden. Um der implizierten Fragmentierung entgegenzuwirken, erfordert dies allerdings eine nachfolgende Komposition der Modelloperationen. Der Ansatz wird als erweitertes Model Management betrachtet, da ein signifikanter Anteil dieser Arbeit die Kompositionen von heterogenen Modelloperationen behandelt. Unterstützt werden zwei unterschiedliche Arten von Kompositionen. Datenfluss-Kompositionen werden verwendet, um Netzwerke von heterogenen Modelloperationen zu beschreiben, die nur durch das Teilen von Ein- und Ausgabe DSMs komponiert werden. Kontext-Kompositionen bedienen sich eines Konzepts, das von deklarativen Modelltransformationen bekannt ist. Dies ermöglicht die Komposition von unabhängigen Transformationsregeln auf unterschiedlichsten Detailebenen. Die in dieser Arbeit eingeführten Kontext-Kompositionen bieten die Möglichkeit eine Menge von unterschiedlichsten Abhängigkeiten als Kontext für eine Komposition zu verwenden – unabhängig davon ob diese Abhängigkeit eine Modelloperation repräsentiert. Zusätzlich müssen die Modelloperationen, die komponiert werden, selber keine Kompositionsaspekte implementieren, was deren Wiederverwendbarkeit erhöht.

Realisiert wird dieser Ansatz durch einen Formalismus der Executable and Dynamic Hierarchical Megamodel genannt wird und auf der originalen Idee der Megamodelle basiert. Auf Basis dieses Formalismus' sind die Konzepte Traceability (hier Localization) und Model Management (hier Execution) umgesetzt.

Acknowledgement

This thesis would not have been possible without my family, friends and colleagues, all of whom supported me during the four years of working on it. More specifically, I wish to thank my lovely wife Melanie Seibel who gave me all the freedom that I needed to finish this thesis, and who took the load off me during all that time. I thank my parents Maria and Johann Seibel who gave me the opportunity to study computer science, and who always supported me during my studies. I thank my supervisor Prof. Dr. Holger Giese who already promoted me during my studies. I also thank him because he gave me the opportunity to write this thesis and because he was a mentor to me during this time. I thank Regina Hebig for her collaboration, the fruitful discussions about model-driven engineering and megamodeling, and for her commitment to giving me feedback for this thesis. I thank Stefan Neumann for being a good and supportive friend and colleague, who especially supported me in the early days of my research. I thank Thomas Vogel for his collaboration in bringing megamodels to the domain of models at runtime. I thank Gregor Gabrysiak for his collaboration in the Hasso Plattner Design Thinking Research Project (HPDTRP). I thank Thomas Beyhl for his commitment on the topic of version-based modeling with megamodels, and for his support in implementing the model management framework. I thank Stephan Hildebrandt who did a great job in implementing Eclipse and EMF-based Story diagrams, which helped me with realizing early prototypes of my thesis. I thank Tobias Hoppe, Martin Hanysz, Arian Treffer, Johannes Dyck, Dmitry Zakharov and Henrik Steudel for realizing new features and for providing additional ideas concerning the model management framework. I thank Mark Liebetrau and Dustin Lange for their support in implementing a very first prototype. Finally, I thank Justine Lera for spending her time on proof-reading my thesis.

Contents

Abbreviations

A ..... Artifacts

A_{A_P} ..... Physical artifacts in an MDE application

A_B ..... Artifact bindings

A_{C_P} ..... Physical artifacts in an MDE configuration

A_{C_t} ..... Artifact type compositions

A_C ..... Artifact compositions

A_R ..... Artifact roles

A_t ..... Artifact types

C_M ..... Relation composition matches

C_S ..... Relation type composition specifications

E_{C_t} ..... Relation type composition graph edges

E_C ..... Relation composition graph edges

E_{E_t} ..... Scheduling type graph edges

E_E ..... Scheduling graph edges

E_O ..... Execution operations

G_{C_t} ..... Relation type composition graph

G_C ..... Relation composition graph

G_{E_t} ..... Scheduling type graph

G_E ..... Scheduling graph

I_C ..... Instantiation conditions

I_S ..... Impact scopes

M_A ..... Configuration megamodel

M_C ..... Configuration megamodel

O_{C_P} ..... Model operations in an MDE configuration

P ..... Parameters

P_B ..... Parameter bindings

P_{C_t} ..... Parameter type connectors

P_C ..... Parameter connectors

P_R ..... Parameter roles

P_t ..... Parameter types

R ..... Relations

R_B ..... Relation bindings

R_{C_t} ..... Relation type compositions

R_C ..... Relation compositions

R_R ..... Relation roles

R_t ..... Relation types

V_{C_t} ..... Relation type composition graph vertices (relation types)

V_C ..... Relation composition graph vertices (relations)

V_{E_t} ..... Scheduling type graph vertices (relation types)

V_E ..... Scheduling graph vertices (relations)

AGG ..... Attributed Graph Grammar

AMMA ..... Atlas Model Management Architecture

AMW ..... Atlas Model Weaving

API ..... Application Programming Interface

ATL ..... Atlas Model Transformation Language

AToM3 ..... A Tool for Multi-formalism and Meta-Modeling

BOTL ..... Bidirectional Object-oriented Transformation Language

CASE ..... Computer Aided Software Engineering

D-MDA ..... Deployment Model-Driven Architecture

DSF ..... Depth-First Search

DSL ..... Domain Specific Language

DSM ..... Domain Specific Model

DSML ..... Domain Specific Modeling Language

ECL ..... Epsilon Comparison Language

EGL ..... Epsilon Generation Language

EMF ..... Eclipse Modeling Framework

EMOF ..... Essential Meta Object Facility

ETL ..... Epsilon Transformation Language

EVL ..... Epsilon Validation Language

GPL ..... General Purpose Language

GPML ..... General Purpose Modeling Language

GReAT ..... Graph Rewriting and Transformation language

iLSI ..... Incremental Latent Semantic Indexing

Jet ..... Java Emitter Templates

LHS ..... Left-Hand Side

LSI ..... Latent Semantic Indexing

MDD ..... Model-Driven Development

MDE ..... Model-Driven Engineering

MOF ..... Meta Object Facility

MOLA ..... MOdel transformation LAnguage

OCL ..... Object Constraint Language

OMG ..... Object Management Group

QVT ..... Query/View/Transformation

RDBMS ..... Relational Database Management System

RHS ..... Right-Hand Side

SQL ..... Structured English Query Language

TGG ..... Triple-Graph Grammars

UML ..... Unified Modeling Language

VIATRA ..... VIsual Automated model TRAnsformations

XML ..... Extended Markup Language

1. Introduction

Contents

* * *

Nowadays, software is omnipresent in our everyday lives because it's an essential part of today's systems. Software-based systems are surrounding us in nearly every domain; e.g., infrastructure (traffic control systems or power plants), automotive (cars), avionics (planes), communication (satellites or mobile phones), security (smart cards), entertainment (television or radio), etc. Generally, a system can be considered as a way of working, organizing or doing one or more tasks according to a plan, program or set of rules [88]. Thus, systems programmatically solve problems by means of collaborating units which work toward a common goal. Software-based systems are usually partially realized by means of software (software systems). Common classes of software systems are embedded systems, enterprise systems or simply desktop systems.

The complexity[1] of new software systems continuously increases due to various reasons. For example, technological advances increase market demands for new applications and features [60], or the integration of software systems leads to various stakeholders, multiple crosscutting domains like cars communicating with traffic control systems and other cars in their environment.

In addition, increased productivity is a major objective in software development because of the ongoing demand for shorter time-to-market cycles and reduced costs [155]. However, the more complex a software system is, the less productive the software development can get. This leads to the so-called productivity gap – the disparity between the rate of technological developments and the productivity rate of of software development [61]. Due to the gap, the failure of software development projects is more probable due to tight economic constraints.

The past has shown that the productivity gap can be lessened by narrowing the problem-implementation gap [60]. This latter is the distance between the domain-relatedness of the implementation language and the actual problem domain. A wide problem-implementation gap results in the certain drawbacks. The less domain-related, that is to say, less abstract, an implementation language is, the less reusable and harder to understand the resulting code will be. For example, implementing a software system using a machine language for a particular microprocessor (e.g., Z80), will couple the resulting executable code to that particular microprocessor because the resulting code relies on microprocessor specific concepts.

[1] The New Oxford American Dictionary [2] defines the term complexity as: *"[...] the state of being intricate or complicated [...]"* whereby intricate means *"[...] very complicated or detailed [...]"* and complicated means *"[...] consisting of many interconnecting parts of elements [...]"*.

Apart from this implementation constraint, it is tedious and error-prone to implement problem domain concepts by means of such languages a wider gap also means more mental effort is required to translate problem domain specific concepts into microprocessor specific concepts.

Closing the problem-implementation gap can be obtained in two different ways. A short-term solution reduces software develoment's increasing complexity by introducing additional technologies which can handle this complexity. For example, support navigation between dependent software artifacts to increase understanding. A long-term solution is to provide a way to only rely on implementation languages which are closer to the actual problem domain. This long-term solution has already been applied several times in the past, also known as paradigm shift. For example, shifting from machine languages to procedural programming languages (C such as Pascal) and from procedural programming languages to object-oriented programming languages (Java or C++). In the latter case, whole software systems are implemented solely using object-oriented programming languages, which are automatically translated into machine executable code (machine language) using a compiler and a linker.[2]

In 1997, the object management group (OMG)[3] introduced the unified modeling language (UML) [128]. The UML has become a de-facto standard specification comprised of a set of (graphical) general-purpose modeling languages (GPML) for describing various aspects of a software system. It became very popular as a technique for documenting software systems in early phases of software development.

More recently, MDE has gained increasing attention; able to be considered as a candidate for the next paradigm shift, as such it also constitutes a huge challenge. MDE promotes domain-specific models (DSMs), domain-specific modeling languages (DSMLs), and model operations as first-class engineering artifacts for software development, in order to increase the level of abstraction and automation. Any kind of automated development activities on models are considered as Model Operations, for instance, model transformations, model checking, model merge, etc. In [41], a domain-specific language (DSL) is defined as: *"[...] a custom language that targets a small problem domain, which it describes and validates in terms of native to the domain [...]".*[4] In MDE, DSMLs eventually have the same status as implementation languages in classical software development [1].

In order to develop software systems, the application of MDE requires a collection of heterogeneous DSMLs since each language might only define a specific aspect of, or perspective on, the software system at a certain level of abstraction. Heterogeneous model operations are thereby becoming increasingly important in order to keep DSMs consistent, for example, transforming models into different perspectives and into different levels of abstraction. Model operations are also important for analysis purposes, and especially support early analysis as to whether required software system properties are satisfied.[5]

1.1. Application Scenario for Model-Driven Engineering

To define a problem statement and to explain the challenges and the contribution of this thesis, one possible and simplified application scenario for MDE is shown (naturally, this is not the only application scenario for MDE; it may be applied to different scenarios, however it is assumed that the same challenges will result, with perhaps only a different focus). This application scenario is of a particular use case where multiple software systems are developed for different customers, in an organizational environment. This particular application scenario will be used to motivate MDE and to explain the resulting challenges. A high-level and simplified overview of the application scenario is shown in Figure 1.1.

It is assumed that an organization is developing multiple software systems within certain domains for multiple customers, who are specifically requesting the development of said software systems. Each software system is developed in a separate project, which can be conducted simultaneously or sequentially with all other projects. It is assumed that developing a software system in a project is achieved through the application of MDE only. The application scenario is a simplification because it does not distinguish between various software development life-cycles such as requirements analysis, design, implementation, testing, or deployment, etc. It also does not show all of the roles involved in a software development

[2] This is further considered as classical software development.
[3] http://www.omg.org/
[4] A DSML is also a DSL.
[5] In the following, any language, regardless of whether it is domain-specific or general-purpose, is considered as a metamodel and instances of a metamodel are models. A metamodel always conforms to a meta-metamodel, which is the modeling language of a modeling language.

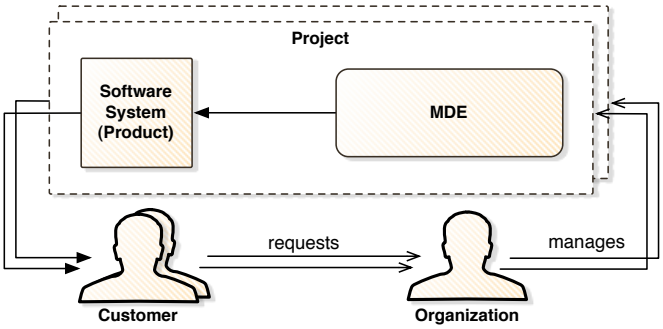

Figure 1.1.: A simplified application scenario for MDE

process, e.g., that of business analysts. Finally, because each life-cycle phase specifies its outcomes by means of models, it is assumed that it is using MDE.

In classical software development, there is a strict separation between tool vendors and tool users. A tool vendor is any organization (internal or external), which provides a ready-to-use integrated development environment (IDE) for programming in a specific language (e.g., Java). A tool user is another organization or person who is using the IDE for doing the actual software development. This strict separation is feasible as long an IDE focuses on a GPL for implementing a software system. However in the case of MDE, a collection of DSMLs has to be employed instead. Thus, this approach is rather infeasible because of at least two reasons. Firstly, a tool vendor must provide an IDE that is tailored for a specific domain of software systems. Secondly, a tool vendor must provide all necessary model operations that are required to obtain all necessary analyses and validations, as well as model transformations between different perspectives and different levels of abstraction. The model operations that are required depend on the languages that are provided. Thus, MDE requires additional setup efforts before it can be applied to develop the actual software system.

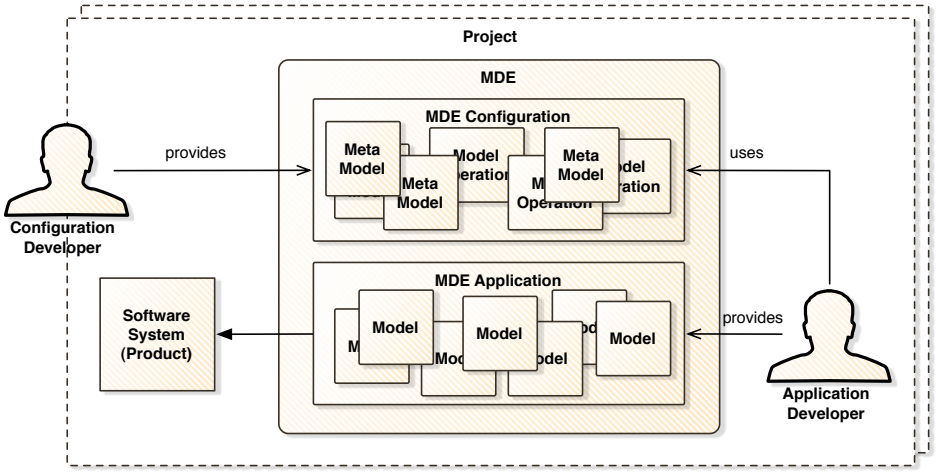

Figure 1.2.: Illustration of an MDE configuration and application

A more detailed view of a project is shown in Figure 1.2. Here, MDE consists of two separate activities – setting up an MDE environment by providing required metamodels and model operations, and applying such a setup for actual software development. In this thesis, the first activity is called configuration of MDE, which results in an MDE configuration, and the second activity is called application of MDE, which results in an MDE application.

An MDE configuration is, from a generalized perspective, a selection of the metamodels and model operations necessitated by the development of a certain software system within a specific project. It is

provided by a role called the configuration developer, who can be an external 3rd party tool vendor, or an internal department that is part of the organization. More concretely, an MDE configuration can be considered as a configurable IDE for modeling. An MDE application is the application of an MDE configuration, which is considered as a collection of models that specify the actual software system that is developed. These models are instances of the metamodels provided by the MDE configuration. Furthermore, the model operations, which are provided by the MDE configuration, can be applied to the models in an MDE application. Thus, an MDE application is basically the use of a configured IDE for modeling.

This perspective on MDE can also be found in literature as well as in early practice. In [63] and [118] there exists an explicitdistinction between a preparation phase and an execution phase for MDE. The preparation phase is similar to the configuration of MDE, which comprises the selection of all necessary metamodels and model operations. The execution phase is similar to the application of MDE by doing the actual modeling and applying model operations.

Such a perspective on MDE is also provided in [136]. The authors present a generative development process, an approach of employing MDE in practice. They distinguish between two types of activity: generator development and application development. The generator development activity concerns itself with implementing all necessary metamodels and model operations, while the application development activity is the actual modeling (development) activity. Thus, the first activity is similar to the configuration of MDE, and the second activity is similar to the application of MDE.

Figure 1.3 shows a UML activity diagram illustrating that configuring and applying MDE is cyclic rather than strictly sequential.

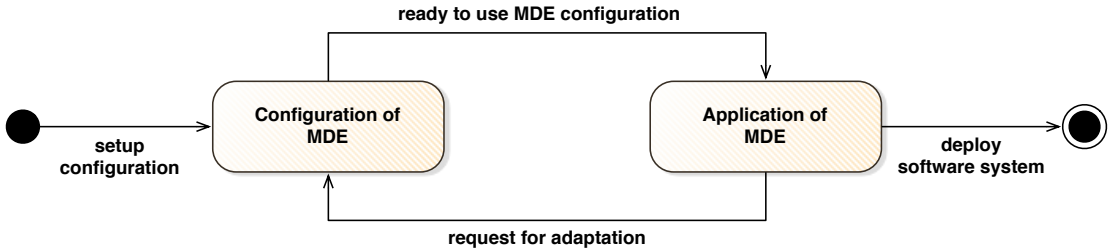

Figure 1.3.: Generic activities for configuring and applying MDE

Organizations may realize that an existing MDE configuration is permanently evolving because the requirements of the software system under development might themselves change [72, 73]. This means that metamodels and model operations may have to be adapted. In certain situations the technological capability of a model operation is insufficient to realize a required adaptation. In such a case the model operation has to be completely re-implemented using another technology, which is time-consuming, error-prone and expensive. It is possible to mitigate this cost in the case of an organizations which is developing software systems for different customers which are nonetheless to some extent similar. In this case, the efforts of configuring MDE for further application can be reduced if previous MDE configurations could be at least partially reused.

1.2. Challenges

As already indicated, MDE is promising but at the same time challenging. A major challenge of MDE, ironically, is caused by one of its greatest benefits: the domain-relatedness. In the following, four challenges to organizations employing MDE shall be described, specifically – the cause and form of each challenge. Each of these All of these stated challenges will be addressed in the sequel of this thesis.

1.2.1. Understanding MDE Applications

Models do not exist in isolation but rather have inherent dependencies upon each other. These dependencies exist because models (partly) represent similar aspects or concerns of a software system but at different levels of abstraction or from different perspectives.

Basically, three types of dependencies can be distinguished: hard references, soft references and semantic connections [106]. Hard and soft references are syntactic dependencies between arbitrary model elements. A hard reference is an explicit reference between model elements, whereas a soft reference is an implicit reference between model elements encoded by means of name equivalence of certain attributes. A semantic dependency is a complex dependency between model elements, which may involve other model elements to define this dependency. In addition, the application of model operations may also introduce new dependencies between models and model elements, which semantics are defined by the model operation itself. How to interpret certain models, or the reason why certain models exist, may be unclear to individual application developers working as part of a large team. To effectively employ an MDE application, application developers must understand why models exist and how they have to be used. Thus, application developers need knowledge about dependencies between models or model elements.

- **(C1)** Understanding MDE applications is challenging due to the increased number of models and therefore their increased number of implicitly existing interdependencies. Furthermore, the models that are interconnected are heterogeneous, as are the dependencies themselves. Thus, dependencies should be able to reflect all kinds of dependencies like soft references, hard references, and semantic connections, as well as the application of model operations.

1.2.2. Changes in MDE Applications

In MDE applications, models are not static but rather subject to continuous change because models are manually complemented or adapted due to changed requirements to a software system under development. Being not aware of existing dependencies or applied model operations may endanger the success of the whole software development project. That is because inconsistencies, due to missed propagation of changes or (re-)application of model operations, may find their way into the final software system, which can decrease its quality [48, 157, 20, 13]. Thus, changes to models may render dependent models to be inconsistent because of implicit intra-model dependencies (such as applied model operations), which leads to the second challenge.

- **(C2)** Changes in MDE applications is challenging because of the increased number of heterogeneous and dependent models and the heterogeneity of dependencies that may occur. If models have changed, these changes may have impact to dependent models. In this case, depending on the kind of dependency, a certain action may have to be applied, e.g., propagating changes manually or (re-)applying a model operation. Furthermore, changes may result in new dependencies or may invalidate existing dependencies.

1.2.3. Applying Model Operations in MDE Applications

The increased heterogeneity of models implies the increased heterogeneity of model operations. The heterogeneity of model operations is influenced by two factors. Firstly, not only a single technology is sufficient to implement all required model operations but rather a set of different technologies is required. Secondly, different kinds of model operations must be implemented, e.g.; model synthesis like model transformations or code generations and model analysis like validation, verification, etc. This leads to the third challenge.

- **(C3)** Applying model operations in MDE applications is challenging because of the increased number of heterogeneous models and therefore the increased number of heterogeneous model operations that have to be applied. Thus, it gets tedious to manually apply all model operations within the right context [14].

1.2.4. Reusable and Adaptable Model Operations in MDE Configurations

Setting up an MDE configuration for further application of MDE is time intensive and costly because of the heterogeneity of metamodels and model operations. MDE configurations might be different for any project. There are two ways of addressing this challenge. Setting up an MDE configuration could be eased by partly reusing former MDE configurations of similar projects. This requires that metamodels

and model operations are designed to support reusability. This is considered for model operations but not for metamodels in this thesis. An opportunity to increase the reusability of model operations is to decompose them into smaller and more reusable model operations [126].

Their reusability depends on the degree of coupling and cohesion. The term coupling is: "*[...] a measure of the strength of interconnection between one module and another [...]*" [176]. To improve loose coupling of a model operation, it should have as few connections to other model operations as possible. The term cohesion is: "*[...] the degree of functional relatedness of processing elements within a single module [...]*" [176]. Only focusing on a single concern – the actual model operation's task – renders a model operation as being highly cohesive. To address high cohesion, any additional concern, i.e. finding the required context for applying a model operation, should be eliminated from the operations' implementation because it decreases cohesion.

Model operations may also get more adaptable because the complexity of each model operation may be reduced. This could improve the scenario where application developers request the adaptation model operations. Therefore, model operations should be implemented as loosely coupled and highly cohesive as possible, which results in the last and major underlying challenge of this thesis.

- **(C4)** Reusable and adaptable model operations in MDE configurations are challenging because it leads to a fragmentation of model operations. Thus, the number of model operations that have to be applied is increasing, which makes the application of MDE even more tedious and error-prone. In addition, the heterogeneity of model operations makes the composition of individual model operations even more complex.

1.3. Goals

In oder to address the aforementioned challenges, four goals are defined in the following. Each goal comes with a set of requirements in oder to achieve the individual goal. These individual goals can also be considered to be goals in specific domains in the context of MDE. Figure 1.4 maps each goal to its related domain.

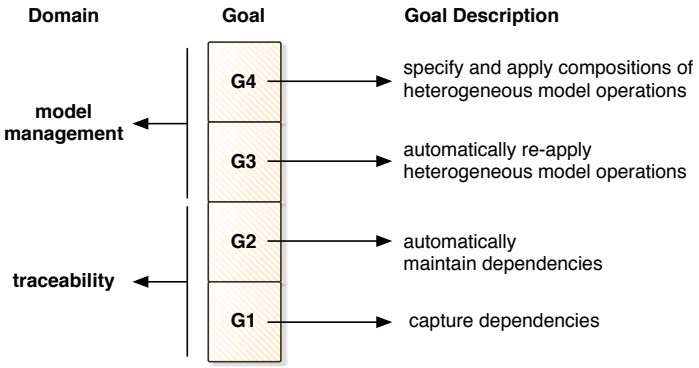

Figure 1.4.: Relationship between thesis's goals and existing domains

The first two goals are common goals of traceability approaches in the context of MDE (see Section 2.1.3). Traceability is basically about capturing all kinds of dependencies between elements of models. Due to the increasing number of dependencies that have to be captured and the changes that are made to models, an important aspect of traceability is automated maintenance. The last two goals are also goals of current model management approaches in the context of MDE (see Section 2.1.4). Recently, model management gains increasing attention, which is basically about managing models and their dependencies caused by the application of model operations.

Figure 1.5 gives an overview of the relationship between the individual goals and the aforementioned challenges. Each cell in this matrix shows the impact of the goal to a given challenge, which can be positive or negative. The figure shows that each challenge is positively affected by at least one goal.

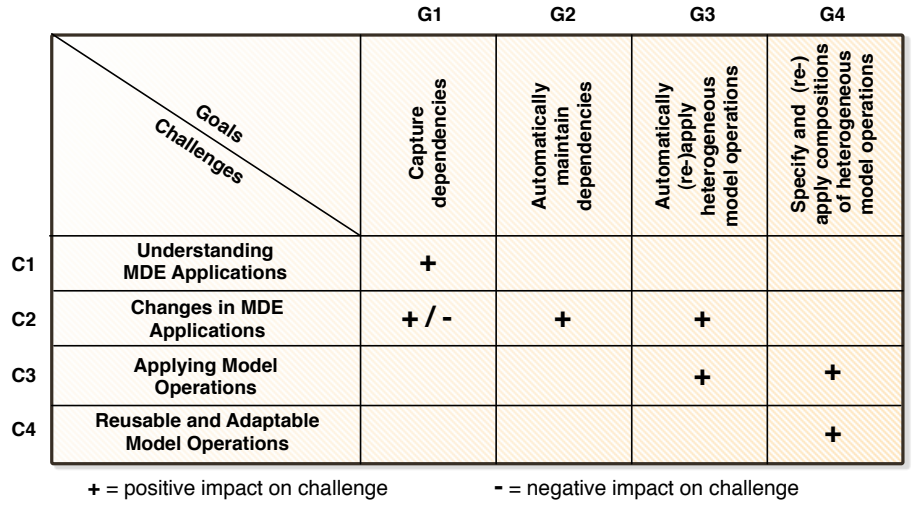

Figure 1.5.: Relationship between thesis's goals and challenges

The first challenge is positively impacted by reaching the first goal because explicitly captured dependencies show why certain models or model elements exist. The second challenge is also positively impacted by reaching the first goal because dependencies are the foundation for impact analysis. Nevertheless, the first goal negatively impacts the second challenge because captured dependencies deteriorate through changes. This negative impact is, however, reversed by achieving the second goal. The second challenge is also positively impacted by reaching the third goal because already applied model operations are automatically re-applied in case of changes. The third challenge can be tackled also by providing a solution to the third goal. That is because the efforts of manual (re-)applications of model operations can be decreased. The fourth goal does also have a positive impact on the third challenge because specifying composition of heterogeneous model operations reduces the number of model operations by providing a compound of model operations that can be applied as a coherent unit. Furthermore, it positively impacts the fourth challenge because it allows defining complex model operations from fine-grained model operations without coupling their implementations. This encourages the decomposition of coarse-grained model operations into fine-grained model operations. Each goal is explained in more detail in the following sections.

1.3.1. Capture Dependencies

The first goal positively impacts the understandability of MDE applications. Making dependencies between models and model elements visible to an application developer, who is going to work with these models, can increase the understandability (cf. [174]). Thus, application developers are aware of dependencies and are able to reason about the intentions of models. Having explicitly captured dependencies has a positive impact on making changes in MDE applications because captured dependencies can be employed to analyze the impact of changes.

In MDE applications, models can be considered at different levels of detail. For example, models are considered as being high-level artifacts[6] in MDE applications while the content of models (model elements) can be considered as low-level artifacts. An approach should be able to capture dependencies at any level of detail (between models and between its elements). This is necessary because model operations can operate on different levels of detail. For example, a model transformation can transform a model into another model or it can transform only a part of one model into a part of another model. Figure 1.6 illustrates dependencies between models and model elements that could also represent the application of different model operations.

Furthermore, in MDE applications it may be necessary to capture dependencies between artifacts that go beyond models that are conform to metamodels, e.g.; projects, folders, files, etc. This is required

[6]The term artifact is used as a superordinate term for all kinds of elements that are considered in a software developments process, e.g., metamodels, models, model elements, documents, etc.

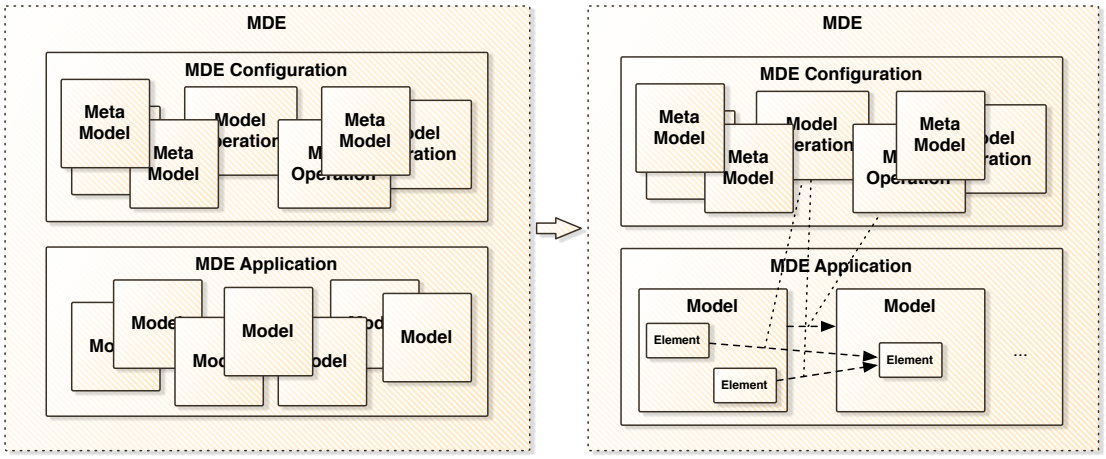

Figure 1.6.: Illustration of capturing dependencies between models and model elements

because certain model operations might be applied between models and other artifacts like folders. For example, a code generation usually takes a model as source and generates a set of files into a specific folder as target. Thus, the folder is related to the code generation.

The issue with most classical traceability approaches in the context of MDE is that they only support capturing dependencies between elements of models [138, 82, 105, 111, 171, 54, 127, 100, 106], which are also known as traceability-in-the-small. In the context of model management, approaches support capturing dependencies between models [32, 31, 14], which are also known as traceability-in-the-large. Recently, approaches emerged that are a combination of both [23, 143, 145, 146, 86]. However, the approaches shown in [23, 86] propose that dependencies between elements of models are captured in explicit traceability models. Thus, dependencies between models have a different syntax than dependencies between their elements. This strict separation between these two perspectives requires specific techniques to use them in combination, e.g., OCL cannot be applied to traverse from high-level dependencies to low-level dependencies. Furthermore, none of these approaches support artifacts beyond models.

Beside the positive impact to understandability of MDE applications and making changes in MDE applications, introducing explicitly captured dependencies also negatively impacts the changeability of an MDE application. Models and model elements are subject to continuous change and thus already captured dependencies may deteriorate or emerging dependencies are not captured.

1.3.2. Automatically Maintain Dependencies

The second goal is primarily motivated to compensate the negative impact of the first goal. Models in an MDE application change frequently, e.g., models are created, removed, updated. Thus, already captured dependencies get suspect or even obsolete. This requires reconsidering whether already captured dependencies should still exist or if certain changes have to be propagated to re-validate the already captured dependency again. Furthermore, it is not clear whether new dependencies exist and thus have to be captured. In any case, not being able to maintain dependencies appropriately can lead to deterioration, which decreases the reliability of the captured dependencies. Therefore, the second goal is automatically maintain dependencies, which means creating new dependencies and deleting or archiving obsolete dependencies. This positively impacts the changeability of an MDE application because only dependencies exist that are reliable.

In the context of MDE, maintaining dependencies is usually obtained by traceability approaches. The automated maintenance of dependencies can be distinguished between semi-automatic [51, 107, 45, 108, 81] and fully automatic approaches [138, 82, 54, 127, 100, 15, 69]. The major issue with semi-automatic approaches is the necessity to manually reason about correctness of automatically captured dependencies.

Recent approaches toward automatic maintenance of dependencies are strongly integrated into model operation technologies [138, 82, 54, 127, 100, 15, 69]. These approaches create traceability as by-product of applying a model operation. But there are also traceability approaches that support a dedicated and

fully automated maintenance of dependencies between elements of models [9, 148]. Nevertheless, these approaches only focus on traceability-in-the-small.

1.3.3. Automatically (Re-)Apply Heterogeneous Model Operations

The third goal is about the (re-)application of heterogeneous model operations in order to automatically re-apply those model operations that have been impacted by changes only. Because changes occur frequently, already applied model operations might have to be re-applied to propagate changes to other models. Doing this manually is tedious and error-prone due to the increased number of heterogeneous models and the increased number of heterogeneous model operations that have to be applied. If the re-application of a model operation is missing, inconsistencies might occur that can be propagated down to the implementation and into the final software system.

Only being able to compose coarse-grained heterogeneous model operations increases the probability that re-applied model operations overwrite their results accidentally if model operations propagate changes toward a shared model. For example, if disjoint changes were made in different models, propagating these changes toward a single model could lead to overwriting propagated changes in this model. This depends on the order of how model operations are re-applied and if the individual model operation is capable of updating models. Independently, if fine-grained model operations are provided and composed, only those parts of a model are going to be overwritten or updated that are actually impacted by changes. Thus, the probability of accidentally overwriting changes can be reduced.

Existing workflow and model transformation chain approaches [125, 92, 170, 161, 12, 129, 53, 76], support re-applying a set of heterogeneous model transformations as a coherent unit. Nevertheless, the major issue with these approaches is that they only support a coarse-grained re-application, which may imply the re-application of model operations or parts of model operations that are not affected by actual changes. The other approaches that support fine-grained compositions of heterogeneous model operations [165, 95, 43] do not explicitly consider the case of re-applying model operations.

1.3.4. Specify and Apply Compositions of Heterogeneous Model Operations

The fourth goal is about the specification of compositions of heterogeneous model operations and their subsequent applications. As many heterogeneous model operations may be required to reach an appropriate degree of automation in application development, it can get tedious and error-prone to apply each model operation individually. Thus, providing a facility to specify compositions of a set of heterogeneous model operations positively impacts the challenge of applying model operations because less (fine-grained) model operations have to be applied manually. The composition of model operations is illustrated in Figure 1.7. The figure illustrates that a collection of model operations can be applied as a coherent unit in MDE applications.

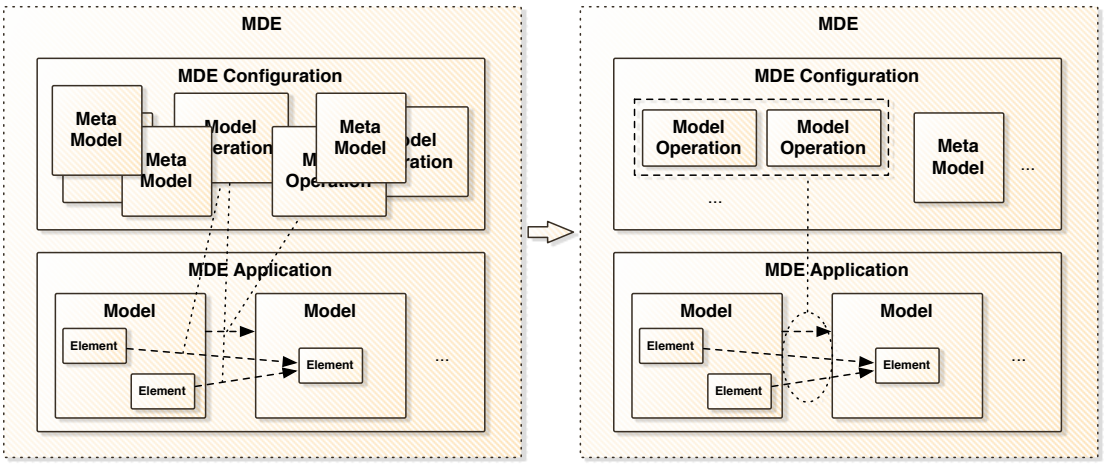

Figure 1.7.: Illustration of composing model operations

An approach must support specifying and applying compositions of heterogeneous models operations (implemented in different technologies). If this is not supported, it restricts the heterogeneity and reusability of an MDE configuration because it is not possible to use a specific technology to implement a specific aspect of a model operation as part of a whole. Furthermore, an approach must support specifying and applying compositions at any level of detail. If an approach only supports the coarse-grained composition of model operations between models, it restricts model operations in the ability of being reusable and adaptable. In certain situations model operations have to extend the ability of other model operations. Doing this at a high-level only requires that the extending model operation has to re-implement navigation concerns that are required only to find the right context for its application.

Another factor that can negatively impact the reusability and adaptability of model operations is if the composition facility is going to couple them to other model operations or technologies. This would increase coupling as well as decrease cohesion because model operations must implement composition concerns. Thus, the composition facility should not require any adaptation to the implementation of model operations, which means considering them as true black-boxes.

In recent years, the demand for approaches in the context of MDE that provide a way to compose heterogeneous model operations into more complex and coherent units is continuously increasing. A standard approach in the context of programming is make [77], which can be considered as a classical build tool. It allows specifying and applying chains of command line operations like compile, link, etc. More recently, the build tool Ant [77] emerged, which is similar to make but closer to the domain of Java programming. Recent approaches that directly emerged in the context of MDE [125, 92, 170, 161, 12, 129, 53, 76] are known as workflow or model transformation chain approaches.

All these approaches are able to build compositions of heterogeneous model operations based on shared source and target models, which makes them completely decoupled but also coarse-grained. These approaches do not support to specify and apply compositions of heterogeneous model operations at different levels of detail (fine-grained).

There are approaches that support compositions of fine-grained model operations even implemented in different technologies [165, 95, 43]. However, these approaches rely on their own traceability information that has to be created and interpreted within the model operations themselves. Thus, the major issue of these approaches is that they rely on the fact that implementations of the model operations have to be adapted that they can create and interpret traceability information. This decreases cohesion because the model operations now have to implement composition concerns. Furthermore, they cannot be considered as true black-box compositions, also if they state to be.

1.4. Contribution

The contribution of this thesis is the development of an approach that combines traceability and model management in the context of MDE. Thus, the approach addresses the four aforementioned challenges, which eases setting up / adapting the configuration of MDE and applying MDE subsequently, by providing a solution to each of the four defined goals.

The traceability aspect of the approach can be considered as a standalone traceability approach that supports capturing all kinds of dependencies between models and beyond, e.g., folders, documents, etc. Dependencies can also be captured at any level of detail, which means they can be captured between models but also between elements of models. This traceability approach does also supports the automated maintenance of dependencies, which is a vital component of any traceability approach. The model management aspect of the approach can be considered as a standalone model management approach that supports managing the application of heterogeneous model operations like model transformations, code generations, etc. Therefore, it supports the initial application of model operations as well as the subsequent (incremental) re-application in case of changes.

Nevertheless, considering these two approaches on their own is not the major contribution of this thesis. The major contribution of this thesis is the specification of complex compositions of heterogeneous model operations at any level of detail including their subsequent (re-)application, which is enabled by the combination of both approaches. Therefore, two different kinds of compositions are provided, which is a data-flow composition and a context composition. Furthermore, the approach enables the complex composition of heterogeneous model operations without requiring to implement additional composition concerns into the model operation. Thus, model operations are considered as true black-boxes.

1.4.1. Concepts

The contributions of this thesis are obtained by a concept called executable and dynamic hierarchical megamodel, which foundation is an extension of the original idea of the megamodel introduced by Bézivin et al. [32]. The executable and dynamic hierarchical megamodel is a DSML that has two concepts on top called localization and execution. These two concepts provide additional functionality in order to achieve the stated goals of this thesis. The localization introduces the automatic maintenance of dependencies, which are captured in the megamodel. The execution supports the (re)-application of compositions of model operations that are captured by the megamodel. The whole approach is embedded into a model management framework that is conceptually illustrated in Figure 1.8.

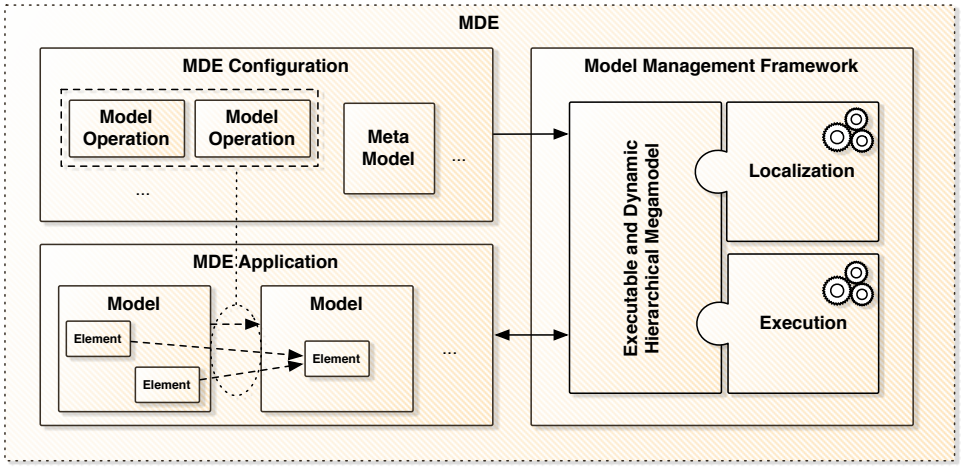

Figure 1.8.: Conceptual overview of a model management framework and its integration into MDE

The figure shows how the model management framework is integrated into MDE, as defined previously. The framework is considered as an integral part of MDE that is located beside an MDE configuration and an MDE application. The model management framework uses the executable and dynamic hierarchical megamodel as an abstraction layer which provides abstract representations of artifacts in an MDE configuration and artifacts in an MDE application. It also captures all kinds of dependencies between these abstract representations. The localization and the execution comes with functionality that depends on the executable and dynamic hierarchical megamodel.

The executable and dynamic hierarchical megamodel is developed in three consecutive steps with each step is developed in a major chapter of this thesis. The first step is the hierarchical megamodel followed by an extension called dynamic hierarchical megamodel. Finally, the dynamic hierarchical megamodel is extended and called executable and dynamic hierarchical megamodel.

The hierarchical megamodel acts as an abstract representation of artifacts that are models, which is similar to the original idea of a megamodel. In addition, the hierarchical megamodel also acts as a representation of model elements thus that is can represent any level of detail. In addition, it is also able to represent artifacts beyond models, e.g., projects, folders, files, etc. The hierarchical megamodel does also distinguish between representing an MDE application (instance perspective) and an MDE configuration (type perspective). As with a megamodel, the hierarchical megamodel supports capturing dependencies explicitly between representations of artifacts. Because of the separation between type and instance perspective, all captured dependencies have an explicit type as their semantic. In addition, the hierarchical megamodel supports capturing dependencies at any level of detail including explicit composition relationships in between.

The localization is a concept that provides means to automated maintenance of captured dependencies in a dynamic hierarchical megamodel. The dynamic hierarchical megamodel is an extension of the hierarchical megamodel that provides more details concerning the semantic of captured dependencies. Two different variations of localizations are introduced, which are called batch localization and incremental localization.

The execution is a concept that provides means to (re-)apply compositions of heterogeneous model

operations. Therefore, an extension of the dynamic hierarchical megamodel is shown called executable and dynamic hierarchical megamodel. This extension allows considering captured dependencies as the application of heterogeneous model operations. Because of the hierarchy, coarse-grained as well as fine-grained heterogeneous model operations can be composed and applied. The execution also supports the (re-)application of individual model operations that are composed of a set of fine-grained and heterogeneous model operations and the incremental (re-)application of heterogeneous model operations that are impacted by changes.

	Concepts / Goals	Hierarchical Megamodel	Dynamic Hierarchical Megamodel	Executable and Dynamic Hierarchical Megamodel
G1	Capture dependencies	x	x	x
G2	Automatically maintain dependencies		x	x
G3	Automatically (re-)apply heterogeneous model operations			x
G4	Specify and apply compositions of heterogeneous model operations			x

Figure 1.9.: Relationship between thesis's primary concepts and goals

Figure 1.9 illustrates at a glance which concepts satisfy which goals. The hierarchical megamodel achieves the first goal only. The dynamic hierarchical megamodel is an extension of the hierarchical megamodel and thus achieves also the first goal and the second goal. As the executable and dynamic hierarchical megamodel is an extension of the dynamic hierarchical megamodel, it also achieves the first and the second goal. Additionally it achieves the third and the fourth goal.

1.4.2. Validation

The validity of this thesis is obtained by showing that the introduced concepts indeed satisfy the previously stated goals and therefore addresses the aforementioned challenges. The validation process is an integral part of this thesis because case studies, including simple application examples, are employed throughout this thesis. They are used to explain the shown concepts as well as to show their feasibility concerning the stated goals. Aspects that are not shown as integral part of this thesis are explicitly discussed in a separate evaluation chapter. The evaluation will demonstrate the expressiveness of the composition of heterogeneous model operations.

1.5. Outline

This section provides a brief tour through this thesis. In Chapter 2, an overview about state-of-the-art in the context of MDE, traceability and model management is presented. This overview is necessary because it presents the foundations of this thesis. That chapter further provides four case studies that are used to explain and evaluate the introduced concepts.

In Chapter 3, the hierarchical megamodel is introduced and formally defined. As already mentioned, the hierarchical megamodel acts as the foundation for the localization and execution. In Chapter 4, the dynamic hierarchical megamodel is introduced as an extension of the hierarchical megamodel and formally defined. The chapter also introduces the localization, which is considered as a traceability approach. In Chapter 5, the executable and dynamic hierarchical megamodel is introduced and formally defined. This megamodel complements the dynamic hierarchical megamodel. At the same time, that chapter defines the execution functionality on top of the executable and dynamic hierarchical megamodel. In the end, that chapter defines a traceability and a comprehensive model management approach.

In Chapter 6, the executable and dynamic hierarchical megamodel is further evaluated by means of application examples of the shown case studies. It further discusses the accomplishment of the stated goals. In Chapter 7, this thesis is delimited against related work in area of traceability and model

management. Finally in Chapter 8, this thesis is concluded and limitations as well as additional future work is discussed.

2. Basics and State-of-the-Art

Contents

* * *

This chapter provides an introduction to MDE, techniques to support it, and an overview of its possible integration into software development (prefigured in the introduction). This chapter additionally provides a conceptual overview of MDE including its fundamental techniques, such as model operations, traceability and model management. This includes an overview of state-of-the-art concerning these fundamental techniques (see Section 2.1). A set of case studies is introduced, be employed later in the thesis for explanation and evaluation (see Section 2.2). A summary of this chapter is provided in Section 2.3.

2.1. Model-Driven Engineering

The fundamental idea of MDE is to provide a complete specification of system only through the use of DSMs. In the context of software engineering, the system is a software system and the introduction of MDE means providing a collection of DSMs to describe the whole software system (or specific aspects of it). From these DSMs, an executable software system should be automatically derived, either obtained by directly interpreting the DSMs, or by using model operations (code generations) to generate executable code from DSMs.

In order to automatically interpret DSMs or generate executable code from DSMs, a DSM must conform to a specific DSML. This is generically explained in terms of models and metamodels in Section 2.1.1, below.

In the context of MDE, model operations are not only applied to generate executable code from DSMs. Generally, since a software system is built of various aspects, versatile DSMs are required to specify the overall software system. A DSM can be used to specify different aspects of a software system and even different levels of abstraction of those same aspects. Thus, model operations are needed in order to be able to translate between different overlapping aspects, and to transform from one level of abstraction into another. In addition, DSMs have to be analyzed concerning various characteristics and requirements even at different levels of abstraction. An overview of different kinds of model operations and available techniques is given in Section 2.1.2.

In developing complex software systems, different aspects of the software system are usually not independent. Even DSMs which specify these different aspects are not independent. These dependencies can lead to serious inconsistencies when continuously changing the DSM, and not propagating these changes appropriately to other DSMs. A common way of handling these issues is the introduction of traceability, an overview of which is given in Section 2.1.3.

As mentioned earlier, the number of model operations in MDE is increasing. Managing the (re-)application of model operations is becoming another dimension of MDE's complexity which has to be treated explicitly. The recent discipline of model management in the context of MDE is about managing this dimension of complexity. In Section 2.1.4, current state-of-the-art in model management is revisited.

Finally, the composition of model operations has lately gained an increasing focus in MDE. Initially, this topic was strongly related to the composition of model transformations, that is, the composition of model transformation rules in homogeneous model transformations. Recently, it has also become a topic in model management, which is about the composition of heterogeneous model operations that are not necessarily model transformations. Section 2.1.5, is a presentation of the basics of model transformation composition.

2.1.1. Models, Metamodels and Meta-Metamodel

A model is not a specific thing, but a model always has a specific purpose. The purpose of a model is to (partially) represent a system, which enables questions to be asked of a model instead of being put directly to the system it represents:

> *"A model is a representation of a given system. For each question of a given set of questions, the model will provide exactly the same answer that the system would have provided in answering the same question."* [14]

But why should I ask a model instead of directly asking a system? This depends on the constitution of a model. In the case of the model being a direct copy of the system, there is no benefit to working with it, because I could directly ask the system instead. In the case of the model being in some way abstracted from the system[1], asking questions of the model rather than the system itself becomes more cost-effective, simple and safe(cf. [142]). If we wish to realize these benefits, a model should not try to capture all aspects of the system it represents, but rather should be an attempt to abstract from those aspects that are not necessary for the specific purpose of answering a set of questions. In this way, an efficiently modeled system is in fact represented by a set of different models, each of which captures or addresses specific aspects of the system [28].

This idea of a model has been used for a long time and in different contexts, e.g.; statistics, biology, ecology, economics, etc. Common to all these discipines is that their models primarily represent an existing system, e.g., the human blood circulatory system by the English physiologist W. Harvey (cf. [28]). In disciplines related to engineering, models are moreover used to develop a system that does not yet exist. Models thus are extended into two types: descriptive (system exists), used to understand a system, or prescriptive (system does not exist), used as a blueprint for building a system. For example, in software engineering models are used to build a software system.

The models' fortune depends on their usability – and this in turn depends upon their comprehensibility. A map can only be used as long as one can interpret its information and therefore, it always conforms to a specific abstract syntax that defines the concept shown by a map, e.g., a legend. This abstract syntax defines the concepts and the relationships between these concepts, and is therefore to be used to interpret the model's information. Thus, a model always conforms to a metamodel. Other examples are programs and programming languages, which are defined in a certain grammar (see [28]). Figure 2.1 summarizes the explained relationships between systems, models and metamodels.

Figure 2.1 also shows that a metamodel conforms to a meta-metamodel. This is a fundamental principle in MDE, which leads to the ability of metamodeling. If a metamodel conforms to a meta-metamodel (acting as the language for the metamodel), it can be precisely specified, supporting automation.[2] Enabling automation is one of the primary principles of MDE, besides raising the level of abstraction. Thus, model operations can be specified directly on metamodels and further applied on models to increase the level of automation in MDE. Lets now turn to a discussion of model operations.

2.1.2. Model Operations

The performing of model operations is a foundational principle for MDE, because they support increasing the automation level. In this thesis, model operations are considered as all kinds of automated devel-

[1] Here, the term abstraction is used as fading out details.
[2] The meta metamodel of all metamodels that are shown in this thesis is Ecore, which is explained in Section B.1.

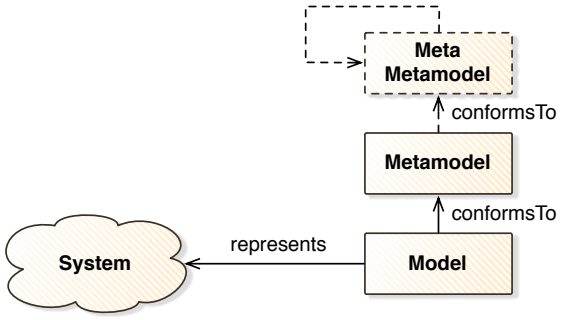

Figure 2.1.: Systems, models, metamodels and meta-metamodels

opment activities that are applied to models. It is further distinguished between two kinds of model operations – model synthesis and model analysis.[3] Principally, model analysis is similar to model synthesis because the outcome of any analysis is any information that is synthesized by analyzing models. Nevertheless, this thesis distinguishes between activities that are used to analyze models, and activities that are used to synthesize models.

2.1.2.1. Model Synthesis

Usually in the literature, model synthesis is known as model transformations. Currently, a plethora of model transformation approaches exist in literature as well as in practice. The type of model transformation varies due to the type artifacts that are used or produced by the model transformation. Commonly known types of model transformations are model-to-model, model-to-text and text-to-model transformations. Furthermore, an important property of model transformations is the number of models that are taken as input and output, where either figure may be single or multiple.

2.1.2.1.1. Model-to-Model Transformation A model-to-model transformation is applied to transform models into different perspectives or to different levels of abstraction, and such transformations can have various characteristics [117].

A model-to-model transformation can be defined as either endogenous or exogenous. The transformation is endogenous if the metamodels of the source and target models are similar, and is exogenous if the metamodels are different. Therefore, a model transformation that merges two models representing different versions into a new model would be endogenous because all models have the same metamodel.

A model-to-model transformation can be defined to be either in-place or out-place. An in-place model transformation just updates an existing model, which implies that the source and target model of an in-place model transformation is similar. Thus, in-place model transformations are always endogenous. In case of an out-place model transformation, the source and target models are not similar but their metamodels might be similar. Thus, out-place model transformations can be either endogenous or exogenous.

A model-to-model transformation can implement different execution directions. Usually, model transformations are considered to be uni-directional, meaning they can only be executed from source to target. Bi-directionally executable model transformation is called model synchronization, where changes in the target model can be propagated back to the source model. Due to these various characteristics of model-to-model transformations, a huge set of applications are possible, e.g.; merge, optimization, adaptation, refactoring, simplification, migration, reverse engineering, etc.

On the other hand, many different technologies for realizing model-to-model transformations have been established during the last decade. All these technologies have different strengths and weaknesses due to their slightly different focus. These different technologies can be divided into three classifications: declarative model transformations (VIATRA [166], Kent Model Transformation Language [10, 11], Tefkat [104], UMLX [172], AToM3 [103], BOTL [114], TGG [147, 68, 69], MOLA [87], AGG [159]), imperative

[3]The terms synthesis and analysis have also been used in [22] with a similar meaning.

(operational) model transformations (Story diagrams[4] [177, 58, 65, 64], Kermeta [119], Xtend [131]) and hybrid model transformations (GReAT [7], ATL [84, 83], QVT [100, 124], ETL [97]).

The specification of declarative model transformations focus on *what* has to be transformed, while the specification of imperative model transformations focus on *how* something has to be transformed. Thus, declarative model transformations are usually rule-based, with each rule describing a specific mapping that defines what has to be transformed. How the transformation is actually obtained is hidden in the transformation technology. Imperative model transformations are operational and thus the specification of such transformations explicitly contains a control flow that defines in detail how the transformation is obtained. Hybrid model transformations are a mixture of both, and thus support declarative as well as operational aspects.

2.1.2.1.2. Model-to-Text Transformations Model-to-text transformations are very common, and are more commonly known as code generations, e.g., generating Java or C++ code from UML models. These kinds of transformations are important to MDE because they are usually used to automatically generate executable code. Some existing technologies for model-to-text transformations are Xpand [130], JET [5] or EGL [141].

2.1.2.1.3. Text-to-Model Transformations Text-to-model transformations are a more recently-developed technique, employed to automatically create metamodel-based models from grammar-based (textual) models. Some recent technologies for this kind of transformations are Xtext [132] or EMFText [75].

2.1.2.2. Model Analysis

The term model analysis comprises all activities concerned with analyzing models with resepct to certain properties. Model analyses are necessary for verification or validation purposes, e.g., for analyzing the consistency of models, etc. Thus, depending on the property and kind of model, different analyses can be applied. Structurally, a model analysis takes one or many models as input and provides a model or some report (which can also be considered as model) as output. Some of them are explained in later sections.

- **Structural comparison** – is employed for comparing the structure of two or more models. Technologies for this kind of analysis are EMF compare [4] and ECL [94] for comparing Ecore [3] models or UMLDiff [175] for comparing UML models.
- **Constraint checking** – is employed for checking structural constraints (conditions) on models by employing technologies like OCL [139], EVL [98] and Xcheck [6].
- **Verification** – is employed for ensuring that behavioral models satisfy specific state reachability properties using model checking technologies as shown in [67].
- **Simulation** – is employed for simulating the behavior of models concerning specific properties like throughput or congestion in performance-based simulations. Such simulations can be applied, e.g., on petri nets [89].

2.1.3. Traceability

Traceability is an emerging discipline that established itself in the domain of requirements engineering in the early 1970s, and is considered as a fundamental part of software development today. Because traceability is important in many different domains, diverse definitions of the term traceability exist in literature.

In the context of requirements traceability (cf. [137]), the most prominent definition of the term traceability is given by Gotel and Finkelstein in [70]: *"[...] the ability to describe and follow the life of a requirement, in both a forwards and backwards direction (i.e., from its origins, through its development and specification, to its subsequent deployment and use, and through periods of ongoing refinement and iteration in any of these phases) [...]"*. Their definition implies that traceability either originates from requirements, or results in requirements.

[4] Story diagrams are used several times in this thesis. Thus, they are explained in more detail in Section B.2.

The term traceability is considered to be more generic in the domain of software traceability, which is defined by Spanoudakis and Zisman in [157] as: *"[...] ability to relate artefacts created during the development of a software system to describe the system from different perspectives [...]"*. Other definitions in this context are provided by Aizenbud-Reshef et al. in [8]: *"[...] we regard traceability as any relationship that exists between artifacts involved in the software-engineering life cycle [...]"*, and by Shaham-Gafni and Hartman in [153]: *"[...] a relationship between entities: a set of source entities, and a set of target entities. The exact meaning of the relationship depends on the context in which it is used [...]"*.

Applying traceability usually requires that certain activities be carefully applied: [174].

- **Planning and preparing** – is the activity that is obtained during the planning phase of a software project. It includes identifying the kinds of artifacts to be created during the project, and identifying the kinds of relations that will be traced. Furthermore, required tools are configured for later usage.

- **Recording** – is the activity of creating traceability information while conducting the software project. The traceability information is persistently stored in data structures or traceability schemes as prepared in the previous activity. Recording can be obtained online (prospective) or offline (retrospective). Online means that relations are automatically created as a by-product of development activities, e.g., model transformations. Offline means that relations are automatically or manually created after the development activity has been completed.

- **Using** – is the activity of leveraging previously recorded traceability information. This activity is versatile, and depends on the need of using the traceability information, and on the stakeholder interested in this information. A common example is impact analysis.

- **Maintaining** – is basically two activities. The first is maintaining the structure of traceability, which means either changing the kind of traceability information that is recorded, or adding new kinds of traceability information. The second activity is to maintain recorded traceability information in case some artifacts have changed.

We shall now introduce state-of-the-art traceability approaches, classified by means of their traceability schemes and by their recording and maintenance capabilities.

2.1.3.1. Traceability Schemes (Representations)

The representation of traceability information can be realized in various ways. Three common representation forms are explained in the following. A very ancient way of representing traceability links is using traceability matrices. In a traceability matrix, each field represents a possible connection between two artifacts. The semantic of the traceability link can be defined by using different colors in the matrix, or by using names for different types of traceability links. Representing traceability links as hyperlinks (also known as cross-references) is the typical form when the artifacts are document-based. Thus, references or links are directly written beside a requirement or some artifact in a document. A user can click on this link to navigate to the connected artifacts.

A recent way of capturing traceability links is by using metamodel-based traceability models. For example, in [113] an approach called AMW is introduced, which provides a generic metamodel (weaving model) for traceability that can be extended for each pair of models that should be related. In [15] another traceability metamodel is demonstrated, which contains the notion of link types as well as composite links, which can be considered as hierarchical traceability links. In [50] and [49] another generic metamodel is shown, which is similar to the weaving model shown in [113]. Other approaches that provide traceability using metamodels are shown in [105, 111, 120, 96, 171, 39, 127, 42, 134, 81, 71, 102].

2.1.3.2. Recording and Maintaining Traceability

Automated recoding and maintenance of traceability links is necessary because there may implicitly exist many thousands of dependencies between all kinds of artifacts. The various approaches can be divided between prospective (online) and retrospective (offline) types (cf. [19]).

2.1.3.2.1. Retrospective A retrospective approach creates traceability links ex post facto from a set of given artifacts. Classical traceability approaches are retrospective and rely on information retrieval methods to automate traceability link creation. In contrast to prospective approaches, retrospective approaches are applicable in settings where no change information is available. Currently, there is a plethora of approaches that support retrospective traceability between all kinds of artifacts [113, 111, 8, 107, 45, 106, 81, 102, 19, 148].

2.1.3.2.2. Prospective A prospective approach generates traceability links in situ, e.g., by directly analyzing actions or events (e.g., [51, 52, 18, 15, 108, 109, 110, 19]). In MDE, a common way of realizing prospective traceability is to generate traceability links as by-product of model transformations (e.g., [36, 82, 54, 171, 68, 69]). An inherent benefit of prospective approaches is that they are efficient and scalable because they are incremental by nature. However, they either require a tight integration into existing environments or technologies (e.g., model transformations), or are restricted to capturing traceability links from behavioral information alone.

2.1.3.2.3. Recall and Precision In information retrieval, dependencies between software artifacts cannot be formally specified. Thus, heuristic methods like latent semantic indexing (LSI) (e.g., [80]) are employed to determine the similarity between software artifacts by means of textual similarity. The inherent benefit is that traceability links between informal as well as formal software artifacts could be established with only one common or pre-defined heuristic. However, using a heuristic implies manual post-processing because required traceability links may be not automatically established, or traceability links could be falsely established (false positives). Thus, the quality of information retrieval approaches is "lower" (measured by means of precision and recall [59]) than in non-information retrieval approaches.

Recall is defined as the ratio between the number of relevant and retrieved traceability links and the number of relevant traceability links as shown in the following equation. *Relevant* is a set of traceability links that should be generated by the method for a given query or rule and *Retrieved* is the set of traceability links that are actually returned by a method for a given query or rule. A recall of 1.0 means that all traceability links that are retrieved are also relevant.

$$Recall = \frac{|Relevant \cap Retrieved|}{|Relevant|}$$

Nevertheless, only considering recall as a quality metric for information retrieval methods is insufficient because simply retrieving all possible traceability links would also result in a recall of 1.0. Thus, another metric is required – precision. Precision is defined as the ratio between the number of relevant and retrieved traceability links and the number of retrieved traceability links, which is shown in the following equation.

$$Precision = \frac{|Relevant \cap Retrieved|}{|Retrieved|}$$

A precision of 1.0 means that only relevant traceability links are retrieved. Thus, the quality of an information retrieval method measured using probabilistic similarity must consider recall and precision.

On the contrary, formal methods, i.e. pattern matching, can also be employed to establish traceability links (e.g., [152]). A formal method does not necessarily require any manual post-processing because the dependencies are precisely specified and do not contain any vague statements (as in the case of information retrieval). Because of their precision, formal methods are best applied in a context of formal and precisely specified artifacts, such as models that conform to a precisely specified metamodel.

2.1.3.3. Using Traceability

The use of traceability can also be differentiated into requirements traceability and software traceability. Winkler and Pilgrim determined the following usages in [174]. Typical usage of requirements traceability might be: providing system adequateness; validating artifacts; improving changeability; extracting metrics; monitoring progress; assessing the development process; understanding the software system; tracking the rationale of the software system; establishing accountability; finding reusable elements or finding best-practices, etc. Concerning software traceability, potential usages could be: supporting design

decisions; understanding and managing artifacts; understanding and debugging model transformations; deriving usable visualizations; change impact analysis; synchronizing models, etc.

The benefits of applying traceability to software development are manifold, and neglecting traceability in software development comprises inherent risks, e.g., less maintainable software or even software defects due to inconsistencies or omissions (see [174]). Nevertheless, in practice, traceability runs the risk of being counterproductive if not applied pragmatically and effectively. If traceability information maintenance is insufficiently automated, users start spending most of their time on manually maintaining links. Even worse, if artifacts evolve but traceability information is not maintained appropriately, inconsistencies occur which can lead to wrong assumptions etc., which can lead to software defects.

2.1.4. Model Management

The term model management has been in use since the nineteen-seventies, indicative of the large amount of research which has been undertaken in this field. Perhaps unsurprisingly, there is no clear consensus about the actual meaning of model management. Different researches from different contexts defined and applied the term model management differently. In the sense of a classical model management, the meaning of the term depends on what model means. In early approaches, models were either treated as procedures or executable functional units (cf. [33, 34, 35, 133]), or as data that can be analyzed by procedures (cf. [37]).

In the beginning of this century, the term generic model management was introduced by Bernstein et al. [26], and evolved in subsequent works of the authors [25, 115, 116, 90, 156]. According to their perspective, the primary goal of model management is defined as: *"[...] develop a set of powerful high-level operators that simplify the programming of such applications, and increase the productivity of developers by an order of magnitude [...]"* in [116] and *"[...] the development of new technologies and mechanisms to support the integration, evolution and matching of data models at the conceptual and logical design level [...]"* in [156]. Another definition for generic model management is given in [144]: *"[...] model management has emerged as a way to address these complexities by proposing that model relations be expressed as first class objects called model mappings and that generic operators be defined that could be used to manipulate models and mappings in a sound way to achieve various modeling goals [...]"*.

Thus, in generic model management models are considered as data stored in a database, which are manipulated by applying generic model management operators. The operators are only defined on a common model for generic model management by using a common formalism to reflect as many heterogeneous models as possible. The major problem of generic model management is that expressivity is lost due to abstraction (using the common formalism). Nevertheless, generic model management can be seen as the transition from classical model management in the last century to more recent model management activities in the context of MDE.

In this thesis, as well as in other recent model management approaches, the term model management is considered as the opportunity to define relationships (mappings) between heterogeneous artifacts as first-class citizens, and to apply operations to manipulate artifacts to achieve various software development goals.

2.1.4.1. Model Management with Megamodels

Today, the term model management is common in the domain of MDE. Bézivin et al. [32] further introduced the term global model management. The term global came from the need for a global vision, which can be provided by the modeling-in-the-large, an activity of establishing and using global relationships between macroscopic entities (models and metamodels) by ignoring internal details of these macroscopic entities, which is realized by the so-called megamodel (cf. [31]). Megamodeling can be used to manage a collection of related models as well as to manage automated operations like model transformations (these operations are not generic as in generic model management).

The term megamodel occurs in several publications (cf. [29, 23, 55, 56, 135]) and is used as fundamental model to (global) model management.

Bézivin et al. were amongst the first founders of the term megamodel. Their view of the term megamodel has evolved during the recent years. The first definition was given in [29]: *"[...] a megamodel is a model with other models as elements [...]"*. In 2007, this definition is further complemented in [23]: *"[...] a megamodel contains relationships between models [...]"*. Bézivin et al. propose different

applications of megamodels. In [29, 32, 31, 14], megamodels are applied to support MDE by using it for model management. Thus, megamodels provide a global view on models. By contrast, in [23], megamodels are applied to facilitate traceability between models and their elements. In [24] it is shown how to apply megamodels to the management of complex systems like the Bugzilla[5] system in Eclipse.

J. M. Favre can also be seen as one of the founders of the term megamodel, who, in comparison to Bézivin et al., has a strong theoretical focus on megamodels. In 2004, he gave the following definition in [55]: *"[...] the idea behind a megamodel is to define the set of entities and relations that are necessary to model some aspect about MDE [...]"*. In [56], he provided an extension of this definition: *"[...] a megamodel is a model that represents this kind of complex arrangements without entering into the details of each artifact [...]"*. J. M. Favre applies megamodels to model MDE [55, 56, 57]. He does not focus on the applicability of megamodels but on reasoning about relations that can exist in the context of MDE. He defines exemplary patterns in MDE by using the megamodels.

Perovich et al. provide the following definition of the term megamodel in [135]: *"[...] a megamodel is a model composed of related models [...]"*. They facilitate megamodels to design software architectures in as shown in [135]. In their approach, a megamodel defines a software architecture, and design decisions are encoded in the megamodel as model transformations that are connected to relations between models.

Vignaga et al. provided another application of megamodels in [167]. They assume that megamodels contain MDE artifacts and have to be updated if those artifacts are manipulated. Such manipulations are seen as "programs on megamodels". To ensure type-safety, weaving models have to be typed well, because they consider weaving models as the relationships in a megamodel.

2.1.4.2. Model Management with Macromodels

In [144] it was mentioned that model management supports software development by dealing with a collection of related models, which seems to be the leitmotif of subsequent works of Salay et al. [143, 145, 146].

Salay introduced the term macromodel. A first definition of the term macromodel was given in [143]: *"[...] a macromodel is a graphical model whose elements denote models and whose edges denote model relations [...]"*. Later, another and slightly different definition of the term macromodel was given by Salay et al. in [146]: *"[...] a macromodel consists of elements denoting models and links denoting intended relationships between these models with their internal details abstracted away [...]"*. Thus, a macromodel is basically a model over models that additionally captures relationships.

In [143, 145, 146], Salay et al. have shown how they apply macromodels. They use macromodels to capture a software designer's intention by defining relationships between different models representing different views of the system to be developed. Therefore, a macromodel specifies some pattern that defines a certain intention of a software designer while instances of that macromodel represent current models of the system. Based on the macromodel and an instance of that model, an automated analysis is provided that estimates if the intention of a software designer is satisfied.

The macromodeling approach of Salay et al. does not focus on operations to manipulate models but rather on the analysis of a software designer's intention. Their approach can be considered as the so-called traceability-in-the-large.

2.1.5. Composition of Model Transformations

As explained in the introduction, MDE configurations may have to be (partly) reused in different software development projects or during their development they may evolve from time to time. This implies that model operations may have to be reused. By decomposing complex model transformations into smaller and less complex transformation units (rules), the reusability of model transformations can be increased [126]. However, when decomposing a model transformation, it has eventually to be composed, in order to once again build a coherent model transformation.

In [44], Czarnecki and Helsen have exhaustively analyzed the characteristics of model transformations including characteristics that are related to the composition of model transformation rules. This includes rule scheduling, a mechanism defining the order in which transformation rules are applied. They distinguish between implicit and explicit rule scheduling. Implicit rule scheduling means that no explicit specification exists that defines the order of applying the transformation rules. Explicit rule scheduling,

[5] https://bugs.eclipse.org/bugs/

as the opposite of implicit rule scheduling, means that an explicit specification exists that determines the order of applying transformation rules.

Explicit rule scheduling is further divided between internal and external rule scheduling. Explicit and internal rule scheduling means that the order of applying transformation rules is specified explicitly within the transformation rules, e.g., a transformation rule directly invokes another transformation rule. Explicit and external rule scheduling means that the order of applying transformation rules is separated from the specification of the transformation rules.

In the case of explicit rule scheduling, the scheduling of model transformation rules can also be broken down into three kinds. Firstly, model transformation rules are scheduled by means of an external specification, e.g., a state machine or a script that invokes model transformation rules. Secondly, model transformation rules are scheduled by means of shared sources and targets. Thus, if the target of one model transformation rule is the source of another model transformation rule, an explicit scheduling is given. This is usually known as a workflow or chain and is subsequently called data-flow composition. Thirdly, model transformation rules are scheduled by means of requiring the application of another model transformation rule as context for its own application. This kind of composition is usually applied in declarative model transformation approaches, which is further called context composition.

2.2. Case Studies

In the following, three case studies from different domains are presented. Each case study manifests certain challenges discussed in the introduction. Some of the case studies explain the approach of this thesis, while others are used for a more detailed evaluation of the approach. These case studies are simplified so as to focus on specific issues and because they are used for proof-of-concept purposes only.

2.2.1. Deployment Model-Driven Architecture (D-MDA)

This case study is shown because it is employed to explain how the approach of this thesis is capturing dependencies explicitly and how dependencies are automatically maintained.

The deployment model-driven architecture (D-MDA) case study was a research project in cooperation with CA Labs[6]. Its aim was to improve time-to-value and product quality by enabling CA Services and CA R&D to evaluate faster IT solution alternatives. The architecture supports highest quality adjustments of deployment configuration options in order to choose the best customer fitted IT solution. D-MDA was an envisioned tool and platform tightening the collaboration between Customers, CA R&D and CA Services around architectural requirements. The idea of this case study was to enable the exchange of architectural knowledge based on a simple set of modeling languages, minimal documentation efforts, and automation capabilities.

The set of core modeling languages in D-MDA are reference architectures, solution architectures and IT infrastructures. A reference architecture is an architectural description of CA software products, whose goal is to enable CA Services to focus on creating deployment plans, known as solution architectures or IT solutions. These plans are based on reference architectures provided by CA R&D. A reference architecture is basically a model of the configuration possibilities of software products. A solution architecture is developed by selecting components from CA products' portfolio (reference architectures), and configuring these components according to customer needs. Solution architectures are deployed on physical machines, which are specified in separate IT infrastructures. An IT infrastructure reflects the concrete physical infrastructure of a customer requesting a solution architecture. In the end, the solution architecture is a solution that provides a certain service required by a customer.

Figure 2.2 shows on overview of D-MDA and the involved roles at CA. It shows that CA R&D is responsible for providing reference architectures for the software products they develop. These reference architectures are provided to CA Services that are going to configure these software products for CA customers in order to developer IT solutions. This is managed by creating solution architectures based on IT infrastructures, which reflect the IT of a CA customer. Because CA Services usually also integrate software products into IT solutions that are not developed by CA R&D in-house, solution architectures can be used as starting point for creating new reference architectures for 3rd party software products, which can subsequently used by other CA services. Thus, reference architectures should guide the

[6]CA Labs is the research department of CA Inc.; http://www.ca.com

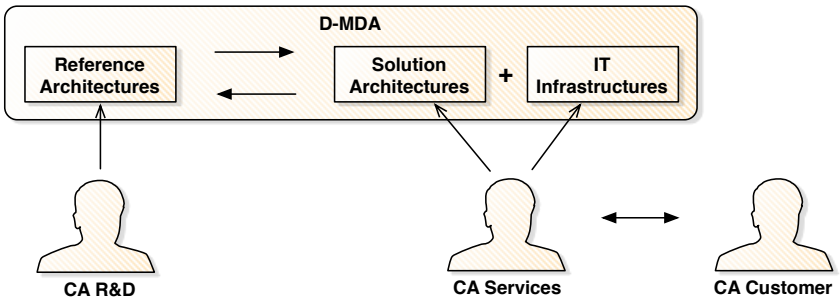

Figure 2.2.: Overview of D-MDA

development of solution architectures but not hinder the development. Therefore, solution architectures should also be developed without necessarily having all reference architectures available. This is realized by loosely couple these two modeling languages by means of soft references in between only.

The reference architecture has two possible applications. On the one hand, it is used for developing a software product. Thus, it describes a high-level architecture, which is made of components at a high-level of abstraction. On the other hand, it is used to capture all valid configurations of a software product comprising structural issues as well as component attributions, which are important for the subsequent deployment of the software product. In this thesis, a simplified metamodel of the reference architecture is used, which is sufficient for proof-of-concept. The metamodel is shown in Figure 2.3.

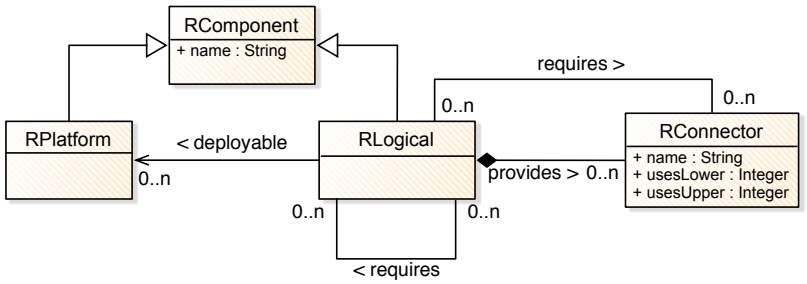

Figure 2.3.: Reference architecture metamodel

A reference architecture is built of components (RComponent) and connectors (RConnector). A component can be either a platform (RPlatform) or a logical component (RLogical). A logical component is a representation of a software product that is deployable to platforms, which is specified by the deployable association. A connector is a communication interface between logical components. A logical component can provide connectors as well as require them, expressed by the associations provides and requires. A connector has two integer attributes usesLower and usesUpper, which represent lower and upper bounds for the number of components that can use a connector provided by another component.

As an example, a simple reference architecture is shown in Figure 2.4. The example describes a security product by means of four logical components.[7]

SecurityAgents check for security breaches by validating if all security policies[8] are satisfied on the platform to which the SecurityAgents are deployed. Security policies are specified within a CentralPolicyDistributor and further distributed to all indirectly connected SecurityAgents via the Policy connector. Therefore, all security policies are first distributed to all PolicyGateways using the Policy connector. PolicyGateways are used for workload purposes when distributing security policies to SecurityAgents. If the number of SecurityAgents increases, new PolicyGateways need to be connected to the CentralPolicyDistributor via the Policy connector to keep the number of SecurityAgents per PolicyGateway low. The Reporter is used

[7] All elements are instances of the according metamodel but are shown by means of a concrete syntax to improve readability. Logical components are visualized by means of rectangles. Platforms are also visualized by means of rectangles but with a (P) in the label. Connectors are shown by mean of circles, which is known from UML component diagrams.

[8] Security policies are not explicitly modeled here.

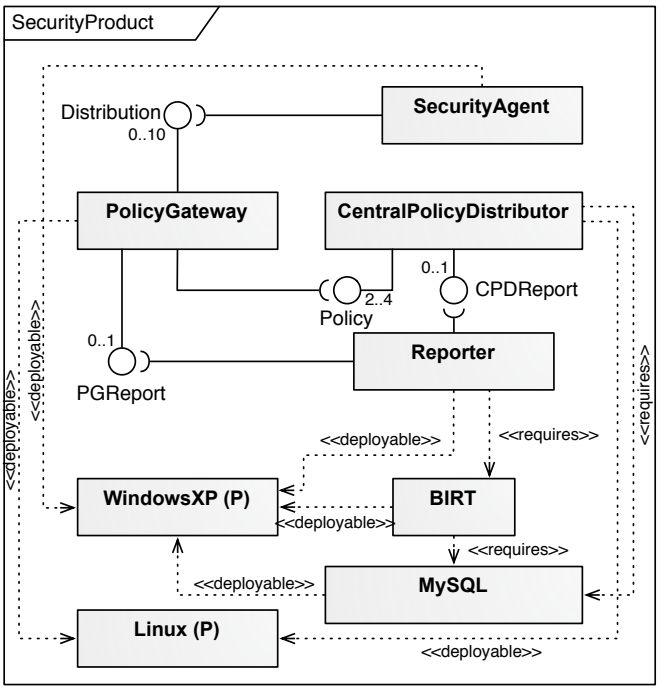

Figure 2.4.: Exemplary reference architecture of a security product

to filter logging information, which are sent from all SecurityAgents to the PolicyGateway and subsequently to the CentralPolicyDistributor via PGReport and CPDReport connectors, respectively.

This reference architecture shows even more information. The numbers next to connectors are the attributes usesLower and usesUpper required by the related RConnector class in the reference architecture metamodel. For example, the Policy connector requires at least two and at most four PolicyGateway components to be connected to it. The stereotyped connections are references of the type deployable between the classes RLogical and RPlatform. For example, the Reporter component is deployable to platforms of type WindowsXP.

A solution architecture is a configuration and deployment specification for specific software products that can be defined by reference architectures. Since a reference architecture is used as a blueprint for supporting the configuration and deployment of these software products, there are implicit dependencies between solution architectures and reference architectures because software products in solution architectures can be configurations of the related software products in reference architectures. A severely reduced version of the solution architecture metamodel is shown in Figure 2.5.

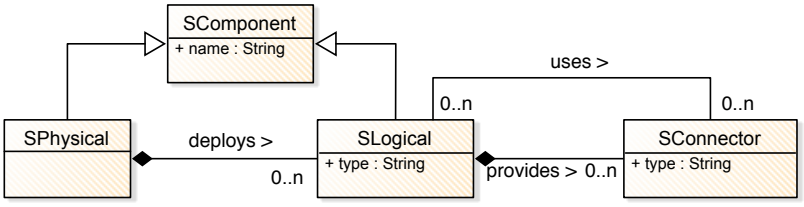

Figure 2.5.: Solution architecture metamodel

A solution architecture is built of components (SComponent) and connectors (SConnector). A component can either a physical component (SPhysical) or a logical component (SLogical). A logical component is the configuration of a software product that should be deployed to a physical component, which is specified by the deploys association. A physical component is just a placeholder for a concrete physical machine that is available on-site at the customer premises.

A logical component can further provide connectors, denoted by the provides association, and also use different connectors, denoted by the uses association. The attribute type is used to relate logical components and connectors from solution architectures and logical components and connectors in reference architectures, respectively. Thus, whenever the attribute type of an SLogical is equal to the attribute name of an RLogical, an implicit dependency exists because the SLogical configures the RLogical.

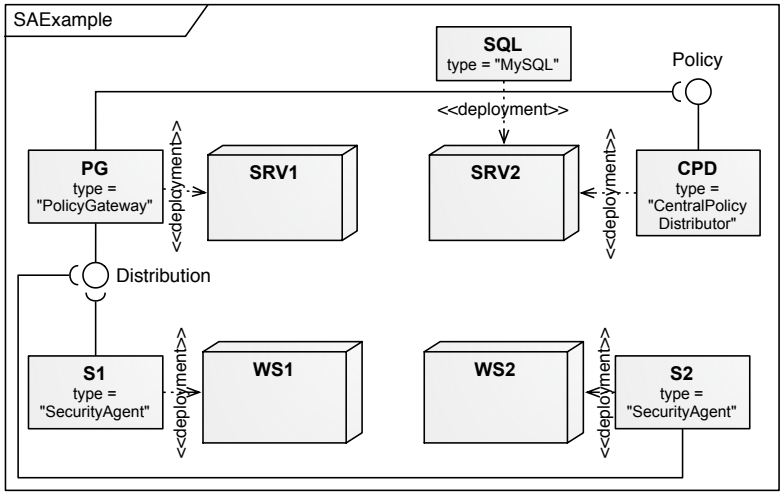

Figure 2.6.: Exemplary solution architecture deploying the security product

Figure 2.6 shows a simple solution architecture that conforms to the metamodel of Figure 2.5. The solution architecture is named SAExample and consists of four logical components (denoted as two-dimensional rectangles), which are instances of SLogical, four representations of physical components (denoted as three-dimensional rectangles), which are instances of SPhysical, and two connectors (denoted as circles), which are instances of SConnector.

This example shows a partial configuration of the software products defined in the reference architecture of Figure 2.6. It configures a PolicyGateway (PG), two SecurityAgents (S1 and S2), and a CentralPolicyDistributor (CPD). The CPD provides a connector called Policy that is used by PG, which further provides the connector Distribution that is used by S1 and S2. The SecurityAgents S1 and S2 should observe the physical machines called WS1 and WS2, respectively, where they should be deployed. S1 and S2 get their security policies from PG through the connector Distribution. PG itself receives its security policies from CPD via the connector Policy. PG and CPD should be deployed on different physical machines called SRV2 and SRV1.

An IT infrastructure is a concrete physical infrastructure reflecting the physical IT of a customer. It is used as a deployment target of solution architectures. For this purpose a metamodel has been defined, shown in a severely reduced version in Figure 2.7.

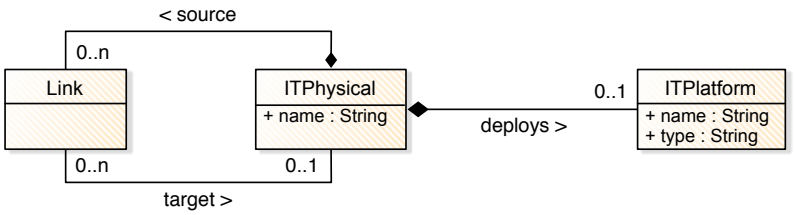

Figure 2.7.: IT infrastructure metamodel

An IT infrastructure is built of physical components (ITPhysical), platforms (ITPlatform) and links (Link) between physical components. A physical component is linked with other physical components by means of a Link using the source and target associations. A platform can be deployed on a physical component

by means of the deploys association. A platform has an attribute type, which is used to identify mappings between instances of RPlatform specified in reference architectures and instances of ITPlatform specified in IT infrastructures. In this simplified version of this case study, it is assumed that a physical component can only host a single platform.

Furthermore, the deployment of logical components in solution architectures is defined by individual implicit mappings between physical components in solution architectures and physical components in IT infrastructures. For simplification, it is defined that such a mapping exists whenever the value of the attribute name of these components are equal. The deployment of logical components to physical components in solution architectures defines that a deployment should exist. Whenever a mapping of the related instance of SPhysical to an instance of ITIPhysical exists, a deployment exists because there is a target to which it can be deployed in the IT infrastructure. All logical components in solution architectures that are thus deployed to physical components in IT infrastructures are meant to be executed on a platform that is deployed on the same physical components in the IT infrastructure.

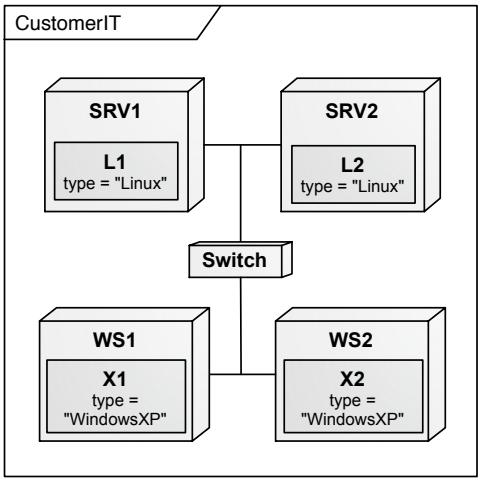

Figure 2.8.: Exemplary customer IT infrastructure

Figure 2.8 shows a simple example of an ITI of a hypothetical customer. The IT infrastructure named CustomerIT consists of five physical components (denoted as three-dimensional rectangles), which are instances of ITPhysical, and four platforms (rectangles nested into physical components), which are instances of ITPlatform. The links between physical machines are denoted as simple connections in between. The example shows a simple IT infrastructure with physical components SRV1, SRV2, WS1, WS2 and Switch and platforms L1, L2, X1, X2 and the deployment of these platforms on physical components. Thus, the hypothetical customer owns these five physical devices and their respective deployed platforms.

One of the challenges in this case study is to explicitly capture the dependencies between reference architectures and solution architectures, and between solution architectures and IT infrastructures. Due to the loose coupling of these modeling languages, dependencies only implicitly exist (soft references). For example, a logical component in a solution architecture may configure a logical component in a reference architecture, defined by their attributes. Furthermore, solution architectures as well as reference architectures can get quite large and changes occur frequently. Thus, automated maintenance of dependencies is required.

Additionally, if logical components or connectors in solution architectures are mapped to logical components or connectors in reference architectures, certain well-formedness constraints should be analyzed. This is also achieved by means of capturing and automatically maintaining dependencies.

2.2.2. UML Software Development

This case study is aligned to the case study that is shown in [93]. It uses UML to generate java code that is subsequently compiled. Furthermore, UML is also used to generate a schema used to generate SQL code, which can be used to initialize a relational database management system (RDBMS).

Because the UML metamodel is too complex for illustration purposes, a simplified UML metamodel is provided, as shown in [124]. The shown UML metamodel only supports modeling simplified UML class diagrams, which is shown in Figure 2.9.

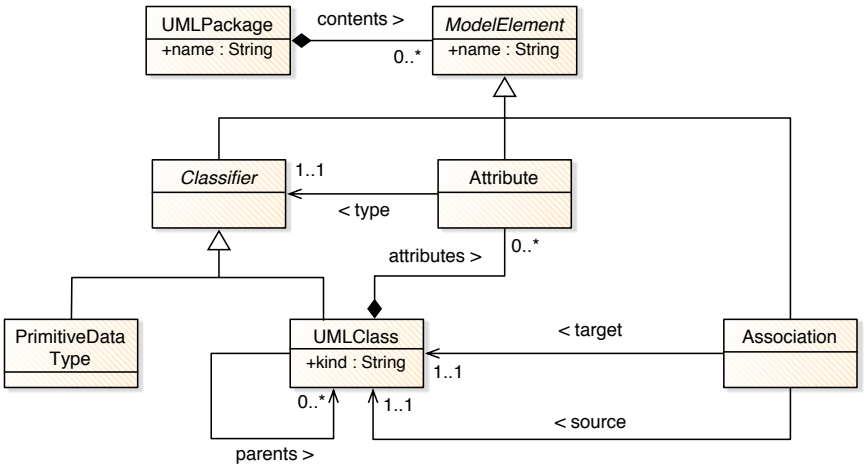

Figure 2.9.: Simplified UML metamodel for UML class diagrams

The simplified UML metamodel consists of a UML package (UMLPackage) that contains model elements (ModelElement). Model elements are either classifiers (Classifier), attributes (Attribute) or associations (Association). Furthermore, a classifier can either be a primitive data type (PrimitiveDataType) or a UML class (UMLClass). Primitive data types are, e.g., strings, integers, floats, doubles, etc. A UML class can extend multiple other UML classes (parents association) and is related to a set of attributes (attributes association). An attribute has an explicit type, which is a classifier. An attribute is considered as complex if the type is a UML class. It is considered as primitive if the type is a primitive data type. The association is connecting two UML classes via source and target associations.

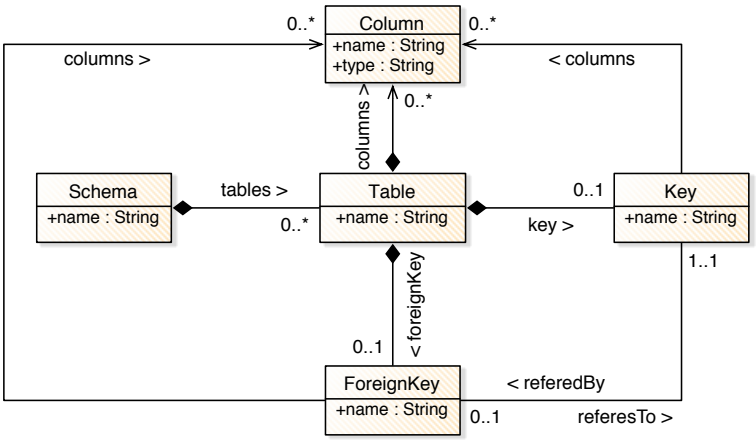

Figure 2.10.: Schema metamodel

The metamodel, which is shown on Figure 2.10, is a simple schema metamodel for RDBMS. The metamodel is aligned to the schema metamodel as shown in [124]. A schema (Schema) has a set of tables (Table). Each table has a set of columns (Column), a key (Key) and a foreign key (ForeignKey). A key and a foreign key may refer to a set of columns (columns association) and a foreign key may refer to a key (refersTo association).

There are various reasons for introducing this case study. Firstly, it comprises several model operations that have to be applied to generate Java code as well as SQL code. From a high-level perspective, these

model operations interact because generating SQL code requires transforming the UML model into a schema model. Secondly, the study is used to demonstrate how the shown approach is going to specify compositions of multiple model operations and how to (re-)apply them accordingly. Based on the model transformation from a simplified UML model to a schema model, which is shown in [124], a similar model transformation is built from fine-grained model operations. This will show the capabilities of the approach in building complex model operations from individual and fine-grained model operations.

2.2.3. Embedded Systems Development

This study is concerned with a scenario in which a complex and legacy model transformation had to be extended by additional functionality. It is an industrial case study in cooperation with dSPACE[9] where a tool chain has been developed, which transforms SysML[10] models into AUTOSAR[11] models within Eclipse and further transforms these AUTOSAR models into SystemDesk[12] conform AUTOSAR models (see [66]).

Because of the new timing extension of AUTOSAR (see [21]), textual SysML timing requirements must now be transformed into structural AUTOSAR latency timing constraints. Since re-implementing a complete model transformation in another model transformation language is time-consuming, error-prone and expensive, the model transformation should be extended without changing the original implementation. Thus, the extension should be seamlessly composed into the existing model transformation.

SysML, which reuses a subset of UML, is used in systems engineering for developing embedded systems. Thereby, in the automotive domain, parts of these SysML models, which are important for software engineering, are subsequently refined in AUTOSAR for further software development. Thus, automotive companies that develop the overall system architecture or subsystems using SysML need to transform relevant parts of those models into AUTOSAR models.

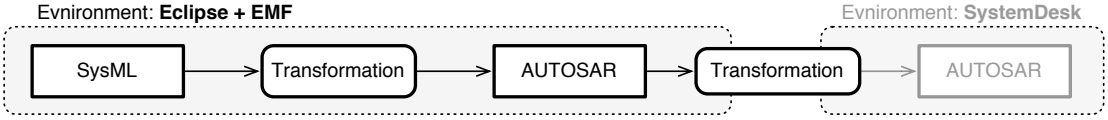

Figure 2.11.: Complete tool chain of the case study

The developed tool chain is shown in Figure 2.11. The SysML and AUTOSAR models are provided in an EMF compatible representation. The model transformation between these models is realized by means of a TGG transformation. A second model transformation bridges the technological gap between Eclipse and EMF and the tool SystemDesk from dSPACE.

In the current release of AUTOSAR, defining timing constraints becomes possible by a recently developed timing extension. Thus, the model transformation between SysML and AUTOSAR had to be extended, such that timing requirements specified in SysML models are also transformed into timing constraints of AUTOSAR models. Such AUTOSAR timing constraints are an inherent part of architectural models. Since much effort was invested in the development of the actual model transformation between SysML and AUTOSAR models, it should be reused for this purpose.

However, because the requirements in SysML are formulated textually, they cannot be handled by the model transformation implemented using TGGs, because it is a structural model transformation language that does not support text parsing. Thus, the model transformation had to be extended by another technology or language for handling the transformation of textual requirements without enhancing the existing transformation technology or language. Therefore, a separate and independent model transformation has been implemented, which is implemented in plain Java because Java is well suited for text parsing. This additional model transformation only transforms individual requirements of SysML models into latency timing constraints of AUTOSAR models.

Figure 2.12 shows a simple example, which is the software architecture of a fuel system controller as

[9] http://www.dspace.com
[10] http://www.sysml.org/specs.htm
[11] AUTOSAR 4.0; http://autosar.org/
[12] SystemDesk is an environment to model embedded systems from dSPACE.

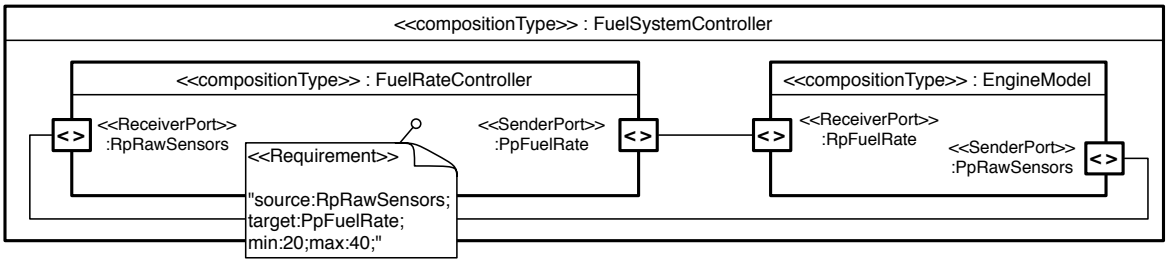

Figure 2.12.: Software architecture of the fuel system controller

an SysML model including an SysML requirement.[13] The FuelRateController computes the fuel injection rate, which depends on the current speed of the engine. The engine is represented through the EngineModel. Due to the real-time requirements of an automotive engine, the FuelRateController is subject to specific timing requirements that are reflected in form of the shown SysML requirement.

This thesis will be show how the approach can be employed to extend this legacy model transformation by means of another model transformation without needing to change the original model transformation implementation.

2.3. Summary

This chapter has introduced the basics of MDE as models, metamodels and model operations. It has been explained that models are used as abstract representations of systems. Furthermore, we have introduced the idea of precisely specified metamodels being able to leverage the usage of model operations in MDE, which is one of its building blocks for increasing the level of automation.

In addition, two techniques of supporting the practicability of MDE have been introduced, namely, traceability and model management. Traceability has been introduced as a technique to capture dependencies between all kinds of artifacts that are necessary for the understanding of a collection of artifacts. It has also been mentioned that traceability is a foundation for various other techniques like impact analysis, change propagation, etc.

Model management is promising in the context of MDE because it can be applied to ease MDE in various ways. The term global model management has been recently introduced by Bézivin et al. and is used to manage a collection of related models from a global perspective using megamodels. A megamodel can be used to define global relationships between models or to apply model operations between models, for example. State-of-the-art has been shown for both techniques. Furthermore, four case studies have been introduced, which are employed to explain and evaluate the concepts that are shown in this thesis. The case studies are from different domains, e.g., software configuration to embedded system development. In each case study a specific application scenario is highlighted that is addressed by the concepts under investigation.

The approach demonstrated by this thesis leverages the configuration of MDE. This is realized by encouraging the decomposition of heterogeneous model operations by providing means to specify compositions of decomposed model operations without needing to change the implementation of the model operations. The approach provides the data-flow as well as the context composition of model operations as explained in Section 2.1.5. The approach supports the application of MDE by providing techniques to automatically maintain the existence of dependencies and to automate the (re-)application of heterogeneous model operations. This is obtained based on the specifications that are provided when configuring MDE using this approach.

[13] This is an example shipped with SystemDesk.

3. Hierarchical Megamodels

Contents

* * *

This chapter provides an approach to explicitly capture all kinds of dependencies between artifacts in MDE applications by means of hierarchical megamodels. In addition, this chapter provides the foundation for the specification of compositions of heterogeneous model operations.

3.1. Conceptual Introduction

As already discussed in the previous chapter, a megamodel can be considered as a homogeneous and global view on a collection of heterogeneous artifacts and their dependencies, which describe a snapshot of the implementation of a software system.[1] A megamodel is a homogeneous view on a collection of heterogeneous artifacts because for each heterogeneous artifact it only captures an abstract representation that removed any detail that is not necessary for further management purposes (e.g., attributions). Thus, for any heterogeneous artifact, an homogeneous representation is provided by the megamodel. A megamodel is a global view because it only represents high-level artifacts, which are models in its entirety.

Due to the abstraction, which a megamodel provides, it is possible to explicitly capture dependencies between any kinds of artifacts by capturing them between representations of these artifacts in the megamodel. Thus, dependencies can be captured in a uniform way for any kind of artifact. Furthermore, the abstraction of megamodels supports model management activities on a heterogeneous set of artifacts in a uniform way by applying them directly on a megamodel or using information from a megamodel, i.e. navigation, impact analysis, etc.

The hierarchical megamodel is an extension in several dimensions of the core idea of the megamodel introduced by Bézivin et al. Generally, the hierarchical megamodel is a homogeneous but *progressive view* on a collection of heterogeneous artifacts. The hierarchical megamodel is a progressive view on a collection of heterogeneous artifacts because it captures artifacts at any level of detail including explicitly captured hierarchy relationships lying inbetween. In addition, the hierarchical megamodel captures heterogeneous dependencies between a set of heterogeneous artifacts in a homogeneous way even at different levels of detail including explicit hierarchy relationships.

The hierarchical megamodel further distinguishes between artifacts and dependencies that either act as types or instances, the necessity for which will be clear in later chapters of this discussion. Thus, the hierarchical megamodel is split up into an application megamodel, an idea is similar to the original notion of megamodels from Bézivin et al. except the notion of hierarchy, and a configuration megamodel,

[1] This thesis basically considers the megamodel of Bézivin et al. as the foundation (see [29] and [23]).

which only contains artifacts and dependencies that act as types for artifacts and dependencies in an application megamodel.

3.1.1. Representation of MDE Configurations and MDE Applications

The hierarchical megamodel is defined by two separate megamodels – the configuration megamodel and the application megamodel – as illustrated in Figure 3.1. The figure shows that everything that is going to be represented by a hierarchical megamodel is further declared to be the physical level while the hierarchical megamodel is declared to be the logical level.

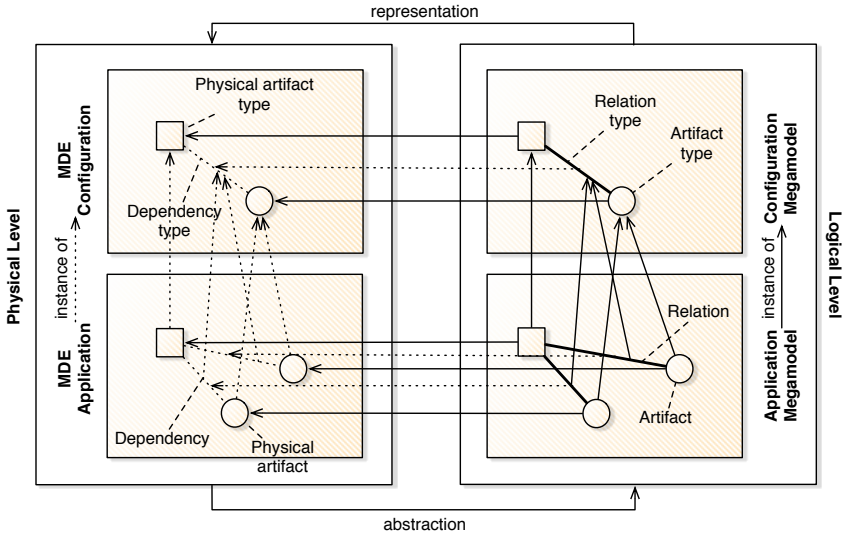

Figure 3.1.: Abstract representation of MDE configurations and applications

An MDE configuration is considered as a set of physical artifacts and implicitly existing physical dependencies similar to an MDE application (e.g., a metamodel is a model which metamodel is a metametamodel). Thus, an MDE configuration could also be represented by means of an application megamodel.

Nevertheless, the goal of the hierarchical megamodel is to support an application developer by providing traceability and model management facilities. Thus, the application megamodel only represents an MDE application from an instance perspective while the configuration megamodel is considered as a complement to the application megamodel by representing an MDE configuration from a type perspective. These two perspectives are used to provide enhanced model management facilities to an application developer, as will be shown in this thesis.

Thus, an MDE configuration is considered as a collection of physical artifact types, which are metamodels that act as types or models, and physical dependency types that interconnect physical artifact types.[2] An MDE application is considered as a collection of physical artifacts, which are primarily models that are instances of metamodels in MDE configurations, and physical dependencies that interconnect physical artifacts.[3]

A physical dependency type is defined to be any kind of dependency between physical artifact types. For example, if instances of two metamodels can depend on each other because of some reason, a physical dependency type should exist between these two metamodels including a description or specification of the reason.

A physical dependency is defined to be any kind of dependency between physical artifacts. For example, if two models depend on each other, a physical dependency should exist between these two models. A physical dependency is always an instance of a physical dependency type. This instantiation relationship declares that a physical dependency is a concrete occurrence of a certain physical dependency type.

[2]Model operations are considered in subsequent chapters.
[3]Physical artifacts can also be folders, files, documents etc., which is explained in Section A.2.3.2.

Thus, a physical dependency type can be considered as the class or semantic of a collection of physical dependencies that instantiate that type.

A configuration megamodel is defined to be a collection of artifact types, including the relation types which obtain between those artifact types. An artifact type is an abstract representation of a physical artifact type, which is explicitly captured by means of a representation relationship in between. A relation type captures and either explicitly or implicitly represents an existing physical dependency type by interconnecting a set of artifact types that represent physical artifact types connected by the physical dependency type. A relation type is an abstract representation of a physical dependency type because it only interconnects their representations (artifact types). The representation relationship between relation types and physical dependency types is not explicitly captured because it cannot be assumed that physical dependency types always explicitly exist. For example, in the case where two independently specified metamodels depend on each other, a physical dependency type should exist between these metamodels. However, this is usually not the case because they have been specified independently. Therefore, the representation relationship is rather encoded as a combination of capturing the physical dependency type and a description of the semantic of the physical dependency type.

An application megamodel is defined to be a collection of artifacts and relations in between those artifacts. An artifact is an abstract representation of a physical artifact, which is explicitly captured by means of a representation relationship in between. If a physical artifact is an instance of a physical artifact type, then the representation of the physical artifact (artifact) is an instance of the representation of the physical artifact type (artifact type).

A relation captures and represents an explicitly or implicitly existing physical dependency by interconnecting artifacts that represent physical artifacts connected by the physical dependency. A relation is an abstract representation of a physical dependency because it only interconnects representations of physical artifacts. The representation relationship between relations and physical dependencies also only implicitly exists because physical dependencies may only implicitly exist. Thus, the representation relationship between relations and physical dependencies is rather a combination of what the relation interconnects and the type of the relation, which is given by the instantiation relationship between relations and relation types. A relation is always an instance of a relation type, if the represented physical dependency is an instance of a dependency type that is represented by the relation type.

The representation relationship is always a one-to-one mapping and supports identifying the physical counterpart of some artifact type, artifact, relation or relation type (also the other way around). This makes the hierarchical megamodel valuable because it allows the results of reasoning on the hierarchical megamodel to be mapped back to the actual physical level, and it allows querying information directly on the physical artifacts that use information from the hierarchical megamodel, e.g., whether two physical artifacts depend on each other.

3.1.2. Representation of Hierarchy

In general, physical artifacts can be considered as being hierarchically structured, which is not surprising since hierarchy is a common method for reducing complexity [154]. For example, a model hosts a collection of model elements, where each model element may be the parent of further model elements. The hierarchical megamodel is able to capture hierarchical structures that exist at physical level. However, hierarchy is considered differently in a configuration megamodel compared with in an application megamodel due to their different perspectives.

The configuration megamodel captures and represents capabilities for building hierarchical structures in an MDE application, while the application megamodel captures and represents actual hierarchical structures in an MDE application concerning their capabilities as specified in an MDE configuration. Figure 3.2 illustrates the idea of hierarchy in hierarchical megamodels.[4]

An application megamodel can represent physical artifacts at any level of detail (e.g., models and model elements) and is not restricted to representing high-level physical artifacts (models). Furthermore, an application megamodel explicitly captures the hierarchy relationships between physical artifacts. In this thesis, hierarchy between physical artifacts is considered to be an existential relationship that distinguishes between superior physical artifacts influencing the life-cycle of subordinate physical artifacts. That is to say, a model is superior to its model elements, which are subordinates. If the model is going to

[4]Instantiation relationships are not visualized and representation relationships are only partially visualized for reasons of readability.

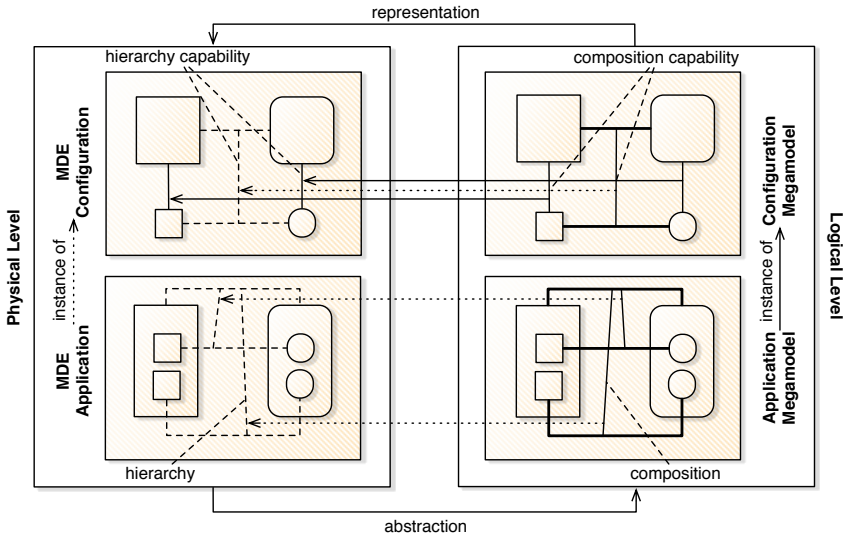

Figure 3.2.: Abstract representation of hierarchy in hierarchical megamodels

be deleted, all its model elements will be deleted, too. Such a semantic of hierarchy is well known and is also considered as composition, e.g., composition association in UML [128]. The abstract representation of hierarchy in an MDE application is further considered as composition.

A configuration megamodel is not intended to capture the hierarchical structure of physical artifact types in the same way as is captured in application megamodels for physical artifacts. A configuration megamodel rather captures the capabilities of defining hierarchy between physical artifacts in MDE applications. Thus, if two physical artifact types are in a relationship, which enables one instance to be subordinate or superior to the other instance, this relationship will be explicitly captured between the representing artifact types. This is further considered as a composition capability.

The representation relationship between composition capabilities, hierarchy capabilities of artifact types, and physical artifact types is captured explicitly because it is assumed that physical artifact types have explicit concepts for defining hierarchy capabilities (e.g., composition associations). The representation relationship between compositions, artifact hierarchies, and physical artifacts is also explicitly captured because it is assumed that physical artifacts instantiate the concepts for defining hierarchy capabilities to build hierarchy. Thus, also the composition between artifacts is considered as an explicit instance of a composition capability between artifact types.

Furthermore, physical dependencies may be structured hierarchically, which is considered to be an existential relationship between dependencies, defining that a subordinate physical dependency depends on the existence of a superior physical dependency. Thus, the semantic of hierarchy between physical dependencies is considered to be conceptually similar to the semantic of hierarchy between physical artifacts. Hierarchy between physical dependencies is explicitly captured by means of composition relationships between relations.

A configuration megamodel does not capture hierarchy between physical dependency types, but rather captures the capabilities of defining hierarchy between physical dependencies. Thus, if two physical dependency types somehow define a relationship, which enables one instance being subordinate to another instance, this relationship between physical dependency types will be explicitly captured between the representing relation types. This is also called composition capability but in reference to that between relation types.

The representation relationship between compositions, composition capabilities of physical dependencies, and physical dependency types is implicitly captured because physical dependencies and physical dependency types may only exist implicitly.

3.1.3. Overview

The introduced hierarchical megamodel has to be specified by a configuration developer in order for an application developer to use it appropriately. Figure 3.3 shows the use cases of the employed hierarchical megamodel.

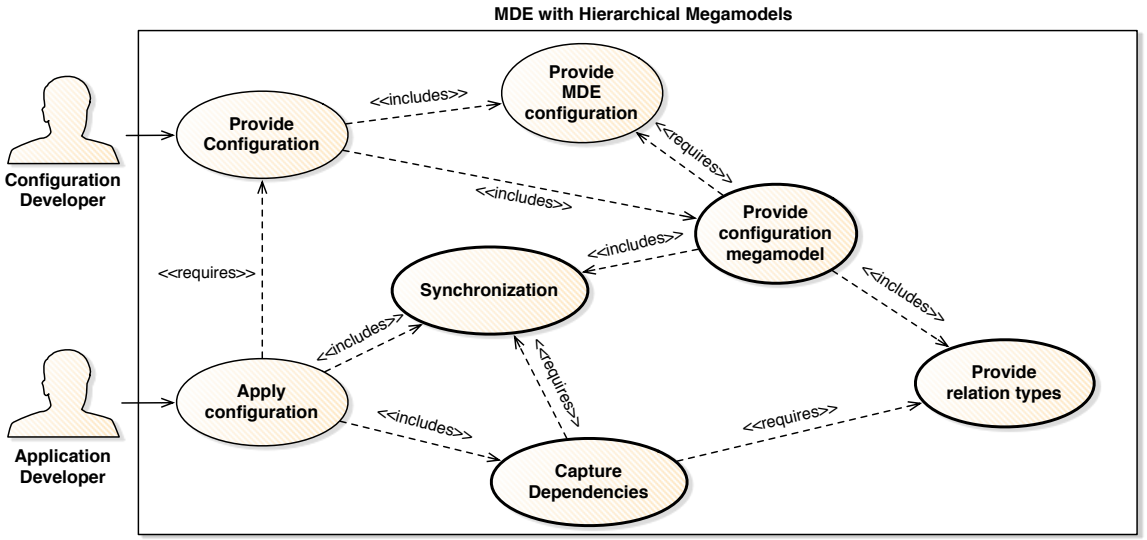

Figure 3.3.: Use cases for using hierarchical megamodels and applying the synchronization

As mentioned in the introduction of this thesis, a configuration developer is responsible for providing a configuration, which now implies the provision of an MDE configuration and a configuration megamodel for that MDE configuration. Providing a configuration megamodel further includes providing relation types and the application of a synchronization. The synchronization automatically creates artifact types in a configuration megamodel from physical artifact types in an MDE configuration. Thus, an MDE configuration comes with a configuration megamodel that provides additional information that is further required for traceability and model management.

Based on the configuration, an application developer applies the provided configuration to define an MDE application. In addition, an application developer can capture dependencies in an application megamodel based on a given configuration megamodel. Capturing dependencies further requires the application of a synchronization, which automatically keeps the artifacts of the application megamodel in sync with physical artifacts in an MDE application.

The remainder of this chapter will define in more detail the concept of the hierarchical megamodel. The synchronization concept will additionally be introduced, automatically providing a consistent view on an MDE configuration and an MDE application. The synchronization is considered as a substantial component of the model management framework, which is required because model management facilities can only operate correctly with a consistent view. Figure 3.4 illustrates these concepts, that will be shown in this chapter, and how they relate to one another.

As part of the model management framework, first the hierarchical megamodel will be defined (see Section 3.2). This comprises the definition of a configuration megamodel and an application megamodel, which realize the hierarchical megamodel, as explained previously. Because physical artifacts and physical artifact types change, especially in MDE applications, the artifact types and artifacts have to be updated accordingly in order to be a consistent representation. This is automatically obtained by synchronization, located between an MDE configuration, an MDE application and a hierarchical megamodel (see Section 3.3).

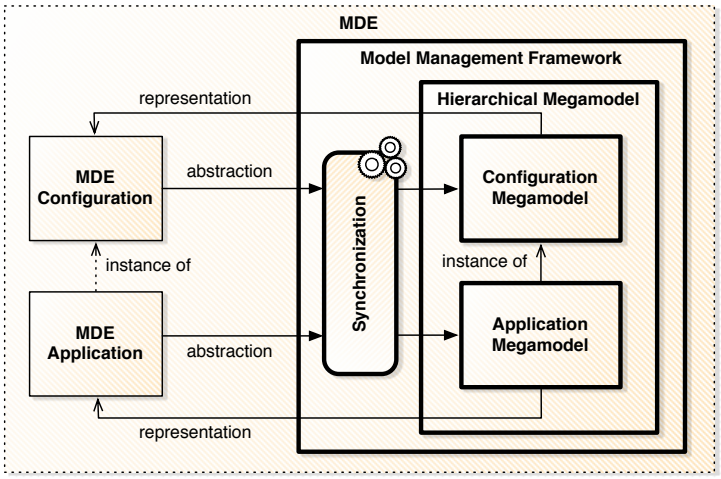

Figure 3.4.: Conceptual integration of hierarchical megamodels

3.2. Hierarchical Megamodels

The hierarchical megamodel is separately explained and defined by first introducing the configuration megamodel (see Section 3.2.1) and then the application megamodel (see Section 3.2.2). The instantiation relationship between them is shown together with the application megamodel.

The configuration megamodel as well as the application megamodel are both explained and defined by means of a metamodel, which explains the structure of configuration and application megamodels, respectively. For all necessary concepts a concrete syntax is defined. Based on the concrete syntax, application examples from the case studies are shown. Finally, formal definitions and additional constraints on these definitions are provided for configuration and application megamodels, respectively.

3.2.1. Configuration Megamodels

A configuration megamodel represents an MDE configuration from a type perspective and thus defines the capabilities of an application megamodel. The primary concepts of a configuration megamodel are shown in the metamodel of Figure 3.5.

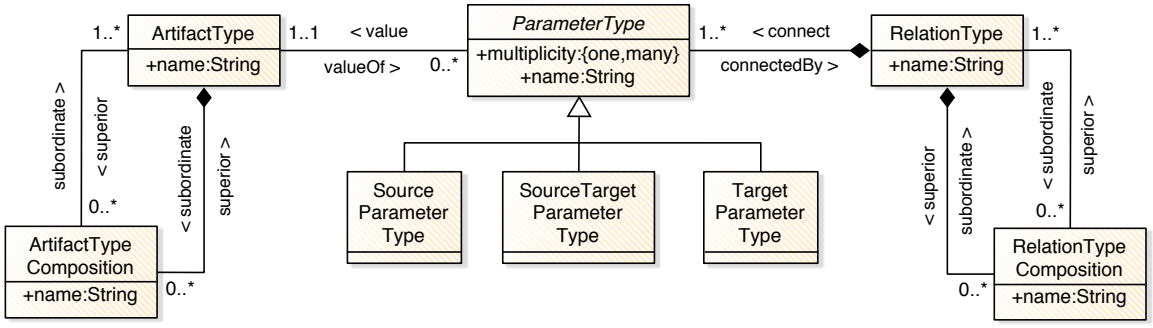

Figure 3.5.: Metamodel of the configuration megamodel

Each class of this metamodel represents a specific concept of the configuration megamodel, and its associations define relationships between these concepts. These concepts are artifact type (ArtifactType), parameter type (ParameterType) and relation type (RelationType). The specializations of ParameterType are used to indicate a specific direction of parameter types and therefore relation types. The hierarchy capabilities of a configuration megamodel are defined by means of the two concepts artifact type composition (ArtifactTypeComposition) and relation type composition (RelationTypeComposition).

3.2.1.1. Artifact Types and Relation Types

An artifact type is an abstract representation of exactly one physical artifact type in an MDE configuration (e.g., metamodel or metamodel element). The representation relationship is captured explicitly but is not shown because it depends on the kind of physical artifact type that is represented. A relation type captures and abstractly represents any physical dependency type that might implicitly or explicitly exist between physical artifact types. Thus, a relation type defines the semantic or a class of physical dependencies.

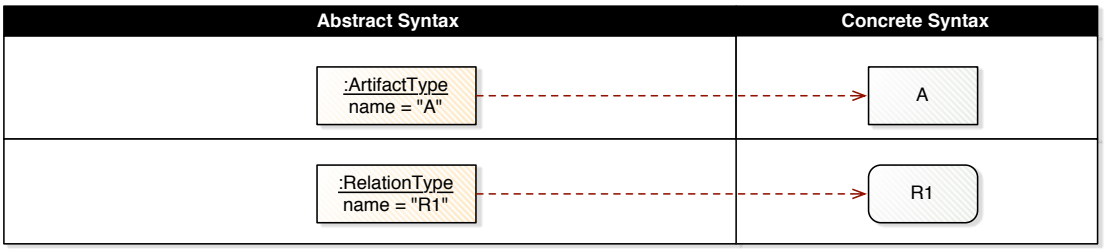

Figure 3.6.: Concrete syntax of artifact types and relation types

Figure 3.6 shows the concrete syntax of an artifact type and a relation type by mapping the abstract syntax of a configuration megamodel to elements shown as concrete syntax.[5] The figure visualizes an artifact type by means of a rectangle while a relation type is visualized by means of a rounded rectangle. The name of an artifact type and a relation type (defined by the name attributes) are labels within the rectangles.

Furthermore, a relation type is defined to be an n-ary connection between artifact types (with $n > 0$), realized by means of parameter types. A parameter type is employed to complement a relation type because it acts as connector between a relation type and an artifact type. A relation type can be connected to a set of parameter types with each parameter type using exactly one artifact type as its value. A parameter type is also employed to qualify the connection between a relation type and an artifact type. Using a parameter type as connector is required because a relation type can be connected to the same artifact type multiple times but with a different meaning. Thus, a parameter type supports distinguishing between similar artifact types connected to the same relation type.

A parameter type also provides the concept of multiplicities. A parameter type can have two different multiplicities, which are *one* and *many*. The multiplicity is employed to define how many artifacts of a certain artifact type can be connected to an instance of this relation type. A multiplicity of one means exactly one artifact and many means exactly one artifact or more than one artifact.

Additionally, a relation type is considered as being directed. The direction of a relation type is realized by means of parameter types, which are connected to the relation type, because a parameter type defines a specific direction. The direction of a parameter type is defined by means of the specializations of ParameterType. A source parameter type (SourceParameterType) indicates that an artifact type of the parameter type is considered as a source of the connected relation type. A source & target parameter type (SourceTargetParameterType) indicates that the artifact type of the parameter type is considered as a source and target of the connected relation type. Finally, a target parameter type (TargetParameterType) indicates that the artifact type of the parameter type is considered as target of the connected relation type.

Figure 3.7 shows the concrete syntax of a relation type in combination with parameter types and artifact types. It shows a relation type R1 that is connected to four parameter types, which are visualized by means of double-headed arrows. A source parameter type is directed from an artifact type to a relation type, a source & target parameter type is headed in both directions, and a target parameter type is directed from a relation type to an artifact type.

The figure also shows that the target parameter type of R1 has a multiplicity of many, which is denoted by means of a plus (+) symbol. Thus, a set of instances of artifact type A can be connected to an instance of the relation type R1 by means of that parameter type. The shown source parameter type has a name

[5] The abstract syntax of a configuration megamodel is shown in the UML object diagram notation.

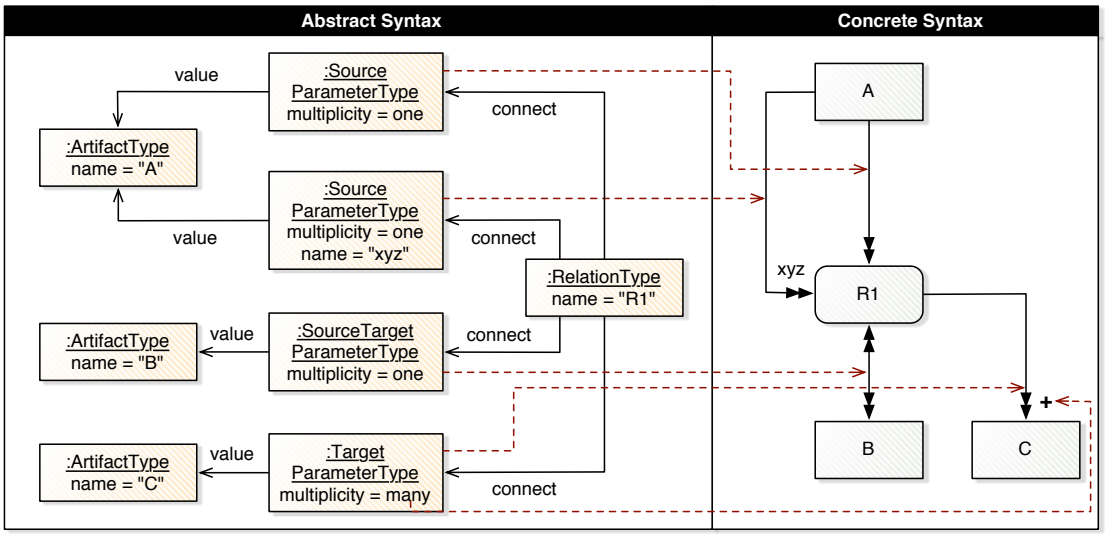

Figure 3.7.: Concrete syntax of parameter types

set (xyz), which is optional. The name is visualized as label beside the double-headed arrow. The name is used to explicitly distinguish between parameter types that have similar artifact types and relation types.

The relation type R1 has two source parameter types with both having the artifact type A as their value. This means that an instance of R1 must be connected to exactly two instances of the artifact type A (these can be similar instances as in case of artifact type A).

Figure 3.8.: Partial view on a configuration megamodel from the D-MDA case study (high-level)

Figure 3.8 shows a cutout of a configuration megamodel from the D-MDA case study shown in Section 2.2.1. The configuration megamodel is shown by means of the concrete syntax.[6] The configuration megamodel shows abstract representations of the metamodels (ReferenceArchitecture, SolutionArchitecture and ITInfrastructure) and dependency types that can indicate overlaps between instances of these metamodels.

The relation type SADependsOnRA is connected to the artifact type SolutionArchitecture and the artifact type ReferenceArchitecture. The multiplicity of the parameter type between SADependsOnRA and ReferenceArchitecture is set to the value many. Thus, this relation type defines that logical components in a solution architecture can be configurations of logical components in a reference architecture.

The figure shows another relation type called SADeployedToITI, which also takes the artifact type SolutionArchitecture as source and the artifact type ITInfrastructure as target. This relation type defines that a solution architecture can be deployed on an IT infrastructure, which means that logical components of a solution architecture can be deployed on physical components of that IT infrastructure.

3.2.1.2. Hierarchical Artifact Types

Hierarchy between physical artifacts is defined by means of their connecting existential dependencies (composition), and hierarchy between physical artifact types is considered to be the capability of hierarchy between physical artifacts (composition capability). These composition capabilities between artifact types are specified by means of artifact type compositions. Thus, an artifact type composition specifies

[6] It only shows high-level artifact types and high-level dependency types.

a superior/subordinate relationship between a set of artifact types. An artifact type composition has exactly one artifact type as superior, and a non-empty set of artifact types as subordinates. This means that the superior artifact type is also superior to all subordinate artifact types of the artifact composition type. Additionally, it means that all artifact types that are subordinate to the artifact type composition are also subordinate to the artifact type that is superior to the artifact type composition.

An artifact type can be subordinate to a set of artifact type compositions, which means that an instance of that artifact type can be composed into different artifacts (not simultaneously). Furthermore, an artifact type can have a set of artifact type compositions as subordinates, which means that it can distinguish between different types of compositions. Finally, an artifact type is responsible for an artifact type composition that defines the subordinates of that artifact type, which is defined by the composition association (subordinate) between ArtifactType and ArtifactTypeComposition. This design decision has been made because it corresponds to the semantic of a composition association.

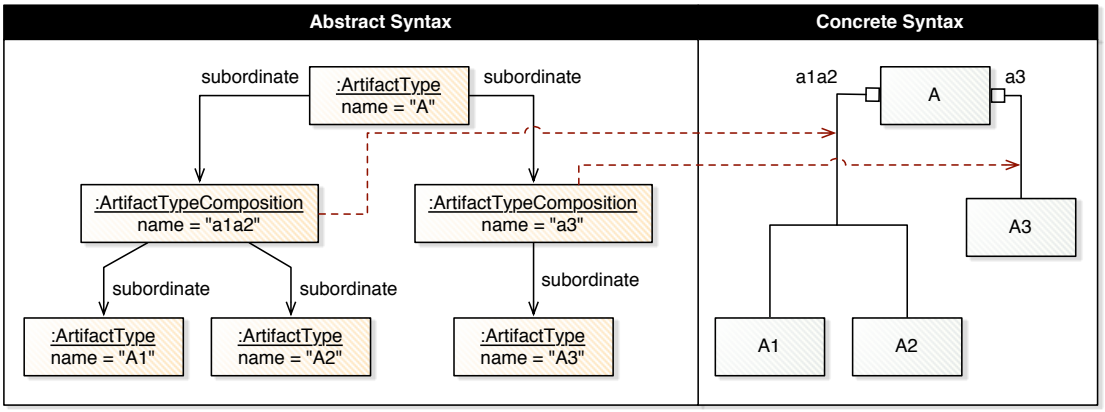

Figure 3.9.: Concrete syntax of artifact type compositions

Figure 3.9 shows the concrete syntax of artifact type compositions. The figure shows four artifact types A, A1, A2 and A3, whereby A1, A2 and A3 are defined to be subordinate to A. This is realized by two artifact type compositions. An artifact type composition is visualized by means of connections with a square on the side of the artifact type that is superior to the artifact type composition. The name of the artifact type composition is visualized as a label that is shown beside the square of the connection.

An artifact type composition allows for the qualification of the composition capability between artifact types. This is necessary because physical artifact types may have various different hierarchy capabilities, defined as shown in Figure 3.10.

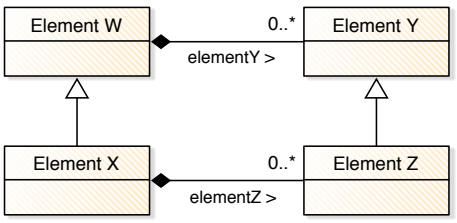

Figure 3.10.: Exemplary metamodel for illustrating the hierarchy capabilities

The figure shows a simple metamodel with four classes Element W; Element X that is a specialization of Element W; Element Y; and Element Z that is a specialization of Element Y. Element W is connected via a composition association (elementY) to Element Y and Element X is connected via a composition association (elementZ) to Element Z. Representing this metamodel in a configuration megamodel results in five artifact types as shown in Figure 3.11.

The artifact type Metamodel represents the metamodel as a whole, and the other four artifact types represent the four classes (metamodel elements) of the metamodel. The metamodel itself has the capa-

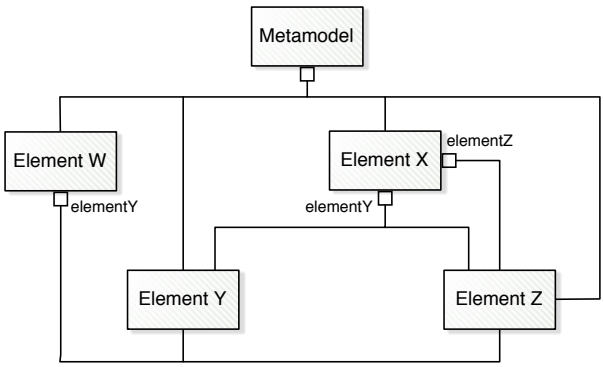

Figure 3.11.: Representation of the example metamodel

bility to be superior to all classes in the metamodel; this is represented by an artifact type composition with artifact type Metamodel as superior, and all other four artifact types as subordinates.

Element W has the capability to be superior to Element Y and Element Z because Element Z is directly related via the composition association (elementY). The same association is indirectly related to Element Z because it is a specialization of Element Y. Thus, artifact type Element W has an artifact type composition named elementY, which represents the composition association elementY. This artifact type composition has the artifact type Element W as superior, and the artifact types Element Y and Element Z as subordinates, in order to correctly represent the capabilities of the composition association.

The artifact type Element X has two different artifact type compositions. One represents the composition association elementY and the other one the composition association elementZ; this is because the class Element X is a specialization of the class Element W, and therefore also inherits its associations. Thus, instances of Element X can compose instances of Element Y and instances of Element Y and Element Z in two different ways. It is further entailed that artifact type Element X has two different artifact type compositions (elementY and elementZ).

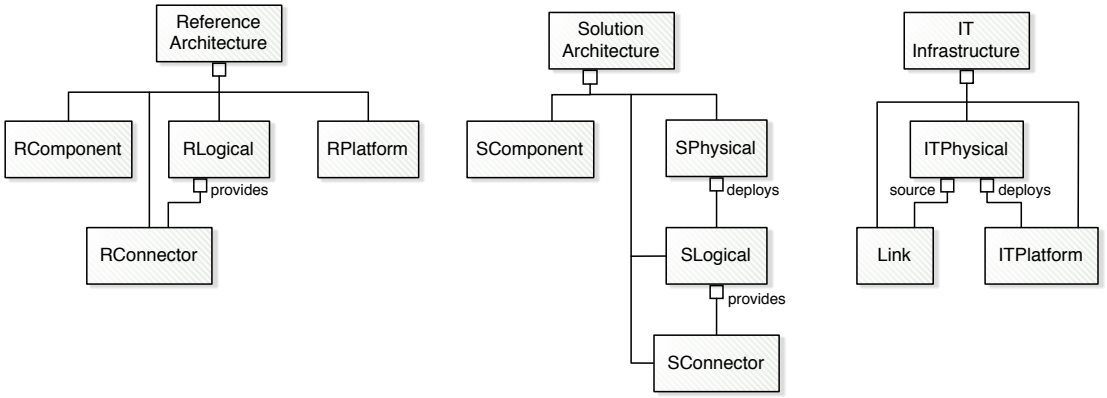

Figure 3.12.: Representation of metamodels from the D-MDA case study

Figure 3.12 shows a configuration megamodel that represents all physical artifact types from an MDE configuration of the first case study, including artifact type compositions.

The top-most artifact types represent the metamodels from the case study. Each metamodel is capable of composing all of its metamodel elements, which are defined by the artifact type compositions from the metamodel representations to all representations of the metamodel elements of the according metamodels. Instances of the artifact type RLogical are capable of composing instances of the artifact type RConnector because the representation of RLogical has a direct composition association to the representation of RConnector. The name of the artifact type composition is similar to the name of the association (provides). The artifact type SPhysical is connected to the artifact type SLogical via an artifact type composition

called deployed and SLogical is further connected to SConnector via an artifact type composition called provides. The artifact type ITPhysical is connected to two artifact types Link and ITPlatform via two different artifact type compositions named source and deploys because the representation of ITPhysical has two different composition associations.

3.2.1.3. Hierarchical Relation Types

Hierarchy between physical dependencies is defined by means of the existential dependencies between them(composition), and hierarchy between physical dependency types is considered to be the capability of building hierarchical structures between physical dependencies (composition capability). These composition capabilities between relation types are specified by means of relation type compositions.

Thus, a relation type composition specifies a superior/subordinate relationship between a set of relation types. A relation type composition has exactly one relation type as subordinate and a non-empty set of relation types as superiors. This means that the subordinate relation type is also subordinate to all superior relation types of the relation type composition. Additionally, it means that all relation types that are superior to the relation type composition are also superior to the relation type that is subordinate to the relation type composition.

A relation type can be subordinate to a set of relation type compositions, which means that an instance of that relation type can be composed into various relations (even simultaneously). In comparison to the artifact type composition, a relation type uses relation type compositions to define superior relation types, whereas an artifact type uses artifact type compositions to define subordinate artifact types. Thus, each artifact type is responsible for defining the capability of compositing its subordinates, which couples the superiors to the subordinates. This is legitimate in the case of artifact types because the composition capabilities of physical artifact types are almost predefined by means of their representations (metamodels).

In case of relation types however, superior relation types should not be coupled to subordinate relation types because it makes them less reusable and flexible. Inverting the composition direction provides more flexibility because superiors are no longer responsible for defining the capability of composing subordinates, but rather subordinates are responsible for defining their composition capability on their own. Thus, superior relation types are decoupled from subordinate relation types.

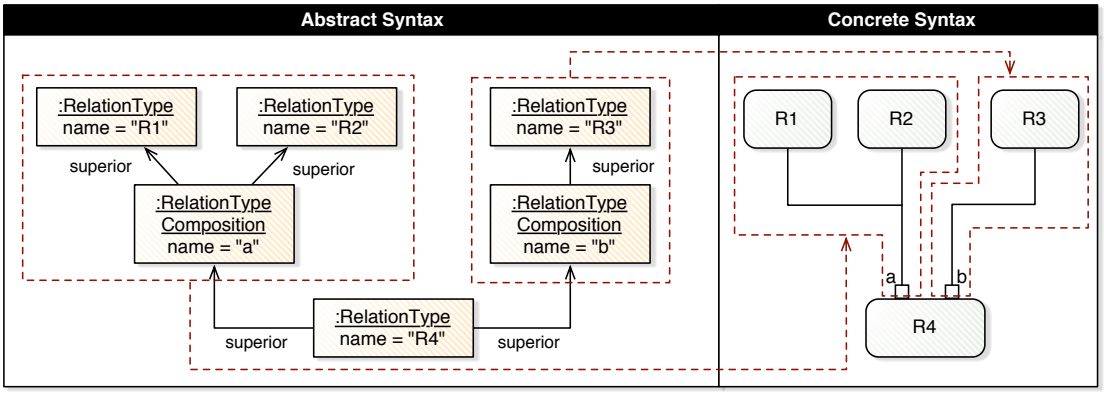

Figure 3.13.: Concrete syntax of relation type compositions

Figure 3.13 shows the concrete syntax of relation type compositions.[7] A relation type composition is visualized by means of a connection between a relation type that is subordinate to the relation type composition and a set of relation types that are superior to the relation type composition. The connection has a square on the side of the relation type, which is subordinate to the relation type composition. Furthermore, the optional name of the relation type composition is shown as a label beside the square.

The figure shows a relation type R4, which has two relation type compositions named a and b as superior. The relation type composition a has the relation type R1 and R2 as superiors whereas the

[7] Artifact types and parameter types are not shown due to readability.

relation type composition b has relation type R3 as superior. Thus, instances of relation type R4 require an instance of R1 and R2 or an instance of R3 as context for their own composition.

By now, a relation type composition does not imply that a relation type, which is superior to another relation type, is connected to artifact types that are higher-level than artifact types that are connected to the subordinate relation type. Both situations are possible and indeed useful in certain application scenarios. This way, a relation type composition may define two different kinds of relation type compositions, which are bottom-up and top-down. This is not explicitly defined by any concept but rather implicitly defined by the constellation of artifact types, artifact type compositions, relation types and relation type compositions.

3.2.1.3.1. Bottom-Up Relation Type Composition A relation type composition is declared to be bottom-up if all relation types that are superior to another relation type are connected to artifact types that are higher-level than or similar to artifact types that are connected to the subordinate relation type.

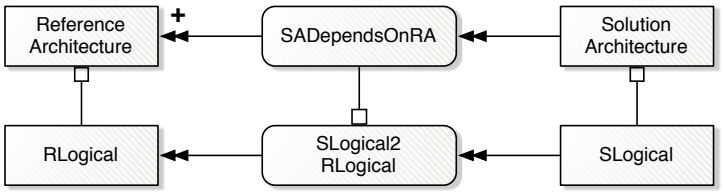

Figure 3.14.: Specification of a relation type SLogical2RLogical composed into SADependsOnRA

Figure 3.14 shows a relation type from the case study, which defines the relation type SADependsOnRA in more detail. The relation type SLogical2RLogical is defined between the artifact types SLogical and RLogical, whereby SLogical is the source and RLogical is the target of the relation type. This relation type defines that an artifact type SLogical depends on an artifact type RLogical because the representation of SLogical can use the representation of RLogical explicitly as type.

The relation type SLogical2RLogical has the relation type SADependsOnRA as superior relation type, which means that instances of SLogical2RLogical require the existence of an SADependsOnRA for their own existence. Thus, SLogical2RLogical defines a more detailed type of dependency than SADependsOnRA because it is used to capture dependencies between SLogical and RLogical artifacts, which are constituents of a SolutionArchitecture artifact and a ReferenceArchitecture artifact, respectively.

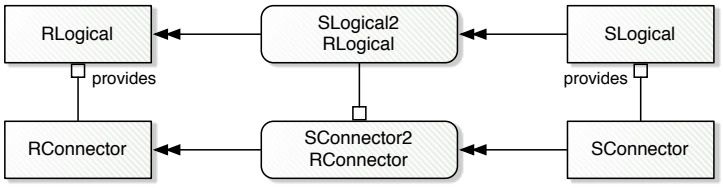

Figure 3.15.: Specification of a relation type SConnector2RConnector composed into SLogical2RLogical

Figure 3.15 shows another bottom-up relation type composition of a relation type called SConnector2RConnector. This relation type has an artifact type SConnector as source and an artifact type RConnector as target. The relation type indicates that an instance of the artifact type SConnector can depend on an instance of the artifact type RConnector, in the case of RConnector acting as an explicit type for SConnector.

The relation type composition defines that SConnector2RConnector uses the previously shown relation type SLogical2RLogical as superior relation type. Thus, an artifact SConnector only depends on an artifact RConnector in the context of an artifact SLogical and an artifact RLogical which depend on each other, indicated by a relation of type SLogical2RLogical. Therefore, artifacts of a relation type SConnector2RConnector must interconnect artifacts of type SConnector and artifacts of type RConnector if they depend on each other and if their context (SLogical and RLogical) depends on each other.

3.2.1.3.2. Top-Down Relation Type Composition A relation type composition is declared to be top-down if all relation types that are superior to another relation type are connected to artifact types which are lower-level than or similar to artifact types that are connected to the subordinate relation type.

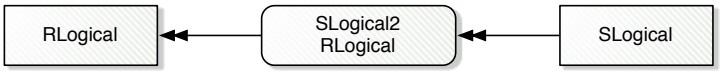

Figure 3.16.: Alternative specification of SLogical2RLogical without composition

For example, Figure 3.16 shows another version of the relation type SLogical2RLogical that was previously shown in Figure 3.14. This time SLogical2RLogical is defined without having any relation type composition, which means that instances of SLogical2RLogical can exist independently of any other relation.

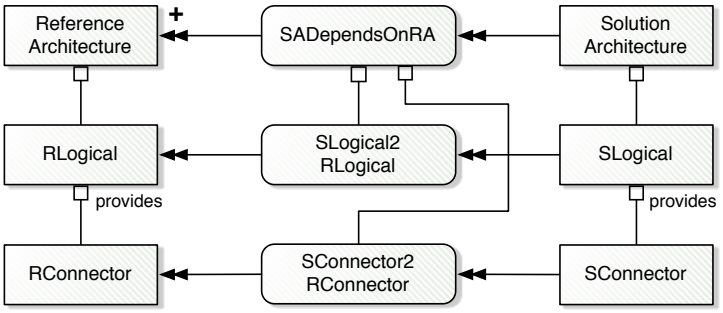

Figure 3.17.: Alternative specification of SADependsOnRA composed into SLogical2RLogical or SConnector2RConnector

Figure 3.17 shows a revision of the relation type SADependsOnRA. This time SADepndsOnRA defines two different relation type compositions. They define that instances of SADependsOnRA require an instance of SLogical2RLogical, or an instance of SConnector2RConnector, for their own existence. Thus, SLogical2RLogical and SConnector2RConnector define a condition for the existence of instances of SADependsOnRA.

Both relation type compositions are declared to be top-down because SADependsOnRA is connected to artifact types that are higher-level than artifact types connected to SLogical2RLogical and SConnector2RConnector.

3.2.1.4. Formal Definitions and Constraints

The configuration megamodel concepts introduced here will now be formally defined, for their subsequent use in this thesis. Furthermore, the behavior of the main concepts of this thesis are defined by relying on these formal definitions. The definitions specify when a configuration megamodel is a conforming representation of an MDE configuration and when a configuration megamodel is considered to be well-formed.

First of all, an MDE configuration is formally defined as shown in Definition 3.2.1.

3.2.1 Definition (MDE Configuration) An MDE configuration is a tuple (A_{C_P}, O_{C_P}) where A_{C_P} is a set of physical artifact types that can be instantiated in MDE applications, and O_{C_P} is a set of model operations that can be applied in MDE applications.

The configuration megamodel, which has been shown in Figure 3.5, is now formally defined. The formal definition of the configuration megamodel is shown in Definition 3.2.2 and provides all necessary concepts and relationships between them. In general, the associations of metamodels are formally defined by means of mapping functions. The names of these functions correspond to the names of the associations. Sub classification and attributions of classes are also defined by mapping functions.

3.2.2 Definition (Configuration Megamodel) Given an MDE configuration (A_{C_P}, O_{C_P}), a configuration megamodel M_C is a 5-tuple $(A_t, A_{C_t}, P_t, R_t, R_{C_t})$ where A_t is a finite set of artifact types, A_{C_t} is a finite set of artifact type compositions, P_t is a finite set of parameter types, R_t is a finite set of relation types, and R_{C_t} is a finite set of relation type compositions.

- The relationships between artifact types and other concepts of the configuration megamodel are defined by the following mapping functions:
  - $represent_{A_t} : A_t \to A_{C_P}$ maps every artifact type to exactly one physical artifact type that declares to be represented by the artifact type, and $abstract_{A_{C_P}} : A_{C_P} \to A_t \cup \{\epsilon\}$ maps every physical artifact type to at most one artifact type that declares to be the abstract representation of the physical artifact type.
  - $sub_{A_t} : A_t \to \mathcal{P}(A_{C_t})$ defines the subordinate association from ArtifactType to ArtifactTypeComposition and maps every artifact type to a set of artifact type compositions with each artifact type composition of the set declared to be subordinate to the artifact type. $sup_{A_{C_t}} : A_{C_t} \to A_t$ defines the superior association from ArtifactTypeComposition to ArtifactType and maps every artifact type composition to exactly one artifact type that is declared to be superior to the artifact type composition.
  - $sup_{A_t} : A_t \to \mathcal{P}(A_{C_t})$ defines the superior association from ArtifactType to ArtifactTypeComposition and maps every artifact type to a set of artifact type compositions with each artifact type composition of the set declared to be superior to the artifact type. $sub_{A_{C_t}} : A_{C_t} \to \mathcal{P}(A_t) \setminus \{\emptyset\}$ defines the subordinate association from ArtifactTypeComposition to ArtifactType and maps every artifact type composition to a non-empty set of artifact types with each artifact type of the set declared to be subordinate to the artifact type composition.
- The relationships between relation types and other concepts of the configuration megamodel are defined by the following mapping functions:
  - $connect_{R_t} : R_t \to \mathcal{P}(P_t) \setminus \{\emptyset\}$ defines the connected association between RelationType and ParameterType and maps every relation type to a non-empty set of parameter types with each parameter type of the set declared to be connected by the relation type. $connect_{P_t} : P_t \to R_t$ defines the connectedBy association between ParameterType and RelationType and maps every parameter type to a exactly one relation type that is declared to be connected to the parameter type.
  - $sub_{R_t} : R_t \to \mathcal{P}(R_{C_t})$ defines the subordinate association from RelationType to RelationTypeComposition and maps every relation type to a set of relation type compositions with each relation type composition of the set declared to be subordinate to the relation type. $sup_{R_{C_t}} : R_{C_t} \to \mathcal{P}(R_t) \setminus \{\emptyset\}$ defines the superior association from RelationTypeComposition to RelationType and maps every relation type composition to a non-empty set of relation types with each relation type of the set declared to be superior to the relation type composition.
  - $sup_{R_t} : R_t \to \mathcal{P}(R_{C_t})$ defines the superior association from RelationType to RelationTypeComposition and maps every relation type to a set of relation type compositions with each relation type composition of the set declared to be superior to the relation type. $sub_{R_{C_t}} : R_{C_t} \to R_t$ defines the subordinate association from RelationTypeComposition to RelationType and maps every relation type composition to exactly one relation type that is declared to be subordinate to the relation type composition.
- Additional concepts of parameter types and relationships between parameter types and other concepts of the configuration megamodel are defined by the following mapping functions:
  - $val_{P_t} : P_t \to A_t$ defines the value association from ParameterType to ArtifactType and maps every parameter type to exactly one artifact type that is declared to be the value of the parameter type. $val_{A_t} : A_t \to \mathcal{P}(P_t)$ defines the valueOf association from ArtifactType to ParameterType and maps every artifact type to a set of parameter types with each parameter type declared to have the artifact type as value.
  - $dir_{P_t} : P_t \to \{S, ST, T\}$ defines the direction of a parameter type and maps every parameter type to exactly one of the directions S (source), ST (source & target) or T (target).

- $multi_{P_t} : P_t \rightarrow \{1, \star\}$ defines the multiplicity attribute of ParameterType and maps every parameter type to either 1, if the parameter type has a one multiplicity, or $\star$, if the parameter type has a many multiplicity.

The formal definition of the configuration megamodel provides five sets, with each set representing a specific class of the metamodel shown in Figure 3.5. Thus, instances of a certain class are elements in the according set, e.g., instances of the class ArtifactType are elements in A_t.

A configuration megamodel is defined to be an abstract representation of an MDE configuration. Therefore, it must comply with certain conformance conditions, which are defined in Definition 3.2.3.

3.2.3 Definition (Conform MDE Configuration Representation) A configuration megamodel $M_C = (A_t, A_{C_t}, P_t, R_t, R_{C_t})$ is a conform representation of an MDE configuration (A_{C_P}, O_{C_P}) if the following conditions are satisfied:

- Every artifact type is a unique representation of a physical artifact type, which is defined as $\forall a_t, a'_t \in A_t : a_t \neq a'_t \Rightarrow represent_{A_t}(a_t) \neq represent_{A_t}(a'_t)$.
- Every artifact type is a conform representation of a physical artifact type, which is defined as $\forall a_t, a'_t \in A_t, \exists a_{C_P}, a'_{C_P} \in A_{C_P} : a_{C_P} = represent_{A_t}(a_t) \wedge a'_{C_P} = represent_{A_t}(a'_t) \wedge a'_{C_P}$ is subordinate to $a_{C_P} \Rightarrow a'_t \in subs_{A_t}(a_t)$. Thus, every hierarchy capability relationship between physical artifact types are captured by means of artifact type compositions between abstractions of these physical artifact types.
- Every relation type $r_t \in R_t$ is a unique and conform representation of a physical dependency type. In addition, if between a set of relation types a relation type composition exists, a hierarchy capability relationship must exist between a set of physical dependency types that are represented by the set of relation types.

Thus, a configuration megamodel is a conform representation of an MDE configuration, if it only provides unique and conform representations of physical artifact types and physical dependency types. However, the conform representation of relation types is not formally defined because no formal definition of physical dependency types is provided in an MDE configuration. This is because physical dependency types may only implicitly exist, e.g., in the minds of configuration developers or language providers.

The metamodel and the formal definition of the configuration megamodel do not provide any further restrictions. Because a configuration developer manually maintains relation types, further restrictions to the specification of relation types must be provided. This is necessary because the specification of a relation type has to follow certain well-formedness conditions. This is defined for the configuration megamodel as shown in Definition 3.2.4.

3.2.4 Definition (Well-Formed Configuration Megamodel) A configuration megamodel $M_C = (A_t, A_{C_t}, P_t, R_t, R_{C_t})$ is well-formed if the following condition is satisfied:

- Every relation type $r_t \in R_t$ that has a parameter type p_t with a many multiplicity connected, cannot be connected to any other parameter type with similar artifact type as value, which is defined as $\forall r_t \in R_t, p'_t \in connect_{R_t}(r_t), \exists p_t \in connect_{R_t}(r_t) : p_t \neq p'_t \wedge multi_{P_t}(p_t) = \star \Rightarrow val_{P_t}(p_t) \neq val_{P_t}(p'_t)$.

This restriction is necessary for further automation purposes as will be introduced in subsequent chapters.

3.2.2. Application Megamodels

The second part of the hierarchical megamodel is the application megamodel. An application megamodel represents an MDE application from an instance perspective. The metamodel of the application megamodel is shown in Figure 3.18.

Every class of the metamodel represents a concept of the application megamodel. The associations are used to define specific relationships between these concepts,which are artifact (Artifact), parameter (Parameter) and relation (Relation). The hierarchy relationships between physical artifacts and physical

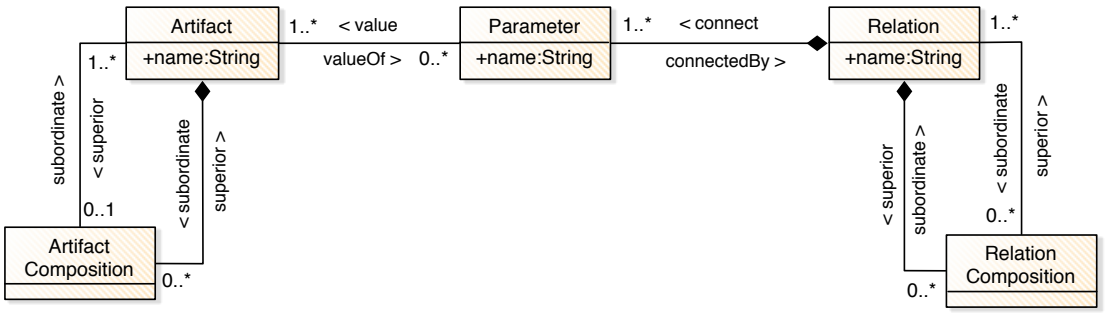

Figure 3.18.: Metamodel of the application megamodel

dependencies are represented by means of the concepts artifact composition (ArtifactComposition) and relation composition (RelationComposition), respectively.

In the conceptual introduction it was mentioned that an application megamodel is considered as an instance of a configuration megamodel. This is expressed by explicit instantiation relationships between the corresponding concepts of the application megamodel and the individual concepts of the configuration megamodel, which is shown by an extension of both metamodels in Figure 3.19.[8]

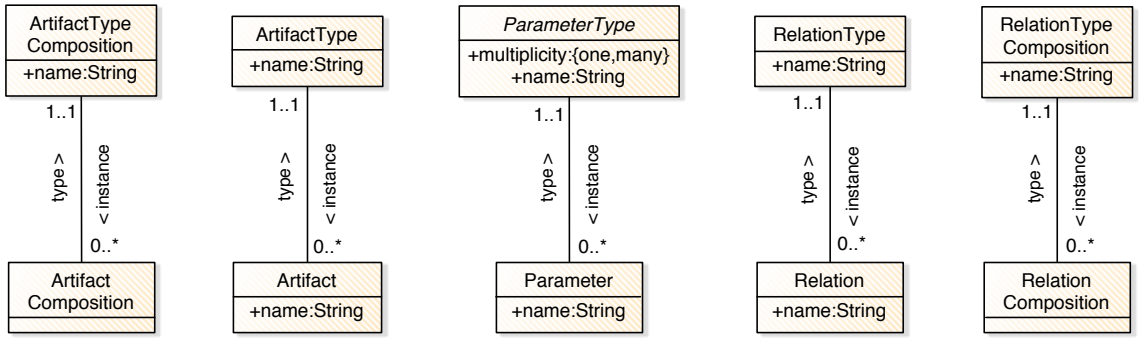

Figure 3.19.: Instantiation relationships between configuration and application megamodel concepts

The metamodel shows that an artifact is an instance of an artifact type, a relation is an instance of a relation type, a parameter is an instance of a parameter type, an artifact composition is an instance of an artifact type composition, and a relation composition is an instance of a relation type composition. By implication, any concept of the configuration megamodel acts as type of a corresponding concept in the application megamodel.

Every concept (artifact, artifact composition, parameter, relation, relation composition) in an application megamodel is always an instance of exactly one corresponding concept in a configuration megamodel (artifact type, artifact type composition, parameter type, relation type or relation type composition). For example, an artifact is always an instance of an artifact type. Conversely, a concept in a configuration megamodel can be the type of multiple concepts in an application megamodel, e.g., an artifact type can be instantiated by multiple artifacts.

3.2.2.1. Artifacts and Relations

An artifact is an abstract representation of exactly one physical artifact in an MDE application (e.g., model or model element). An artifact is always an instance of an artifact type if the representation of the artifact is physically an instance of the artifact type's representation. The representation relationship between an artifact and a physical artifact is not explicitly shown in the metamodel because it also depends on the kind of physical artifact that is represented.

[8]The metamodel does not show relationships that were already introduced.

A relation is explicitly captured and represents exactly one physical dependency in an MDE application (e.g., overlap between model elements). A relation is always an instance of a relation type, which explicitly defines the type of the relation.

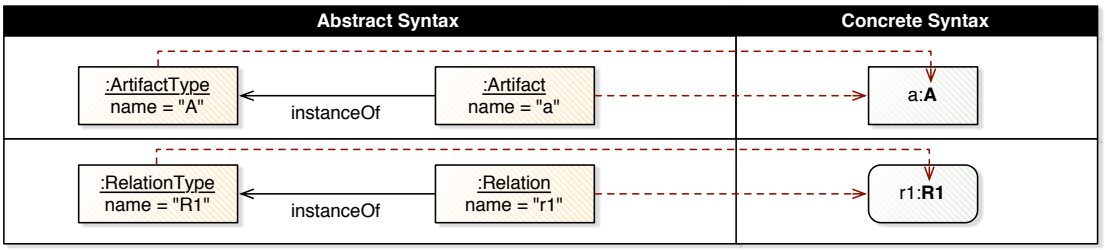

Figure 3.20.: Concrete syntax of artifacts and relations

Figure 3.20 shows the concrete syntax of an artifact and a relation. An artifact is visualized by means of a rectangle while a relation is depicted as a rounded rectangle. Furthermore, because an artifact and a relation is always an instance of an artifact type and a relation type, respectively, their type is shown as part of the concrete syntax. Thus, the name of an artifact and a relation also contains the name of their type separated by a colon.

A parameter is similar to a parameter type. Thus, it is employed as connector between a relation and a set of artifacts. In comparison to a parameter type, a parameter has at least one (but potentially many) artifacts as value because a parameter is considered as an instance of a parameter type. This is due to a potential many multiplicity of the instantiated parameter type. A parameter must have at least one artifact as value because a connector that is not connecting something is meaningless. A relation must be connected to at least one parameter because the relation may represent n-ary dependencies (with $n > 0$). The direction of a parameter is not set by the parameter itself, but is defined by the parameter type it instantiates. Thus, if the type of a parameter is a source parameter type, the parameter is defined to be a source parameter.

A relation always has an artifact context, which is defined as a set of artifacts that are connected to the relation via parameters.

Figure 3.21 illustrates the concrete syntax of a relation r1 of type R1 (the relation type R1 has been shown in Figure 3.7), which is connected to parameters and artifacts. The relation is connected to four parameters, where each is an instance of a parameter type connected to R1. The parameters are visualized similarly to parameter types by means of double-headed arrows. The directions of parameters depend on the direction of their types. Furthermore, the names of parameters are defined by their types.

The shown target parameter has two artifacts c1 and c2 of type C as values. This is legitimate because the type of this parameter has a multiplicity of many.

An example is shown Figure 3.22. The application megamodel represents three models from the D-MDA case study's application example, which are a security product (SecurityProduct), a solution architecture example (SAExample) and a customer IT (CustomerIT). This application megamodel is considered as an instance of the configuration megamodel that is shown in Figure 3.8.

The application megamodel represents two physical dependencies by means of relations. The relation sa2ra is an instance of the relation type SADependsOnRA, and thus denotes that the solution architecture SAExample comprises logical components that depend on logical components from the reference architecture SecurityProduct. The relation sa2it is an instance of the relation type SADeployedToITI, which implies that logical components of the solution architecture SAExample are defined to be deployed to physical components that are part of the IT infrastructure CustomerIT.

SADependsOnRA and SADeployedToITI are both uni-directional relation types. This direction is chosen because the decisions whether dependencies of that type exist are made in solution architectures by specifying logical components accordingly.

3.2.2.2. Hierarchical Artifacts

Hierarchy between physical artifacts is defined by means of the hierarchical relationships which exist between them (composition). A hierarchy relationship between physical artifacts is explicitly captured

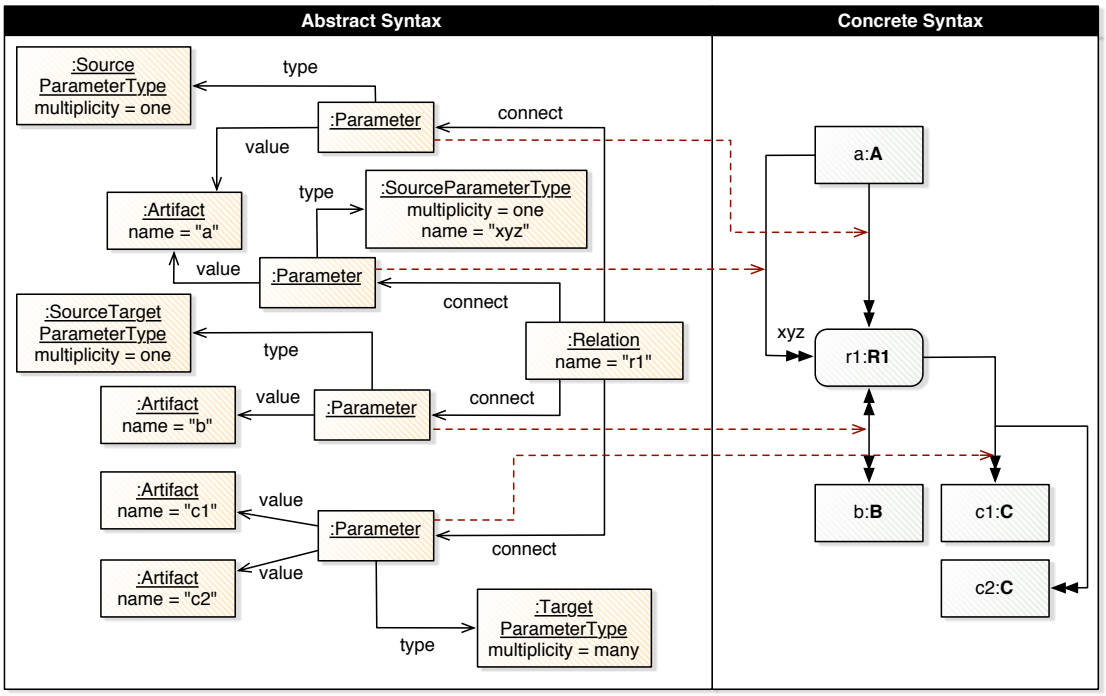

Figure 3.21.: Concrete syntax of parameters

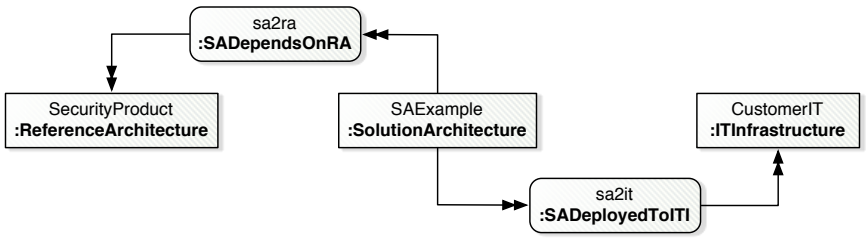

Figure 3.22.: Partial view on an application megamodel from the D-MDA case study (high-level)

by means of an artifact composition that is defined between artifacts representing these physical artifacts.

An artifact composition is always an instance of an artifact type composition, which defines the capabilities of building hierarchies between instances of artifact types related to the artifact type composition. A physical artifact can only be composed into one other physical artifact simultaneously. Thus, an artifact has at most one artifact composition as superior, which itself has exactly one artifact as superior. On the other side, a physical artifact can compose multiple other physical artifacts. Thus, an artifact composition may have multiple artifacts as subordinates. Furthermore, a physical artifact may have to compose multiple artifacts, but by means of different kinds of compositions. Thus, an artifact may have multiple artifact compositions as subordinates.

Considering artifacts as vertices of a graph and artifact compositions as edges of a graph, the resulting graph is always acyclic because artifacts can only be subordinate to one other artifact. Thus, an artifact cannot compose itself but only other artifacts of the same type.

Figure 3.23 illustrates the concrete syntax for artifact composition. The figure shows four artifacts `a`, `a1`, `a2` and `a3` of types `A`, `A1`, `A2` and `A3` (the artifact type compositions of these artifact types are shown in Figure 3.9). Furthermore, two artifact compositions are shown which are visualized by means of connections with a black square on the side of the artifact, which is responsible for the artifact composition (superior). Thus, `a` composes `a1` and `a2` by an artifact composition of type `a1a2`, and `a3` by an artifact composition of type `a3`.

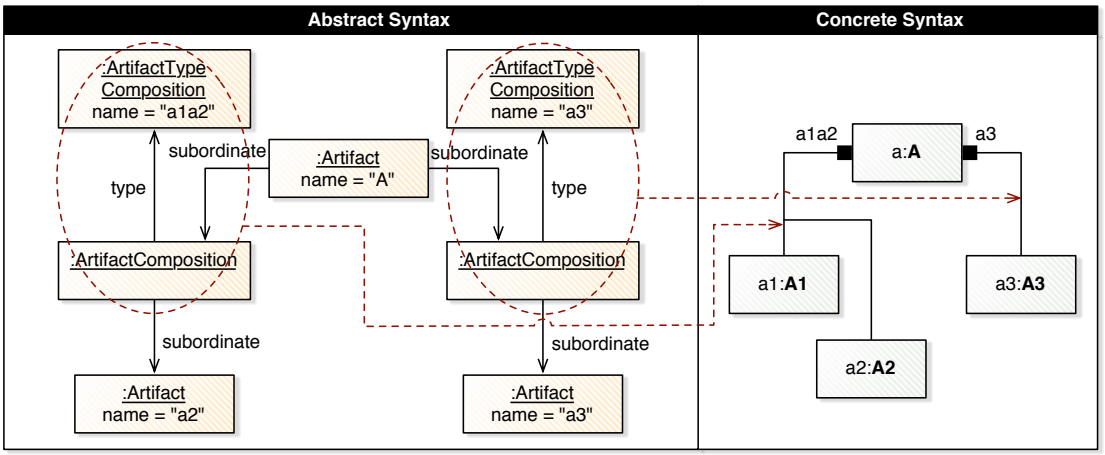

Figure 3.23.: Concrete syntax of artifact compositions

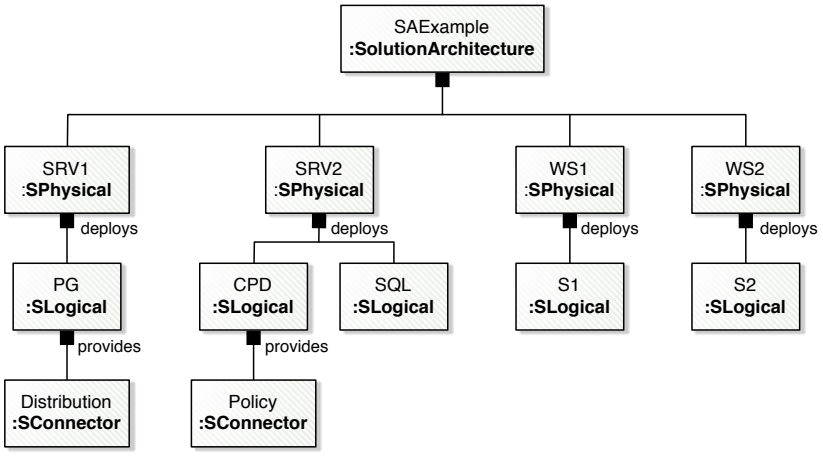

Figure 3.24.: Representation of physical artifacts including artifact compositions from the D-MDA case study

Figure 3.24 shows an artifact SAExample, which represents the solution architecture SAExample, and a set of artifacts that are direct or indirect subordinates of SAExample (this is only a subset of all artifacts). Direct subordinates of the artifact SAExample are the artifacts SRV1, SRV2, WS1 and WS2. The artifacts PG, CPD, SQL, S1 and S2 are composed either by SRV1, SRV2, WS1 or WS2 because they are defined to be deployed to them. Furthermore, the artifacts PG and CPD compose the artifacts Distribution and Policy, respectively, because artifacts of type SConnector are composed by artifacts of type SLogical using artifact compositions of type provides.

3.2.2.3. Hierarchical Relations

Hierarchy between physical dependencies is defined by means of the hierarchy relationships between them (composition), which is explicitly represented by means of relation compositions. A relation composition specifies a superior/subordinate relationship between a set of relations. A relation composition has exactly one relation as subordinate and a non-empty set of relations as superiors. This means that the subordinate relation is also subordinate to all superior relations of the relation composition. It also means that all relations that are superior to the relation composition are also superior to the relation that is subordinate to the relation composition.

A relation uses relation compositions to define superior relations whereas an artifact uses artifact

compositions to define subordinate artifacts. This composition direction between relations is chosen because it is similar to the relation type composition direction between relation types.

A relation might have a composition context. The composition context of a relation is defined by all relation compositions that have the relation as subordinate. Thus, a relation can have multiple relation compositions in its composition context. If the type of a relation has no relation type composition as superior, the relation will have no composition context because it does not need any other relation for its own existence.

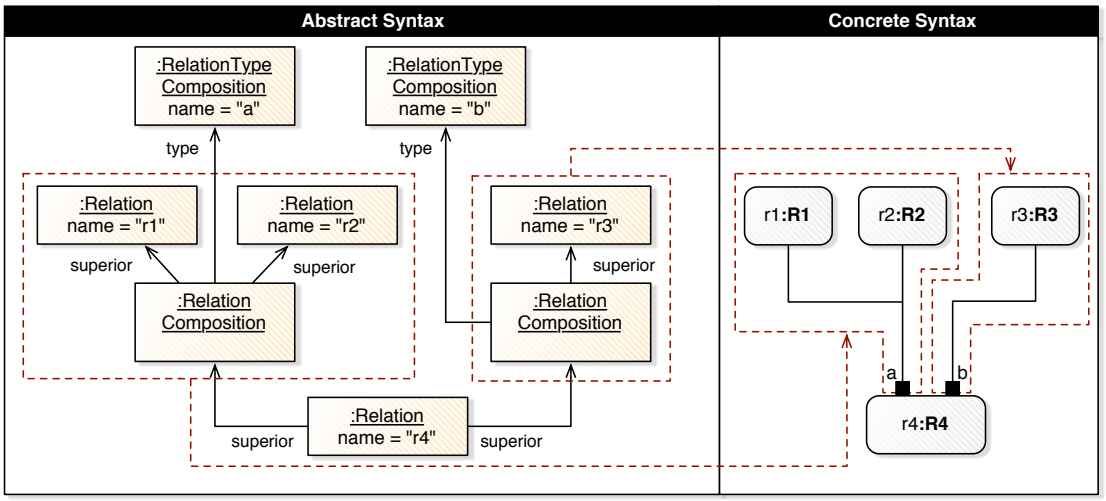

Figure 3.25.: Concrete syntax of relation compositions

The concrete syntax for relation compositions is shown in Figure 3.25, which shows an instance r4 of the relation type R4 that has been shown in Figure 3.13. A relation composition is visualized by means of a connection from a subordinate relation to all superior relations. The connection has a filled square on the side of the subordinate relation, and the name of the relation type composition is shown as a label beside the filled square. The figure shows that the relation r4 has two relation compositions of type a and b. The relation composition of type a has two relations as superior, which are r1 of type R1 and r2 of type R2. The relation composition of type b has one relation as superior, which is r3 of type R3.

In Sections 3.2.1.3.1 and 3.2.1.3.2, it was explained that a relation type composition can be bottom-up or top-down. This also holds for relation compositions. A relation composition is bottom-up if its type is bottom-up, and top-down if its type is top-down. In the following subsections, examples for both kinds of relation compositions are given.

3.2.2.3.1. Bottom-Up Relation Composition The application megamodel, which is shown in Figure 3.26, only shows bottom-up relation compositions. It shows only instances of relation types that have been introduced in Figure 3.8, 3.14 and 3.15.

The relation slrl1 of type SLogical2RLogical is defined between artifact SA1 and SecurityAgent because the representation of SA1 has a soft reference to the representation of SecurityAgent by means of the attribute named type (see Figure 2.6). This relation is composed into the relation sara because sara is defined between the artifacts SAExample and SecurityProduct, which directly and indirectly compose the artifacts SA1 and SecurityAgent. The same holds for the relation slrl2 of the same type, but between artifacts PG and PolicyGateway.

The existence of the relation scrc1 of type SConnector2RConnector defines that the representation of the artifact Distribution of type SConnector has a soft reference to the representation of the artifact Distribution of type RConnector (see Figure 2.6). The relation scrc1 is composed into the relation slrl2 because slrl2 is connected to artifacts that directly compose artifacts connected to scrc1.

3.2.2.3.2. Top-Down Relation Composition The application megamodel, which is shown in Figure 3.27, shows the same situation but with three top-down relation compositions and one bottom-up relation

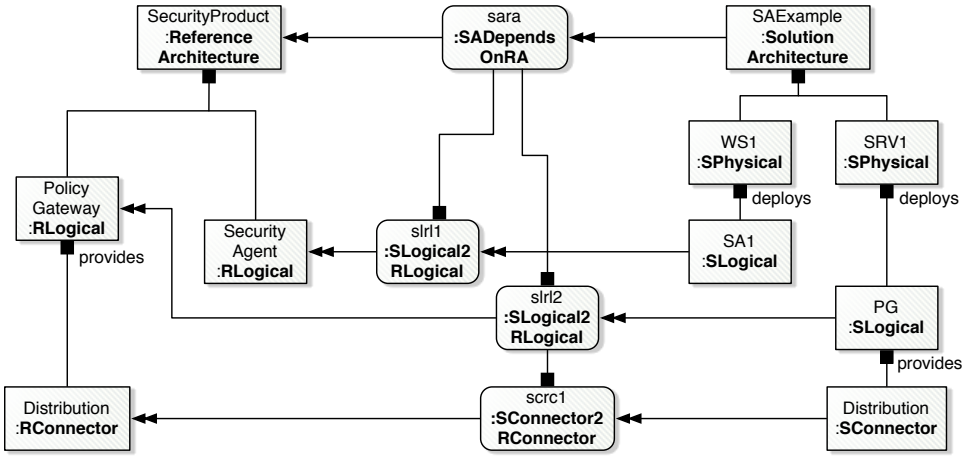

Figure 3.26.: Application megamodel with bottom-up relation compositions

composition. The relations that are shown are instances of relation types that were shown in Figure 3.15, 3.16 and 3.17.

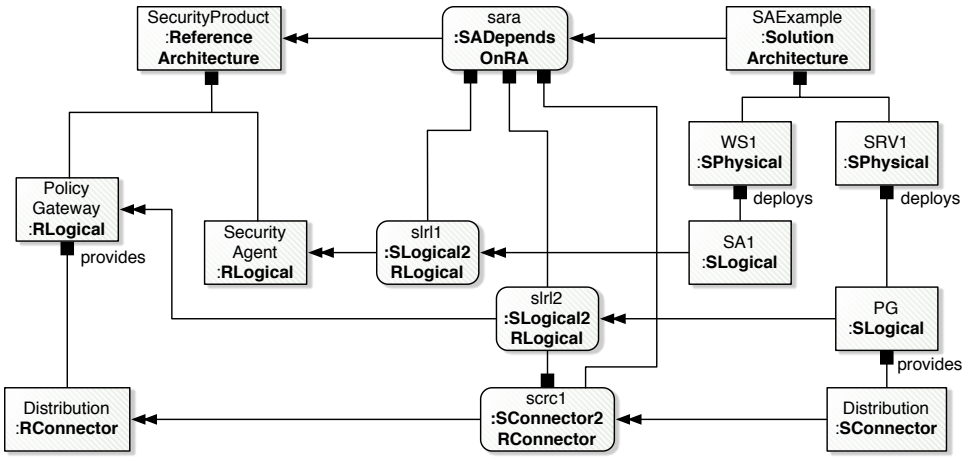

Figure 3.27.: Application megamodel with top-down and bottom-up relation compositions

The relations slrl1 and slrl2 of type SLogical2RLogical do exist without being composed into any other relation. Thus, they do not need the relation sara of type SADepndsOnRA as in the previous example. The relation sara is composed into three relations by means of three different relation compositions. Thus, this relation indicates that SAExample depends on the SecurityProduct because of the existence of the relations slrl1, slrl2 and scrc1. The relation scrc1 is similar to the equally named relation in the previous example because their relation types are similar.

3.2.2.4. Formal Definitions and Constraints

The previously introduced application megamodel will now be formally defined. These definitions are required to define the conformance between an application megamodel and an MDE application, as well as an application megamodel and the configuration megamodel it instantiates. These definitions are further used when it comes to define the behavior of this approach. First, a sufficient definition of an MDE application is given, which is required for the definition of an application megamodel.

3.2.5 Definition (MDE Application) For a given MDE configuration (A_{C_P}, O_{C_P}), an MDE application is a set of physical artifacts A_{A_P} with $\forall a_{A_P} \in A_{A_P}, \exists a_{C_P} \in A_{C_P} : a_{A_P}$ is an instance of a_{C_P}.

As shown in this definition, the MDE application is just a set of physical artifacts A_{A_P} that are instances of physical artifact types A_{C_P} of a given MDE configuration. An MDE application is represented by an application megamodel, which is formally defined as shown in Definition 3.2.6. This definition refers to the metamodel of the application megamodel as shown in Figure 3.18.

3.2.6 Definition (Application Megamodel) Given an MDE application A_{A_P}, an application megamodel M_A is an 5-tuple (A, A_C, P, R, R_C) where A is a finite set of artifacts, A_C is a finite set of artifact compositions, P is a finite set of parameters, R is a finite set of relations, and R_C is a finite set of relation compositions.

- The relationships between artifacts and other concepts of the application megamodel are defined by the following mapping functions:
  - $represent_A : A \to A_{A_P}$ maps every artifact to exactly one physical artifact that declares to be represented by the artifact, and $abstract_{A_{A_P}} : A_{A_P} \to A \cup \{\epsilon\}$ maps every physical artifact to at most one artifact that is declared to be the abstract representation of the physical artifact.
  - $sub_A : A \to \mathcal{P}(A_C)$ defines the subordinate association from Artifact to ArtifactComposition and maps every artifact to a set of artifact compositions with each artifact composition of the set declared to be subordinate to the artifact. $sup_{A_C} : A_C \to A$ defines the superior association from ArtifactComposition to Artifact and maps every artifact composition to exactly one artifact that is declared to be superior to the artifact composition.
  - $sup_A : A \to A_C \cup \{\epsilon\}$ defines the superior association from Artifact to ArtifactComposition and maps every artifact to at most one artifact composition that is declared to be superior to the artifact. $sub_{A_C} : A_C \to \mathcal{P}(A) \setminus \{\emptyset\}$ defines the subordinate association from ArtifactComposition to Artifact and maps every artifact composition to a non-empty set of artifacts with each artifact of the set declared to be subordinate to the artifact composition.
- The relationships between relations and other concepts of the application megamodel are defined by the following mapping functions:
  - $connect_R : R \to \mathcal{P}(P) \setminus \{\emptyset\}$ defines the connected association between Relation and Parameter and maps every relation to a non-empty set of parameters with each parameter of the set declares to be connected by the relation. $connect_P : P \to R$ defines the connectedBy association between Parameter and Relation and maps every parameter to a exactly one relation that is declared to be connected to the parameter.
  - $sub_R : R \to \mathcal{P}(R_C)$ defines the subordinate association from Relation to RelationComposition and maps every relation to a set of relation compositions with each relation composition of the set declared to be subordinate to the relation. $sup_{R_C} : R_C \to \mathcal{P}(R) \setminus \{\emptyset\}$ defines the superior association from RelationComposition to Relation and maps every relation composition to a non-empty set of relations with each relation of the set declared to be superior to the relation composition.
  - $sup_R : R \to \mathcal{P}(R_C)$ defines the superior association from Relation to RelationComposition and maps every relation to a set of relation compositions with each relation composition of the set declared to be superior to the relation. $sub_{R_C} : R_C \to R$ defines the subordinate association from RelationComposition to Relation and maps every relation composition to exactly one relation that is declared to be subordinate to the relation composition.
- Additional concepts of parameters and relationships between parameters and other concepts of the application megamodel are defined by the following mapping functions:
  - $val_P : P \to \mathcal{P}(A) \setminus \{\emptyset\}$ defines the value association between Parameter and Artifact and maps every parameter to a non-empty set of artifacts with each artifact declared to be a value of the parameter. $val_A : A \to \mathcal{P}(P)$ defines the valueOf association between Artifact and Parameter and maps every artifact to a set of parameters with each parameter declared to have the artifact as value.
  - $dir_P : P \to \{S, ST, T\}$ defines the direction of a parameter and maps every parameter to exactly one of the directions S (source), ST (source & target) or T (target) with $\forall p \in P : dir_P = dir_{P_t}(type_P(p))$.

The formal definition of the application megamodel provides five sets, with each set representing a specific class of the metamodel shown in Figure 3.18. Thus, instances of a certain class are elements in the relevant set, e.g., instances of the class Artifact are elements in A.

As already mentioned in the introduction of this chapter, an application megamodel is considered as an instantiation of a configuration megamodel. In Figure 3.19, the instantiation relationships between the individual concepts of an application megamodel and a configuration megamodel were shown. These relationships are now formally defined by means of the formal definition of the configuration and application megamodel.

3.2.7 Definition (Instantiation) Given a configuration megamodel $M_C = (A_t, A_{C_t}, P_t, R_t, R_{C_t})$ and an application megamodel $M_A = (A, A_C, P, R, R_C)$, the instantiation relationships between the individual concepts of both megamodels are defined by the following mapping functions:

- $type_A : A \rightarrow A_t$ defines the type association between Artifact and ArtifactType and maps every artifact to exactly one artifact type that is declared to be type of the artifact. $instance_{A_t} : A_t \rightarrow \mathcal{P}(A)$ defines the instance association between ArtifactType and Artifact and maps every artifact type to a set of artifacts with each artifact of the set declared to be an instance of the artifact type.
- $type_R : R \rightarrow R_t$ defines the type association between Relation and RelationType and maps every relation to exactly one relation type that is declared to be type of the relation. $instance_{R_t} : R_t \rightarrow \mathcal{P}(R)$ defines the instance association between RelationType and Relation and maps every relation type to a set of relations with each relation of the set declared to be an instance of the relation type.
- $type_P : P \rightarrow P_t$ defines the type association between Parameter and ParameterType and maps every parameter to exactly one parameter type that is declared to be type of the parameter. $instance_{P_t} : P_t \rightarrow \mathcal{P}(P)$ defines the instance association between ParameterType and Parameter and maps every parameter type to a set of parameters with each parameter of the set declared to be an instance of the parameter type.
- $type_{A_C} : A_C \rightarrow A_{C_t}$ defines the type association between ArtifactComposition and ArtifactTypeComposition and maps every artifact composition to exactly one artifact type composition that is declared to be type of the artifact composition. $instance_{A_{C_t}} : A_{C_t} \rightarrow \mathcal{P}(A_C)$ defines the instance association between ArtifactTypeComposition and ArtifactComposition and maps every artifact type composition to a set of artifact compositions with each artifact composition of the set declared to be an instance of the artifact type composition.
- $type_{R_C} : R_C \rightarrow R_{C_t}$ defines the type association between RelationComposition and RelationTypeComposition and maps every relation composition to exactly one relation type composition that is declared to be type of the relation composition. $instance_{R_{C_t}} : R_{C_t} \rightarrow \mathcal{P}(R_C)$ defines the instance association between RelationTypeComposition and RelationComposition and maps every relation type composition to a set of relation compositions with each relation composition of the set declared to be an instance of the relation type composition.

Thereby, the instantiation relationship between individual concepts is always defined in both directions. Thus, *type* provides a type for a given instance and *instance* provides a set of instances for a given type.

Because the application megamodel is also an abstract representation, it must comply with certain conformance conditions. These are formally defined in Definition 3.2.3.

3.2.8 Definition (Conform MDE Application Representation) An application megamodel $M_A = (A, A_C, P, R, R_C)$ is a conform representation of an MDE application A_{A_P}, if all of the following constraints are satisfied:

- Every artifact is a unique representation of a physical artifact, which is defined as $\forall a, a' \in A : a \neq a' \Rightarrow represent_A(a) \neq represent_A(a')$.
- Every artifact is a conform representation of a physical artifact, which is defined as $\forall a, a' \in A, \exists a_{A_P}, a'_{A_P} \in A_{A_P} : a_{A_P} = represent_A(a) \wedge a'_{A_P} = represent_A(a') \wedge a'_{A_P}$ is subordinate to $a_{A_P} \Rightarrow a' \in subs_A(a)$. Thus, every hierarchy relationship between tuples of physical artifacts are captured by means of artifact compositions between abstractions of these physical artifacts.

- Every relation $r \in R$ is a unique and conform representation of a physical dependency. In addition, if between a set of relations a relation composition exists, a hierarchy relationship must exist between a set of physical dependencies that are represented by the set of relations.

As shown in this definition, an application megamodel M_A is a conform representation of an MDE application, if every artifact is a unique representation of a physical artifact, which also represents the hierarchy of a physical artifact correctly. Furthermore, every relation must be a unique representation, too. This also includes the representation of hierarchy relationships between physical dependencies. Artifacts must be unique representations because the application megamodel should be able to deterministically navigate from an artifact to a physical artifact and back again. This also holds for relations.

Because an application megamodel is an instance of a configuration megamodel, an application megamodel has to adhere to certain conditions to be a conform instantiation of a configuration megamodel.

3.2.9 Definition (Conform Configuration Megamodel Instantiation) An application megamodel $M_A = (A, A_C, P, R, R_C)$ is a conform instantiation of a configuration megamodel $M_C = (A_t, A_{C_t}, P_t, R_t, R_{C_t})$ written as $M_A \models M_C$, if the following conditions are satisfied:

- Every artifact $a \in A$ must be a conform instance of an artifact type $a_t \in A_t$ defined by $\forall a \in A, \exists a_t \in A_t : a_t = type_A(a) \Rightarrow a \models a_t$.
- Every artifact composition $a_C \in A_C$ must be a conform instance of an artifact type composition $a_{C_t} \in A_{C_t}$ defined by $\forall a_C \in A_C, \exists a_{C_t} \in A_{C_t} : a_{C_t} = type_{A_C}(a_C) \Rightarrow a_C \models a_{C_t}$.
- Every parameter $p \in P$ must be a conform instance of a parameter type $p_t \in P_t$ defined by $\forall p \in P, \exists p_t \in P_t : p_t = type_P(p) \Rightarrow p \models p_t$.
- Every relation $r \in R$ must be a conform instance of a relation type $r_t \in R_t$ defined by $\forall r \in R, \exists r_t \in R_t : r_t = type_R(r) \Rightarrow r \models r_t$.
- Every relation composition $r_C \in R_C$ must be a conform instance of a relation type composition $r_{C_t} \in R_{C_t}$ defined by $\forall r_C \in R_C, \exists r_{C_t} \in R_{C_t} : r_{C_t} = type_{R_C}(r_C) \Rightarrow r_C \models r_{C_t}$.

Each of these instantiation conformance relationships ($\models$) are individually defined in the following for artifacts, artifact compositions, parameters, relations and relation compositions. An artifact is a conform instance of an artifact type if the conditions that are defined in Definition 3.2.10 are satisfied.

3.2.10 Definition (Artifact Conformance) Given an application megamodel $M_A = (A, A_C, P, R, R_C)$ and a configuration megamodel $M_C = (A_t, A_{C_t}, P_t, R_t, R_{C_t})$, an artifact $a \in A$ is conform to an artifact type $a_t \in A_t$ written as $a \models a_t$ if the following conditions are satisfied:

- The artifact a must be an instance of the artifact type a_t, which is defined as $a_t = type_A(a)$.
- Every artifact composition $a_C \in A_C$ that is subordinate to the artifact a must be an instance of an artifact type composition $a_{C_t} \in A_{C_t}$ that is subordinate to the artifact type a_t, which is formally defined as $\forall a_C \in sub_A(a), \exists a_{C_t} \in sub_{A_t}(a_t) : a_C \models a_{C_t}$.

An artifact composition is a conform instance of an artifact type composition if the conditions that are defined in Definition 3.2.11 are satisfied.

3.2.11 Definition (Artifact Composition Conformance) Given an application megamodel $M_A = (A, A_C, P, R, R_C)$ and a configuration megamodel $M_C = (A_t, A_{C_t}, P_t, R_t, R_{C_t})$, an artifact composition $a_C \in A_C$ is conform to an artifact type composition $a_{C_t} \in A_{C_t}$ written as $a_C \models a_{C_t}$, if the following conditions are satisfied:

- The artifact composition a_C must be an instance of the artifact type composition a_{C_t}, which is defined as $a_{C_t} = type_{A_C}(a_C)$.
- Every artifact $a \in A$ that is subordinate to the artifact composition a_C must be an instance of an artifact type $a_t \in A_t$ that is subordinate to the artifact type composition a_{C_t}, which is formally defined as $\forall a \in sub_{A_C}(a_C), \exists a_t \in sub_{A_{C_t}}(a_{C_t}) : a \models a_t$.

A parameter is a conform instance of a parameter type if the conditions that are defined in Definition 3.2.12 are satisfied.

3.2.12 Definition (Parameter Conformance) Given an application megamodel $M_A = (A, A_C, P, R, R_C)$ and a configuration megamodel $M_C = (A_t, A_{C_t}, P_t, R_t, R_{C_t})$, a parameter $p \in P$ is conform to a parameter type $p_t \in P_t$ written as $p \models p_t$, if the following conditions are satisfied:

- The parameter p must be an instance of the parameter type p_t, which is defined as $p_t = type_P(p)$.
- If the parameter type p_t has a one multiplicity, the parameter p must have exactly one artifact as value, which is defined as $|multi_{P_t}(p_t)| = 1 \Rightarrow |val_P(p)| = 1$.
- If the parameter type p_t has a many multiplicity, the parameter p must have at least one artifact as value, which is defined as $|multi_{P_t}(p_t)| = \star \Rightarrow |val_P(p)| \geq 1$.
- Every artifact $a \in A$ that is a value of the parameter p must be a conform instance of an artifact type $a_t \in A_t$ that is the value of the parameter type p_t, which is defined as $\forall a \in A : a \in val_P(p) \Rightarrow \exists a_t \in A_t : a_t = val_{P_t}(p_t) \wedge a \models a_t$.

A relation is a conform instance of a relation type if the conditions that are defined in Definition 3.2.13 are satisfied.

3.2.13 Definition (Relation Conformance) Given an application megamodel $M_A = (A, A_C, P, R, R_C)$ and a configuration megamodel $M_C = (A_t, A_{C_t}, P_t, R_t, R_{C_t})$, a relation $r \in R$ is conform to a relation type $r_t \in R_t$ written as $r \models r_t$, if the following conditions are satisfied:

- The relation r must be an instance of the relation type r_t, which is defined as $r_t = type_R(r)$.
- If the relation type r_t has at least one relation type composition $r_{C_t} \in R_{C_t}$ as superior, the relation r must provide at least one relation composition $r_C \in R_C$ as superior that is conform to r_{C_t}, which is defined as $|sup_{R_t}(r_t)| \geq 1 \Rightarrow |sup_R(r)| \geq 1 \wedge r_C \models r_{C_t}$.
- The relation r must have a correct artifact context, which holds if the following conditions are satisfied:
  - Every parameter $p \in P$ that is connected to the relation r must be a conform instance of a parameter type $p_t \in P_t$ that is connected to the relation type r_t, which is defined as $\forall p \in connect_R(r), \exists p_t \in connect_{R_t}(r_t) : p \models p_t$.
  - Every parameter type $p_t \in P_t$ that is connected to the relation type r_t must have a corresponding parameter p that is conform to p_t and that is connected to the relation r, which is defined as $\forall p_t \in connect_{R_t}(r_t), \exists p \in connect_R(r) : p \models p_t$.

A relation composition is conforming to a relation type composition if the conditions that are defined in Definition 3.2.14 are satisfied.

3.2.14 Definition (Relation Composition Conformance) Given an application mega-model $M_A = (A, A_C, P, R, R_C)$ and a configuration megamodel $M_C = (A_t, A_{C_t}, P_t, R_t, R_{C_t})$, a relation composition $r_C \in R_C$ is conform to a relation type composition $r_{C_t} \in R_{C_t}$ written as $r_C \models r_{C_t}$ if the following conditions are satisfied:

- The relation composition r_C must be an instance of the relation type composition r_{C_t}, which is defined as $r_{C_t} = type_{R_C}(r_C)$.
- Every relation $r \in R$ that is superior to the relation composition r_C must be a conform instance of a relation type $r_t \in R_t$ that is superior to the relation type composition r_{C_t}, which is formally defined as $\forall r \in sup_{R_C}(r_C), \exists r_t \in sup_{R_{C_t}}(r_{C_t}) : r \models r_t$.
- Every relation type $r_t \in R_t$ that is superior to the relation type composition r_{C_t} must be a type of a relation $r \in R$ that is superior to the relation composition r_C and that is conform to r_t, which is formally defined as $\forall r_t \in sup_{R_{C_t}}(r_{C_t}), \exists r \in sup_{R_C}(r_C) : r \models r_t$.

Thus, an application megamodel is a conform instance of a configuration megamodel if all its elements are conform instances of elements in a configuration megamodel.

3.3. Synchronization

In order to represent MDE configurations and MDE applications by means of hierarchical megamodels, an additional synchronization facility is required. The synchronization facility is separately defined for synchronizing MDE configurations with configuration megamodels and for synchronizing MDE applications with application megamodels. This separation is required because the synchronization works differently for MDE configurations and MDE applications.

Nevertheless, the synchronization shown below is a generic schema that needs to be implemented for specific kinds of physical artifacts and physical artifact types as explained in the implementation chapter (see Section A.2.3). Thus, it is employed as a blueprint or schema for the implementation.

3.3.1. Synchronization of MDE Configurations

The primary goal of this synchronization is to provide a selected set of artifact types that are representations of physical artifact types in an MDE configuration.

The synchronization facility for MDE configurations is an automatic approach. Nevertheless, it requires that a configuration developer has to manually decide which physical artifact types have to be synchronized and when they have to be synchronized. This is sufficient because it is assumed that physical artifact types (e.g., metamodels) do not change that frequently. If they change, it is further assumed that a new version of the physical artifact will be created and the old version still remains.[9] Changing physical artifacts in MDE configurations has to be obtained carefully because certain technologies, e.g., model editor or model operations may depend on them. Thus, just changing these physical artifacts may invalidate such existing technologies. Figure 3.28 shows the necessary use cases for synchronization of MDE configurations.

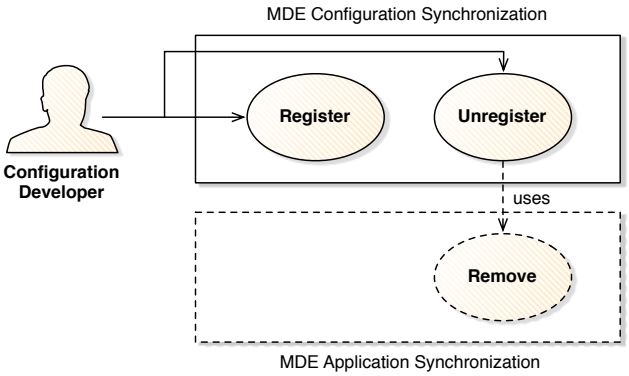

Figure 3.28.: Use cases for synchronization of MDE configurations

The two use cases that are shown in Figure 3.28 are register and unregister, considered to be user-driven, meaning that a user always triggers these operations. In this case, the configuration developer triggers these operations.

In the case of register, the configuration developer chooses a set of physical artifact types from an MDE configuration that should be represented by the configuration megamodel. After triggering register, every registered physical artifact type has an artifact type that is an abstract representation of the physical artifact type. A configuration developer should only register physical artifact types that are not contained by any other physical artifact type.

The register function, as shown in Listing 3.1, takes a set of physical artifact types as input. For every physical artifact type of the given set of physical artifact types it operates as follows. Firstly, it checks whether there is already a representation of that physical artifact type (Line 3). If not, it creates an artifact type as representation (Line 6-7). It further adds an artifact type composition for the first artifact type because all physical artifact types that are contained by the root physical artifact type will be added as subordinate to this artifact type composition (Line 8-9). Secondly, it iterates over all

[9] This is considered as metamodel evolution and basically requires model co-evolution (cf. [47, 38]). However, this is not the focus of this thesis.

```
1  procedure register(A'_{C_P}) // A'_{C_P} ⊆ A_{C_P}
2    forall (a'_{C_P} ∈ A'_{C_P})
3      if (∃a_t ∈ A_t : a'_{C_P} = represent_{A_t}(a_t))
4        continue;
5      else
6        a'_t := create representation of a'_{C_P};
7        A_t := A_t ∪ {a'_t};
8        a_{C_t} := container of a'_t; //a_{C_t} ∈ sub_{A_t}(a'_t)
9        A_{C_t} := A_{C_t} ∪ {a_{C_t}};
10
11       forall (a''_{C_P} ∈ A_{C_P} : a''_{C_P} directly or indirectly contained by a'_{C_P})
12         if (a''_{C_P} does not represent a container)
13           a''_t := create representation of a''_{C_P};
14           A_t := A_t ∪ {a''_t};
15           add a''_t as contained by a_{C_t} thus that a''_t ∈ sub_{A_{C_t}}(a_{C_t});
16         endif
17       endforall
18
19       forall (a''_{C_P} ∈ A_{C_P} : a''_{C_P} directly or indirectly contained by a'_{C_P})
20         if (a''_{C_P} represents a container)
21           A_{C_t} := A_{C_t} ∪ {create representation of a''_{C_P}};
22         endif
23       endforall
24     endif
25   endforall
26 endprocedure
```

Listing 3.1: Synchronization operation: register physical artifact types

physical artifact types that are contained by the considered root physical artifact type (Line 11-17). In this iteration, only physical artifact types are considered that do not act as containers but as types for physical artifacts. For each one of these physical artifact types, an artifact type is created and added to the set of artifact types. Additionally, the artifact type is added as contained by the artifact type composition of the root artifact type. Thirdly, it iterates over the same set of physical artifact types again (Line 19-23). This time, only physical artifact types are considered that act as container (e.g., containment associations). For each of them an artifact type composition is created. Creating an artifact type composition includes setting all required superior and subordinate artifact types.

In case of unregister, a configuration developer chooses a set of physical artifact types that are currently represented by artifact types in a configuration megamodel. All artifact types that represent the chosen physical artifact type will be removed form the configuration megamodel. In addition, all instances of these artifact types will be removed too because artifacts without type are not conform and thus should not exist.

```
1  procedure unregister(A'_{C_P}) // A'_{C_P} ⊆ A_{C_P}
2    forall (a'_{C_P} ∈ A'_{C_P})
3      if (∃a_t ∈ A_t : a'_{C_P} = represent_{A_t}(a_t))
4        forall (a''_{C_P} ∈ A_{C_P} : a''_{C_P} directly or indirectly contained by a'_{C_P})
5          if (∃a'_t ∈ A_t : a''_{C_P} = represent_{A_t}(a'_t))
6            A_t := A_t \ {a'_t}; //also remove all a_{C_t} with a_{C_t} ∈ sub_{A_t}(a'_t)
7          endif
8        endforall
9        forall (a ∈ A : a ∈ instance_{A_t}(a_t))
10         remove(represent_A(a));
11       endforall
12       A_t := A_t \ {a_t};
13     else
14       continue;
15     endif
16   endforall
17 endprocedure
```

Listing 3.2: Synchronization operation: unregister physical artifact types

The unregister operation, as shown in Listing 3.2, also takes a set of physical artifact types as input. For every physical artifact type, which is given as parameter, the operation works as follows. Firstly, it checks whether the physical artifact type is represented by the configuration megamodel (Line 3).

If a representation exists, the unregister operation starts removing all artifact types and artifact type compositions that are directly or indirectly contained by the physical artifact that is currently considered (Line 4-8). Secondly, for each physical artifact that is an instance of the considered physical artifact type, the remove function is invoked (Line 9-11). This function is responsible for removing the artifact and all its direct and indirect subordinates. Finally, the artifact type that represents the considered physical artifact type is removed, too (Line 12).

3.3.2. Synchronization of MDE Applications

The primary goal of synchronization is to provide a selected set of artifacts that are representations of physical artifacts in an MDE application. The synchronization facility for MDE applications is partly semi-automatic and fully automatic. This means that an application developer can decide which physical artifacts should be synchronized and when. Once they are represented by an application megamodel, all physical artifacts that are directly or indirectly subordinate to them will be automatically synchronized. This fully automatic synchronization is necessary because changes to physical artifacts in MDE applications occur frequently. Figure 3.29 shows the required use cases for the synchronization of MDE applications.

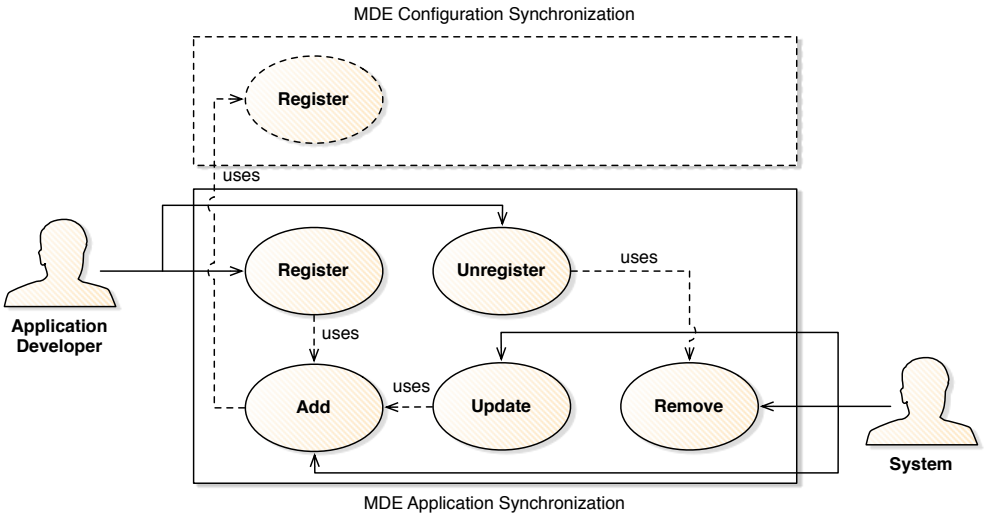

Figure 3.29.: Use cases for synchronization of MDE applications

The use cases that are manually triggered (user-driven) by the application developer are register and unregister. The use cases that are automatically triggered by the system (change-driven), are add, update and remove. These use cases are automatically triggered whenever changes to physical artifacts in an MDE application were made, which are subordinate to a physical artifact that is represented in an application megamodel. The use cases add and remove can also be considered as user-driven because they are implicitly used by the register and unregister use cases.

3.3.2.1. User-Driven Synchronization

The use cases register and unregister are realized by means of two equally named operations. An application developer triggers register whenever a physical artifact in an MDE application should be represented in an application megamodel. It is assumed that register can only be triggered for physical artifacts that are not contained by other physical artifacts that are already represented. If a physical artifact is already registered, all physical artifacts that are directly or indirectly nested into it are automatically synchronized.

The register operation, which is shown in Listing 3.3 is pretty simple. It iterates over a selected set of physical artifacts that are given as parameters. For every physical artifact it checks whether a representation already exists (Line 3). If not, it invokes the add operation using the physical artifact as parameter (Line 6).

```
1  procedure register(A'_{A_P}) // A'_{A_P} ⊆ A_{A_P}
2    forall (a'_{A_P} ∈ A'_{A_P})
3      if (∃a ∈ A : a'_{A_P} = represent_A(a))
4        continue;
5      else
6        add({a'_{A_P}});
7      endif
8    endforall
9  endprocedure
```

Listing 3.3: Synchronization operation: register physical artifacts

```
1  procedure unregister(A'_{A_P}) // A'_{A_P} ⊆ A_{A_P}
2    forall (a'_{A_P} ∈ A'_{A_P})
3      if (∃a ∈ A : a'_{A_P} = represent_A(a))
4        remove({a'_{A_P}});
5      else
6        continue;
7      endif
8    endforall
9  endprocedure
```

Listing 3.4: Synchronization operation: unregister physical artifacts

The unregister operation, which is shown in Listing 3.4, is pretty similar to the register operation that has been shown in Listing 3.3. It iterates over a selected set of physical artifacts that are given as parameters. For every physical artifact in that set, it is checked whether a representation for the considered physical artifact exists (Line 3). If an artifact exists, the remove operation is invoked using the physical artifact as parameter (Line 4).

3.3.2.2. Change-Driven Synchronization

The change-driven synchronization is an incremental approach to keep artifacts in an application megamodel in sync with physical artifacts in an MDE application. Thus, the change-driven synchronization reacts on changes that are coming from the system, which are further called physical change events. These changes are classified in Definition 3.3.1.

3.3.1 Definition (Physical Change Events) A set of physical change events P_E is defined as 3-tuple $(A_{P_E}, U_{P_E}, R_{P_E})$ with $A_{P_E} \subseteq A_{A_P}$ is a finite set of physical artifacts which have been created, $U_{P_E} \subseteq A_{A_P}$ is a finite set of physical artifacts which have been updated, and $R_{P_E} \subseteq A_{A_P}$ is a finite set of physical artifacts which have been removed.

Given a set of physical changes P_E, the change-driven synchronization operations can be triggered accordingly. The add operation is triggered by the system for every physical artifact $a_{A_P} \in A_{P_E}$. Thus, it will automatically create a representation for any newly created physical artifact. The add operation is shown in Listing 3.5.

The add operation does nothing if the physical artifact, which is given as a parameter, has no parent or if the representation of that physical artifact already exists (Line 3). If the physical artifact is directly subordinate to another physical artifact, which is represented in the application megamodel, the artifact that represents the parent of the given physical artifact is estimated (Line 4). Subsequently, the artifact type of the artifact that will be created is estimated (Line 5). If no such artifact type yet exists, the register operation, shown in Listing 3.1, is invoked with the physical artifact as parameter that is the type of the physical artifact (Line 6-8). After the artifact type is estimated, a conform artifact and conform artifact representation is created based on the given superior artifact and artifact type (Line 9-10). Finally, for any physical artifact that is directly contained by the physical artifact that is given, the add operation is called recursively (Line 11-13).

Given a set of physical change events P_E, the system triggers the update operation for every physical artifact in U_{P_E}, which updates the representation for every updated physical artifact. Updating also implies checking whether physical artifacts have been added to or removed from the physical artifact

```
1  procedure add(a_{A_P}) // a_{A_P} ∈ A_{A_P}
2    a'_{A_P} := get direct parent of a_{A_P};
3    if (a'_{A_P} ≠ ϵ ∧ ∀a ∈ A : a_{A_P} ≠ represent_A(a))
4      a' := get representation of a'_{A_P}; //with a'_{A_P} = represent_A(a')
5      a_{C_P} := get type of a_{A_P};
6      if (∀a_t ∈ A_t : a_{C_P} ≠ represent_{A_t}(a_t))
7        register({a_{C_P}});
8      endif
9      a_t := get representation of a_{C_P}; //with a_{C_P} = represent_{A_t}(a_t);
10     a := create representation of a_{A_P}; //with a_t = type_A(a) and a' = sups_A(a);
11     forall (a''_{A_P} directly contained by a_{A_P})
12       add(a''_{A_P});
13     endforall
14   endif
15 endprocedure
```

Listing 3.5: Synchronization operation: add artifact

that has changed. This is necessary because it cannot always be assumed that change events will be triggered for physical artifacts at any level of detail. The update function is shown in Listing 3.6.

```
1  procedure update(a_{A_P}) // a_{A_P} ∈ A_{A_P}
2    a := get representation of a_{A_P}; //with a_{A_P} = represent_A(a)
3    forall (a'_{A_P} directly contained by a_{A_P})
4      if (∀a' ∈ A : a'_{A_P} ≠ represent_A(a'))
5        add(a'_{A_P});
6      else
7        update(a'_{A_P});
8      endif
9    endforall
10   forall (a' ∈ A : a' ∈ subs*(a))
11     if (represent_A(a') ∉ A_{A_P})
12       A := A \ {a'};
13     endif
14   endforall
15 endprocedure
```

Listing 3.6: Synchronization operation: update artifact

The update function first estimates the artifact that represents the physical artifact that is provided as a parameter (Line 2). Then it iterates over all physical artifacts that are directly contained by the provided physical artifact (Line 3-9). For each contained physical artifact, it is checked whether an artifact exists to represent this physical artifact (Line 4). If not, the add function is called to add the missing artifact (Line 5). If a representation exists, the update operation is invoked recursively (Line 7). Afterwards, it iterates over all artifacts that are direct children of the artifact that represents the physical artifact that is provided as parameter (Line 10-13). If one of these artifacts represents a physical artifact that is not part of the MDE application anymore, the artifact will be removed (Line 11-12).

Given a set of physical change events P_E, the system triggers the remove operation for every physical artifact in R_{P_E}, which removes the representation for every removed physical artifact. The remove operation is shown in Listing 3.7.

```
1  procedure remove(a_{A_P}) // a_{A_P} ∈ A_{A_P}
2    a := get representation of a_{A_P} // with a_{A_P} = represent_A(a)
3    forall (a' ∈ A : a' ∈ subs_A(a))
4      remove(represent_A(a'));
5    endforall
6    A := A \ {a'};
7  endprocedure
```

Listing 3.7: Synchronization operation: remove artifact

The remove operation is a recursive operation that removes every artifact that is directly or indirectly subordinate to the representation of the physical artifact that has been removed. This is legitimate because removing a physical artifact causes every physical artifact to also be removed that is directly or

indirectly contained by it.

3.3.2.3. Resulting Change Events

As seen before, a configuration developer and an application developer can influence a configuration and an application megamodel by registering and unregistering physical artifacts and physical artifact types, respectively. An application developer additionally can operate on physical artifacts, which will cause physical change events. These physical change events will then trigger the synchronization to add, update or remove artifacts of an application megamodel accordingly.

All these modifications will result in further change events that are leveraged by further concepts shown later in this thesis. In the following, a definition of potential change events (Definition 3.3.2) is given as well as effects that will cause certain change events.

3.3.2 Definition (Change Events) A set of change events E is defined as 6-tuple $(A_{A_E}, U_{A_E}, R_{A_E}, A_{R_E}, U_{R_E}, R_{R_E})$ with $A_{A_E} \subseteq A$ is a set of added artifacts, $U_{A_E} \subseteq A$ is a set of updated artifacts, $R_{A_E} \subseteq A$ is a set of removed artifacts, $A_{R_E} \subseteq R$ is a set of added relations, $U_{R_E} \subseteq R$ is a set of updated relations, and $R_{R_e} \subseteq R$ is a set of removed relations.

An artifact added event is raised if a physical artifact has been added to an MDE application. An artifact update event emerges if the physical artifact it represents has been updated. This can have various reasons, e.g., changing attributes, setting references, etc. An artifact remove event is triggered if the physical artifact it represents has been removed from an MDE application.

A relation added event is triggered whenever a new relation has been added to an application megamodel. A relation update event is raised whenever something directly related to the relation has been changed, e.g.; if parameters have been connected to or disconnected from a relation; if parameters that are connected to a relation have been changed; if the relation composition of a relation has been changed. A relation remove event emerges if a relation has been removed from an application megamodel.

These change events are exploited by the approach introduced in the following two chapters.

3.4. Summary

In this chapter, the concept of a hierarchical megamodel has been introduced as an extension of the megamodel proposed by Bézivin et al. A hierarchical megamodel is an abstract representation of a heterogeneous landscape of physical artifacts. It provides two separate perspectives on physical artifacts in MDE. Thus, a hierarchical megamodel consists of a configuration megamodel and an application megamodel. A configuration megamodel is a type perspective that represents physical artifact types, e.g., metamodels. On the other side, an application megamodel is an instance perspective that represents physical artifacts, e.g., models. A hierarchical megamodel also captures the hierarchical structure of physical artifacts. Thus, it is not only a global view but rather a holistic view on to a collection of physical artifacts at any level of detail.

A hierarchical megamodel supports capturing physical dependencies. Because of the two perspectives, physical dependency types between physical artifact types can be captured by means of relation types, and physical dependencies between physical artifacts can be captured by means of relations. Due to the hierarchical capabilities, physical dependencies can be captured at any level of detail. Because the hierarchical megamodel is an abstract representation of physical artifacts, a mechanism is required to automatically synchronize physical artifacts and artifacts in the hierarchical megamodel. This has been defined by means of the synchronization.

By now, the hierarchical megamodel can provide the required foundations for the realization of traceability and model management in the context of MDE. It will be successively extended in the following two chapters.

4. Localization: Traceability

Contents

* * *

The previous chapter introduced the hierarchical megamodel, consisting of a type perspective (configuration megamodel) and an instance perspective (application megamodel). In this chapter, the hierarchical megamodel is further extended to support automated maintenance of dependencies by automatically creating and deleting relations, which represent physical dependencies, in application megamodels. Thus, this chapter shows a traceability approach for MDE.

Nevertheless, the major goal of this chapter is to complete the foundation for the composition and (re-)application of heterogeneous model operations, to be elaborated in the subsequent chapter.

4.1. Conceptual Introduction

It has already been shown that artifact types and artifacts are maintained by means of a synchronization. This way, the hierarchical megamodel evolves continuously. However, whenever artifacts change, existing relations, which are connected to these artifacts, may become invalidated. Furthermore, new relations may have to be created due to such changes.

In this chapter, it will be shown how relations are automatically deleted, if they should not exist anymore, and how relations are automatically created if they should exist. It will not be shown how relation types are automatically created or deleted. The relation types are manually maintained by a configuration developer in order to support application developers.[1]

By now, it's clear the hierarchical megamodel is not sufficient for the task of automated maintenance because it does not provide enough information about a relation type to automatically decide whether a relation of a specific type should exist or not. To address this, it shall extend the hierarchical megamodel with additional concepts. That extension is called the dynamic hierarchical megamodel. The term "dynamic" has been chosen because it indicates the hierarchical megamodel automatically adapts to changes by deleting and creating relations.[2]

The basic idea of the dynamic hierarchical megamodel is to enrich relation types with more detailed information that is required to automatically decide about the existence of relations of that type (see Section 4.1.1). Based on the enriched specification of relation types, a localization is provided, which

[1] This does not imply that a configuration developer have to manually maintain relation types. For example, a configuration developer might apply existing techniques that automatically generate detailed mappings (e.g., [113]) between metamodels and use these mappings to automatically create relation types. However, this is not in the focus of this thesis.

[2] The synchronization is not considered as a dynamic property because it is a necessary foundation to the megamodel.

operationalizes the automatic creation and deletion of relations of a set of relation types (see Section 4.1.2).

4.1.1. Detailed Specification of Relation Types

To support automatic reasoning about the existences of relations, it has to be defined when a relation of a certain type should exist and when not. This is accomplished by means of the location of a relation, defined by its artifact context and its composition context. Thus, a relation type is used to specify a correct location, which is leveraged by the subsequent localization.

The artifact context of a relation has been introduced as a set of artifacts that are connected to a relation via parameters. For example, the artifact context of slrl1, shown in Figure 3.26, are the artifacts SA1 and SecurityAgent. The composition context has been introduced as a set of relation compositions that are superior to a relation. For example, the composition context of the relation slrl1 is a single relation composition with the relation sara as superior.

In addition, the artifact context of a relation may be influenced by the composition context of a relation because artifacts in the artifact context may have to be in a certain composition relationship to artifacts that are in artifact contexts of relations that are superior to relation compositions in the composition context. For example, it is assumed that SA1, which is connected to the relation slrl1, is indirectly subordinate to SAExample, which is connected to the relation sara, and that SecurityAgent is directly subordinate to SecurityProduct, which is also connected to the relation sara. This also holds for other examples that have been shown.

In [152], an early version of a dynamic hierarchical megamodel was introduced. In that version, so-called localization rules were employed, which are model operations (Story diagrams) characterizing a location of a relation of a specific type. These localization rules have been used to automatically reason about the correctness of a composition context, an artifact context and the relationship in between. Thus, localization rules explicitly couple multiple relation types to each other, which has a negative impact on the reusability of relation types in different composition contexts, because modifying a relation type composition cannot be obtained without adapting the specification of localization rules. Furthermore, the specification of these localization rules is rather complex due to the amount of information that has to be encoded in such a Story diagram.

In [150], the reusability of localization rules was improved by removing everything from localization rules that is specific to the composition context of a relation. Thus, these localization rules only characterize the artifact context of a relation of a specific type, which reduces the complexity of individual localization rules and further decouples localization rules from other relation types that are used in the composition context. Nevertheless, this implies two things.

Firstly, the composition context of a relation of a specific type has to be characterized somewhere else. This has been done automatically by using a relation type composition directly, as introduced in the previous chapter. However, in that early version a relation type composition does only define a single relation type to be superior to another relation type. Thus, building complex compositions of relations, as it will be demonstrated in Chapter 6, is not possible in [150].

Secondly, the relationship between the artifact context and the composition context of a relation of a specific type has to be specified somewhere else. In that previous version, it was assumed that artifacts in an artifact context are always directly or indirectly subordinate to artifacts connected to relations in the composition context. However, for certain relations this is too restrictive. For example, the relation sara, which has been shown in Figure 3.27, is connected to the artifacts SAExample and SecurityProduct. Here, it is assumed that SAExample is indirectly superior to SA1, PG and Distribution, and that SecurityProduct is directly or indirectly superior to SecurityAgent, PolicyGateway and Distribution. Thus, these kinds of composition cannot be expressed in [150].

The dynamic hierarchical megamodel, introduced in this chapter, overcomes these issues and furthermore can be used to build complex compositions of relations. In order to do so, it provides two additional concepts called relation type composition specification and instantiation condition.

The relation type composition specification allows a configuration developer to characterize relation type compositions by means of patterns over relations, parameters and artifacts. This pattern is then employed to automatically reason about the correctness of a composition context of a relation of a specific type. The relation type composition specification also provides a formalism to characterize the relationship between the artifact context of a relation of a specific type and the relations that are part

of the composition context.

The instantiation condition is quite similar to the adapted localization rule that has been shown in [150]. To automatically reason about the correctness of an artifact context, it is required to reason about physical artifacts that are represented by the artifacts in the artifact context. Because physical artifacts are heterogeneous, it would be infeasible to only support a single technology for automatically reasoning about the artifact context. For example, one instantiation condition might reason about the structure of files while another instantiation condition might reason about the internals of a specific model. Thus, instantiation conditions are an abstraction of all kinds of technologies for this specific purpose. An instantiation condition can be implemented by any kind of model operations that are specified in a way that can be employed as an implementation of an instantiation condition.

4.1.2. Automated Relation Maintenance

The actual process of automatically maintaining the existence of dependencies by maintaining the existence of relations is provided by a localization, which exploits the dynamic hierarchical megamodel. The localization consists of localization operations and a localization strategy that uses localization operations for maintaining all relations in an application megamodel.

In this chapter, two versions of the localization are shown. Firstly, batch localization automatically creates and deletes relations regardless of changes that were made. This localization is useful in situations where no change information is available (initialization). Secondly, incremental localization automatically creates and deletes relations based on changes that were made. Thus, the incremental localization analyzes the impact of changes and only maintains those relations that have been impacted by changes, which increases the efficiency and scalability in comparison to the batch localization.

4.1.3. Overview

The introduced dynamic hierarchical megamodel requires further specification by a configuration developer. Figure 4.1 shows additional use cases and their relationships to already explained use cases as shown in Figure 3.3.

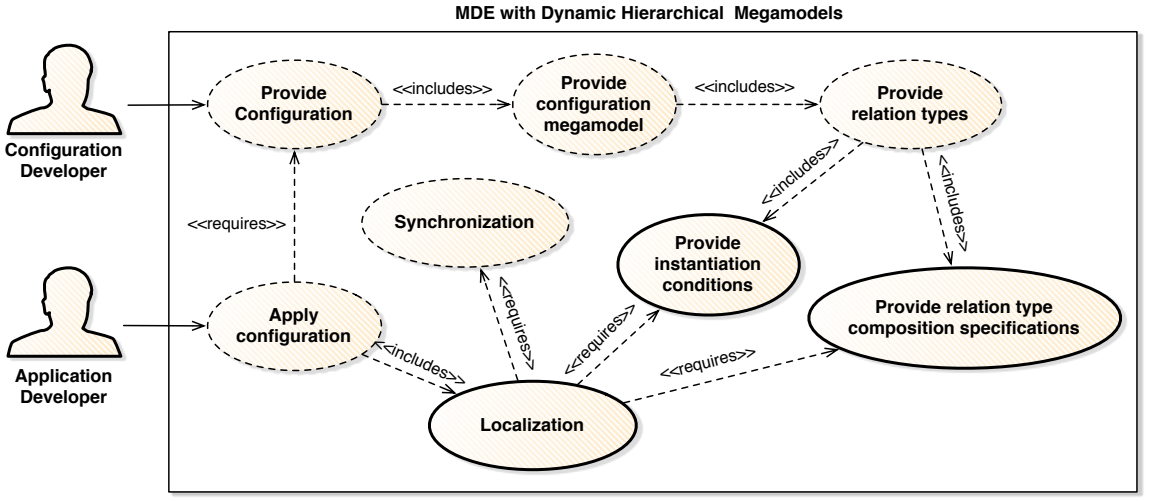

Figure 4.1.: Additional use cases for dynamic hierarchical megamodels and applying the localization

The figure shows that a configuration developer is additionally responsible for providing relation type composition specifications and instantiation conditions, which are included by the specification of relation types. An application developer benefits from these additional specifications because she can use the localization to let relations be automatically maintained. Therefore, an application developer just needs to trigger the localization, which internally requires the synchronization as well as the specification of instantiation conditions and relation type composition specifications.

In the rest of this chapter, the concept of the dynamic hierarchical megamodel will be defined in more detail. In addition, a localization is introduced, which automatically maintains the existence of dependencies in MDE applications by means of relations in an application megamodel.

Figure 4.2 shows how the model management framework is going to be extended by means of dynamic hierarchical megamodels and localization. The hierarchical megamodel is replaced by the dynamic hierarchical megamodel. Synchronization is not impacted because artifact types and artifacts do not change in a dynamic hierarchical megamodel. In addition, localization is introduced as part of the model management framework that uses a configuration megamodel of the dynamic hierarchical megamodel to automatically maintain the relations in an application megamodel of the dynamic hierarchical megamodel.

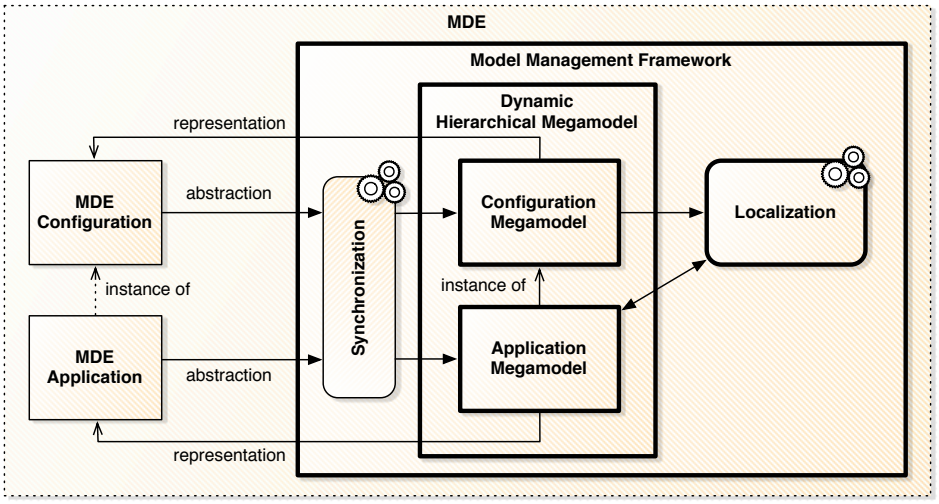

Figure 4.2.: Conceptual integration of dynamic hierarchical megamodels

The dynamic hierarchical megamodel is introduced in Section 4.2 and the localization is shown in Section 4.3.

4.2. Dynamic Hierarchical Megamodels

The dynamic hierarchical megamodel will be explained in two steps. Firstly, the extended configuration megamodel is introduced including all required concepts and formal definitions (see Section 4.2.1). Secondly, the extended application megamodel is introduced also including all required concepts and formal definitions (see Section 4.2.2).

4.2.1. Configuration Megamodels

The primary concepts of a configuration megamodel of a dynamic hierarchical megamodel are shown in the metamodel of Figure 4.3. This metamodel not only shows the extensions but also shows the concepts from the configuration megamodel introduced in Figure 3.5.

The new concepts are relation type composition specification (RelationTypeCompositionSpecification), model operation representation (ModelOperationRepresentation), instantiation condition (Instantiation Condition) that is a specialization of model operation representation, and impact scope (ImpactScope). Furthermore, relation type has an additional attribute named mode. We shall now explain these additional concepts in detail.

4.2.1.1. Relation Type Composition Specifications

The relation type composition specification is employed to provide a detailed characterization of a relation type composition and thus provide a detailed characterization of a composition context of a relation

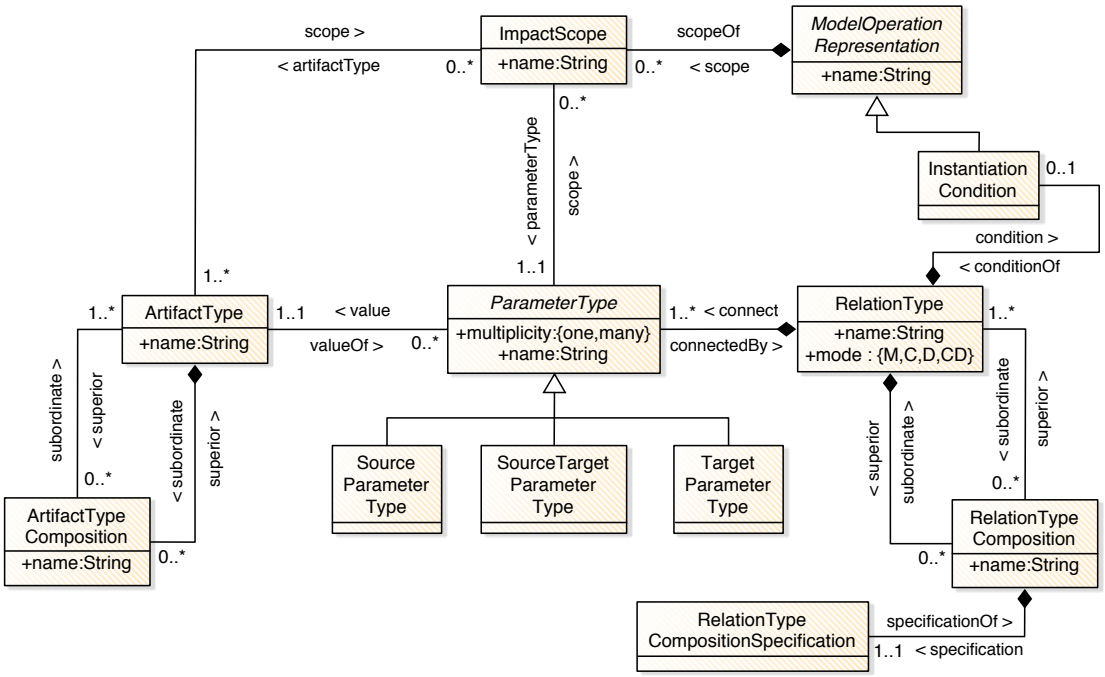

Figure 4.3.: Extended metamodel of the configuration megamodel

of a certain type. Furthermore, the relation type composition specification provides the capability of characterizing a complex composition context. The relation type composition specification is further defined by means of a metamodel that is shown in Figure 4.4.

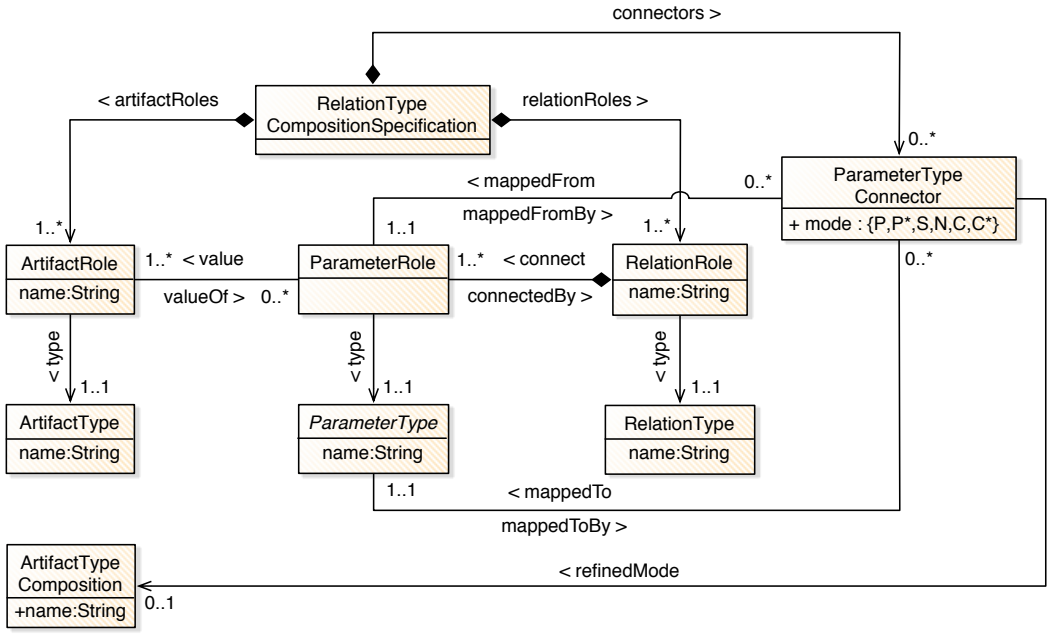

Figure 4.4.: Metamodel of the relation type composition specification

The relation type composition specification is defined for exactly one relation type composition (specification association). If a relation type has multiple superior relation type compositions, each of them has its own relation type composition specification characterizing its particular part of the composition context.

The relation type composition specification comes with further concepts, which are artifact role (ArtifactRole), relation role (RelationRole), parameter role (ParameterRole) and parameter type connector (ParameterTypeConnector). It can be observed that the structure of relation roles, parameter roles and artifact roles is similar to relations, parameters and artifacts in an application megamodel. That is because a set of relation roles, parameter roles and artifact roles is used to define a specific pattern of relations and their connected artifacts, which characterize a specific part of a composition context of a relation.

These roles are considered as instances of relation types, parameter types and artifact types, respectively. This is required when applying this specification to automatically reason about the correctness of a relation composition of that type. However, roles are used instead of relations, parameters and artifacts because roles do not have the purpose of representing something physical. These roles rather define a pattern, specifying a potential situation that does not necessarily exist but could exist in an application megamodel. Thus, artifacts, parameters and relations can be mapped to artifact roles, parameter roles and relation roles, which is called a match. Each match of a relation type composition specification defines a correct relation composition and thus a correct part of the composition context of a relation.

The relationship between the artifact context of a relation and the composition context of a relation is specified by means of a set of parameter type connectors. More precisely, a parameter type connector specifies the relationship between a parameter role in a relation type composition specification and a parameter type that is connected to a relation type that is subordinate to the related relation type composition of the specification. Thus, this relationship is defined by explicitly mapping parameter roles (mappedFrom association) of the relation type composition specification to parameter types (mappedTo association).

A parameter type connector has an attribute named mode, which defines an additional semantic of a parameter type connector. This mode prescribes the composition relationship between artifacts of certain types. A parameter type connector has six pre-defined modes (P, P^*, S, N, C, C^*) that can be used. Every mode is explained in the following:

- P stands for parent and means that an artifact, connected to an instance of the parameter type, must be directly superior to another artifact that is connected to a parameter that is mapped to the parameter role.
- P^* stands for direct or indirect parent of and means that an artifact, connected to an instance of the parameter type, must be directly or indirectly superior to another artifact that is connected to a parameter that is mapped to the parameter role.
- S stands for similar and means that an artifact, connected to an instance of the parameter type, is similar to another artifact that is connected to a parameter that is mapped to the parameter role.
- N stands for neighbor and means that an artifact, connected to an instance of the parameter type, is a sibling of another artifact that is connected to a parameter that is mapped to the parameter role. An artifact is a sibling of another artifact, if they both have the same parent artifact.
- C stands for child and means that an artifact connected to an instance of the parameter type is directly subordinate to an artifact connected to a parameter that is mapped to the parameter role.
- C^* stands for direct or indirect child and means that an artifact connected to an instance of the parameter type is directly or indirectly subordinate to an artifact connected to a parameter that is mapped to the parameter role.

Thus, these modes are used to more precisely define which artifacts are expected to be connected to a parameter of a relation that exists in a specific composition context.

In addition, these relationships can be further refined by means of the refined mode. The refinedMode association is employed for specifying the refined mode, which is associated to an artifact type composition. This means that the relationship between an artifact, connected to an instance of the parameter type, and another artifact, connected to a parameter that is mapped to the parameter role, must be composed by means of an artifact composition as defined by the refined mode.[3]

[3]Basically, other notations could be used to define the semantic of a relationship between composition contexts and artifact contexts. An alternative could be OCL specifications. However, these predefined modes are used to specify a more rigorous definition about the conformance of application megamodels as shown in the sequel to this chapter.

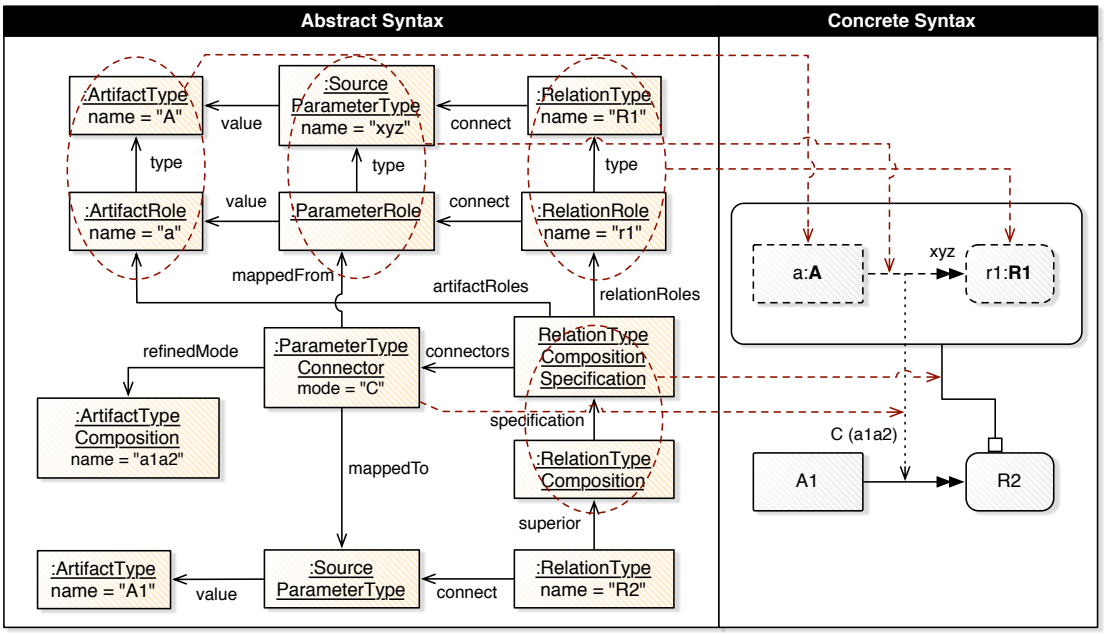

Figure 4.5.: Concrete syntax of relation type compositions specifications

Figure 4.5 shows the concrete syntax of relation type compositions and relation type composition specifications, which are related. This concrete syntax is further used instead of the one shown in Figure 3.13. The figure shows a relation type R2 that has an artifact type A1 connected as source. R2 further has a single relation type composition, which has a relation type composition specification related to it.

A relation type composition specification is visualized by means of a rounded rectangle, which contents are artifact roles, relation roles and parameter roles. These roles are visualized similarly to artifacts, relations and parameters, respectively. The only difference is that the boundaries and connections are dashed. The name of the parameter role is taken from the parameter type it instantiates.

The relation type composition is always connected between exactly one relation type (subordinate to the composition) and exactly one relation type composition specification. Thus, the relation type composition is visualized as a connection between a relation type and a relation type composition specification with a square on the side of the relation type. For any other tuple, a separate connection is visualized.

The visualization of a parameter type connector crosses the boundary of a relation type composition specification. It is visualized as a dotted arrow from a parameter role to a parameter type that is connected to the relation type that is indirectly connected to the relation type composition specification. The mode of a parameter type connector is shown as a label beside the arrow. The refined mode is an extension of the label written in brackets. The label extension of the refined mode displays the name of the artifact type composition that is connected by the refined mode.

As explained in Section 3.2.1.3, relation type compositions can be considered as bottom-up or as top-down relation type compositions. However, the kind of relation type composition cannot be explicitly expressed by means of hierarchical megamodels. The dynamic hierarchical megamodel uses the concept of parameter type connectors to explicitly define the kind of a relation type composition.

4.2.1.1.1. Bottom-Up Relation Type Composition A relation type composition is declared to be bottom-up, if the following conditions are satisfied. At least one parameter type connector of the corresponding relation type composition specification with mode C or C^* is mapped to a parameter type of the subordinate relation type. In addition, all other parameter type connectors must have a mode set to S or N. The relation types SLogical2RLogical and SConnector2RConnector, which has been shown in Section 3.2.1.3.1, are now extended with relation type composition specifications such that they are explicitly composed by means of bottom-up relation type compositions.

Figure 4.6 shows the SLogical2RLogical relation type that is subordinate to a bottom-up relation

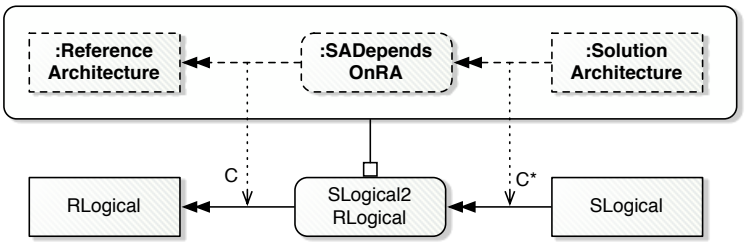

Figure 4.6.: Explicitly specified bottom-up relation type composition (SLogical2RLogical)

type composition with a relation type composition specification that specifies a simple pattern over the relation type SADependsOnRA. The relation type composition specification defines that an instance of SLogical2RLogical requires an instance of SADependsOnRA for its own existence.[4]

For every relation of type SADependsOnRA that acts as superior, the parameter type connectors additionally specify that an artifact of type SLogical must be directly or indirectly subordinate to an artifact of type SolutionArchitecture that is connected as source to the relation of type SADependsOnRA. Furthermore, an artifact of type RLogical must be directly subordinate to an artifact of type ReferenceArchitecture that is connected as target of the relation of type SADependsOnRA.

Thereby, the relation type composition specification ensures that an instance of SLogical always exists in the context of a SolutionArchitecture and that an instance of RLogical always exists in the context of a ReferenceArchitecture and that the SolutionArchitecture depends on the ReferenceArchitecture.

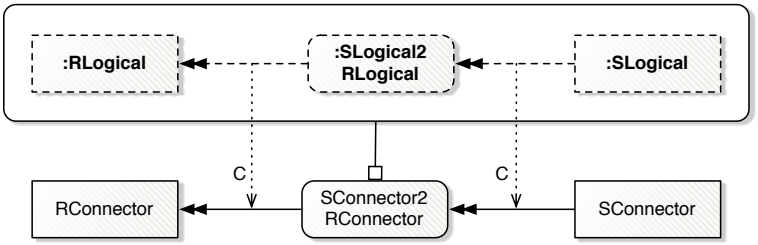

Figure 4.7.: Explicitly specified bottom-up relation type composition (SConnector2RConnector)

Figure 4.7 shows the SConnector2RConnector relation type that is subordinate to a bottom-up relation type composition with a relation type composition specification that defines a simple pattern over the relation type SLogical2RLogical. The semantic of the relation type SConnector2RConnector is that an SConnector depends on an RConnector. Due to the relation type composition specification, the semantic is refined in a way that an SConnector depends on an RConnector only if SConnector is directly subordinate to an SLogical, RConnector is a directly subordinate (C) to an RConnector and the SLogical depends on the RLogical, which is ensured by the existence of a relation of type SLogical2RLogical.

4.2.1.1.2. Top-Down Relation Type Compositions A relation type composition is declared to be top-down, if the following conditions are satisfied. At least one parameter type connector of the corresponding relation type composition specification with mode P or P^* is mapped to a parameter type of the subordinate relation type. In addition, all other parameter type connectors must have a mode set to S or N. In the following top-down example, it is assumed that the relation type SLogical2RLogical is defined, as it has been shown in Figure 3.16.

The relation type SADependsOnRA that is already shown in Figure 3.17 is defined similarly but with relation type composition specifications. Thus, the relation type SADependsOnRA, as shown in Figure 4.8, has two explicitly defined top-down relation type compositions. The relation type composition between SADependsOnRA and SLogical2RLogical is declared to be top-down because SolutionArchitecture is defined to be indirectly superior (P^*) to SLogical and ReferenceArchitecture is defined to be directly

[4]This simple pattern is similar to a relation type composition with only one relation type as superior.

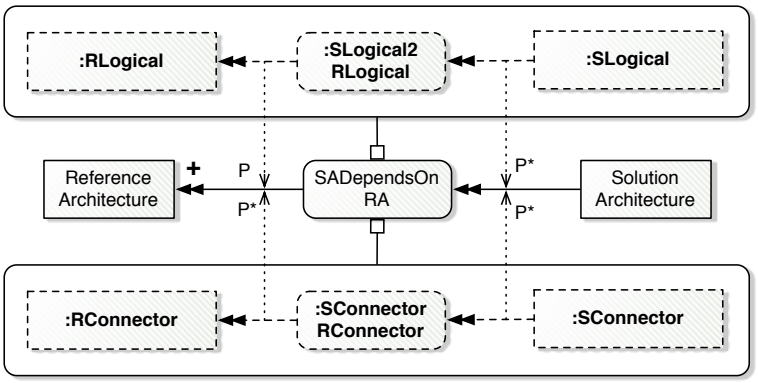

Figure 4.8.: Explicitly specified top-down relation type compositions (SADependsOnRA)

superior (P) to RLogical. Furthermore, the relation type composition between SADependsOnRA and SConnector2RConnector is declared to be top-down because SolutionArchitecture is defined to be indirectly superior (P^*) to SConnector and ReferenceArchitecture is defined to be indirectly superior (P^*) to RConnector.

An instance of SADependsOnRA can exist as soon as a relation of type SLogical2RLogical exists such that an instance of ReferenceArchitecture is a direct superior of an instance of RLogical and an instance of SolutionArchitecture is a direct or indirect superior of an instance of SLogical. An instance of SADependsOnRA can also exist if a relation of type SConnector2RConnector exists such that an instance of ReferenceArchitecture is a direct or indirect superior of an instance of RConnector and an instance of SolutionArchitecture is a direct or indirect superior of an instance of SConnector. An instance of SADependsOnRA can also exist in multiple relation compositions, which are instances of the relation type compositions shown in Figure 3.27.

Thus, SADependsOnRA could be employed to indicate that a SolutionArchitecture depends on a ReferenceArchitecture because of the individual relations to which it is subordinate.

4.2.1.2. Instantiation Conditions

The instantiation condition is a concept to exactly specify the structure of the artifact context of a relation of a certain type. Basically, this could be obtained by any model operation, which can be applied on physical artifacts in order to reason about the artifact context's structure. However, because physical artifacts are potentially heterogeneous, integrating a specific model operation technology might not be sufficient. Additionally, the artifact context of a single relation might consist of several heterogeneous artifacts such that a single technology must even handle different kinds of artifact.

The instantiation condition is employed as an abstraction for model operations, specified in any technology that can be used to express appropriate instantiation conditions. Thus, an instantiation condition is an abstract representation of a model operation employed in an MDE configuration. An instantiation condition is always related to a specific relation type that specifies the artifact context to be analyzed by the instantiation condition. Nevertheless, a relation type must not necessarily have an instantiation condition, if no additional structural restrictions for artifacts in the artifact context exist.

Every model operation, which is employed as an implementation of an instantiation condition, needs to be conform to a relation type that is related to the instantiation condition. This basically means that the application of such a model operation takes the representations of artifacts in an artifact context of a relation of that type as parameters.

Figure 4.9 shows the concrete syntax for relation types that are related to instantiation conditions. The figure shows a relation type R2 that is related to an instantiation condition. The relationship to an instantiation condition is visualized as rectangle placed to the upper left of the relation type labeled with IC.

Figure 4.10 shows the relation type SLogical2RLogical from Figure 4.6. This time the relation type has an instantiation condition related. This instantiation condition characterizes the structure of an

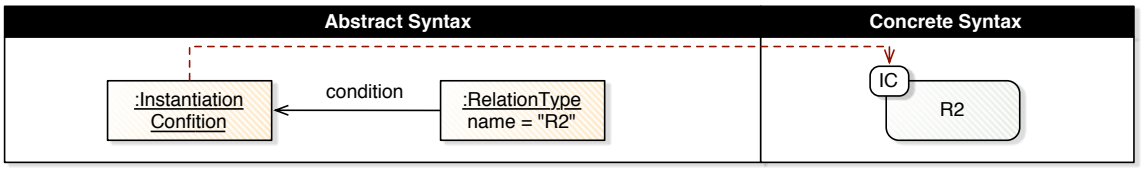

Figure 4.9.: Concrete syntax of instantiation conditions

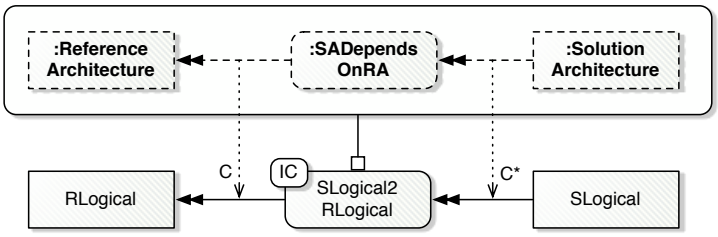

Figure 4.10.: SLogical2RLogical relation type with instantiation condition attached

artifact context of a relation of type SLogical2RLogical. In this case, the artifact context is correct, if a representation of an artifact SLogical uses a representation of an artifact RLogical as type.

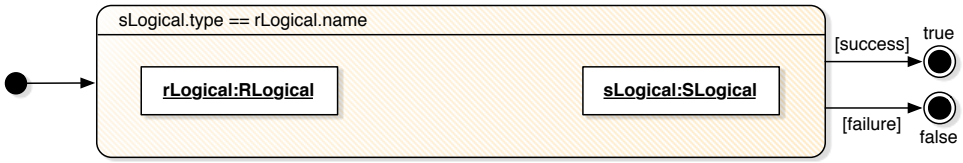

Figure 4.11.: Story diagram implementation of SLogical2RLogical's instantiation condition

Figure 4.11 shows the instantiation condition of SLogical2RLogical implemented as Story diagram. The Story diagram consists of a single Story pattern that awaits an SLogical and an RLogical, which are physical artifacts, as already bound (provided when applying the Story diagram). Based on these two physical artifacts, the Story diagram checks whether the type attribute of SLogical is similar to the name attribute of RLogical (soft reference). If this condition is satisfied, the Story diagram returns true. Else, it returns false, which indicates that the condition is not satisfied for a given artifact context.

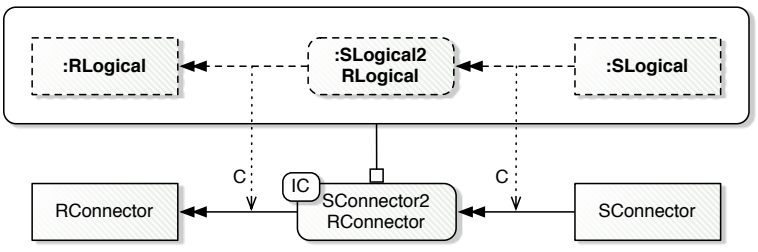

Figure 4.12.: SConnector2RConnector relation type with instantiation condition attached

Another example is shown in Figure 4.12, which is the relation type that has been shown in Figure 4.7, but with an instantiation condition related to it. The instantiation condition of this relation type is also implemented by means of a Story diagram and is shown in Figure 4.13.

The Story diagram simply specifies that the type attribute of the representation of a given artifact SLogical is similar to the name attribute of the representation of a given artifact RConnector (soft refer-

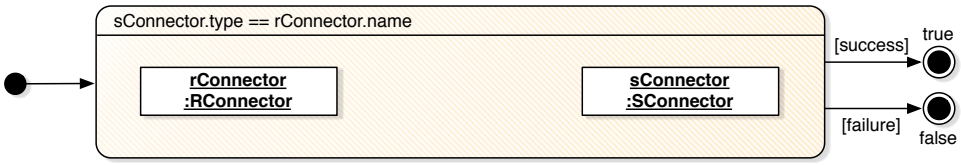

Figure 4.13.: Story diagram implementation of SConnector2RConnector's instantiation condition

ence). If this condition is satisfied for the representation of a given artifact context, the connectors are considered to depend on each other.

The previous examples have shown that instantiation conditions are used to reason about the validity of a certain condition of an artifact context, which are considered as positive conditions. However, instantiation conditions can also be used to refer to conditions that define that a condition does not hold (negative conditions). This is necessary, if relation types are employed to indicate violations of constraints or conditions.

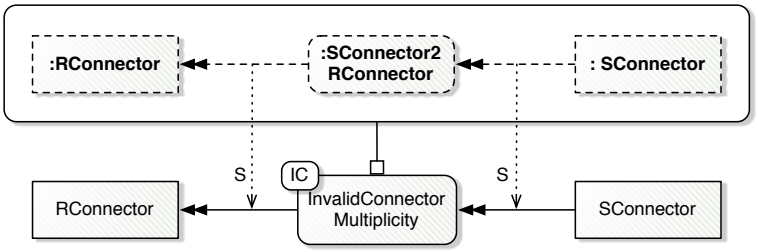

Figure 4.14.: InvalidConnectorMultiplicity relation type for indicating a constraint violation

Figure 4.14 introduces a new relation type that is used to indicate the violation of a specific constraint. If an SConnector depends on an RConnector, it does not mean that the SConnector is conform to the RConnector. For example, an SConnector must respect that only a certain number of SLogical components can use an SConnector. This is defined by means of the usesLower and usesUpper attribute of an RConnector that acts as type of an SConnector. Thus, the relation type InvalidConnectorMultiplicity should only exist between an SConnector and an RConnector if the SConnector is not conform to the RConnector.

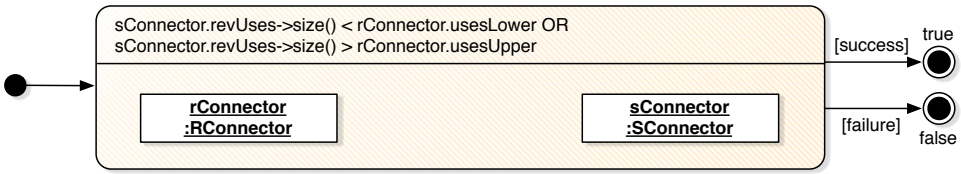

Figure 4.15.: Story diagram implementation of InvalidConnectorMultiplicity's instantiation condition

Figure 4.15 shows the Story diagram that implements this constraint. It returns true, if a given SConnector is used by less SLogical components (using revUses) than specified by the usesLower attribute of a given RConnector. It also returns true, if a given SConnector is used by more SLogical components (using revUses) than specified by the usesUpper attribute of the given RConnector.

Thus, this relation type should always exist, if an SConnector does not correctly instantiate an RConnector. This can be used as a hint to an application developer to correct this violation eventually.

4.2.1.3. Impact Scopes for Instantiation Conditions

As mentioned in the conceptual introduction, the incremental localization uses change information to more precisely maintain the existence of relations. In certain cases, the relation types and instantiation conditions are sufficient to be employed by an incremental localization.

However, this only works if instantiation conditions are specified solely within the scope of an artifact context. This means that the instantiation condition only use physical artifacts that are directly provided by an artifact context including their attributions and references. If an instantiation condition is specified such that it crosses the boundaries of an artifact context, the incremental localization might miss the maintenance of relations, which types are related to such an instantiation condition. That is because the localization will only assume that the correctness of a relation is impacted solely by the composition context and the artifact context.

If the instantiation condition is specified to also reason outside the scope of the artifact context, the instantiation condition must provide a set of impact scopes for that instantiation condition. An impact scope is basically a triple of an instantiation condition, a parameter type and a set of artifact types. Every impact scope belongs to exactly one instantiation condition and is related to exactly one parameter type. The parameter type declares that modifications to artifacts that are directly or indirectly subordinate to an artifact, which is connected to an instance of the parameter type, might impact the related instantiation condition.

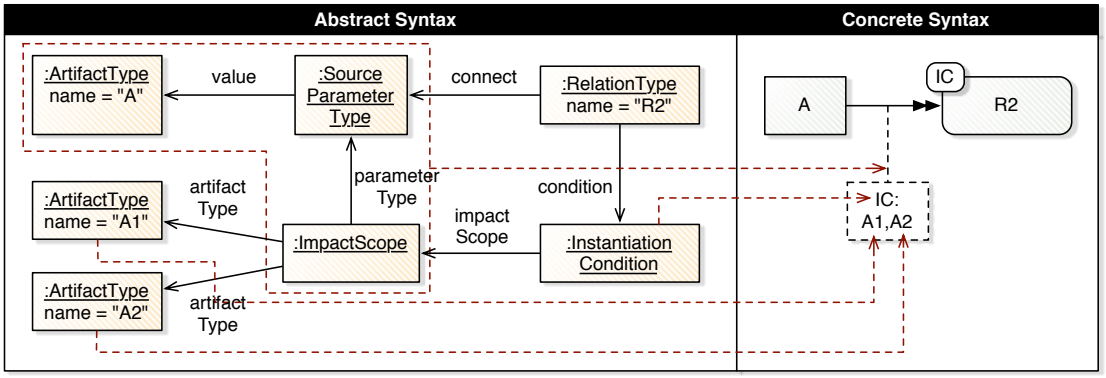

Figure 4.16.: Concrete syntax of impact scopes of instantiation conditions

Figure 4.16 shows the concrete syntax for impact scopes of instantiation conditions. An impact scope is visualized as a dashed rectangle is connected to the related parameter type. The label of the square starts with "IC:", which indicates that this is the impact scope of an instantiation condition, and ends with a comma-separated list of artifact types, which are related to the impact scope.

The figure shows a relation type R2 that is connected to an artifact type A via a source parameter type. It further has an instantiation condition, and a single impact scope related to the source parameter type. The impact scope is defined over artifact types A1 and A2. Thus, the implementation of the instantiation condition reasons about artifacts of type A1 and A2 that are traversed from the artifact of type A in the artifact context. This means that artifacts of type A1 and A2 must be directly or indirectly subordinate to an artifact of type A.

4.2.1.4. Maintenance Modes

The maintenance mode of a relation type defines how relations of a certain type are going to be maintained by the localization. This is necessary, because instances of certain relation types might not be completely maintained or does not have to be maintained at all. For example, higher-level relation types, which are not defined to be in a context composition, are usually manually maintained whereas lower-level relation types, which are defined in context compositions, are rather subject to automated maintenance.

A distinction is made between four different maintenance modes M, C, D and CD, explained as follows:

- M stands for manual, and means that relations of a type whose mode is set to M are only manually maintained. Thus, an application developer has to manually create and delete relations of that type.
- C stands for automatic creation and means that relations of a type whose mode is set to C are automatically created. Thus, an application developer has to manually delete relations of that type eventually.
- D stands for automatic deletion and means that relations of a type whose mode is set to D are automatically deleted. Thus, an application developer has to manually create relations of that type which get automatically deleted in case of invalidation.
- CD stands for automatic creation and deletion and means that relations of a type whose mode is set to CD are automatically created and deleted. Thus, an application developer must not care about maintaining the existence of relations of that type.

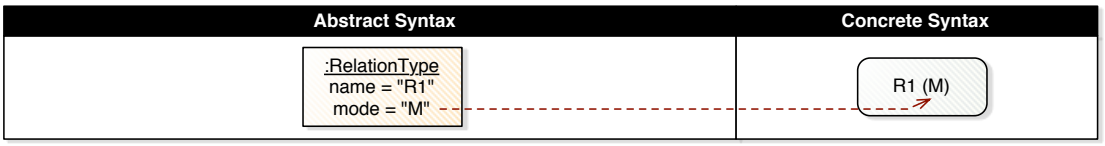

Figure 4.17.: Concrete syntax of relation types with maintenance mode

Figure 4.17 shows the concrete syntax for relation types where a maintenance mode set. The maintenance mode is visualized as an extension to the label of a relation type by adding the mode in braces to the right of the relation type's name. Thus, the figure shows a relation type R1, which is set to maintenance mode M.

Concerning the application example, the relation type SADependsOnRA (see Figure 3.8) is set to M and the relation types SLogical2RLogical (see Figure 4.10), SConnector2RConnector (see Figure 4.12), and InvalidConnectorMultiplicity (see Figure 4.14) are set to CD. Thus, the application developer only needs to define which SolutionArchitecture might depend on which ReferenceArchitecture. The alternative version of SADependOnRA (see Figure 4.8) can also be set to maintenance mode CD, because two architectures are defined to depend on each other if at least one SLogical2RLogical or SConnector2RConnector relation exists between their elements.

4.2.1.5. Formal Definitions and Constraints

The configuration megamodel has been extended as shown in this section. Thus, the definition of the configuration megamodel, which has been shown in Definition 3.2.2, is extended by the definition of the configuration megamodel that is shown in Definition 4.2.1. Everything that has been defined in Definition 3.2.2 still holds but is not shown in the following definition.

4.2.1 Definition (Configuration Megamodel) A configuration megamodel M_C of a dynamic hierarchical megamodel is an 8-tuple $(A_t, A_{C_t}, P_t, R_t, R_{C_t}, C_S, I_C, I_S)$ with C_S is a finite set of relation type context specifications, I_C is a finite set of instantiation conditions and $I_S \subseteq \mathcal{P}(A_t) \setminus \{\emptyset\} \times P_t$ is a finite set of impact scopes. The relationships between these individual concepts of the configuration megamodel are defined by means of mapping functions.

- $spec_{R_{C_t}} : R_{C_t} \rightarrow C_S$ defines the specification association between RelationTypeComposition and RelationTypeCompositionSpecification and maps every relation type composition to exactly one relation type composition specification that is declared to be the detailed specification of the relation type composition. $spec_{C_S} : C_S \rightarrow R_{C_t}$ defines the specificationOf association between RelationTypeCompositionSpecification and RelationTypeComposition and maps every relation type composition specification to exactly one relation type composition.
- $mode_{R_t} : R_t \rightarrow \{M, C, D, CD\}$ defines the mode attribute of RelationType and maps every relation type to either M (manual), C (create only), D (delete only) or CD (create and delete).

- $cond_{R_t} : R_t \rightarrow I_C \cup \{\epsilon\}$ defines the condition association between RelationType and InstantiationCondition and maps every relation type to at most one instantiation condition that is declared to be the instantiation condition of the relation type. $cond_{I_C} : I_C \rightarrow R_t$ defines the conditionOf association between InstantiationCondition and RelationType and maps every instantiation condition to exactly one relation type that is declared to be the relation type of the instantiation condition.

- $represent_{I_C} : I_C \rightarrow O_{C_P}$ maps every instantiation condition to exactly one model operation that is declared to be represented by the instantiation condition, and $abstract_{O_{C_P},I_C} : O_{C_P} \rightarrow \mathcal{P}(I_C)$ maps every model operation to a set of instantiation condition that are declared to be abstract representations of the model operation.

- $scope_{I_C} : I_C \rightarrow \mathcal{P}(I_S)$ defines the scope association between ModelOperationRepresentation (InstantiationCondition) and ImpactScope and maps every instantiation condition to a non-empty set of impact scopes. $scope_{I_S,I_C} : I_S \rightarrow I_C$ defines the scopeOf association between ImpactScope and ModelOperationRerpesentation (InstantiationCondition).

- $scope_{P_t} : P_t \rightarrow \mathcal{P}(I_S)$ defines the scope association between ParameterType and ImpactScope and maps every parameter type to a set of impact scopes. $scope_{I_S,P_t} : I_S \rightarrow P_t$ defines the parameterType association between ImpactScope and ParameterType.

The new concepts that were added to the configuration megamodel are relation type composition specifications C_S, instantiation conditions I_C, and impact scopes I_S including a set of mapping functions in between them.

The relation type composition specification is further defined by means of additional concepts that have been introduced in the metamodel of Figure 4.4. These concepts are formally defined as shown in Definition 4.2.2.

4.2.2 Definition (Relation Type Composition Specification) Given a configuration megamodel $M_C = (A_t, A_{C_t}, P_t, R_t, R_{C_t}, C_S, I_C, I_S)$ a relation type composition specification $c_S \in C_S$ is a 4-tuple (R_R, A_R, P_R, P_{C_t}) with R_R is a finite set of relation roles, P_R is a finite set of parameter roles, A_R is a finite set of artifact roles, and P_{C_t} is a finite set of parameter type connectors. The relationships between these individual concepts are defined by means of mapping functions.

- The mapping functions that define the relationships between roles and other concepts of the relation type composition specification are defined as follows:
  - $connect_{R_R} : R_R \rightarrow \mathcal{P}_{\mathcal{R}}(P_R) \setminus \{\emptyset\}$ defines the connected association between RelationRole and ParameterRole and maps every relation role to a non-empty set of parameter roles with each parameter role of the set declared to be connected by the relation roles. $connect_{P_R} : P_R \rightarrow R_R$ defines the connectedBy association between ParameterRole and RelationRole and maps every parameter role to a exactly one relation role that is declared to be connected to the parameter role.
  - $val_{P_R} : P_R \rightarrow \mathcal{P}(A_R) \setminus \{\emptyset\}$ defines the value association between ParameterRole and ArtifactRole and maps every parameter role to a non-empty set of artifact roles with each artifact role declared to be a value of the parameter role. $val_{A_R} : A_R \rightarrow \mathcal{P}_{\mathcal{R}}(P_R)$ defines the valueOf association between ArtifactRole and ParameterRole and maps a every artifact role to a set of parameter roles with each parameter role declared to have the artifact role as value.

- The mapping functions that define the relationships between parameter type connectors and other concepts of the relation type composition specification are defined as follows:
  - $mapFrom_{P_{C_t}} : P_{C_t} \rightarrow P_R$ defines the mappedFrom association between ParameterTypeConnector and ParameterRole and maps every parameter type connector to exactly one parameter role that is declared to be the source of the parameter type connector. $mapFrom_{P_R} : P_R \rightarrow \mathcal{P}(P_{C_t})$ defines the mappedFromBy association between ParameterRole and ParameterTypeConnector and maps every parameter role to a set of parameter type connectors.
  - $mapTo_{P_{C_t}} : P_{C_t} \rightarrow P_t$ defines the mappedTo association between ParameterTypeConnector and ParameterType and maps every parameter type connector to exactly one parameter type that is declared to be target of the parameter type connector. $mapTo_{P_t} : P_t \rightarrow \mathcal{P}(P_{C_t})$ defines

the mappedToBy association between ParameterType and ParameterTypeConnector and maps every parameter type to a set of parameter type connectors.

- $mode_{P_{C_t}} : P_{C_t} \rightarrow \{P, P^*, S, N, C, C^*\}$ defines the mode attribute of ParameterTypeConnector and maps every parameter type connector to either P (direct superiors), P^* (direct and indirect superiors), S (similar), N (siblings), C (direct subordinates), and C^* (direct and indirect subordinates).
- $refMode_{P_{C_t}} : P_{C_t} \rightarrow A_{C_t} \cup \{\epsilon\}$ defines the refinedMode association between ParameterTypeConnector and ArtifactTypeComposition and maps every parameter type connector to at most one artifact type composition that is declared to be the refined mode of the parameter type connector.

- The mapping functions that define the instantiation relationships between roles and concepts from the configuration megamodel are defined as follows:
  - $type_{A_R} : A_R \rightarrow A_t$ defines the type association between ArtifactRole and ArtifactType and maps every artifact role to exactly one artifact type that is declared to be type of the artifact role.
  - $type_{R_R} : R_R \rightarrow R_t$ defines the type association between RelationRole and RelationType and maps every relation role to exactly one relation type that is declared to be type of the relation role.
  - $type_{P_R} : P_R \rightarrow P_t$ defines the type association between ParameterRole and ParameterType and maps every parameter role to exactly one parameter type that is declared to be type of the parameter role.

The major concepts of the relation type composition specification are relation roles R_R, artifact roles A_R, parameter roles P_R and parameter type connectors P_{C_t}. The roles are used to define a pattern, which describes a potential situation in an application megamodel. The mapping functions define all necessary relationships between the formally defined concepts.

Based on the extension of the configuration megamodel, which has been shown in this chapter, a formal definition for bottom-up and top-down relation type compositions can be given, which has already been informally explained in Section 4.2.1.1.1 and 4.2.1.1.2. A relation type composition is defined to be a bottom-up if the condition defined in Definition 4.2.3 is satisfied.

4.2.3 Definition (Bottom-Up Relation Type Composition) Given a relation type $r_t \in R_t$, a relation type composition r_{C_t} and a relation type composition specification $c_S = (R_R, A_R, P_R, P_{C_t})$ with $r_{C_t} \in sup_{R_t}(r_t) \wedge c_S = spec_{R_{C_t}}(r_{C_t})$, the relation type composition r_{C_t} is in a bottom-up relation type composition if $\forall p'_{C_t} \in P_{C_t}, \exists p_{C_t} \in P_{C_t} : p_{C_t} \neq p'_{C_t} \wedge mode_{P_{C_t}}(p_{C_t}) \in \{C, C^*\} \wedge mode_{P_{C_t}}(p'_{C_t}) \in \{C, C^*, S, N\}$, which means that at least one parameter type connector with mode C or C^* exists and all others may have a mode C, C^*, S or N.

A relation type composition is defined to be top-down if the condition defined in Definition 4.2.4 is satisfied.

4.2.4 Definition (Top-Down Relation Type Composition) Given a relation type $r_t \in R_t$, a relation type composition r_{C_t} and a relation type composition specification $c_S = (R_R, A_R, P_R, P_{C_t})$ with $r_{C_t} \in sup_{R_t}(r_t) \wedge c_S = spec_{R_{C_t}}(r_{C_t})$, the relation type composition r_{C_t} is in a top-down relation type composition if $\forall p'_{C_t} \in P_{C_t}, \exists p_{C_t} \in P_{C_t} : p_{C_t} \neq p'_{C_t} \wedge mode_{P_{C_t}}(p_{C_t}) \in \{P, P^*\} \wedge mode_{P_{C_t}}(p'_{C_t}) \in \{P, P^*, S, N\}$, which means that at least one parameter type connector with mode P or P^* exists and all others may have a mode P, P^*, S or N.

The well-formedness definition of the configuration megamodel from the previous chapter, which has been shown in Definition 3.2.4, has also to be extended because the configuration megamodel from this chapter provides new concepts. Thus, a configuration megamodel is well-formed, if it satisfies the conditions from the previous well-formedness definition and the conditions from the well-formedness definition as follows.[5]

[5] This definition uses mapping functions that are defined in Definition C.1.1. These mapping functions are used as shortcuts to define certain hierarchy capability relationships between artifact types concerning their composition structure.

4.2.5 Definition (Well-Formed Configuration Megamodel) A configuration megamodel $M_C = (A_t, A_{C_t}, P_t, R_t, R_{C_t}, C_S, I_C, I_S)$ is well-formed if the following conditions are satisfied:

- Every impact scope $i_S \in I_S$ that is defined in the scope of an instantiation condition $i_C \in I_C$, which is the condition of a relation type, i_S must be related to a parameter type that is also connected to the relation type. This is formally defined as $\forall i_S \in I_S, \exists i_C \in I_C, r_t \in R_t, p_t \in P_t : i_C = scope_{I_S,I_C}(i_S) \wedge i_C = cond_{R_t}(r_t) \wedge r_t = connect_{P_t}(p_t) \Rightarrow p_t = scope_{I_S,P_t}(i_S)$.
- Every relation type composition specification $c_S \in C_S$ must be well-formed. Thus, $\forall c_S \in C_S, \exists r_t \in R_t : spec_{C_S}(c_S) \in sup_{R_t}(r_t)$ the following conditions must be satisfied:
  - Every relation role $r_R \in R_R$ of the relation type composition specification c_S is a conform instance of a relation type. This means that a relation role r_R must be connected to a parameter role $p_R \in P_R$ that is an instance of a parameter type $p_t \in P_t$ connected to the instance of the relation role r_R and whose artifact roles are instances of the artifact type $a_t \in A_t$ that is connected to the parameter type p_t. This is formally defined as $\forall r_R \in R_R, \forall p_R \in P_R, \forall a_R \in A_R, \exists r'_t \in R_t, \exists p'_t \in P_t : p_R \in connect_{R_R}(r_R) \wedge p'_t \in connect_{R_t}(r'_t) \wedge r'_t = type_{R_R}(r_R) \wedge p'_t = type_{P_R}(p_R) \wedge a_R \in val_{P_R}(p_R) \Rightarrow val_{P_t}(p'_t) = type_{A_R}(a_R)$.
  - Every parameter role that is an instance of a parameter type with a multiplicity of one must be connected to at most one artifact role, which is defined as $\forall p_R \in P_R, \exists p'_t \in P_t : p'_t = type_{P_R}(p_R) \wedge multi_{P_t}(p'_t) = 1 \Rightarrow |val_{P_R}(p_R)| = 1$.
  - Every parameter type connector of the relation type composition specification c_S must be mapped to a parameter type that is connected to the relation type r_t that uses c_S as a relation type composition specification, which is defined as $\forall p_{C_t} \in P_{C_t} : connect_{R_t}(r_t) \in mapTo_{P_{C_t}}(p_{C_t})$.
  - Every parameter type connector of the relation type composition specification c_S must be mapped to distinct parameter types defined as $\forall p_{C_t} \in P_{C_t}, p'_{C_t} \in P_{C_t} : p_{C_t} \neq p'_{C_t} \Rightarrow mapTo_{P_{C_t}}(p_{C_t}) \neq mapTo_{P_{C_t}}(p'_{C_t})$.
  - Every parameter type connector of the relation type composition specification c_S must be defined in a way that the relation type r_t can be composed into a situation as specified by c_S. This primarily depends on parameter type connectors, the mode and the refined mode. The condition that must be satisfied for every parameter type connector is different for each combination of mode and refined mode. Thus, $\forall p_{C_t} \in P_{C_t}, \exists p_R \in P_R, \exists p_t \in P_t : p_R = mapFrom_{P_{C_t}}(p_{C_t}) \wedge p_t = mapTo_{P_{C_t}}(p_{C_t})$ one of the following conditions must be satisfied:
    - If $mode_{P_{C_t}}(p_{C_t}) = P$ and $refMode_{P_{C_t}}(p_{C_t}) = \epsilon$, the artifact type of p_t must be in the set of direct superiors of all artifact roles' types of p_R which is defined by $\forall a_R \in val_{P_R}(p_R) : val_{P_t}(p_t) \in sups_{A_t}(type_{A_R}(a_R))$. Else, if $refMode_{P_{C_t}}(p_{C_t}) = a_{C_t} \in A_{C_t}$, the artifact type of p_t must be in the set of direct superiors of all artifact roles' types by only considering the artifact type composition a_{C_t}, specified by the refined mode, which is defined as $\forall a'_R \in val_{P_R}(p_R) : val_{P_t}(p_t) = sups_{A_t,A_{C_t}}(type_{A_R}(a'_R), a_{C_t})$.
    - If $mode_{P_{C_t}}(p_{C_t}) = P^*$ and $refMode_{P_{C_t}}(p_{C_t}) = \epsilon$, the artifact type of p_t must be in the set of direct or indirect superiors of all artifact roles' types of p_R which is defined by $\forall a'_R \in value_{P_R}(p_R) : value_{P_t}(p_t) \in sups^*_{A_t}(type_{A_R}(a'_R))$. Else, if $refMode_{P_{C_t}}(p_{C_t}) = a_{C_t} \in A_{C_t}$, the same condition as in case of $mode_{P_{C_t}}(p_{C_t}) = P$ must be satisfied.
    - If $mode_{P_{C_t}}(p_{C_t}) = S$ and $refMode_{P_{C_t}}(p_{C_t}) = \epsilon$, the artifact type of p_t must be similar to all artifact roles' types which is defined as $\forall a'_R \in value_{P_R}(p_R) : val_{P_t}(p_t) = type_{A_R}(a'_R)$.
    - If $mode_{P_{C_t}}(p_{C_t}) = N$ and $refMode_{P_{C_t}}(p_{C_t}) = \epsilon$, the artifact type of p_t must be in the set of siblings of all artifact roles' types of p_R which is defined by $\forall a'_R \in val_{P_R}(p_R) : val_{P_t}(p_t) \in sibs_{A_t}(type_{A_R}(a'_R))$. Else, if $refMode_{P_{C_t}}(p_{C_t}) = a_{C_t}$, the artifact type of p_t must be in the set of siblings of all artifact roles' types by only considering the artifact type composition a_{C_t}, specified by the refined mode, which is defined as $\forall a'_R \in val_{P_R}(p_R) : val_{P_t}(p_t) \in sibs_{A_t,A_{C_t}}(type_{A_R}(a'_R), a_{C_t})$.
    - If $mode_{P_{C_t}}(p_{C_t}) = C$ and $refMode_{P_{C_t}}(p_{C_t}) = \epsilon$, the artifact type of p_t must be in the set of direct subordinates of all artifact roles' types of p_R which is defined by $\forall a'_R \in$

$val_{P_R}(p_R) : val_{P_t}(p_t) \in subs_{A_t}(type_{A_R}(a'_R))$. Else, if $refMode_{P_{C_t}}(p_{C_t}) = a_{C_t}$, the artifact type of p_t must be in the set of direct subordinates of all artifact roles' types by only considering the artifact type composition a_{C_t}, that is specified by the refined mode, which is defined as $\forall a'_R \in val_{P_R}(p_R) : val_{P_t}(p_t) \in subs_{A_t,A_{C_t}}(type_{A_R}(a'_R), a_{C_t})$.

* If $mode_{P_{C_t}}(p_{C_t}) = C^*$ and $refMode_{P_{C_t}}(p_{C_t}) = \epsilon$, the artifact type of p_t must be in the set of direct and indirect subordinates of all artifact roles' types of p_R which is defined by $\forall a'_R \in val_{P_R}(p_R) : val_{P_t}(p_t) \in subs^*_{A_t}(type_{A_R}(a'_R))$. Else, if $refMode_{P_{C_t}}(p_{C_t}) = a_{C_t}$, the same condition as in case of $mode_{P_{C_t}}(p_{C_t}) = C$ must be satisfied.

The additional well-formedness definition, which has been shown previously, only provides additional well-formedness conditions for the specification of a relation type composition specification, except one condition that is required for impact scopes. Because an impact scope is always related to exactly one instantiation condition, an impact scope must be related to a parameter type that is connected to a relation type, itself related to the instantiation condition of the impact scope. The well-formedness condition for the relation type composition specification is necessary because the pattern, which can be defined via roles, in a relation type composition specification has to comply to the capabilities of the types the roles instantiate.

Therefore, all parameter roles that are connected to a relation role must have a type that is connected to the type of the relation role, and all artifact roles that are values of a parameter role must have a type that is the value of the type of the parameter role.

The multiplicity of the type of a parameter role must be respected by the parameter role. Thus, if the type of a parameter role has a one multiplicity, the parameter role must have exactly one artifact role as value. The parameter type connector must be specified properly. This depends on the combination of parameter role (mappedFrom) and parameter type (mappedTo). The parameter type must be connected to a relation type that has a relation type composition which defines the relation type composition specification that contains the parameter type connector. Furthermore, the parameter role must be in the same relation type composition specification as the parameter type connector.

In addition, the mode and refined mode of a parameter type connector do also influence the well-formedness. Thus, the mode and refined mode define a specific relationship between the parameter role and the parameter type related via the parameter type connector. For example, if the mode of a parameter type connector is set to C, the parameter type must have an artifact type as value that is directly subordinate the artifact type which is the type of the value of the parameter role. If this does not hold, the relation type composition specification cannot be applied successfully.

A model operation that is represented by an instantiation condition must also satisfy certain conditions to be appropriately employed as an instantiation condition. These conditions are summarized in a definition for instantiation condition conformance as shown in Definition 4.2.6.

4.2.6 Definition (Conform Instantiation Condition Implementation) An instantiation condition $i_C \in I_C$ that is related to a relation type r_t is conform to a model operation $o_{C_P} \in O_{C_P}$ written as $i_C \models o_{C_P}$, if the following conditions are satisfied:

- The model operation o_{C_P} must return true (1) or false (0) only.
- The model operation o_{C_P} must provide a corresponding parameter for every parameter type p_t that is connected to the relation type r_t. For every parameter type with a many multiplicity, the model operation o_{C_P} must provide a corresponding parameter that takes a set of physical artifacts of a corresponding type.
- The model operation o_{C_P} must only reason on physical artifacts that are represented by artifacts that are directly or indirectly subordinate to artifacts in the given artifact context of a relation of type r_t.
- The model operation o_{C_P} must be side-effect free.

The first condition is necessary because instantiation conditions are interpreted as boolean expressions. The second condition defines that the signature of a model operation must be similar to the signature of the relation type that is related to the instantiating condition. This condition ensures that the model operation can be applied on relations of the relation type.

The third condition is required by incremental localization strategies because they use changes to the artifact context as trigger for re-applying instantiation conditions. If the implemented model operation is going to reason about artifacts outside the scope of the artifact context, required re-applications will be missing. Thus, model operations can reason about attributions and references of physical artifacts that are represented by artifacts which are in scope of artifacts in the artifact context.

The fourth condition is based on the required assumption that a model operation of an instantiation condition does not manipulate the MDE application. This is necessary for the subsequent localization, which assumes that instantiation conditions cannot impact the result of other instantiation conditions (independence). Nevertheless, model operations can create temporary artifacts or variables for complex computations but they should not be represented by the application megamodel as artifacts.

4.2.2. Application Megamodels

The primary concepts of an application megamodel of a dynamic hierarchical megamodel are shown in the metamodel of Figure 4.18. This metamodel shows an extension of the application megamodel's metamodel as was shown in Figure 3.18.

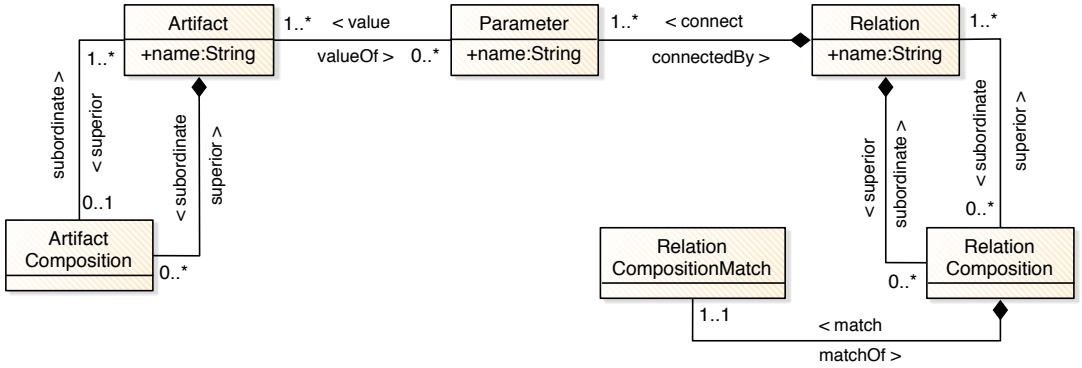

Figure 4.18.: Extended metamodel of the application megamodel

The only new concept to the extended application megamodel is the relation composition match RelationCompositionMatch, which is always owned by a relation composition (match association). This concept is given a detailed explanation in the following.

4.2.2.1. Relation Composition Matches

A relation composition match is employed to define a match of a relation type composition specification. Thus, for each relation composition, a correct relation composition match must exist. The metamodel of a relation composition match is shown in Figure 4.19.

A relation composition match consists of a set of artifact bindings, relation bindings and parameter bindings. Each binding is a mapping between exactly one role and exactly one instance that is matched to that role. A relation composition can be considered as an instance of a relation type composition specification. Thus, for each role of a relation type composition specification, a relation composition match must provide a binding that maps this role to an instance that is a correct match of the role.

Furthermore, a relation composition match consists of a set of parameter connectors. A parameter connector is an instance of a parameter type connector and is a mapping between two parameters. The parameter that is connected via a mappedFrom association is connected to a relation in the composition context of the relation that is subordinate to the relation composition, and the parameter that is connected via a mappedTo association is connected to the relation that is subordinate to the relation composition.

A relation composition match is not explicitly visualized by the concrete syntax because these details are not shown to application developers. These details are rather important for the localization. Nevertheless, it is introduced in detail because it is integral part of the shown approach. The context

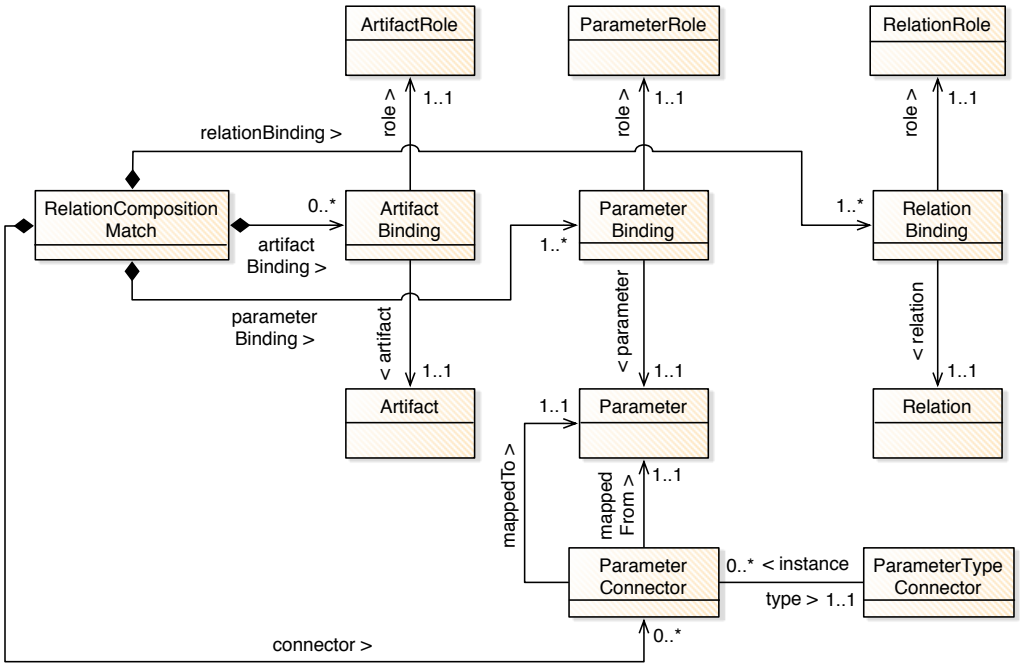

Figure 4.19.: Metamodel of the relation composition match

composition of a relation is visualized by means of relation compositions only, as already been shown in Figure 3.25.

4.2.2.2. Formal Definitions and Constraints

The application megamodel has also been extended in this chapter. Thus, the definition of the application megamodel, which is shown in Definition 4.2.7, is considered as an extension of the application megamodel that has been shown in Definition 3.2.6.

4.2.7 Definition (Application Megamodel) Given an MDE application A_{A_P}, an application megamodel M_A is an 6-tuple (A, A_C, P, R, R_C, C_M) with C_M is a finite set of relation composition matches. The relationships between the individual concepts of an application megamodel are defined by means of mapping functions:

- $match_{R_C} : R_C \rightarrow C_M$ defines the match association between RelationComposition and RelationCompositionMatch and maps every relation composition to exactly one relation composition match that is declared to be a match of a relation type composition specification. $match_{C_M} : C_M \rightarrow R_C$ defines the matchOf association between RelationCompositionMatch and RelationComposition and maps every relation composition match to exactly one relation type composition.

Thus, the only extension to the application megamodel is a set of relation composition matches C_M, which are related to relation compositions R_C. However, a relation composition match provides further concepts as shown in the metamodel of Figure 4.19. Thus, a relation composition match is formally defined as shown in Definition 4.2.8.

4.2.8 Definition (Relation Composition Match) A relation composition match $c_M \in C_M$ is a 4-tuple (A_B, P_B, R_B, P_C) with $A_B \subseteq A_R \times A$ is a finite set of artifact bindings that are tuples of artifact roles and artifacts, $P_B \subseteq P_R \times P$ is a finite set of parameter bindings that are tuples of parameter roles and parameters, $R_B \subseteq R_R \times R$ is a finite set of relation bindings that are tuples of relation roles and relations, and P_C is a finite set of parameter connectors. The relationships between the individual concepts of a relation composition match are defined by means of mapping functions:

- $mapFrom_{P_C} : P_C \rightarrow P$ defines the mappedFrom association between ParameterConnector and Parameter and maps every parameter connector to exactly one parameter that is bound to a parameter role via a parameter binding.
- $mapTo_{P_C} : P_C \rightarrow P$ defines the mappedTo association between ParameterConnector and Parameter and maps every parameter connector to exactly one parameter that is connected to a relation that has a superior relation composition with this match.
- $type_{P_C} : P_C \rightarrow P_{C_t}$ defines the type association between ParameterConnector and ParameterTypeConnector and maps every parameter connector to exactly one parameter type connector that is declared to be a type of the parameter connector. $instance_{P_{C_t}} : P_{C_t} \rightarrow \mathcal{P}(P_C)$ defines the instance association between ParameterTypeConnector and ParameterConnector and maps every parameter type connector to a set of parameter connectors with each parameter connector of the set declared to be an instance of the parameter type connector.

The relation composition match is considered as a match of a relation type composition specification. Thus, it introduces three kinds of bindings - A_B, P_B, and R_B, and a set parameter connectors P_C, which contain instances of parameter type connectors of a certain relation type composition specification. Thereby, every relation composition must provide a relation composition match because every relation composition is an instance of a relation type composition, which is specified by a relation type composition specification.

In the previous chapter, conditions have been shown that are employed to define whether an application megamodel is conform to a configuration megamodel. Because the configuration megamodel as well as the application megamodel has been extended in this chapter, slight extensions to the conformance have to be made.

The first extension is made for the relation composition conformance that has been introduced in Definition 3.2.14. Definition 4.2.9 can only be considered as an addition to Definition 3.2.14. Thus, only the additional condition is required.

4.2.9 Definition (Relation Composition Conformance) Given an application megamodel $M_A = (A, A_C, P, R, R_C, C_M)$ and a configuration megamodel $M_C = (A_t, A_{C_t}, P_t, R_t, R_{C_t}, C_S, I_C, I_S)$, a relation composition $r_C \in R_C$ is conform to a relation type composition $r_{C_t} \in R_{C_t}$ written as $r_C \models r_{C_t}$ if the following conditions are satisfied:

- The relation composition match $c_M \in C_M$ that is related to the relation composition must be a conform match of the relation type composition specification $c_S \in C_S$ that is related to the relation type composition r_{C_t}, which is formally defined as $match_{R_C}(r_C) \models spec_{R_{C_t}}(r_{C_t})$.

Furthermore, a relation composition match must conform to a relation type composition specification. A relation composition match conforms to a relation type composition specification if the conditions that are defined in Definition 4.2.10 are satisfied.

4.2.10 Definition (Relation Composition Match Conformance) Given an application megamodel $M_A = (A, A_C, P, R, R_C, C_M)$ and a configuration megamodel $M_C = (A_t, A_{C_t}, P_t, R_t, R_{C_t}, C_S, I_C, I_S)$, a relation composition match $c_M = (A_B, P_B, R_B, P_C) \in C_M$ is conform to a relation type composition specification $c_S = (R_R, A_R, P_R, P_{C_t}) \in C_S$ written as $c_M \models c_S$ if the following conditions are satisfied:

- Every role in c_S must have a corresponding binding in c_M and every binding in c_M must have a corresponding role in c_S. This is separately defined for artifact roles, parameter roles and relation roles:
  - Every artifact role in c_S has a corresponding artifact binding in c_M and every artifact binding in c_M has a corresponding artifact role in c_S, which is defined as $\forall a_R \in A_R, \exists (a'_R, a) \in A_B : a'_R = a_R$ and $\forall (a'_R, a) \in A_B, \exists a_R \in A_R : a'_R = a_R$. Furthermore, every artifact binding must map an artifact role to an artifact with similar artifact type, which is defined as $\forall (a_R, a) \in A_B : type_{A_R}(a_R) = type_A(a)$.
  - Every parameter role in c_S has a corresponding parameter binding in c_M and every parameter binding in c_M has a corresponding parameter role in c_S, which is defined as $\forall p_R \in$

$P_R, \exists(p'_R, p) \in P_B : p'_R = p_R$ and $\forall(p'_R, p) \in P_B, \exists p_R \in P_R : p'_R = p_R$. Furthermore, every parameter binding must map a parameter role to a parameter of similar parameter type, which is defined as $\forall(p_R, p) \in P_B : type_{P_R}(p_R) = type_P(p)$.

- Every relation role in c_S has a corresponding relation binding in c_M and every relation binding in c_M has a corresponding relation role in c_S, which is defined as $\forall r_R \in R_R, \exists(r'_R, r) \in R_B : r'_R = r_R$ and $\forall(r'_R, r) \in R_B, \exists r_R \in R_R : r'_R = r_R$. Furthermore, every relation binding must map a relation role to a relation with similar relation type, which is defined as $\forall(r_R, r) \in R_B : type_{R_R}(r_R) = type_R(r)$.

- Every binding must build a tuple of a role and an element from an application megamodel that are conform to each other, which is defined for parameter bindings and relation binding as shown in the following:
  - For every parameter binding $(p_R, p) \in P_B$ and artifact binding $(a_R, a) \in A_B$ with an artifact role a_R that is a value of the parameter role p_R, it must hold that also the artifact a is a value of the parameter p. This is formally defined as $\forall(p_R, p) \in P_B, \forall(a_R, a) \in A_B : a_R \in val_{P_R}(p_R) \Rightarrow a \in val_P(p)$.
  - For every relation binding $(r_R, r) \in R_B$ and parameter binding $(p_R, p) \in P_B$ with an parameter role p_R that is connected to the relation role r_R, it must also hold that the parameter p is connected to the relation r. This is formally defined as $\forall(r_R, r) \in R_B, \forall(p_R, p) \in P_B : p_R \in connect_{R_R}(r_R) \Rightarrow p \in connect_R(r)$.

- For every parmeter type connector $p_{C_t} \in P_{C_t}$ that is specified in c_S, a conform instance $p_C \in P_C$ must exist in c_M. Thus, $\forall p_{C_t} \in P_{C_t}, \exists p_C \in P_C : p_{C_t} = type_{P_C}(p_C)$ the following conditions must be satisfied:
  - There must exist a parameter binding $(p_R, p) \in P_B$ with $p_R = mapFrom_{P_{C_t}}(p_{C_t}) \wedge p = mapFrom_{P_C}(p_C)$, which defines that for each parameter connector a correct parameter binding must exist. Furthermore, $mapTo_{P_{C_t}} = type_P(mapTo_{P_C}(p_C))$, defines that a parameter connector must be mapped to a parameter of a conform type.
  - The parameter connector must be mapped to a parameter that is connected to a relation r that is subordinate to the relation composition r_C that is related to the relation composition match c_M, which is defined as $\exists r \in R : mapTo_{P_C}(p_C) \in connect_R(r) \wedge r = sub_{R_C}(r_C) \wedge r_C = match_{C_M}(c_M)$.

- For every parameter connector $p_C \in P_C$, artifacts that are values of a parameter, which is mapped to the parameter connector p_C, must be in a certain hierarchy relationship to at least one artifact, which is a value of the parameter that is mapped from the parameter connector p_C. This hierarchy relationship is specified by means of the mode and refined mode of the relation type connector p_{C_t} that is the type of the parameter connector p_C. Thus, $\forall p_C \in P_C, a \in A, \exists a' \in A, p_{C_t} \in P_{C_t} : p_{C_t} = type_{P_C}(p_C) \wedge a' \in val_P(mapFrom_{P_C}(p_C)) \wedge a \in val_P(mapTo_{P_C}(p_C))$ one of the following conditions must be satisfied:
  - If the mode of p_{C_t} is set to P without refinement, a must be a direct parent of a' defined as $mode_{P_C}(p_{C_t}) = P \wedge refMode_{P_C}(p_{C_t}) = \epsilon \Rightarrow a = sups_A(a')$. Else, if the mode of p_{C_t} is set to P and a refined mode is set, a must be a direct parent of a' by only considering a_{C_t}, which is defined as $\exists a_{C_t} \in A_{C_t} : mode_{P_C}(p_{C_t}) = P \wedge refMode_{P_C}(p_{C_t}) = a_{C_t} \Rightarrow a = sups_{A,A_{C_t}}(a', a_{C_t})$.
  - If the mode of p_{C_t} is set to P^* without refinement, a must be a direct or indirect parent of a' defined as $mode_{P_C}(p_{C_t}) = P^* \wedge refMode_{P_C}(p_{C_t}) = \epsilon \Rightarrow a \in sups^*_A(a')$. Else, if the mode of p_{C_t} is set to P^* and a refined mode is set, a must be a direct or indirect parent of a' by only considering a_{C_t}, which is defined as $\exists a_{C_t} \in A_{C_t} : mode_{P_C}(p_{C_t}) = P^* \wedge refMode_{P_C}(p_{C_t}) = a_{C_t} \Rightarrow a \in sups^*_{A,A_{C_t}}(a', a_{C_t})$.
  - If the mode of p_{C_t} is set to S without refinement, a must be a similar to a' defined as $mode_{P_C}(p_{C_t}) = S \wedge refMode_{P_C}(p_{C_t}) = \epsilon \Rightarrow a = a'$.
  - If the mode of p_{C_t} is set to N without refinement, a must be a neighbor of a' defined as $mode_{P_C}(p_{C_t}) = N \wedge refMode_{P_C}(p_{C_t}) = \epsilon \Rightarrow a \in sibs_A(a')$. Else, if the mode of p_{C_t} is set to

N and a refined mode is set, a must be a neighbor of a' by only considering a_{C_t}, which is defined as $\exists a_{C_t} \in A_{C_t} : mode_{P_C}(p_{C_t}) = N \wedge refMode_{P_C}(p_{C_t}) = a_{C_t} \Rightarrow a \in sibs_{A,A_{C_t}}(a', a_{C_t})$.

- If the mode of p_{C_t} is set to C without refinement, a must be a direct child of a' defined as $mode_{P_C}(p_{C_t}) = C \wedge refMode_{P_C}(p_{C_t}) = \epsilon \Rightarrow a \in subs_A(a')$. Else, if the mode of p_{C_t} is set to C and a refined mode is set, a must be a direct child of a' by only considering a_{C_t}, which is defined as $\exists a_{C_t} \in A_{C_t} : mode_{P_C}(p_{C_t}) = C \wedge refMode_{P_C}(p_{C_t}) = a_{C_t} \Rightarrow a \in subs_{A,A_{C_t}}(a', a_{C_t})$.
- If the mode of p_{C_t} is set to C^* without refinement, a must be a direct or indirect child of a' defined as $mode_{P_C}(p_{C_t}) = C^* \wedge refMode_{P_C}(p_{C_t}) = \epsilon \Rightarrow a \in subs^*_A(a')$. Else, if the mode of p_{C_t} is set to C^* and a refined mode is set, a must be a direct or indirect child of a' by only considering a_{C_t}, which is defined as $\exists a_{C_t} \in A_{C_t} : mode_{P_C}(p_{C_t}) = C^* \wedge refMode_{P_C}(p_{C_t}) = a_{C_t} \Rightarrow a \in subs^*_{A,A_{C_t}}(a', a_{C_t})$.

- If a parameter of the relation r is not the source of a parameter connector, then the artifacts of that parameter are not restricted to a specific set which is defined as $\forall p_C \in P_C, \exists p \in connect_R(r) : p \neq mapTo_{P_C}(p_C) \Rightarrow \forall a \in val_P(p) : a \in A$.

The condition for the conformance of a relation composition match is built from four blocks of conditions. The first block defines that for every binding in c_M there is a corresponding role in c_S. This defines that there are not too many, or missing, matches in c_M. Furthermore, it defines conditions assuring that every binding in c_M relates a role and an element of the application megamodel, where the role and the element have similar types. For example, the artifact role must have a similar type to the artifact in the binding.

The second block defines that relations, parameters, and artifacts, which are matched by the bindings of c_M, are a correct match for the pattern that is specified in c_S. Therefore, it defines that every relation that is bound to a relation role must provide parameters that correspond to parameter roles of the relation role. The same is defined for every parameter that is bound to a parameter role with respect to their artifacts and artifact roles.

The third block is concerned with the conformance of parameter connectors in c_M as regards their types in c_S. Thus, for every parameter type connector in c_S a conform parameter connector p_C must exist in c_M. The first condition defines that a parameter binding p_B must exist such that the parameter role is mapped from p_{C_t} and that the parameter is mapped from p_C. The second condition defines that a parameter connector p_C is mapped to a parameter such that the parameter type connector p_{C_t} is mapped to a parameter type which is similar to the type of the parameter. The third condition defines that every parameter connector p_C maps to a parameter that is connected by a relation that is subordinate to the relation composition of the relation composition match c_M.

The last block defines the correctness of a parameter connector concerning the mode and refined mode specified by the instantiated parameter type connector.

4.3. Localization

The goal of a localization is to automatically create new relations if necessary and to automatically delete relations if they are not conform anymore. This section provides two distinct localizations. Nevertheless, before introducing them, the required operations for localization are discussed. These localization operations are introduced and discussed on an abstract level without focusing on a specific localization. Figure 4.20 shows two use cases with each represents one of the localizations that will be shown. It also shows that an application developer is responsible for triggering the localization.

4.3.1. Localization Operations

Localization is realized by means of three localization operations – namely, update, delete and create. To illustrate the effect of these individual localization operations, two examples are given.

Figure 4.21 shows an example that has already been shown in Figure 3.27. It shows that the artifact PG of type SLogical has been changed, the effect of which is that the instantiation condition of the relation slrl2 is no longer satisfied. This must cause the deletion of slrl2 (1). This deletion must further cause the deletion of the relation composition that is superior to sara and subordinate to slrl2 because it is not

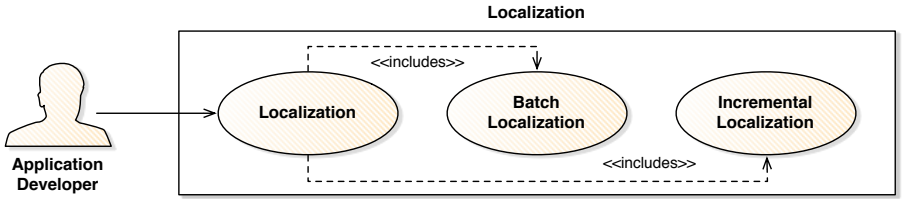

Figure 4.20.: Use cases of the localization

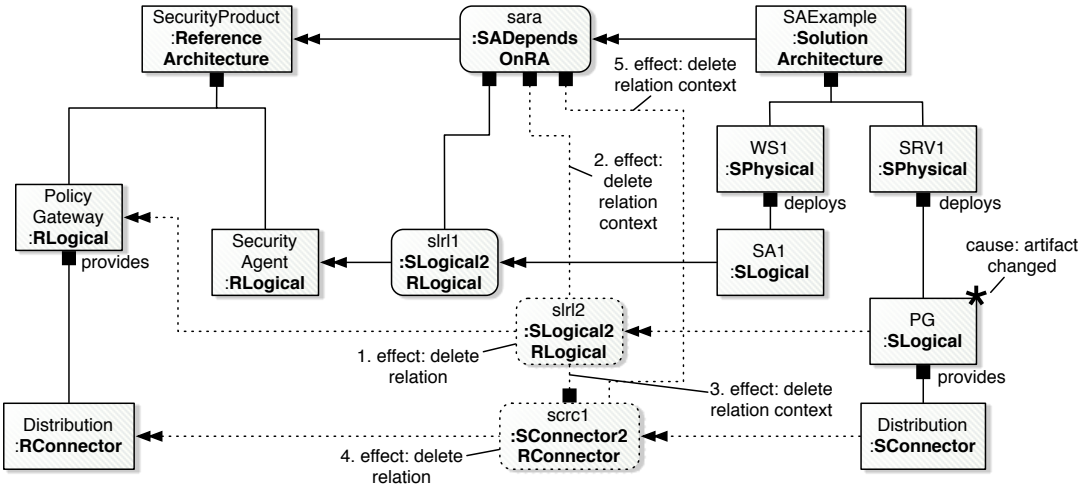

Figure 4.21.: Example of showing the effect of deleting and updating relations

correct anymore (2). Furthermore, the relation composition that is superior to scrc1 and subordinate to slrl2 must also be deleted because the conformance is violated (3). Subsequently, the relation scrc1 has to be deleted, because it has no relation composition left (4). Finally, the relation composition that is superior to sara and subordinate to scrc1 must be deleted (5), because the relation scrc1 has been previously deleted.

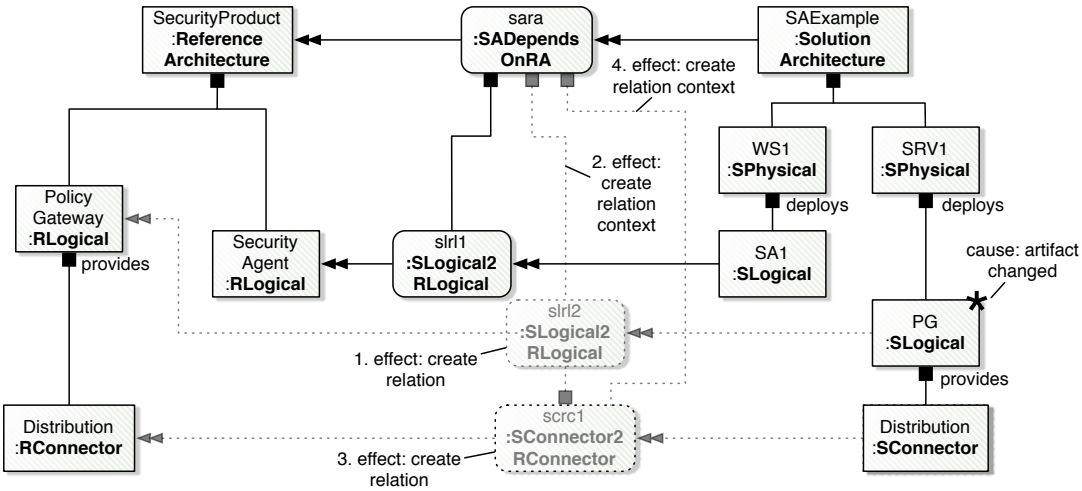

Figure 4.22.: Example of showing the effect of creating and updating relations

Figure 4.22 shows the situation of Figure 4.21 after localization. Based on that application megamodel,

it shows that the artifact PG has been changed again, such that the instantiation condition of the relation slrl2 of type SLogical2RLogical is once again satisfied. Thus, the relation slrl2 must be created between PG and PolicyGateway (1). Because of the creation of slrl2, the relation composition that is superior to sara and subordinate to slrl2 must created (2). The creation of the relation slrl2 must imply the creation of the relation scrc1 with a relation composition that is superior to scrc1 and subordinate to slrl2 (3). Finally, the relation composition that is superior to sara and subordinate to scrc1 must be recreated (4) because scrc1 can be matched by a relation composition superior to sara.

4.3.1.1. Update

The update operation is not responsible for creating or deleting relations, but is only responsible for updating the artifact context and the composition context of existing relations. As a result, the update operation is built of two update operations on existing relations.

On the one hand, an update operation has to maintain the artifact context of existing relations. This is necessary because a parameter may have a type whose multiplicity is many. Thus, if an artifact is going to be created, a parameter might have to be connected to the new artifact. In any case, such a change does not necessarily cause a new relation nor does it cause the deletion of a relation.

On the other hand, an update operation does have to maintain the composition context of existing relations. This is necessary because a relation can exist in multiple relation compositions, which build the composition context of a relation. For example, the relation sara of type SADependsOn exists in three relation compositions at the same time. As long as a relation has at least one correct relation composition left, the relation might still remain. However, due to changes to artifacts or relations, individual relation compositions might violate their conformance. These relation compositions must be deleted without deleting the relation, which is also obtained by the update operation. Furthermore, new relation compositions might have to be created for an existing relation because of new artifacts or relations. The creation of individual relation compositions for existing relations is also task of the update operation.

The update operation is influenced by the delete operation and the create operation, explained below.

4.3.1.2. Delete

The delete operation has to delete relations if necessary, that is, when the relation does not conform to its relation type ($r \models r_t$). If the relation has an instantiation condition, it also depends on the result of applying the instantiation condition to the artifact context of the relation. If the instantiation condition is not satisfied, the relation must be deleted. Furthermore, a relation should only be deleted if the maintenance mode of the corresponding relation type is set to D or CD.

The delete operation is influenced by the update operation because it may delete relation compositions that are not conform. This can invalidate relations and thus the delete operation will delete those relations. The update operation is also influenced by the delete operation because deleting a relation may influence the relation composition of another relation. Thus, after deleting a relation other relations might have to be updated.

4.3.1.3. Create

The create operation has to create as yet not existing relations if necessary, that is, when a correct location for a relation of a specific type exists. If the corresponding type of the relation has no relation type composition as superior, a relation must be created in each artifact context that is valid to the relation and if the instantiation condition is satisfied (if available). If the corresponding relation type has relation type compositions, the relation must also have a set of relation compositions as superior for all matches of relation type composition specifications related to the relation type compositions. Furthermore, a relation should only be created if the maintenance mode of the corresponding relation type is set to C or CD.

The create operation is not influenced by the delete operation because relations are automatically created only by finding the right relations for a relation composition and the right artifact context. The update operation has no influence on that. The update operation is however influenced by the create operation because new relations may complete relation compositions where certain relations are missing.

4.3.2. Batch Localization

The batch localization does not require any change information to work properly. Thus, it can be employed even when no change information is available.

4.3.2.1. Batch Localization Operations

Atomic localization operations are implementations of the abstract localization operations shown in Section 4.3.1, and can be used in the context of the batch localization. Each of the batch localization operations is explained in more detail in the following.

4.3.2.1.1. Update The batch localization implements the update operation by means of two separate update operations. The first update operation is called *updateCompositionContext* and takes a relation r as parameter. The task of this operation is to update the composition context of a given relation r. This means to add all missing relation compositions and to remove those relation compositions that are not conform to its corresponding relation type composition anymore. This operation is completely shown in Listing 4.1.

```
1  procedure updateCompositionContext(r)
2    forall (r_C ∈ sup_R(r))
3      if (r_C ⊨ type_{R_C}(r_C) is not satisfied)
4        R_C := R_C \ {r_C};
5      endif
6    endforall
7    r_t := type_R(r);
8    forall (r_{C_t} ∈ sup_{R_t}(r_t))
9      forall (r' ∈ createAllRelations(r_{C_t}, r_t))
10       if (similarArtifactContext(r, r'))
11         combine(r, r');
12       endif
13     endforall
14   endforall
15 endprocedure
```

Listing 4.1: Batch localization operation: update composition context

The operation consists of two parts. The first part checks for every relation composition, which is superior to r, whether it conforms to its corresponding type (Line 2-6). If not, the relation composition is removed (Line 4). The second part tries to add new relation compositions to r (Line 7-14). Therefore, it iterates over all relation type compositions of the relation type r_t of r. For each relation type composition, a set of temporary and conform relations R' is created, with each relation $r' \in R'$ having a relation composition that instantiates the considered relation type composition. That is obtained by invoking the operation *createAllRelations* (Line 9), which is shown in Listing 4.6. Then, each temporary relation r', which has an artifact context similar to r, is combined with the considered relation r (10-12). Combining means that all relation compositions of r' are moved to r, if they are not yet available. In the end, all relations have up-to-date relation compositions.

The second update operation is called *updateArtifactContext*, which also takes a relation r as parameter. This operation is responsible for adding created artifacts to and removing deleted artifacts from the artifact context of r if necessary. Listing 4.2 shows this operation in detail. The operation only updates the parameters of a relation whose type has a many multiplicity because all other changes will lead to creating new relations or deleting existing relations.

Thus, the operation considers any parameter p with a parameter type that has a many multiplicity (Line 3). If the type of the parameter p is the target of at least one parameter type connector, the artifacts of the parameter p are updated to a set of artifacts which are estimated by invoking the operation *getCompatibleArtifacts* (Line 4-5). This operation will estimate only artifacts with respect to the mode and refined mode of related parameter type connectors. Else, p is set to all available artifacts of the type that can be set to p (Line 6-7).

4.3.2.1.2. Delete The delete operation of the batch localization is implemented by a single delete operation called *deleteRelation*, which takes a single relation r as parameter. The operation is deleting a given relation r if necessary. Listing 4.3 shows the operation in more detail.

```
1  procedure updateArtifactContext(r)
2    r_t := type_R(r);
3    forall (p ∈ connect_R(r) : multi_{P_t}(type_P(p)) = ⋆)
4      if (∃p_{C_t} : p_{C_t} ∈ mapTo_{P_t}(type_P(p)))
5        val_P(p) := getCompatibleArtifacts(p);
6      else
7        val_P(p) := instance_{A_t}(val_{P_t}(p_t));
8      endif
9    endforall
10 endprocedure
```

Listing 4.2: Batch localization operation: update artifact context

```
1  procedure deleteRelation(r) : boolean
2    delete := false;
3    if (r ⊨ type_R(r) not satisfied)
4      delete := true;
5    else
6      i_C := cond_{R_t}(type_R(r));
7      if (i_C ≠ ϵ ∧ eval(i_C, r) = false))
8        delete := true;
9      endif
10   endif
11   if (delete)
12     R := R \ {r};
13     return true;
14   endif
15   return false;
16 endprocedure
```

Listing 4.3: Batch localization operation: delete relation

The operation first checks if the relation r is conform to its corresponding type (Line 3). If this is not the case, the relation will be deleted immediately (Line 11-13). If the relation r is conform, it is further checked if it also satisfies the instantiation condition attached to its type (Line 6-7). If it has no instantiation condition, the relation still conforms to its type. If the evaluation of the instantiation condition is false, the relation will be deleted, too. The operation returns true if a relation is deleted, and false if not.

4.3.2.1.3. Create The create operation of the batch localization is a more complex operation that is separated into several operations that invoke each other. The main operation is called *createRelations* and is shown in Listing 4.4. It takes a relation type r_t as parameter and creates a set of all possible relations R' of the same type as output. All created relations must conform and satisfy their instantiation condition, if available.

The operation has three basic parts. The first part (Line 3-11) creates all conform relations that do not require any relation composition for their own existence. The second part (Line 12-23) creates all conform relations that require at least one relation composition. The third part (Line 25-32) checks whether any created relation also satisfies the instantiation condition that may be provided by its type. If the relation type r_t has no relation type composition (Line 3), a new relation r of type r_t is created and the operation *createAllRelations*, shown in Listing 4.5, is invoked (Line 5). Then, the artifact contexts of all existing relations $r \in R$ with type r_t are compared to all created relations $r' \in R''$ that were just created. If the artifact context of r' is similar to the artifact context of any existing relation $r \in R$, the relation r' is removed from the set R''. After all relations in R'' are traversed, the relations left in R'' are added to the set R'.

If the relation type r_t has at least one relation type composition r_{C_t}, for each relation type composition a set of conform relations is created and added to a set R'' (Line 14). All of these relations exist in a specific relation composition of type r_{C_t}. This is accomplished by invoking the operation *createAllRelations* that is shown in Listing 4.6. For each relation $r' \in R''$ it is checked if another relation $r \in R \cup R'$ of same type exists with a similar artifact context (Line 16). If such a relation r exists, it has to be checked if r has a composition context that is similar to r'. If not, the relation context of r' is added to r, and r' is withdrawn because only one relation of the same type can exist in a similar artifact context (Line 17-18).

```
1  procedure createRelations(r_t) : R'  // with R' ⊆ R
2    R' := ∅;
3    if (sup_{R_t}(r_t) = ∅) // relation type is not defined in a composition
4      r := createRelation(r_t); // with r_t = type_R(r)
5      R'' := createAllRelations(r_t, r, ∅, ∅);
6       forall (r' ∈ R'')
7        if (∃r ∈ R : r_t = type_R(r) ∧ similarArtifactContext(r, r'))
8          R'' := R'' \ {r'};
9        endif
10     endforall
11     R' := R'';
12   else // relation type is defined in composition(s)
13     forall (r_{C_t} ∈ sup_{R_t}(r_t))
14       R'' := createAllRelations(r_{C_t}, r_t);
15       forall (r' ∈ R'')
16         if (∃r ∈ R ∪ R' : r_t = type_R(r) ∧ similarArtifactContext(r, r'))
17           combine(r, r');
18           R'' := R'' \ {r'};
19         endif
20       endforall
21       R' := R' ∪ R'';
22     endforall
23   endif
24   //select relations that satisfy their instantiation condition
25   i_C := cond_{R_t}(r_t);
26   forall (r' ∈ R')
27     if (i_C = ϵ ∨ (i_C ≠ ϵ ∧ eval(i_C, r') = true))
28       R := R ∪ {r'};
29     else
30       R' := R' \ {r'};
31     endif
32   endforall
33   return R';
34 endprocedure
```

Listing 4.4: Batch localization operation: create relations for a given relation type

Now, all relations $r' \in R'$ conform and have to be checked whether they satisfy their instantiation condition, if available (Line 25-32). Thus, if a relation $r' \in R'$ has no instantiation condition or it has an instantiation condition that evaluates to true, the relation is persistently added to the set of relations R. Else, it is removed from the set R'. Finally, all relations that were added to R are returned by returning R' (Line 33).

```
1  procedure createAllRelations(r_t, r, R', P'_t) : R'  //with R' ∩ R = {∅}
2    p'_t := nextParamterType(r_t, P'_t);
3    if (p'_t = ϵ)
4      R' := R' ∪ r;  //with r ⊨ r_t
5    else
6      P'_t := P'_t ∪ p'_t;
7      A' := instance_{A_t}(val_{P_t}(p'_t));
8      if (multi_{P_t}(p'_t) = ⋆)
9        complement(r, p'_t, A');
10       createAllRelations(r_t, r, R', P'_t);
11     else
12       forall (a' ∈ A')
13         r' := copy(r);
14         complement(r', p'_t, {a'});
15         createAllRelations(r_t, r', R', P'_t);
16       endforall
17     endif
18   endif
19   return R';
20 endprocedure
```

Listing 4.5: Batch localization operation: create conform relations without relation type composition

The operation called *createAllRelations*, that is shown in Listing 4.5, creates all possible relations that are conform to a given relation type r_t. All relations that are created by this operation have no relation composition because their type r_t has no relation type composition. This is basically reached by instantiating a relation of the given type into any combination of artifact contexts that can be build

from currently existing artifacts.[6]

The relations R' and parameter types P'_t are required as parameters because of the operation's recursive definition. The set of relations R' is used as the result of the operation and the set of parameter types P'_t is employed for determining the termination condition of the recursion. Within each invocation, the operation determines a value for a currently considered parameter type. Each parameter type is only considered once, and thus the number of recursions is delimited by the number of parameter types connected to r_t.

The operation first determines the next parameter type that is considered by using the *nextParameter Type* operation. This just returns a parameter type p'_t that is connected to r_t but not yet in the set P'_t (Line 2). If P'_t already contains all parameter types of r_t, it returns nothing (ϵ) which indicates that the considered relation r must now conform to r_t and thus can be added to the set of conform relations R' (Line 3-4).

If a parameter type p'_t is still left, it is first added to the set P'_t (Line 6). Then, a set of artifacts A' is estimated by getting all instances of the artifact type that is the value of the currently considered parameter type p'_t (Line 7). It is important that only artifacts are in the set that are not yet the value of any other parameter connected to r, because two parameters cannot refer to the same artifact.

If p'_t has a many multiplicity, the relation r is complemented with a parameter whose values are all artifacts in A'. After complementation the operation is recursively invoked (Line 8-10). Else, if p'_t has a one multiplicity, for each artifact $a' \in A'$ a copy of the relation r is created. Each copy r' is complemented with a parameter whose value is the currently considered a'. Furthermore, a recursion (branch) is started for each complemented copy r' (Line 12-16). If the recursion stops, the set of conform relations R' is returned.

The operation *createAllRelations*, that is shown in Listing 4.6, acts as indirection between the operation that is shown Listing 4.4, which is invoking, and 4.7, which is invoked. The following operation creates all possible relation compositions r'_C of a given relation type composition r_{C_t}. For each relation composition, it creates a set of conform relations within that relation composition.

```
1  procedure createAllRelations(r_Ct, r_t) : R'  //with  R' ∩ R = {∅}
2    R' := {∅};
3    R'_C := createAllRelationCompositions(r_Ct);
4    forall (r'_C ∈ R'_C)
5      r := createRelation(r_t, r'_C);
6      R' := R' ∪ createAllRelations(r_t, r, r_Ct, r'_C, ∅, ∅);
7    endforall
8    return R';
9  endprocedure
```

Listing 4.6: Batch localization operation: create conform relations with relation type composition

As a first step, the operation creates all possible relation compositions R'_C for a given relation type composition r_{C_t} by invoking the operation *createAllRelationCompositions* (Line 3). The operation is a pattern matching approach using the context specification of r_{C_t} as pattern, and context matches of composition contexts as matches of that pattern.[7]

For each relation composition $r'_C \in R'_C$, which is created by the pattern matching, a new relation r of type r_t is created with a relation composition r'_C (Line 5). Based on the relation r, the relation type r_t, the relation composition r'_C, and the relation type composition r_{C_t}, all possible relations R' that conform to r_t are created by invoking the operation *createAllConformRelations* (Line 6), which is shown in Listing 4.7.

This operation *createAllRelations*, which is shown in Listing 4.7, works similarly to the operation shown in Listing 4.5. The only difference is that the set of artifacts, which are used as values for the parameters of the relations, is computed differently (Line 7). The complementation of relations with

[6] This operation has a high computational complexity because the number of artifact contexts, and therefore relations, depends on the number of artifacts ($|A|$) and the number of parameters of a relation ($|P|$). The number of artifact contexts is approximately $|A|^{|P|}$. To keep this number low, the number of parameters should be kept as low as possible, especially if relations do not exist in any relation composition.

[7] The pattern matching is technically realized by interpreting a Story diagram that implements the relation type composition specification and automatically creates all possible relation compositions and matches. The Story diagrams are automatically created from relation type composition specifications and are applied by using the Story diagram interpreter (see [65, 64]).

```
1  procedure createAllRelations(r_t, r, r_Ct, r_C, R', P'_t) : R'  //with  R' ∩ R = {∅}
2    p'_t := nextParameterType(r_t, P'_t);
3    if (p'_t = ϵ)
4      R' := R' ∪ r;  // with  r ⊨ r_t
5    else
6      P'_t := P'_t ∪ p'_t;
7      A' := getCompatibleArtifacts(p'_t, r_Ct, r_C);
8      if (multi_Pt(p'_t) = many)
9        complement(r, p'_t, r_C, A');
10       createAllRelations(r_t, r, r_Ct, r_C, R', P'_t);
11     else
12       forall (a' ∈ A')
13         r' := copy(r);
14         complement(r', p'_t, r_C, {a'});
15         createAllRelations(r_t, r', r_Ct, r_C, R', P'_t);
16       endforall
17     endif
18   endif
19   return R';
20 endprocedure
```

Listing 4.7: Batch localization operation: create conform relations for a given relation composition match

parameters also creates parameter connectors (Line 9 and 14) and the copy operation also copies the relation composition (match) of a relation (Line 13). The set of artifacts A' is estimated with respect to the parameter type connector that is defined in the relation type composition specification of r_{C_t} and which is mapped to p'_t. If parameter type connectors exist that are mapped to the parameter type p'_t, the artifacts in A' will follow the restrictions that are defined by the mode and the refined mode of the parameter type connectors.

4.3.2.2. Sufficient Batch Localization Strategy

Now that all batch localization operations have been introduced, a first batch localization strategy is shown that operationalizes the shown batch localization operations in order to automatically maintain relations in an application megamodel. The batch localization does not use any change information, and thus every time it is applied it will maintain all relations in an application megamodel.

To better explain the batch localization, a sufficient batch localization strategy is shown first in Listing 4.8. This strategy already respects the dependencies between the individual localization operations by first updating relations, deleting relations and finally creating relations.

```
1  procedure sufficientBatchLocalization()
2    changed := true;
3    while (changed)
4      changed := false;
5      forall (r ∈ R : mode_Rt(type_R(r)) ∉ {M})
6        updateCompositionContext(r);
7        updateArtifactContext(r);
8        if (mode_Rt(r_t) ∈ {D, CD})
9          result := deleteRelation(r);
10         if (result)
11           changed := true;
12         endif
13       endif
14     endforall
15     forall (r_t ∈ R_t : mode_Rt(r_t) ∈ {C, CD})
16       R' := createRelations(r_t);
17       if (R' ≠ ∅)
18         changed := true;
19       endif
20     endforall
21   endwhile
22 endprocedure
```

Listing 4.8: Sufficient batch localization strategy

The whole operation is applied several times until a fix-point is reached, which is defined by a surrounding while loop (Line 3-21). In each iteration of the while-loop, the strategy first updates and deletes relations by iterating over all relations that are defined not to be maintained manually (Line 5-

14). For each relation r, the relation is first updated by invoking *updateCompositionContext* and then *updateArtifactContext* on r (Line 6-7). Subsequently, if the relation r should be automatically deleted, the *deleteRelation* operation is invoked on r (Line 8-13). If at least one relation has been deleted, the result of the *deletionRelation* operation is true (Line 9-12). This indicates that another iteration is required because the composition context of existing relations might incomplete or missing. After updating and deleting relations, new relations are going to be created (Line 15-20). For each relation type r_t that should be automatically maintained (Line 15), the *createRelations* operation is invoked on r_t (Line 16). If at least one relation has been created (Line 17-19), another iteration is required because new relations might have to be added to composition contexts of existing relations.

The overall fix-point of this operation is reached, if an iteration has not deleted or created any relation. That a fix-point always exists, and therefore termination is guaranteed, is discussed in Section 4.3.2.2.1. If such a fix-point is reached, the resulting application megamodel only contains relations that conform, and all dependencies are captured by relations.[8] Otherwise, another iteration would be necessary. This is discussed as correctness in Section 4.3.2.2.2.

The major issue of the sufficient batch localization strategy is efficiency, which is discussed in Section 4.3.2.2.3.

4.3.2.2.1. Termination The termination of the sufficient batch localization strategy depends on the existence of a fix-point. Such a fix-point always exists because of the following reasons:

- The deletion of relations always has a fix-point because the number of relations in an application megamodel is finite and the number of successful delete operation applications is strongly monotonously decreasing. The number of delete operation applications is strongly monotonously decreasing because in each iteration at least one relation will be deleted or no more relations are left for deletion. Thus, not later than $n + 1$ rounds, with n as the number of existing relations, no relations are left for deletion and the fix-point is reached.

- The creation of relations always has a fix-point because the number of relation types and artifact contexts is finite and constant. Additionally, the number of successful create operation applications is strongly monotonously increasing. The number of artifact contexts is finite and constant because applying update, delete or create operations does not create any new artifact contexts. The number of create operation applications is strongly monotonously increasing because in each iteration at least one relation will be created. With each creation of a relation the number of available artifact contexts for the corresponding relation type is decreasing. Thus, after $m + 1$ rounds, with m as the number of potential artifact contexts, no artifact contexts are left and a fix-point is reached.

- The combination of both always has a fix-point. That is because the deletion of relations and the creation of relations have fix-points and the deletion of relations does not lead to the creation of relations, and vice-versa. The deletion of relations does not lead to further creations of relations because a relation is only deleted if it is not conform or if the instantiation condition is not satisfied, whereby the instantiation condition reasons about the artifact context. Hence, a relation will not be recreated into the same artifact context or composition context unless the artifact context has changed, which is not allowed during maintenance. The creation of relations does not lead to the deletion of relations because a relation is only created if a conform artifact context, which satisfies the instantiation condition, and a conform relation composition, exists. Thus, any relation that is created conforms and therefore will not be deleted unless the artifact context changes or new relations are added.

4.3.2.2.2. Correctness The correctness of the sufficient batch localization algorithm depends on whether all non-conform relations that need to be deleted have been deleted and all dependencies that need to be captured have been captured by means of relations. The sufficient batch localization algorithm is correct if it terminates because the termination implies a fix-point, which further implies that no more relations can be deleted and created.

[8] This only holds for relations which types are defined to be automatically maintained.

4.3.2.2.3. Efficiency The sufficient batch localization strategy is not guaranteed to be efficient because in the worst case the number of iterations is $max(n, m) + 1$ where n is the number of relations, and m is the number of potential artifact contexts. Nevertheless, in the best case only 2 iterations are required because everything is processed in the first iteration, and nothing in the second iteration. The number of iterations depends on the ordering of the relations and relation types when applying the deletion of relations and the creation of relations. Finding an optimal ordering of relations and relation types can guarantee only a single iteration. This is exploited by the localization strategy shown in the next section.

4.3.2.3. Optimized Batch Localization Strategy

A second batch localization strategy is called optimized batch localization strategy, which overcomes the previously mentioned inefficiency of the sufficient batch localization strategy because it does not require a fix-point iteration. However, the missing of a fix-point iteration may influence the correctness of the strategy. To ensure correctness without requiring a fix-point iteration, a strict partial ordering over a set of relations and relation types is introduced. This ordering allows for requiring only a single iteration.

```
1  procedure optimizedBatchLocalization()
2    G_C := createCompositionGraph(R);
3    R' := getRootRelations(R); // with R' ⊆ R
4    S := topologicalSort(G_C, R'); // with S is an ordered set
5    forall (r_i ∈ S : mode_{R_t}(type_R(r_i)) ∉ {M})
6      updateCompositionContext(r_i);
7      updateArtifactContext(r_i);
8      if (mode_{R_t}(instanceOf_R(r_i)) ∈ {D, CD})
9        deleteRelation(r_i);
10     endif
11   endforall
12   G_{C_t} := createCompositionTypeGraph(R_t);
13   R'_t := getRootRelationTypes(R_t); // with R'_t ⊆ R_t
14   S_t := topologicalSort(G_{C_t}, R'_t); // with S_t is an ordered set
15   forall (r_{t_i} ∈ S_t : mode_{R_t}(r_{t_i}) ∈ {C, CD})
16     if (∃(r_{t_i}, r_{t_i}) ∈ E_{C_t})
17       while (true)
18         R' := createRelations(r_{t_i});
19         if (R' = ∅)
20           break;
21         endif
22       endwhile
23     else
24       createRelations(r_{t_i});
25     endif
26   endforall
27 endprocedure
```

Listing 4.9: Optimized batch localization strategy

The optimized batch localization strategy is shown in Listing 4.9. This operation is pretty similar to the sufficient batch localization strategy. The first difference is that it is not wrapped in a while-loop that requires a fix-point for termination. The second difference is that it creates a strictly partially ordered set of relations (Line 2-4) and a strictly partially ordered set of relation types (Line 12-14). The ordered set of relations is employed for traversing relations for update and deletion of relations (Line 5-11). The ordered set of relation types is employed for traversing relation types for the creation of relations (Line 15-26).

A set of relations $S \subseteq R$ is a strictly partially ordered set, if it is the result of topologically sorting a relation composition graph G_C, starting from relations $R' \subseteq R$ that do not depend on any other relations (Line 2-4).[9] Thus, at least one relation must exist that does not depend on any other relation.

The relation composition graph G_C is a directed acyclic graph over relations and its edges encode a specific dependency between relations (see Definition 4.3.1).

4.3.1 Definition (Relation Composition Graph) A relation composition graph G_C is a directed acyclic graph (V_C, E_C) with $V_C = R$ being a finite set of relations, which act as vertices of the graph, and $E_C \subseteq V_C \times V_C$, which is a finite set of directed edges. It further holds that $\forall (r, r') \in E_C, \exists c_M =$

[9] The *topologicalSort* operation is explained in Section B.3.2.

$(A_B, P_B, R_B, P_C) \in C_M, (r_R, r'') \in R_B : r = sub_{R_C}(match_{C_M}(c_M)) \wedge r' = r''$, which means that r depends on the existence of r'.

For example, as shown in Figure 4.21, the relation sara depends on SLogical2RLogical relations because these relations are in relation compositions of sara. The relation composition graph G_C is acyclic because the application megamodel does not allow relations that refer to themselves in relation compositions.

A set of relation types $S_t \subseteq R_t$ is a strictly partially ordered set if it is the result of topologically sorting a relation type composition graph G_{C_t}, starting from relation types $R'_t \subseteq R_t$ that do not depend on any other relation types (Line 12-14). Because of the topological sort operation, at least one relation type must exist that does not depend on any other relation type.

The relation type composition graph G_{C_t} is a directed graph over relation types and its edges encode a specific dependency between relation types (see Definition 4.3.2).

4.3.2 Definition (Relation Type Composition Graph) A relation type composition graph G_{C_t} is a directed graph (V_{C_t}, E_{C_t}) with $V_{C_t} = R_t$ being a finite set of relation types, which act as vertices of the graph, and $E_{C_t} \subseteq V_{C_t} \times V_{C_t}$, which is a finite set of directed edges. Edges exist if $\forall (r_t, r'_t) \in E_{C_t}, \exists c_S = (A_R, P_R, R_R, P_{C_t}) \in C_S, r_R \in R_R : r_t = sub_{R_{C_t}}(spec_{C_S}(c_S)) \wedge r'_t = type_{R_R}(r_R)$, which means that r_t is subordinate to r'_t.

For example, as shown in Figure 4.14, the relation type InvalidConnectorMultiplicity depends on the SConnector2RConnector relation type because SConnector2RConnector occurs in the relation type composition (specification) of InvalidConnectorMultiplicity. The relation type composition graph G_{C_t} is not acyclic. However, only cycles of size one are allowed. Otherwise the correctness of this strategy is cannot be ensured (see Section 4.3.2.3.2).

Because the relation type composition graph G_{C_t} may contain cycles of size one, the optimized strategy still requires a while-loop that requires a fix-point (Line 17-22). This is necessary due to the correctness of the strategy. Invoking the creation of relations of similar type requires multiple iterations because a previous iteration may create relations that occur in relation compositions of relations created in a subsequent iteration.[10] That this while-loop always terminates is shown in Section 4.3.2.3.1.

4.3.2.3.1. Termination The termination only needs to be shown for the while-loop in Line 17-22. This loop always has a fix-point and therefore terminates, because the create relation operation is strongly monotonously increasing and the number of artifact contexts is finite and constant in each iteration. In each iteration, only the number of potential relation compositions might increase but not the number of artifact contexts. Thus, a fix-point always exists after a finite number of iterations because the number of available artifact contexts is monotonously decreasing.

4.3.2.3.2. Correctness Because the optimized batch localization strategy does not provide an overall fix-point, the correctness of this strategy depends on the order of how update, delete and create operations are applied. In addition this strategy depends on the order in which relations are updated and deleted, and upon the order in which relations types are traversed for creating relations. This strategy is correct for the following reasons:

- The creation of relations can be applied after updating and deleting relations because the creation of relations does not influence the update and deletion of relations. This is true because relations that have been created are conform, and thus do not have to be updated nor deleted afterwards. The deletion of relations must be applied after updating relations because updating relations may cause their deletion. This is true because updating relations can influence the composition context and thus the conformance. Thus, applying update after delete would be incorrect.

- For each relation, first the update operations must be invoked and subsequently the delete operation. This interleaving is necessary because deleting a relation may require updating another relation that existentially depends on the deleted relation.

[10] Such relation types must always provide an alternative relation type composition that does not refer to itself. Otherwise, instances of this relation type will not be automatically created.

- Applying update and delete interleaved on any relation $r \in R$ requires a specific ordering of R because deleting a relation r that exists in the composition context of another relation $r' \in R$ requires subsequent invocation of update and delete on r' because the composition context of r' might change and thus r' might be deleted, too. First applying delete on r' and subsequently on r would be incorrect if each relation is only traversed once. Thus, the relations must be traversed in an order such that relations which exist in the relation compositions of other relations are traversed first; this is respected by the strictly partially ordered set S.
- Applying the create operation on any relation type only once also requires a specific ordering of the relation types R, which is given by the strictly partially ordered set of relation types S_t. This ordering is necessary because first creating relations of type $r'_t \in R_t$ that depends on another relation type $r_t \in R_t$ may lead to another result compared with the other way around. Thus, the relation types must be traversed in an order such that relation types which are used in the relation type composition (specification) of other relation types are traversed first.
- The relation type composition graph must not provide cycles except for cycles of size one. For cycles of size one, the create operation must be invoked on the same relation type until a fix-point is reached (Line 17-22). Else, the creation of relations might be missed.

4.3.3. Incremental Localization

The incremental localization, which is introduced in this section, can improve efficiency and scalability of the localization because it leverages change information to more precisely maintain the existence of relations.

4.3.3.1. Incremental Localization Operations

The localization operations for the incremental localization are slightly different from those for the batch localization because they are applied in a more fine-grained manner. Furthermore, the impact on the individual localization operations is defined and operationalized.

4.3.3.1.1. Update The incremental localization implements the update operation by means of two new update operations called *removeRelationCompositions* and *addRelationCompositions*, which substitute the *updateCompositionContext* operation of the batch localization (see Listing 4.1). The *update ArtifactContext* operation from the batch localization, which has been shown in Listing 4.2, is reused by the incremental localization.

```
1  procedure removeRelationCompositions(r, R'_C) : boolean // with R'_C ⊆ sup_R(r)
2    changed := false;
3    forall (r_C ∈ R'_C)
4      if (r_C ⊨ type_{R_C}(r_C) is not satisfied)
5        R_C := R_C \ {r_C};
6        changed := true;
7      endif
8    endforall
9    return changed;
10 endprocedure
```

Listing 4.10: Incremental localization operation: remove existing relation compositions

The *removeRelationCompositions* operation, which is shown in Listing 4.10, removes only those relation compositions of r that no longer conform. Furthermore, it only checks a specific subset of relation compositions $R'_C \subseteq sup_R(r)$ because it is assumed that only those relation compositions potentially do not conform. Thus, for any relation composition $r_C \in R_C$ the operation checks whether the relation composition conforms to its corresponding type. If at least one relation composition has been removed, the operation returns *true* and else $false$.

The *addRelationCompositions* operation, which is shown in Listing 4.11, creates all necessary relation compositions of a given relation r that do not yet exist but potentially should exist. It only considers a subset of relation type compositions $R'_{C_t} \subseteq sup_{R_t}(type_R(r))$ because it is assumed that only instances of those relation type compositions have to be added to r.

```
1  procedure addRelationCompositions(r, R'_{C_t}) : boolean // with R'_{C_t} ⊆ sup_{R_t}(type_R(r))
2    changed := false;
3    r_t := type_R(r);
4    forall (r_{C_t} ∈ R'_{C_t})
5      forall (r' ∈ createAllRelations(r_{C_t}, r_t, r))
6        if (similarArtifactContext(r, r'))
7          combine(r, r');
8          changed := true;
9        endif
10     endforall
11   endforall
12   return changed;
13 endprocedure
```

Listing 4.11: Incremental localization operation: add new relation compositions

Therefore, the operation iterates over any relation type composition $r_{C_t} \in R'_{C_t}$ (Line 4-11). For every relation type composition $r_{C_t} \in R'_{C_t}$ it creates a set of temporary relations by invoking the operation *createAllRelations*, which is already shown in Listing 4.13. For every temporary relation r', which has been created, the operation checks whether the temporary relation r' has an artifact context that is similar to the artifact context of r (Line 6). In this case, it further checks if the relation compositions of r' are not yet relation compositions of r. If so, the relation compositions of r' are added to r' by invoking the operation *combine* (Line 6-7).

4.3.3.1.2. Delete The delete operation of the incremental localization is similar to the delete operation of the batch localization as introduced in Listing 4.3.

4.3.3.1.3. Create The incremental localization reuses the create operation *createRelations*, introduced in Listing 4.4 by the batch localization. In addition, another create operation is introduced that is facilitated by the incremental localization strategy, which is shown in Listing 4.12.

```
1  procedure createRelations(r_t, r, R'_{C_t})
2    R' := ∅;
3    forall (r_{C_t} ∈ R'_{C_t})
4      R'' := createAllRelations(r_{C_t}, r_t, r);
5      forall (r'' ∈ R'')
6        if (∃r' ∈ R ∪ R' : r_t = type_R(r') ∧ similarArtifactContext(r', r''))
7          combine(r', r'');
8          R'' := R'' \ {r''};
9        endif
10     endforall
11     R' := R' ∪ R'';
12   endforall
13   //check instantiation condition on new relations
14   i_C := cond_{R_t}(r_t);
15   forall (r' ∈ R') //with ∀r' ∈ R' : r' ⊨ r_t
16     if (i_C = ϵ ∨ (i_C ≠ ϵ ∧ eval(i_C, r') = true))
17       R := R ∪ {r'};
18     endif
19   endforall
20 endprocedure
```

Listing 4.12: Incremental localization operation: create relations for a given relation type and relation type composition

The *createRelations* operation, shown in Listing 4.12, is a subset of the *createRelations* operation shown in Listing 4.4. Two additional parameters are thereby taken into account – a relation r and a set of relation type compositions $R'_{C_t} \subseteq sup_{R_t}(r_t)$. It is assumed that r has been created or updated. Updated means that a parameter of that relation has a new artifact as value. The set of relation type compositions R'_{C_t} is considered as the relation type compositions into which relations of type r_t might be created. Thus, the operation first creates a set of temporary relations R' with all necessary relation compositions (Line 3-12). For each temporary relation $r' \in R'$, the instantiation condition is evaluated, if available (Line 14-19). If the relation r' is considered as a conform relation, also satisfying the instantiation condition, it is added to the set of relations R.

The *createAllRelations* operation (see Listing 4.13) invoked in the previously shown operation (Line 4) is an adaptation of the *createAllRelations* operation shown in Listing 4.6.

```
1 procedure createAllRelations(r_Ct, r_t, r) : R' //with R' ∩ R = ∅
2   R' := ∅;
3   R'_C := createAllRelationCompositions(r_Ct, r); // adapted line
4   forall (r'_C ∈ R'_C)
5     r := createRelation(r_t, r'_C);
6     R' := R' ∪ createAllRelations(r_t, r, r_Ct, r'_C, ∅, ∅);
7   endforall
8   return R';
9 endprocedure
```

Listing 4.13: Incremental localization operation: create conform relations with relation type composition

This adaptation takes an additional parameter, the relation r, used only to create all possible relation compositions that include the relation r (Line 3). The rest of this operation is similar to the operation shown in Listing 4.6.

4.3.3.2. Incremental Localization Strategy

The major issues of batch localization are efficiency and the questionable scalability. The latter is questionable because it does not rely on changes but rather on whether the whole application megamodel is going to be maintained for deleting relations and creating new relations. The incremental localization strategy will depend on changes that actually occur, and thus the number of changes primarily affects efficiency and scalability. The incremental localization strategy is shown in Listing 4.14.

```
1  procedure incrementalLocalization()
2    //1. update and delete relations
3    E := getChanges(); // with E = (C_A, U_A, D_A, C_R, U_R, D_R)
4    while (E ≠ ∅)
5      updateAndDeleteRelations(E);
6      E' := getChanges();
7      E := E' \ E;
8    endwhile
9    //2. create relations
10   E := getChanges();
11   while (E ≠ ∅)
12     createRelations(E);
13     E' := getChanges();
14     E := E' \ E;
15   endwhile
16 endprocedure
```

Listing 4.14: Incremental localization strategy

The incremental strategy consists of two separate while-loops with both terminating if a fix-point is reached. The first while-loop is wrapped around updating and deleting relations incrementally (Line 4-8), and the second while-loop is wrapped around creating relations incrementally (Line 11-15). Both while-loops have the same fix-point, which is reached if no more changes were made ($E = \emptyset$). The while-loops are constructed as follows. In each iteration of the while-loops, the localization strategy first invokes the operation *updateAndDeleteRelations*, which updates and deletes a set of relations that are impacted by changes in E (Line 5), or the operation *createRelations* (Line 12), which creates a set of relations based on changes in E. Afterwards, the set of changes E is updated (Line 6-7 and 13-14). This is done by first querying an up-to-date set of changes E' and then removing already processed changes. If no more changes were made in an iteration of the while-loops, E will not contain any changes and a fix-point is reached.

The operation *updateAndDeleteRelations* is shown in Listing 4.15. The operation iterates through existing relations S in a strict partial order defined by the relation composition graph G_C. Furthermore, only relations are traversed that are at least defined as not manually maintained. Each relation $r \in S$, which is impacted by some change event in E, is going to be updated (Line 7-18) or deleted (Line 19-21). Updating a relation is achieved in three steps.

```
1  procedure updateAndDeleteRelations(E)
2    G_C := createCompositionGraph(R);
3    R' := getRootRelations(R); // with R' ⊆ R
4    S := topologicalSort(G_C, R');
5    forall (r ∈ S : mode_{R_t}(type_R(r)) ∉ {M})
6      // update relation
7      relationCompositionChanged := false;
8      R'_C := getImpactedRelationCompositions(r, D_R ∪ U_R);
9      if (removeRelationCompositions(r, R'_C))
10       relationCompositionChanged := true;
11     endif
12     R'_{C_t} := getImpactedRelationTypeCompositions(instanceOf_R(r), C_R ∪ U_R);
13     if (addRelationCompositions(r, R'_{C_t}))
14       relationCompositionChanged := true;
15     endif
16     artifactContextChanged := false;
17     if (relationCompositionChanged ∨ r ∈ U_R ∨ isArtifactContextImpacted(r, C_A ∪ U_A ∪ D_A))
18       updateArtifactContext(r);
19       if (mode_{R_t}(instanceOf_R(r)) ∈ {D, CD})
20         deleteRelation(r);
21       endif
22     endif
23   endforall
24 endprocedure
```

Listing 4.15: Incremental localization: update and delete relations

In the first step, relation compositions R'_C that are potentially no longer conforming are removed from the considered relation r by invoking the operation *removeRelationCompositions* (Line 9), shown in Listing 4.10. The relation compositions $r'_C \in R'_C$ are estimated by analyzing the impact of changes in E by invoking the operation *getImpactedRelationCompositions*, shown in Listing 4.16.

```
1  procedure getImpactedRelationCompositions(r, R') : R'_C // with R'_C ⊆ sup_R(r)
2    R'_C := ∅;
3    forall (r' ∈ R')
4      forall (r_C ∈ sup_R(r))
5        c_M := match_{R_C}(r_C); // with c_M = (A_B, P_B, R_B, P_C)
6        if (∃(r_R, r'') ∈ R_B : r'' = r')
7          R'_C := R'_C ∪ {c_M};
8        endif
9      endforall
10   endforall
11   return R'_C;
12 endprocedure
```

Listing 4.16: Update and delete relations: relation compositions impacted by changes

The *getImpactedRelationCompositions* operation takes a relation r and a set of relations R' as parameters, which have been deleted or updated. For every relation $r' \in R'$ and relation composition r_C, which is superior to the given relation r, it checks if r' is in the match of r_C (Line 5-8). Thus, the result is a set of relation compositions R'_C of r, which are impacted by relations in R'.

In the second step, instances of relation type compositions R'_{C_t} might have to be added to the relation r under consideration, realized by invoking the operation *addRelationCompositions* (Line 13) as shown in Listing 4.11. The relation type compositions in R'_{C_t} are estimated by analyzing the impact of changes in E (Line 12). The impact is analyzed by the operation *getImpactedRelationTypeCompositions*, shown in Listing 4.17.

The *getImpactedRelationTypeCompositions* operation analyzes the impact of created and updated relations R' to a given relation type $r_t \in R_t$, which is the type of the considered relation r. Therefore, it iterates over all relations $r' \in R'$ and relation type compositions r_{C_t}, which are superior to the given relation type r_t. In each iteration, r_{C_t} is added to the resulting set of relation type compositions R'_{C_t}, if the relation type composition specification c_S of r_{C_t} contains a relation role whose type is similar to an impacted relation r (Line 5-8). Therefore, the result is a set of relation type compositions R'_{C_t} that are impacted by changed relations.

In the third step, the artifact context of the considered relation r is updated by invoking the *update ArtifactContext* operation (Line 18), shown in Listing 4.2. However, r only needs to be updated if a relation composition superior to r has actually changed, the considered relation r is directly impacted

```
1  procedure getImpactedRelationTypeCompositions(r_t, R') : R'_{C_t}  // with R'_{C_t} ⊆ sup_{R_t}(type_R(r))
2    R'_{C_t} := ∅;
3    forall (r' ∈ R')
4      forall (r_{C_t} ∈ sup_{R_t}(r_t))
5        c_S := spec_{R_{C_t}}(r_{C_t}); // with c_S = (R_R, A_R, P_R, P_{C_t})
6        if (∃ r_R ∈ R_R : type_R(r') = type_{R_R}(r_R))
7          R'_{C_t} := R'_{C_t} ∪ {r_{C_t}};
8        endif
9      endforall
10   endforall
11   return R'_{C_t};
12 endprocedure
```

Listing 4.17: Update and delete relations: relation type compositions impacted by changes

by a changed artifact, or the artifact context of r is impacted by artifacts in $C_A \cup U_A \cup D_A$ that have been created, updated or deleted. The artifact context of a relation r is directly impacted if an artifact has been deleted that was connected to a parameter of the relation r, which results in a change event $r \in U_R$. Checking if changed artifacts impact the artifact context of a relation r is implemented by the operation *isArtifactContextImpacted*, shown in Listing 4.18.

```
1  procedure isArtifactContextImpacted(r, A') : boolean
2    r_t := type_R(r);
3    forall (a' ∈ A')
4      if (∃ p_t ∈ connect_{R_t}(r_t) : val_{P_t}(p_t) = type_A(a'))
5        return true;
6      endif
7      if (∃ p ∈ connect_R(r) : a' ∈ val_P(p))
8        return true;
9      endif
10     i_C := cond_{R_t}(r_t);
11     if (i_C ≠ ϵ)
12       forall ((A'_t, p'_t) ∈ scope_{I_C}(i_C))
13         if (type_A(a') ∈ A'_t)
14           return true;
15         endif
16       endforall
17     endif
18   endforall
19   return false;
20 endprocedure
```

Listing 4.18: Update and delete relations: is relation's artifact context impacted by changes

A relation r is impacted by a set of created, updated and deleted artifacts A' due to the following reasons. If the type of the relation has a parameter type, whose value is the type of an artifact $a' \in A'$, the artifact context of the relation r is impacted (Line 4-6). If it has a parameter whose values contain artifacts in A', the artifact context of a relation r is impacted (Line 7-9). If the type provides an instantiation condition i_C with a scope that relates to an artifact type, which is the type of an artifact $a' \in A'$, the artifact context of a relation r is also impacted (Line 10-17).

Finally, the considered relation r might have to be deleted, accomplished by invoking the *deleteRelation* operation (Line 20), introduced in Listing 4.3. However, a relation r is only considered for deletion if its type is set to be automatically deleted, the artifact context of r has been changed, or the artifact context of r is impacted by artifacts U_A that have been updated.

After updating and deleting relations, the incremental localization strategy tries to create necessary relations, by triggering the *creationRelations* operation, shown in Listing 4.19.

The *createRelations* operation iterates over all relation types in a strict partial order defined by the relation type composition graph G_{C_t}. However, only relation types are considered that are defined to be automatically created. For each of these relation types $r_t \in S_t$ it is first checked if a new relation composition might exist into which a relation of type r_t can be created (Line 6-7). In this case, the operation *createRelations*, which has been shown in Listing 4.12, is invoked (Line 7). The set of relation type compositions R'_{C_t}, which is used as parameter, is estimated by invoking *getImpactedRelationTypeCompositions*. This operation has been shown in Listing 4.17.

Afterwards, the *createRelations* operation is invoked (see Listing 4.4) on the considered relation type

```
1  procedure createRelations(E)
2    R'_t := getRootRelationTypes(R_t); // with R'_t ⊆ R_t
3    G_Ct := createCompositionTypeGraph(R_t); // with G_Ct = (V_Ct, E_Ct)
4    S_t := topologicalSort(G_Ct, R'_t);
5    forall (r_t ∈ S_t : mode_Rt(r_t) ∈ {C, CD})
6      R'_Ct := getImpactedRelationTypeCompositions(r_t, C_R ∪ U_R);
7      createRelations(r_t, r', R'_Ct);
8      if (isArtifactContextImpacted(r_t, C_A ∪ U_A ∪ D_A))
9        createRelations(r_t);
10     endif
11   endforall
12 endprocedure
```

Listing 4.19: Incremental localization: create relations

r_t, if changes to artifacts can impact the creation of relations of that type (Line 8-9). The impact is analyzed by invoking $isArtifactContextImpacted$, which is shown in Listing 4.20, on the considered relation type and all changed artifacts in $C_A \cup U_A \cup D_A$ (Line 8).

```
1  procedure isArtifactContextImpacted(r_t, A') : boolean
2    r_t := type_R(r);
3    forall (a' ∈ A')
4      a'_t := type_A(a');
5      if (∃p_t ∈ connect_Rt(r_t) : val_Pt(p_t) = a'_t)
6        return true;
7      endif
8      i_C := cond_Rt(r_t);
9      if (i_C ≠ ϵ)
10       forall ((A'_t, p'_t) ∈ scope_IC(i_C))
11         if (a'_t ∈ A'_t)
12           return true;
13         endif
14       endforall
15     endif
16   endforall
17   return false;
18 endprocedure
```

Listing 4.20: Create relations: is the artifact context of a relation type impacted by changes

The $isArtifactContextImpacted$ operation is similar to the operation that has been shown in Listing 4.18. The only difference is this operation takes a relation type as parameter instead of a relation. Thus, it only analyzes whether types of changed artifacts have an impact on the creation of relations of the considered relation type.

4.3.3.2.1. Termination Termination is a relevant property of the incremental localization strategy because it uses two while-loops that require the existence of fix-points in order to terminate. These two while-loops always have a fix-point for the same reason that deleting and updating relations, and creating relations of the sufficient and optimized batch localization strategies, have a fix-point (cf. Section 4.3.2.2.1).

4.3.3.2.2. Correctness The incremental localization strategy uses a similar order for updating, deleting and creating relations as that used by the optimized batch localization strategy. Thus, it first completely updates and deletes existing relations and then continues by creating all necessary relations. It further uses the same strict partial order for traversing relations, when updating and deleting relations, and relation types, when creating relations.

However, the correctness of the incremental localization strategy also depends on whether all necessary update, delete and create operations are triggered. This strategy is correct for the following reasons:

- The incremental localization strategy updates and deletes all necessary relations because all necessary change events are correctly processed. This is true because in the first iteration all changes made by an application developer are processed and in any subsequent iteration changes of the previous iteration are processed.

Taking Figure 4.21 as an example, in the first iteration only one change event PG $\in U_A$ is available. This change event triggers the deletion of the relation slrl2, which creates a change event slrl2 $\in D_R$ that is processed in the next iteration. In the second iteration, this change event triggers the deletion of the relation compositions from sara to slrl2 and from scrc1 to slrl2. These changes will raise two further change events scrc1 $\in U_R$ and sara $\in U_R$. In the third iteration, these change events will first trigger the deletion of the relation scrc1 and subsequently the removal of the relation composition from sara to scrc1. These changes will raise two further change events scrc1 $\in D_R$ and sara $\in U_R$. Finally, in the fourth iteration no more changes are conducted.

- The incremental localization strategy creates all necessary relations because all necessary change events are correctly processed. This is true because in the first iteration all changes made by an application developer and all changes caused by the update and deletion of relations are processed. In any subsequent iteration, the changes of the previous iteration are processed.

  For a better illustration, Figure 4.22 is taken as an example. Assume that an application developer has just changed the artifact PG such that it once more depends on the artifact PolicyGateway. Then, the iteration of the second while-loop has only one change event in E, – PG $\in U_A$ – that does not cause any update or deletion in the first while-loop. Thus, the first while-loop terminates and the second is initiated. This change event will cause the creation of the relation slrl2. The second iteration has one change event, which is slrl2 $\in C_R$. This change event first triggers the creation of scrc1 and subsequently the creation of the relation composition from sara to slrl2. Thus in the third iteration, two change events are available, which are scrc1 $\in C_R$ and sara $\in U_R$. Both changes will result in the creation of the composition context from sara to scrc1. The fourth iteration only has a single change event, which is sara $\in U_R$. However, this change does not cause any further change because all necessary relations and relation compositions already exist.

4.4. Summary

In this chapter, the concept of a hierarchical megamodel has been extended to the concept of a dynamic hierarchical megamodel. A dynamic hierarchical megamodel is similar to a hierarchical megamodel except it includes a more detailed definition of relation types and relations. This extension is necessary to realize the automated maintenance of relations, which promotes the demonstrated approach as a traceability approach.

A relation type has been extended by means of two additional concepts – an instantiation condition, and a relation type composition specification, which is related to a relation type composition. An instantiation condition has been defined as an abstract representation of a model operation that is used to reason additionally about the correctness of an artifact context of a relation of a certain type. Thus, it is used to decide whether a relation should exist in a certain artifact context, because it satisfies a certain structural condition, or not. A relation type composition specification provides additional details to a relation type composition. Thus, the relation composition of a relation can be exactly specified by means of a pattern over relations and artifacts. This additional specification is leveraged to automatically reason about the correctness of the relation composition of a relation.

Based on a dynamic hierarchical megamodel, localization has been introduced. Localization is a mechanism to automatically maintain the existence of relations at any level of detail. Therefore, it uses the concepts of instantiation conditions and relation type composition specifications. Two different localizations have been introduced in this chapter – batch localization and incremental localization. The batch localization can be applied as it is whereas the incremental localization requires additional change events to maintain only those relations that have impacted by changes. If change information is available, the incremental localization should be preferred due to scalability reasons.

By now, the dynamic hierarchical megamodel and the localization mechanism can be employed as a traceability approach in the context of MDE, including the automated maintenance of relations representing physical dependencies in MDE applications. The next chapter will demonstrate additional extensions, making this approach applicable in the context of model management. It will use the concepts shown in this chapter to enable complex compositions of heterogeneous model operations at any level of detail.

5. Execution: Model Management

Contents

* * *

In the previous chapter, the dynamic hierarchical megamodel was introduced, enabling traceability in the context of MDE, which includes automated maintenance of dependencies.

In this chapter, the dynamic hierarchical megamodel is further extended to enable model management with respect to the composition and application of heterogeneous model operations. This extension is called an executable and dynamic hierarchical megamodel. The executable and dynamic hierarchical megamodel enables the specification of complex compositions of heterogeneous model operations and their subsequent (re-)application. The (re-)application can be conducted in different ways. The approach supports (re-)applying individual model operations and (re-)applying all model operations that have been impacted by changes.

5.1. Conceptual Introduction

The executable and dynamic hierarchical megamodel is introduced in two steps. Firstly, we shall consider the capability of specifying complex compositions of heterogeneous model operations (see Section 5.1.1). Secondly, the capability of (re-)applying compositions of heterogeneous model operations is intorduced, based on specification(see Section 5.1.2).

5.1.1. Specification of Compositions of Heterogeneous Model Operations

The dynamic hierarchical megamodel enables the automated maintenance of relations at any level of detail. Additionally, these relations can exist in compositions to other relations captured by the relation composition concept, which is explicitly specified by means of relation type compositions (specifications) between corresponding relation types. The composition of heterogeneous model operations can be realized in two different ways.

Firstly, model operations are composed by means of relation compositions, called the context composition of model operations. Thus, model operations use the application of other model operations, or just simple relations, as their composition context. This kind of composition is known from declarative model transformations (e.g., TGG or QVT relation).

Secondly, model operations are composed by means of sharing artifacts. For example, if the target of one model operation is also the source of another model operation, these two model operations are considered to be in a composition called data-flow composition. This is a classical composition and is prominent in workflow or model transformation chain approaches.

The basic idea of the executable and dynamic hierarchical megamodel is to consider certain relations as executable units (relations), which is similar to the application (or potential application) of a model operation in a specific location. Therefore, a relation type can be considered as an abstract representation of a model operation by means of a new concept called execution operation. An execution operation is syntactically similar to an instantiation condition. However, an execution operation is employed as an abstraction for any kind of heterogeneous model operation, and not just for model operations that are used to reason about the correctness of a relation. Furthermore, a relation not only captures the application of a model operation, but also that a model operation can or will be applied. As a consequence, applying a model operation is similar to first creating a relation into a concrete location and subsequently executing the relation in that location.

In combination with instantiation conditions, model operations that are represented by relation types do not need to implement navigation concerns because this is already implemented by the localization and the instantiation conditions of relation types. Thus, the clear separation between specifying the required application context and the specification of the actual model operation task is leveraged, enhancing the principle of loose coupling and high cohesion, as stated in [176], but for model operations.

Combining this perspective on relations and relation types, the context composition of model operations is realized by means of relation compositions. Thus, the context composition of heterogeneous model operations is specified by means of relation type compositions and their specifications as shown in the previous chapter. The data-flow composition of model operations does not require any additional concept because it is realized by overlapping artifacts connected to parameters of relations. Nevertheless, the specification of data-flow compositions requires an additional concept, introduced in this chapter as module. A module is a specialization of a relation type. It allows the explicit specification of data-flow compositions based on relation types in the context of a module.

In [152] and [151] a previous version of this approach has been published. In [152] only data-flow compositions have been realized. However, in that version a data-flow composition could not be specified on the type level, but rather it implicitly exists between relations that share artifacts. In [151] it has been shown how heterogeneous model operations can be specified in context compositions. Nevertheless, that approach only supports composition contexts with a single relation, which does not allow the specification of complex context compositions. Furthermore, that approach does not support an explicit specification of data-flow compositions.

5.1.2. (Re-)Application of Compositions of Heterogeneous Model Operations

In the introduction, it was explained that applying every model operation on its own is tedious – in complex application scenarios, the number of model operations that have to be (re-)applied is rather high. Thus, application developers have to be supported by treating a set of model operations, which exist in compositions, as a coherent and applicable model operation. This can decrease the number of model operations that have to be (re-)applied by application developers. Therefore, the executable and dynamic hierarchical megamodel introduces a facility called execution, a process that interleaves localization and the application of a set of heterogeneous model operations by executing executable relations.

Furthermore, physical artifacts in MDE applications are subject to continuous change, and already applied model operations may be invalidated due to changes to their sources. In order to re-validate these model operations, they have to be re-applied. This re-application is necessary to once more bring the output of a model operation into a consistent state.

Because the application of a model operation is captured by means of an executable relation, a model operation can be re-applied by re-executing the same executable relation again. However, the number of invalidated model operations may be high in complex application scenarios, and it is tedious and error-prone to manually re-apply all necessary model operations. Missing the re-application of certain model operations may lead to further, even undetected, inconsistencies. The execution also supports application developers in this case. It provides the facility to automatically re-apply all model operations that have been directly or indirectly invalidated by means of changes.

In [152] the idea of executing a dynamic hierarchical megamodel has already been published, but only applied to data-flow compositions. Whilst that approach could only apply model operations in a batch-like fashion, this chapter demonstrates an extension which also supports automated (re-)application. In [151] the approach, shown in [152], has been extended to support the application of context compositions.

However, that version does also not support the automated (re-)application of heterogeneous model operations.

5.1.3. Overview

Extending the model management framework by means of executable and dynamic hierarchical megamodels provides MDE with model management capabilities, resulting in further use cases to a configuration developer and an application developer. These use cases are shown and highlighted in Figure 5.1.

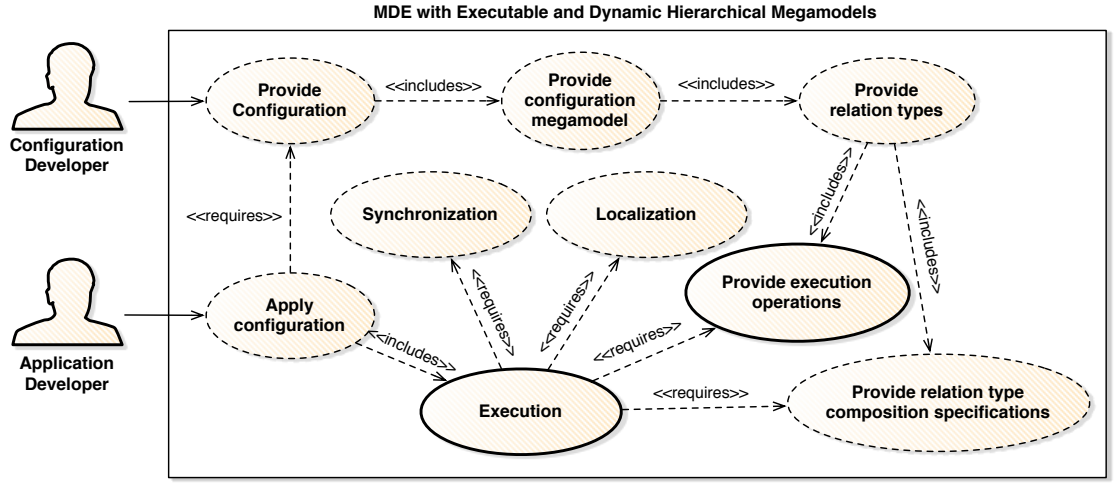

Figure 5.1.: Additional use cases for the application of execution and executable and dynamic hierarchical megamodels

A configuration developer is able to integrate heterogeneous model operations by specifying execution operations that are related to relation types. Furthermore, a configuration developer is able to specify compositions of heterogeneous model operations by specifying relation type compositions and specifications or modules. An application developer only needs to trigger execution when necessary.

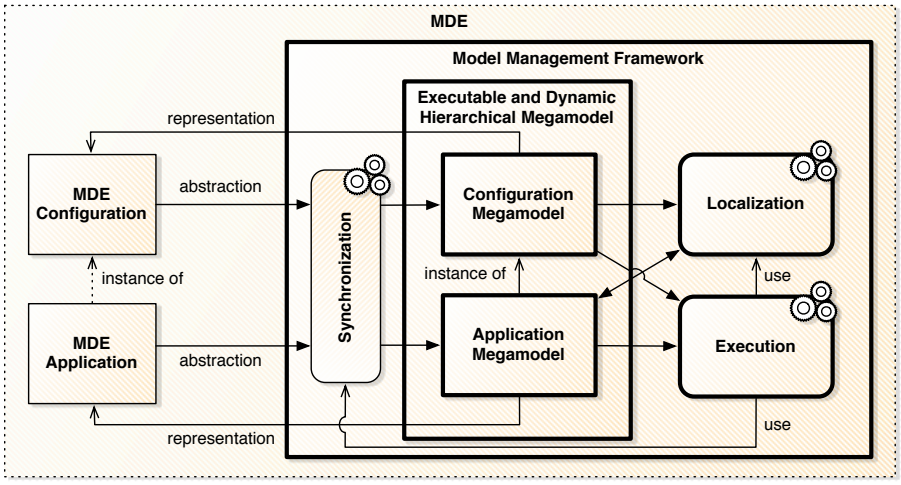

Figure 5.2.: Conceptual integration of executable and dynamic hierarchical megamodels

The effect of the executable and dynamic hierarchical megamodel on the model management framework is shown in Figure 5.2. It shows that the executable and dynamic hierarchical megamodel comes with adaptations to the configuration and application megamodel from the hierarchical megamodel (see

Section 5.2). Furthermore, the localization of the previous chapter has to be slightly adapted so as to function correctly with executable relations, and to be correctly interleaved with execution.

Execution is the second part of this chapter and covers everything regarding the (re-)application of model operations and impact analysis (see Section 5.3). Execution uses information from a configuration megamodel and an application megamodel in order to determine the set of model operations that have to be (re-)applied.

5.2. Executable and Dynamic Hierarchical Megamodels

The executable and dynamic hierarchical megamodel is explained in two steps. Firstly, the extended configuration megamodel will be introduced including all concepts and a formal definition (see Section 5.2.1). Secondly, the extended application megamodel is introduced including all concepts and a formal definition (see Section 5.2.2).

5.2.1. Configuration Megamodels

The configuration megamodel of the executable and dynamic hierarchical megamodel is a simple extension of the configuration megamodel of the previous chapter (see Figure 4.3). Figure 5.3 shows the metamodel of the extended configuration megamodel. The only new concept of this configuration megamodel is the execution operation (ExecutionOperation), which is a specialization of a model operation representation (ModelOperationRepresentation).

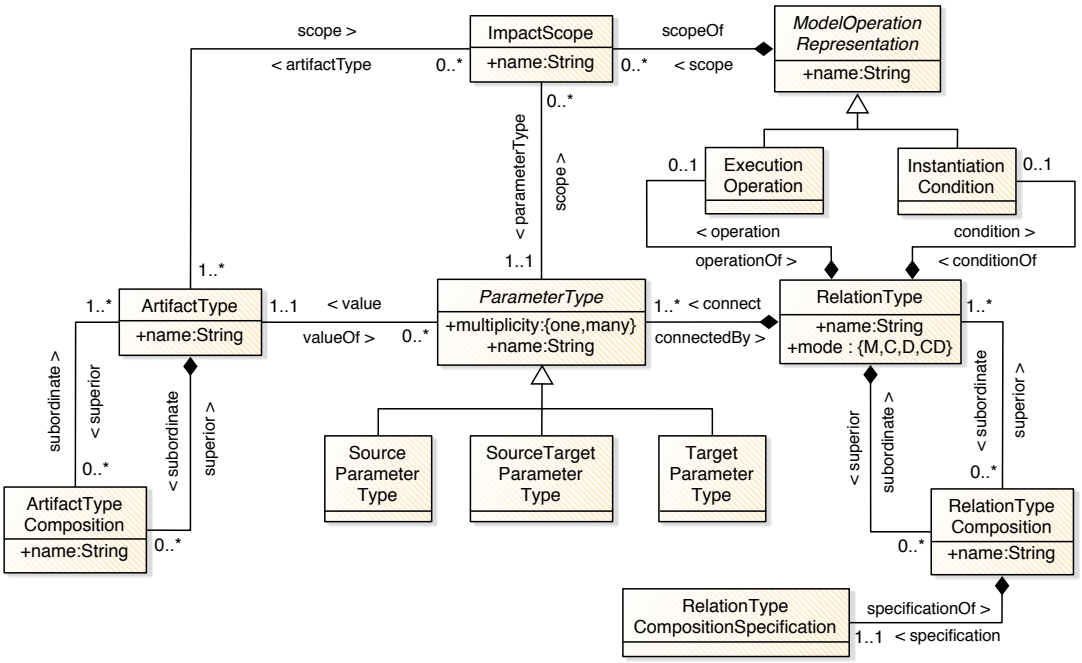

Figure 5.3.: Extended metamodel of the configuration megamodel

5.2.1.1. Execution Operations

An execution operation is an abstract representation of a model operation provided by an MDE configuration. An execution operation is always related to a relation type (operationOf association). If a relation type is related to an execution operation (operation association), the relation type is declared to be the signature of the related model operation. Thus, parameter types and artifact types related to such relation types have a slightly different semantic, since before executing a relation of such a type for the first time, the artifact(s) related to the target parameter does not yet exist.

A source parameter type defines an input of the model operation related to the relation type. Thus, the model operation expects a physical artifact of the defined type as input. Furthermore, the physical artifact will not be modified by the model operation. A target parameter type defines an output of the model operation related to the relation type. Thus, one can expect a physical artifact of the defined type as output. If the model operation is applied for the first time, the physical artifact will be initially created. If the model operation is re-applied, the physical artifact is already available and will be overwritten or updated, depending on the capabilities of the actual model operation. A source & target parameter type defines another input of the model operation related to the relation type. Thus, the model operation expects a physical artifact of the defined type as input. In addition, one can expect that the physical artifact will be overwritten or updated when (re-)applying the model operation.[1]

A relation type with an execution operation has an effect on the conformance of an instantiation condition that can be connected to the same relation type. Model operations that are represented by instantiation conditions must not provide inputs for physical artifacts whose types are represented by the value of target parameter types of the relation type. This is necessary because a physical artifact, whose type is the value of a target parameter type, is created when executing the relation. Thus, they cannot be considered when reasoning about the validity of relations by using instantiation conditions.

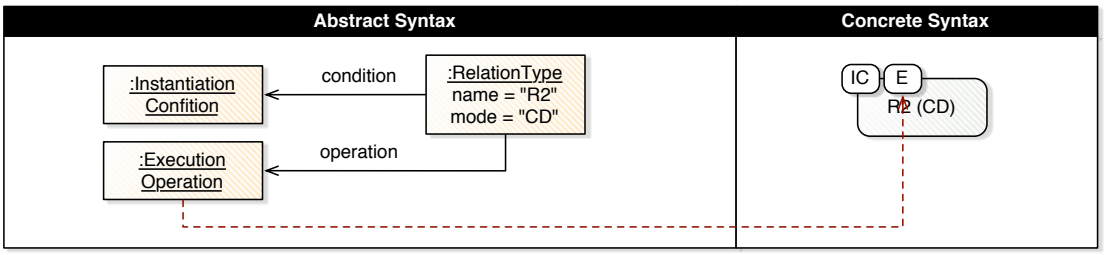

Figure 5.4.: Concrete syntax of relation types with execution operations

The concrete syntax of execution operations is illustrated in Figure 5.4 and is almost similar to the illustration of instantiation conditions. A label named E is shown to the top-left of a relation type, if the relation type is related to an execution operation. The figure shows a relation type R2 with an execution operation and an instantiation condition related.

To better illustrate the concept of execution operations, Figure 5.5 shows a relation type UML2Java that is related to an execution operation. This relation type represents the signature of a model operation taken from the application example, introduced in the case study in Section 2.2.2.

Figure 5.5.: Illustration of a relation type UML2Java with execution operation attached

The model operation, represented by the relation type UML2Java, generates Java code into a given project from a set of UML packages. Thus, the relation type UML2Java has an artifact type UMLPackage as source. The multiplicity of the source parameter type is set to many because the code generation can process multiple physical artifacts of type UMLPackage. Project is source & target because it is required as location for generating the code.[2]

Due to the abstraction by means of execution operations, it does not matter whether the model operation is actually implemented in ATL, QVT, or Java, nor how the model operation is implemented. It is only important that the model operation conforms to the relation type of the execution operation,

[1] Section C.2.1 shows several examples of model operations that can be represented by means of relation types and execution operations.

[2] The artifact types Project, Folder and File are used to define the types of artifacts that represent the filesystem (see Section A.2.3.2).

meaning that the model operation must take a set of physical artifacts of type UMLPackage as source and a physical artifact of type Project that is also used as target for the code generation.

5.2.1.2. Impact Scopes for Execution Operations

It has already been explained that the execution will use an impact analysis to determinate a set of executable relations that have to be re-executed due to changes. However, this only works if the model operations, represented by execution operations, only reason within the scope of the artifact context of relations. Nevertheless, this assumption is too restrictive as in the case of instantiation conditions. Figure 5.5 shows a relation type that is the signature of a coarse-grained model operation. The impact analysis of the execution might miss the re-execution of relations of that type because the model operation reasons about physical artifacts that are outside the scope of physical artifacts represented by artifacts in the artifact context (e.g., UMLClass, UMLAssociation, etc.).

If a model operation is implemented to reason outside the scope of an artifact context of a relation of a certain type, the execution operation must provide a set of impact scopes. The impact scope for execution operations is similar to the impact scope for instantiation conditions.

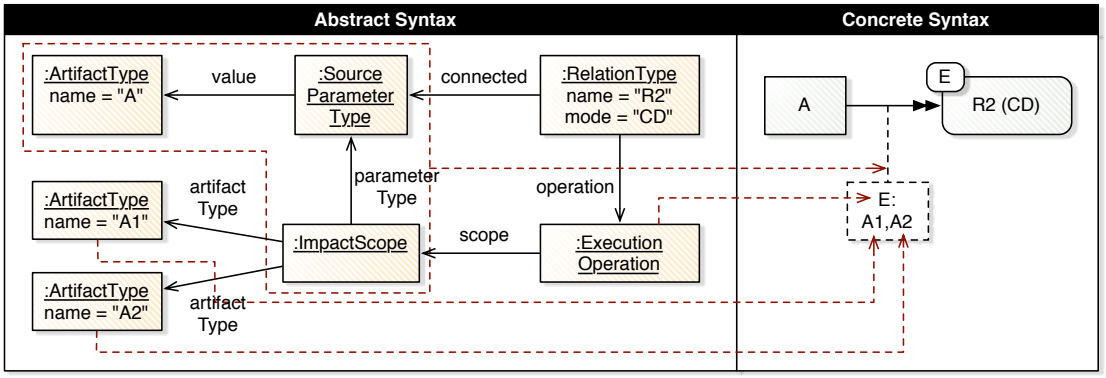

Figure 5.6.: Concrete syntax of impact scopes for execution operations

The concrete syntax of impact scopes for execution operations is shown in Figure 5.6. The impact scope for execution operations is visualized similarly to the impact scope for instantiation conditions. The only difference is that the dashed rectangle contains a label "E:" as prefix instead of "IC:" in order to distinguish between them.

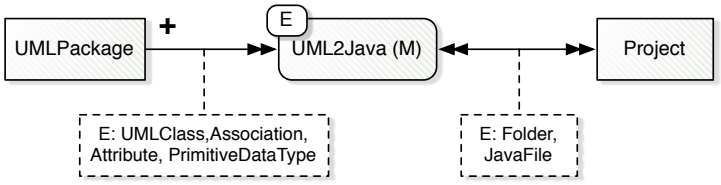

Figure 5.7.: Illustration of a relation type UML2Java with execution operation and impact scopes

To correctly analyze the impact of changes for relations of type UML2Java, the relation type is extended with necessary impact scopes for the execution operation as shown in Figure 5.7. It shows that the relation type now explicitly captures that changes to physical artifacts of type UMLClass, Association, Attribute or PrimitiveDataType impact the source parameter type and that changes to physical artifacts of type Folder and JavaFile impact the source & target parameter type. This means that changes to all artifacts of types that are specified in the related impact scopes and that are directly or indirectly subordinate to an artifact of type UMLPackage potentially impact a relation of type UML2Java.

5.2.1.3. Specification of Context Compositions

Still, the relation type, as extended in Figure 5.7, is problematic if UMLCLass artifacts are going to change frequently. If the implementation of the model operation or the technology does not support incremental re-generation, changing only one UMLClass artifact will cause a complete re-generation. A complete re-generation may overwrite all changes that were made afterwards to the generated Java code.

There are kinds of model operations that support the specification of incremental model operations, e.g., TGG [68]. However, a model operation is not necessarily incremental because their technology or language does not support that. If the technology is not incremental, the model operation must be explicitly implemented to be incremental, which is a complex task. Nevertheless, if model operations can be decomposed into fine-grained model operations, changes can be propagated which are more fine-grained. Thus, it is not required to re-apply a coarse-granular model operation but rather only those fine-granular model operations that are actually impacted by changes. However, the decomposition of model operations requires a subsequent composition to once again treat them as a single coherent and executable unit.

The executable and dynamic hierarchical megamodel allows decomposition of model operations and subsequent composition of them by means of the composition of relations and relation types.

Figure 5.8.: Illustration of a relation type UML2Java without execution operation and impact scopes

Figure 5.8 shows a modified version of the relation type as shown in Figure 5.5. This UML2Java relation type has the same signature but is no longer related to an execution operation. Instead, five additional relation types are introduced in the following, which will be composed and related to more fine-grained model operations doing the same overall task in combination.

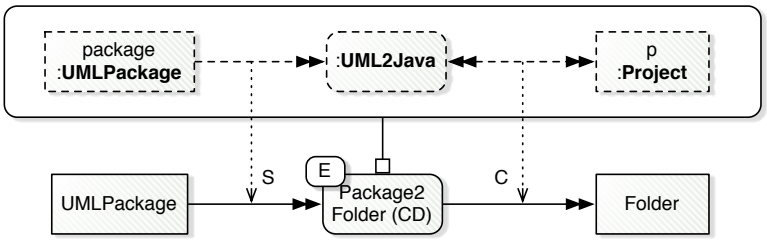

Figure 5.9.: Illustration of a relation type Package2Folder composed into UML2Java

The first relation type is called Package2Folder and is illustrated in Figure 5.9. This relation type defines the signature of a model operation that creates a Folder from a UMLPackage. If a relation of type UML2Java exists, a relation of type Package2Folder will be created for every UMLPackage that is source to the relation of type UML2Java. Thus, for every UMLPackage in that context, a Folder, which represents the UMLPackage, will be created in the Project that is source & target of the relation of type UML2Java. Every of these relations of type Package2Folder will be context composed into the relation of type UML2Java.

The second relation type is called Class2JavaClass and is shown in Figure 5.10. This relation type defines the signature of another model operation that is responsible for only creating a single JavaClass from a UMLClass into a Folder. By specifying a context composition for Class2JavaClass, as shown in the figure, a JavaFile is created into a Folder, which is created from a UMLPackage by Pacakge2Folder, for every UMLClass of that UMLPackage.

The third relation type is called Assoc2JavaClass and is shown in Figure 5.11. This relation type defines the signature of a model operation that is responsible for updating a given JavaFile with a piece of code that defines an Association between two physical artifacts of type UMLClass. The context composition of Assoc2JavaClass further defines that the JavaFile must be created from the UMLClass (src), which is

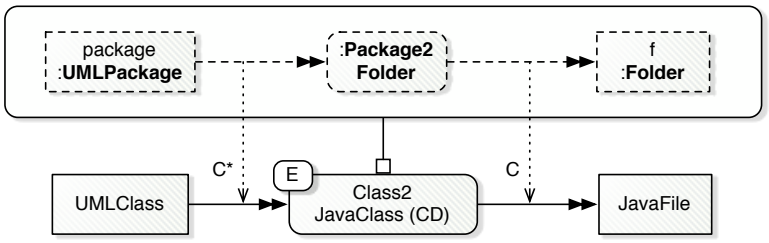

Figure 5.10.: Illustration of a relation type Class2JavaClass composed into Package2Folder

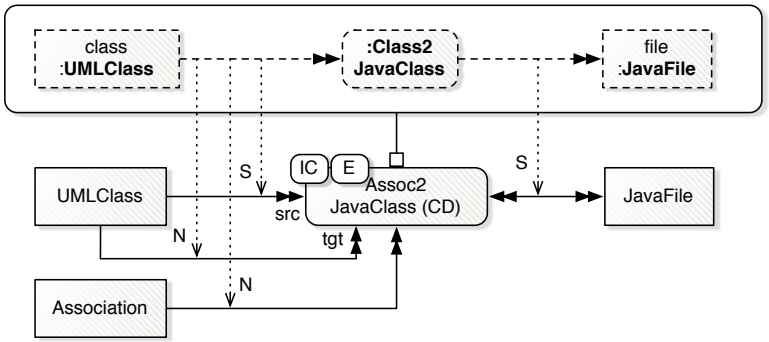

Figure 5.11.: Illustration of a relation type Assoc2JavaClass composed into Class2JavaClass

connected to Assoc2JavaClass (defined by the Class2JavaClass in the relation type composition specification and the respective parameter type connectors). Thus, for every UMLClass (tgt) and Association, which are defined to be siblings of the UMLClass (src), a piece of association code is generated into the JavaFile.

The instantiation condition ensures that the Association artifact uses the UMLClass, connected via src, as source and the UMLClass, connected via tgt, as target.

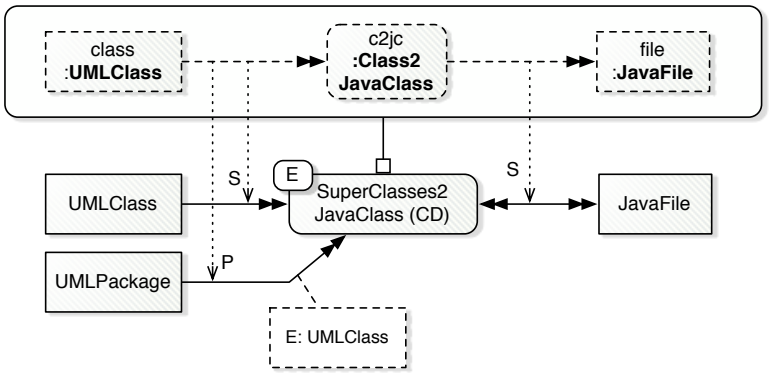

Figure 5.12.: Illustration of a relation type SuperClasses2JavaClass composed into Class2JavaClass

The fourth relation type is called SuperClasses2JavaClass and defines the signature of a model operation that extends a JavaFile with a piece of code that encodes the inheritance relationships of the java class represented by the JavaFile. Due to the relation type composition and specification, the JavaFile that is extended must be generated from the UMLClass, which is ensured by Class2JavaClass and the parameter type connectors. For the generation of the inheritance code, the relation type SuperClasses2Java also takes an artifact type UMLPackage as source. The model operation is generating code for every UMLClass in that UMLPackage that is a direct or indirect super class of the given UMLClass. The source parameter

type (UMLPackage) of SuperClasses2JavaClass and the execution operation has an impact scope, which is related to UMLClass artifacts. This is necessary, because a UMLClass can have multiple other UMLClass artifacts that exist in the context of the related UMLPackage artifact.

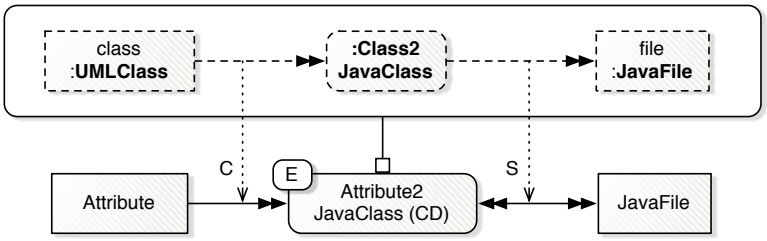

Figure 5.13.: Illustration of a relation type Attibute2JavaClass composed into Class2JavaClass

The last relation type Attribute2JavaClass, as shown in Figure 5.13, is the signature of a model operation that generates a piece of Java code from a single Attribute artifact into a given JavaFile. The context composition additionally specifies that for every JavaFile, Java code for every Attribute of the UMLClass is generated, corresponding to the JavaFile. This is ensured by the Class2JavaClass relation type in the relation type composition specification and the parameter type connectors.

Thus, all these relation types can be considered as a coherent unit that provides the same model operation as the coarse-grained model operation represented by the execution operation and relation type shown in Figure 5.7. However, changes to individual UMLClasses, Association or Attribute artifacts will not cause a complete re-application but rather the re-application of fine-grained model operations that are actually impacted.

5.2.1.4. Specification of Data-Flow Compositions using Modules

Various model operations can also be composed by means of data-flow compositions. Up to now, there is no explicit concept in the configuration megamodel to specify data-flow compositions. Currently, data-flow compositions occur in application megamodels by creating executable relations that have shared artifacts. This section provides another concept called module that allows for specifying data-flow compositions based on relation types in a configuration megamodel.

To explain the concept of a module, a simple example from the case study in Section 2.2.2 is taken. The goal is to automatically transform a UMLPackage into an SQLFile that contains code that can be used to automatically setup an SQL database. This code generation is obtained in two steps. First, a UMLPackage is transformed into a Schema, which is a model of a relational database. Second, this Schema is used to generate an SQLFile. Thus, both model operations can be considered as a single operation that can be composed via data-flow composition.

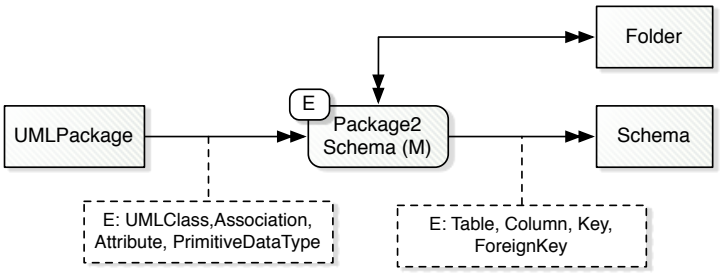

Figure 5.14.: Illustration of a relation type Package2Schema

Figure 5.14 shows a relation type Package2Schema, which is the signature of a model operation that transforms all UMLClass artifacts of a single UMLPackage into a Schema. The Folder, which is connected as source & target, acts as a container for the Schema. Thus, the model operation, which is represented by the execution operation of the relation type, will create a complete Schema into the given Folder.

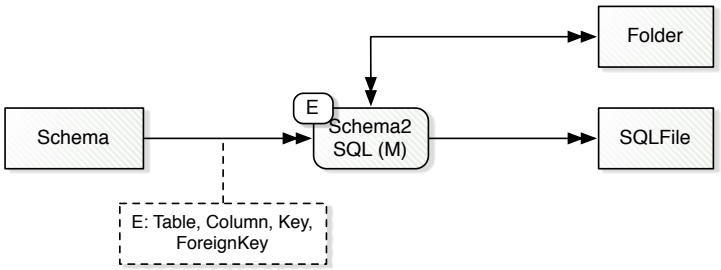

Figure 5.15.: Illustration of a relation type Schema2SQL

The relation type Schema2SQL, which is shown in Figure 5.15, defines the signature of another model operation that generates an SQLFile from a given Schema. The model operation also takes an explicit Folder as source & target, which acts as the container for the generated SQLFile.

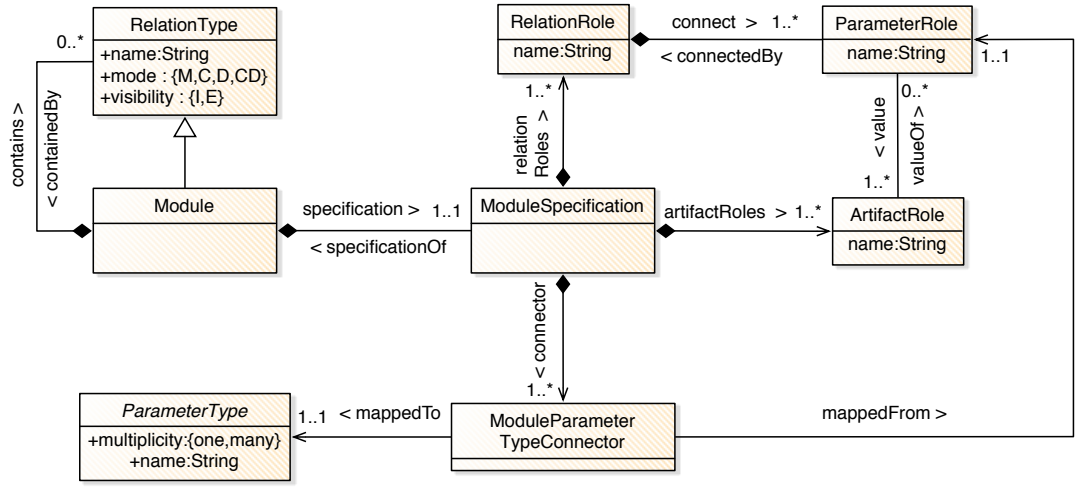

Figure 5.16.: Metamodel of modules

Figure 5.16 shows an extension of the configuration megamodel by means of an extension of the metamodel. The primary concept of this metamodel is the module (Module). A module is a specialization of a relation type and is specified by means of a module specification (ModuleSpecification). A module specification is defined by a set of relation roles (RelationRole), parameter roles (ParameterRole) and artifact roles (ArtifactRole), which are used to define a pattern over relations, parameters and artifacts. Such a pattern is similar to the pattern that can be specified by means of relation type composition specifications.[3]

Every relation role in a module specification must be an instance of a relation type that is contained (contains association) by the related module. The module is responsible for the relation types that are contained by the module. A module also introduces a visibility concept for relation types. Thus, a relation type has an additional attribute named visibility, which can be either I (internal) or E (external). If a module contains a relation type, which visibility is set to I, other relation types outside the module are not allowed to compose themselves into a relation of that type.

A module specification further provides a concept called module parameter type connector (Module ParameterTypeConnector), which is a mapping between a parameter role in the module specification and a parameter type related to the module. A module does not only contain relation types but also other modules, which enables recursion.[4]

[3]The roles are similar to the roles shown in the metamodel of Figure 4.4.
[4]Recursion of modules is not further discussed in this thesis.

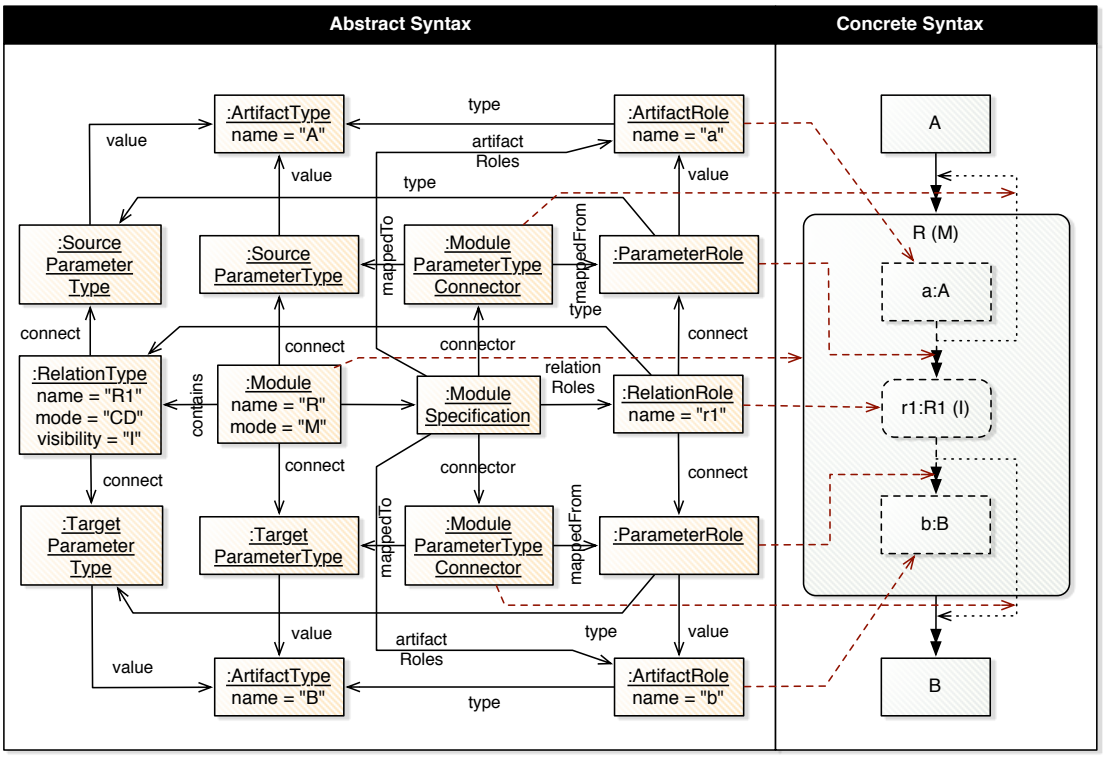

Figure 5.17.: Concrete syntax of modules

Figure 5.17 shows the concrete syntax of modules, which are visualized similarly to a relation type. The only difference is that a module contains a pattern, visualized similarly to the pattern in the relation type composition specification (see Figure 4.5). The module parameter type connector is visualized by means of a dashed arrow from a parameter role to a parameter type connected to the module.

Thus, the figure shows a module R, which has an artifact A as source and an artifact B as target. The module contains a single relation type R1, whose visibility is set to *I*. The module specification describes a simple pattern of a relation role r1 of type R1 that has an artifact role a of type A as source and an artifact role b of type B as target. The module specification has two module parameter type connectors. One of them defines that an artifact, represented by the artifact role a, is similar to an artifact connected as source to the module. The other one defines that an artifact, represented by the artifact role b, is similar to an artifact connected as target to the module.

By means of the module concept, a data-flow composition between the relation types Package2Schema and Schema2SQL can be explicitly specified as shown in Figure 5.18. The figure shows a module Pacakge2SQL, which is responsible for generating an SQLFile from a UMLPackage. The module has two additional source parameter types named schema and sql. These two parameter types are employed to define which Folder artifact is later connected to Package2Schema and which Folder artifact is connected to Schema2SQL.

The module specification describes a pattern over the relation types Package2Schema and Schema2SQL, thus they are defined to be in a data-flow composition because the target of Package2Schema is the source of Schema2SQL.

The semantic of a module is explained by mapping it to a set of relation types that are in specific context compositions. In the following, the resulting relation types from the module Package2SQL are shown. These relation types can be automatically generated from a module to subsequently use them for execution.

Figure 5.19 shows the plain relation type for the module Package2SQL. It is similar to the module but without a module specification. In addition, for every relation role in the module specification a relation type can be created.

Figure 5.20 shows the relation type Package2Schema resulting from the relation role p2s of type Pack-

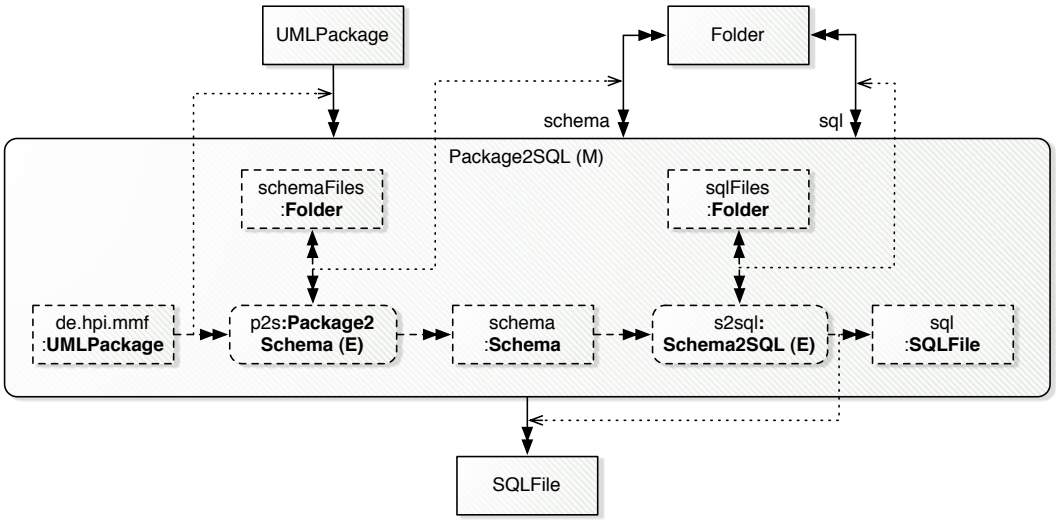

Figure 5.18.: Illustration of a module Package2SQL

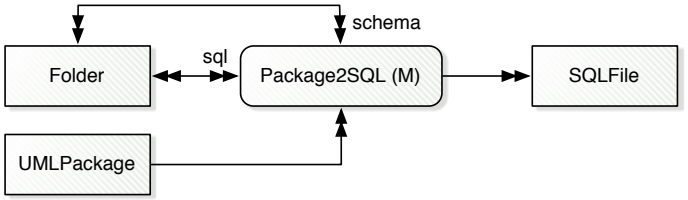

Figure 5.19.: Translation of the module Package2SQL (Part 1)

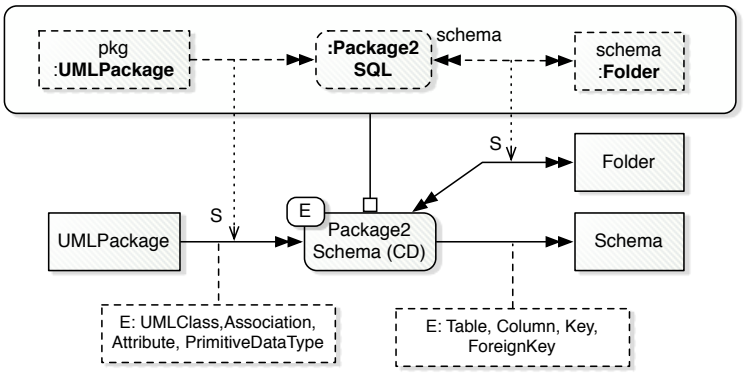

Figure 5.20.: Translation of the module Package2SQL (Part 2)

age2Schema in the module. The parameter types are similar to the relation type contained by the module but with the addition of the relation type composition. It specifies that a relation of type Package2Schema only exists in a composition context that contains a relation of type Package2SQL. Furthermore, a UML-Package artifact connected as source to a Package2Schema relation must be similar to a UMLPackage artifact that is connected to the Package2SQL relation in the composition context.

Figure 5.21 shows the relation type Schema2SQL that results from the relation role s2sql of type Schema2SQL. It now has a relation type composition, whose specification defines a pattern over a relation Package2SQL and Package2Schema which share a UMLPackage artifact as source. Furthermore, a relation Schema2SQL has a Schema artifact as source that is similar to the Schema artifact that is target of the

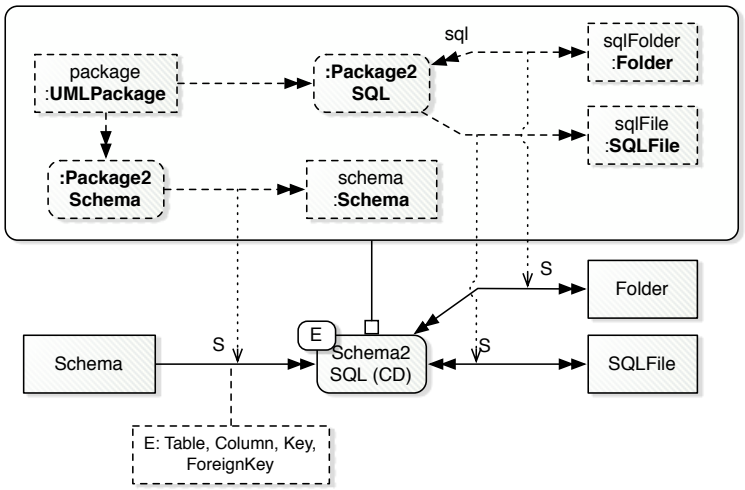

Figure 5.21.: Translation of the module Package2SQL (Part 3)

Package2Schema relation in the composition context. It has a Folder artifact as source & target that is similar to the Folder artifact that is source & target of the Package2SQL artifact in the composition context. Finally, it has an SQLFile artifact as source & target that is similar to the SQLFile artifact that is target of the Package2SQL relation in the composition context. Thus, these three relation types can be used to execute (simulate) a module.

5.2.1.5. Formal Definitions and Constraints

The configuration megamodel of executable and dynamic hierarchical megamodels is a minor extension of the configuration megamodel of dynamic hierarchical megamodels as shown in Definition 4.2.1. The extended configuration megamodel is shown in Definition 5.2.1. It shows only the concepts and relationships which have been introduced in this chapter.

5.2.1 Definition (Configuration Megamodel) The configuration megamodel M_C of the executable and dynamic hierarchical megamodel is a 9-tuple $(A_t, A_{C_t}, P_t, R_t, R_{C_t}, C_S, I_C, I_S, E_O)$ with E_O is a finite set of execution operations. The relationships between these individual concepts of the configuration megamodel are defined by means of mapping functions:

- $represent_{E_O} : E_O \rightarrow O_{C_P}$ maps every execution operation to exactly one model operation that is declared to be represented by the execution operation. $abstract_{O_{C_P},E_O} : O_{C_P} \rightarrow \mathcal{P}(E_O)$ maps every model operation to a set of execution operations that are declared to be abstract representations of the model operation.

- $exec_{R_t} : R_t \rightarrow E_O \cup \{\epsilon\}$ defines the operation association between RelationType and ExecutionOperation and maps every relation type to at most one execution operation that is declared to be the execution operation of the relation type. $exec_{E_O} : E_O \rightarrow R_t$ defines the operationOf association between ExecutionOperation and RelationType and maps every execution operation to exactly one relation type that is declared to be the relation type of the execution operation.

- $scope_{E_O} : E_O \rightarrow \mathcal{P}(I_S)$ defines the scope association between ModelOperationRepresentation (ExecutionOperation) and ImpactScope and maps every execution operation to a non-empty set of impact scopes. $scope_{I_S,E_O} : I_S \rightarrow E_O$ defines the scopeOf association between ImpactScope and ModelOperationRerpesentation (ExecutionOperation).

The extension to the configuration megamodel is a set of execution operations E_O, which are structurally similar to instantiation conditions. Because of this extension, the well-formedness definition of the configuration megamodel, which has been shown in Definition 4.2.5, has to be extended too.

5.2.2 Definition (Well-Formed Configuration Megamodel) A configuration megamodel $M_C = (A_t, A_{C_t}, P_t, R_t, R_{C_t}, C_S, I_C, I_S, E_O)$ is well-formed, if the following conditions are satisfied:

- Every impact scope $i_S \in I_S$ that is defined in the scope of an execution operation $e_O \in E_O$, which is the execution operation of a relation type, must be related to a parameter type that is also connected to that relation type. This is formally defined as $\forall i_S \in I_S, \exists e_O \in E_O, r_t \in R_t, p_t \in P_t : i_C = scope_{I_S,E_O}(i_S) \wedge e_O = exec_{R_t}(r_t) \wedge r_t = connect_{P_t}(p_t) \Rightarrow p_t = scope_{I_S,P_t}(i_S)$.

The shown well-formedness definition is only a complement to the well-formedness definitions that have been shown in previous chapters (see Definition 3.2.4 and 4.2.5). Thus, a configuration megamodel of an executable and dynamic hierarchical megamodel must satisfy all of these definitions. It additionally defines that if an impact scope is related to an execution operation, the impact scope must be related to a parameter type that is connected to a relation type that represents the signature of the execution operation.

Every model operation that is represented by an execution operation must comply with a set of conditions in order to be applicable as implementation of the execution operation. These conditions are defined as shown in Definition 5.2.3.

5.2.3 Definition (Conform Execution Operation Implementation) An execution operation $e_O \in E_O$ that is related to a relation type r_t conforms to a model operation $o_{C_P} \in O_{C_P}$ written as $e_O \models o_{C_P}$, if the following conditions are satisfied:

- The model operation o_{C_P} must provide a corresponding parameter for every parameter type p_t that is connected to the relation type r_t. For every parameter type with a many multiplicity, the model operation o_{C_P} must provide a corresponding parameter that takes a set of artifacts of a certain type.

- The model operation o_{C_P} must only reason on physical artifacts that are represented by artifacts that are directly or indirectly subordinate to artifacts in the given artifact context of a relation of type r_t.

- Every artifact whose type is related to a source parameter type can only be read by the model operation. The model operation may also read artifacts that are directly or indirectly subordinate to the artifact.

- Every artifact whose type is related to a source & target parameter type, can only be read, manipulated or overwritten by the model operation. The model operation can also read, create, manipulate or overwrite artifacts that are directly or indirectly subordinate to the artifact.

- Every artifact which type is related to a target parameter type can only be manipulated or overwritten by the model operation. The model operation can also create, manipulate or overwrite artifacts that are directly or indirectly subordinate to the artifact.

These conditions are pretty similar to the conditions for a conforming implementation of an instantiation condition as shown in Definition 4.2.6. However, a model operation that implements an execution operation also has to take care of how artifacts are processed.

In case of a relation type that is related to an execution operation and an instantiation condition, the conformance of the instantiation condition implementation changes. In this case, the implementation of the instantiation condition only provides a corresponding parameter for every source or source & target parameter but not for target parameters. This is necessary because corresponding artifacts of target parameters do not exist before the first execution. Thus, the conformance definition of instantiation condition implementations is changed as shown in Definition 5.2.4.

5.2.4 Definition (Conform Instantiation Condition Implementation) An instantiation condition $i_C \in I_C$ that is related to a relation type r_t with $exec_{R_t}(r_t) \in E_O$ conforms to a model operation $o_{C_P} \in O_{C_P}$ written as $i_C \models o_{C_P}$, if the following conditions are satisfied:

- The model operation o_{C_P} must return true (1) or false (0) only.

- The model operation o_{C_P} must provide a corresponding parameter for every source or source & target parameter type p_t that is connected to the relation type r_t. For every source or source & target parameter type with a many multiplicity, the model operation o_{C_P} must provide a corresponding parameter that takes a set of artifacts of a corresponding type.
- The model operation o_{C_P} must only reason on physical artifacts that are represented by artifacts that are directly or indirectly subordinate to artifacts in the given artifact context of a relation of type r_t.
- The model operation o_{C_P} must be side-effect free.

5.2.2. Application Megamodels

The application megamodel is similar to the application megamodel that has been shown in the previous chapter. Nevertheless, executable relations are discussed in more detail, as well as further examples for context compositions and data-flow compositions.

5.2.2.1. Executable Relations

A relation, whose type is related to an execution operation, is called an executable relation. An executable relation represents the application or the future application of a model operation that is represented by the execution operation related to the type of the executable relation. Applying a model operation by executing an executable relation is further called execution.

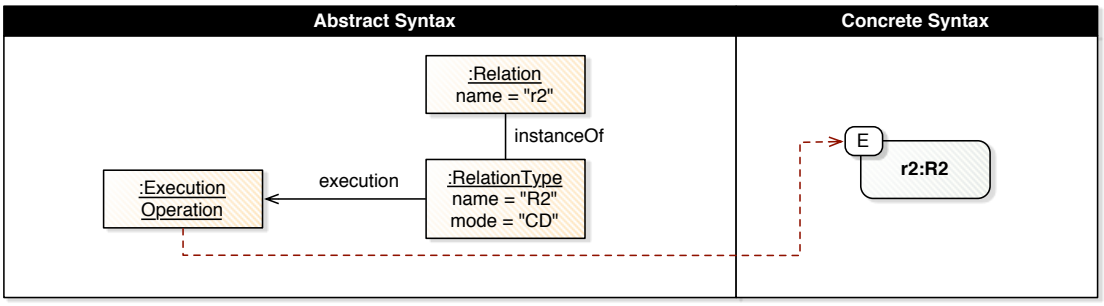

Figure 5.22.: Concrete syntax of executable relations

The concrete syntax of the application megamodel is slightly extended for executable relations as shown in Figure 5.22. Executable relations are marked with a rounded rectangle with a label named E, which is similar to the visualization of relation types with an execution operation. This indicates that this is not a normal relation but an executable relation.

5.2.2.2. Context Compositions

To explain what a context composition in an application megamodel looks like, an exemplary application megamodel is shown that instantiates the relation types introduced in this chapter. The example already shows a situation after executing the executable relations. It is thus the result of the execution, which is shown in the next section.

However, first a simple example model is shown in Figure 5.23, which acts as starting point.

The shown model is an instance of the simplified UML metamodel, which has been introduced in Section 2.2.2. The model consists of a UMLPackage named de.hpi.mmf, a UMLClass named ClassA and another UMLClass named ClassB and an Association named a2b which uses ClassA as source and ClassB as target.

Based on this simple model, an application developer manually creates a relation p2p of type UML2Java and then applies the execution. The resulting application megamodel is shown in Figure 5.24.

The figure shows four executable relations p2f, a, b and a2b, which all exist in a context composition, and a non-executable relation p2p. The executable relation p2f exists in the context of p2p, which is not

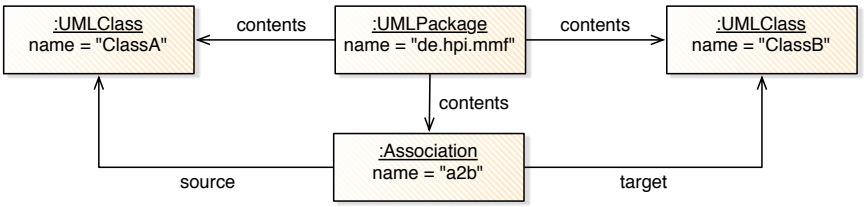

Figure 5.23.: Exemplary UML class diagram using the simplified UML metamodel

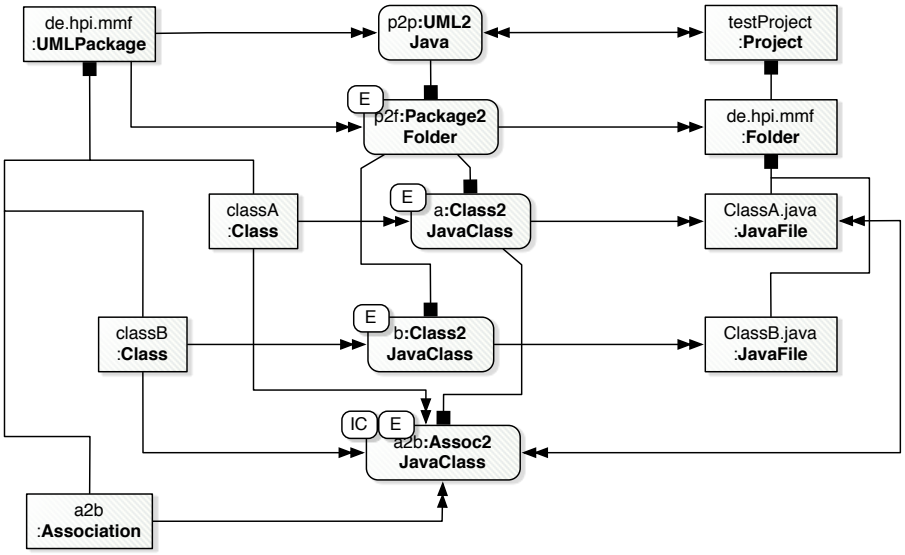

Figure 5.24.: Example of context compositions in an application megamodel

executable. Thus, a model operation is context composed into a non-executable relation. The executable relations a and b are context composed into the same executable relation p2f. Finally, the executable relation a2b exists in the context of the executable relation a because it is connected to classA that represents the source of the association connected to a2b. Even though p2p is not executable, it can also be considered as an executable relation (not technically) that is implemented by a set of fine-grained model operations.

5.2.2.3. Data-Flow Compositions

Another view on the application megamodel is shown in Figure 5.25. It shows two more executable relations p2s and s2sql that are arranged in a data-flow composition. In this simple example, an application developer has to apply the execution twice – first, manually creating the relation p2s of type Package2Schema. Then, applying the execution on p2s the first time, which creates a Schema artifact. Subsequently, an application developer manually creates another relation s2sql of type Schema2SQL. This relation takes the previously created schema as source parameter. Finally, an application developer executes s2sql, which creates an SQLFile into a Folder for SQL files (sqlFiles).

The relation p2s and s2sql are in a data-flow composition because the information from the UMLPackage de.hpi.mmf is propagated into an SQLFile sql via two model operations applied by the executable relations p2s and s2sql. Further, the data-flow requires that first p2s has to be executed and subsequently s2sql because s2sql requires the updated artifact schema of type Schema.

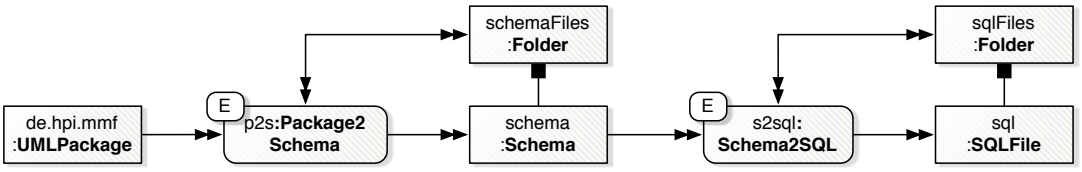

Figure 5.25.: Example of a data-flow composition between p2s and s2sql in an application megamodel

5.2.2.4. Formal Definitions and Constraints

The relation conformance, as it has been defined in Definition 3.2.13, only holds for non-executable relations. In case of executable relations, another definition of relation conformance is provided. This definition is called executable relation conformance and is shown in Definition 5.2.5.

5.2.5 Definition (Executable Relation Conformance) Given an application megamodel $M_A = (A, A_C, P, R, R_C, C_M)$ and a configuration megamodel $M_C = (A_t, A_{C_t}, P_t, R_t, R_{C_t}, C_S, I_C, I_S)$, an executable relation $r \in R$ conforms to a relation type $r_t \in R_t$ written as $r \models_E r_t$, if the following conditions are satisfied:

- The executable relation r must be an instance of the relation type r_t, which is defined as $r_t = type_R(r)$.
- If the relation type r_t has at least one relation type composition $r_{C_t} \in R_{C_t}$ as superior, the executable relation r must provide at least one relation composition $r_C \in R_C$ as superior that is conform to r_{C_t}, which is defined as $|sup_{R_t}(r_t)| \geq 1 \Rightarrow |sup_R(r)| \geq 1 \wedge r_C \models r_{C_t}$.
- The executable relation r must have a correct artifact context, which holds if the following conditions are satisfied:
  - Every source or source & target parameter p that is connected to the executable relation r must be an instance of a source or source & target parameter type p_t that is connected to the relation type r_t. This is defined as $\forall p \in connect_R(r), \exists p_t \in connect_{R_t}(r_t) : p_t = type_P(p) \wedge dir_P(p) \in \{S, ST\} \wedge dir_{P_t}(p_t) \in \{S, ST\}$.
  - Every source or source & target parameter type p_t that is connected to the relation type r_t must be a type of a source or source & target parameter p that is connected to the executable relation r. This is defined as $\forall p_t \in connect_{R_t}(r_t), \exists p \in connect_R(r) : p \in instance_{P_t}(p_t) \wedge dir_P(p) \in \{S, ST\} \wedge dir_{P_t}(p_t) \in \{S, ST\}$.

The only difference is that in case of executable relations, only source and source & target parameters have to be connected. A target parameter will be created when executing the relation for the first time.

5.3. Execution

In this section, the execution for executable and dynamic hierarchical megamodels is presented, which provides model management capabilities to MDE applications. The execution itself requires the localization from the previous chapter because it is necessary for the execution.

Figure 5.26 shows all use cases of the execution that are directly triggered by an application developer and that are necessary but hidden to an application developer.

An application developer can trigger the execution, which has two different implementations (strategies). An individual execution strategy is one which (re-)executes a relation and all executable relations that are directly or indirectly composed into that relation via context compositions (see Section 5.3.3.1). The other strategy is a complete execution strategy, which (re-)executes all necessary executable relations that are impacted by changes (see Section 5.3.3.2).

Both strategies use an adapted localization, which wraps the incremental localization strategy from the previous chapter. The adapted localization is used for estimating relations that are impacted by changes and making them accessible to the execution (see Section 5.3.1).

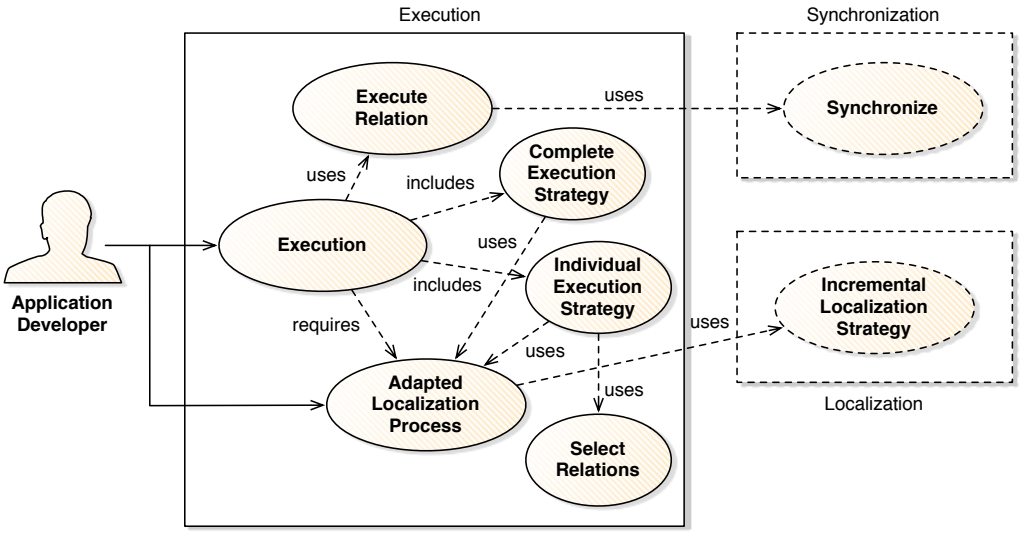

Figure 5.26.: Use cases of the execution

Before applying the execution, the application megamodel must be up-to-date. This means that the application megamodel has been synchronized and the localization has been previously applied. This ensures that only conforming artifacts and relations exist, by automatically triggering the synchronization and the adapted localization before triggering the actual execution. The schema of triggering the execution is shown as UML activity diagram in Figure 5.27.

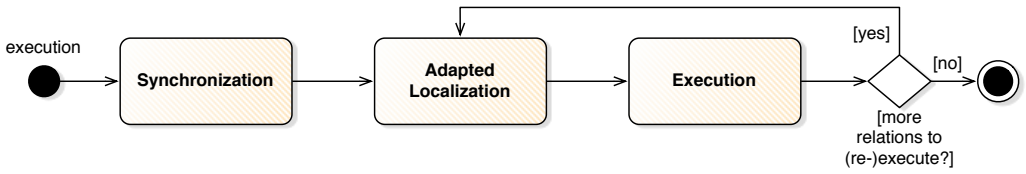

Figure 5.27.: Schematic sequence of the execution

Thus, an application developer does not need to care about triggering the synchronization and localization before execution. The figure also shows that the execution requires interleaving the adapted localization with the execution. This loop is required because execution operations have side-effects. Thus, the (re-)execution of executable relations may cause the creation of new artifact contexts for other (executable) relations, which may have to be executed subsequently.

The execution further uses an execute relation operation as shown in Section 5.3.2, which is an abstraction mechanism for applying all kinds of model operations. Within this operation the synchronization is explicitly triggered to instantaneously create abstract representations of physical artifacts that have been created by applying a model operation. Thus, the synchronization is also part of the loop.

5.3.1. Adapted Localization

When it comes to the automated maintenance of executable relations, adaptations of certain concepts from previous chapters are required because executable relations have a slightly different semantic than usual relations.

Thus, the localization must be slightly adapted too, to also support executable relations. Nevertheless, only the operations $similarArtifactContext$ and $getNextParameterType$ have to be adapted. The operation $similarArtifactContext$ is adapted such that it does not take target parameters into account. Now, two relations are similar if their source parameters and source & target parameters are similar.

The adapted operation *getNextParameterType* only returns source parameter types and source & target parameter types if the considered relation type has an execution operation.

Additionally, the adapted localization has two primary tasks. Firstly, relations must capture all dependencies and all existing relations have to be in conformity, obtained by invoking the localization from the previous chapter. Secondly, it must maintain a set of executable relations that potentially have to be (re-)executed due to changes. This set of executable relations can be used by the execution strategies to only execute those relations that have to be executed. Listing 5.1 shows the adapted localization, which is employed in the sequel of this chapter.

```
1  procedure adaptedLocalization()
2    //1. invoke localization
3    incrementalLocalization();
4    //2. estimate relations that need to be (re-)executed
5    E := getChanges(); // with E = (C_A, U_A, D_A, C_R, U_R, D_R)
6    R* := getRelationsForExecution() \ D_R;
7    forall (r ∈ C_R | operation_{R_t}(type_R(r) ≠ ε)) // directly impacted
8      R* := R* ∪ {r};
9    endforall
10   forall (r ∈ R : operation_{R_t}(type_R(r)) ≠ ε)
11     if (∃a ∈ C_A ∪ D_A ∪ U_A, p ∈ connect_R(r) : dir_{P_t}(type_P(p)) ∈ {S, ST} ∧ a ∈ val_P(p)) // target artifact
           deleted
12       R* := R* ∪ {r};
13       continue;
14     endif
15     if (∃a ∈ D_A, p ∈ connect_R(r) : dir_{P_t}(type_P(p)) ∈ {T} ∧ a ∈ val_P(p))
16       R* := R* ∪ {r};
17       continue;
18     endif
19     forall (p ∈ connect_R(r) : dir_{P_t}(type_P(p)) ∈ {S, ST})
20       if (∃a ∈ C_A ∪ U_A ∪ D_A, a' ∈ val_P(p), e_O ∈ E_O, i_S = (A'_t, p_t) ∈ I_S : i_S ∈ scope_{E_O}(e_O) ∧ p_t = type_P(p) ∧
             e_O = operation_{R_t} (type_R(connect_P(p))) ∧ type_A(a) ∈ A'_t ∧ a ∈ subs*_A(a')) // indirectly impacted
21         R* := R* ∪ {r};
22         break;
23       endif
24     endforall
25   endforall
26   setRelationsForExecution(R*);
27   resetChanges();
28 endprocedure
```

Listing 5.1: Wrapped and adapted incremental localization

The operation first invokes the incremental localization (Line 3). Then, a set of changes E and a set of executable relations R^*, which have to be (re-)executed, are estimated (Line 5-6). The first modification to R^* is to remove those relations that have been deleted by the localization (Line 6). Subsequently, all executable relations that were just created are added to R^* (Line 7-9). This is necessary because newly created executable relations await eventual execution.

Afterwards, all executable relations are added to R^*, which need to be re-executed due to changes (Line 10-25). Firstly, all executable relations are added to R^* that are directly impacted by an artifact a, which just has been created, deleted or updated (Line 11-14). Secondly, all executable relations are added to R^*, if an artifact from a target parameter has been deleted (Line 15-18). These relations need to be (re-)executed because they reflect the same situation as before execution. All relations are added to R^* which source and source & target parameters are indirectly impacted by one of these artifact changes (Line 19-24).

Finally, the modified set R^* is made available to the subsequent applications by invoking *setRelations ForExecution* (Line 26). The set of changes is further reseted by invoking *resetChanges* (Line 27). This is necessary because synchronization and localization have been applied and have completely processed all changes. Thus, there is no need to further keep these changes.

5.3.2. Executing Relations

Executing a single relation, which has a related execution operation, requires additional treatment. Before applying an execution operation, not yet existing physical artifacts have to be created by the model management framework because it is responsible for setting up the target parameters of executable relations. The newly created physical artifacts have to be synchronized immediately and the resulting

artifacts have to be set as the value of the new target parameters. Thus, after executing a relation, the relation must conform to its relation type ($r \models r_t$).

```
1  procedure executeRelation(r) // with r ∈ R
2    //pre-processing
3    A* := ∅; // with A* ⊆ P × A_AP
4    r_t := type_R(r);
5    forall (p_t ∈ connect_Rt(r_t) : dir_Pt(p_t) = T)
6      if (∄p ∈ connect_R(r) : p_t = type_P(p)) //not yet exist
7        A* := A* ∪ {(createParameter(r, p_t), createPhysicalArtifact(r, p_t))};
8      else
9        if (val_P(p) = ∅)
10         A* := A* ∪ {(p, createPhysicalArtifact(r, p_t))};
11       endif
12     endif
13   endforall
14   //processing
15   apply(r, A*);
16   //post-processing
17   synchronize();
18   forall ((p, a_AP) ∈ A*)
19     val_P(p) := abstract_A_AP(a_AP);
20     connect_R(r) := connect_R(r) ∪ {p};
21   endforall
22 endprocedure
```

Listing 5.2: Execute a relation

The operation for executing an executable relation is shown in Listing 5.2. It consists of a pre-processing part (Line 3-13), a processing part (Line 15) and a post-processing part (Line 17-21).

In the pre-processing part, for any target parameter type p_t of the type of r that does not yet have an instance connected to r, a new parameter of type p_t and a new physical artifact is created. Both are added as a tuple to A^* (Line 6-7). If a parameter p of type p_t is already connected to p, but without an artifact as value, only a new physical artifact is created and a tuple of p and the new physical artifact is added to A^* (Line 8-12). In both cases the operation *createPhysicalArtifact* is invoked. This operation takes a relation and a parameter type as value and returns a physical artifact. The operation also ensures that the physical artifact is created into the context of another physical artifact if defined by a parameter type connector that is mapped to the parameter type p_t. The operation is not shown because it requires physical artifact specific operations that are wrapped by means of abstractions, which is part of the implementation (see Section A.2.3).

In the processing part, the actual model operation is applied by invoking apply with r and A^* as parameters (Line 15). The operation (re-)applies the actual model operation by using physical artifacts connected to source and source & target parameters of r and the target parameters and their values in A^*. The (re-)application of model operations has to be obtained differently for every model operation technology (see Section A.2.4).

In the post-processing part, the synchronization is immediately triggered by invoking *synchronize* (Line 17). After synchronization, for each tuple $(p, a_{A_P}) \in A^*$ a representation of a_{A_P} is set as the value of p (Line 19) and the parameter p is added to the relation r (Line 20).

5.3.3. Execution Strategies

The execution provides two execution strategies for two different purposes, which are both applicable by an application developer. The first execution strategy is called individual execution strategy and is explained in Section 5.3.3.1. The second execution strategy is called complete execution strategy and is explained in Section 5.3.3.2.

5.3.3.1. Individual Execution Strategy

The aim of the individual execution strategy is to provide a facility that enables an application developer to treat individual relations as complex model operations that are composed of a set of fine-grained and heterogeneous model operations. An application developer only needs to provide a relation, which should be (re-)executed. This must not be an executable relation because executable relations may be directly or indirectly context composed into non-executable relations.

The main part of the individual execution strategy is shown as pseudo-code in Listing 5.3.

```
1  procedure executeIndividual(r_s) with r_s ∈ R the selected relation
2    R' := ∅; //set of already considered relations
3    while (true)
4      r := getNextRelation({r_s}, R'); // get next relation for execution
5      if (r = ϵ)
6        break;
7      else if (r ∈ getRelationsForExecution() ∧ r is directly or indirectly composed into r_s)
8        execute(r);
9        adaptedLocalization();
10       setRelationsForExecution(getRelationsForExecution() \ {r});
11     endif
12     R' := R' ∪ r;
13   endwhile
14 endprocedure
```

Listing 5.3: Individual execution strategy

The whole operation is defined in a while-loop, which terminates as soon as a fix-point is reached (Line 3-13). A fix-point is reached if no more relations can be (re-)executed. By invoking the operation *getNextRelation* on R_s, a relation r is chosen, which is potentially (re-)executed next (Line 4). The operation *getNextRelation* takes the set of relations as parameter, which is the minimal set of relations to be (re)-executed, and a set of relations $R' \subseteq R$, which is a set of already (re-)executed relations in previous iterations.

If no relation is selected because no more relations have to be (re-)executed, the while loop is left (Line 5-6). Else, the relation r is further processed if r is in the set of relations to be executed and r is directly or indirectly context composed into the relation r_s (Line 7-11). The relation r is (re-)executed by invoking the *execute* operation, which is shown in Listing 5.2, and passes the selected relation r as parameter (Line 8). After (re-)executing r, new physical artifacts, and therefore artifacts in the application megamodel, might exist. Thus, new relations might need to be created or existing relations might have to be deleted, which is realized by invoking the adapted localization, which is shown in Listing 5.1, immediately after executing the relation (Line 9). Subsequently, the just (re-)executed relation r is removed from the set of relations to be executed (Line 10).

The operation *getNextRelation* may also return relations that will not be (re)-executed. These relations must be collected in a set of already considered relations R' to not consider them again in any further iteration, which is important for reaching a fix-point. Thus, any relation r will be automatically added to R' in the end of each iteration (Line 12).

Figure 5.28 shows the initial situation, which is taken from the example shown in Figure 5.24, where a relation p2p of type UML2Java has just been created manually between artifacts de.hpi.mmf and testProject. Now, an application developer triggers the individual execution strategy for the relation p2p. Then, the adapted localization is triggered for the first time, which results in the creation of the relations p2f, a and b. Because these relations have not been executed before, their target parameters are not yet created. Furthermore, the relation a2b does not yet exist at all because the source and target artifact does not yet exist. The set of relations in *getRelationsForExecution* contains p2f, a and b. These three relations are now also in the set of relations to be (re-)executed.

In the first iteration of the individual execution strategy, *getNextRelation* selects p2p, which is not executed because it is not in the set *getRelationsForExecution*. In the second iteration, p2f is selected and executed. This creates the Folder artifact de/hpi/mmf/ contained by the artifact testProject. In the third iteration, a is selected.[5] The execution of a will create the artifact ClassA.java into the context of the artifact de/hpi/mmf. The invocation of the adapted localization will now cause the creation of the relation a2b. Thus, a2b is now also in *getRelationsForExecution*.

In the fourth iteration, the relation b is selected and immediately executed, which creates the artifact ClassB.java into the context of the artifact de/hpi/mmf/. In the fifth iteration, the relation a2b is selected and executed, which updates the context of the artifact ClassA.java. The fifth iteration is the last iteration in which a relation is executed because there is no more relation $r \in R$ that is not in R' and that is in *getRelationsForExecution* and that is directly or indirectly context composed into p2p.

The operation *getNextRelation* is shown in Listing 5.4 in more detail.

[5]Selecting b would also be legitimate in this iteration because both are independent of each other.

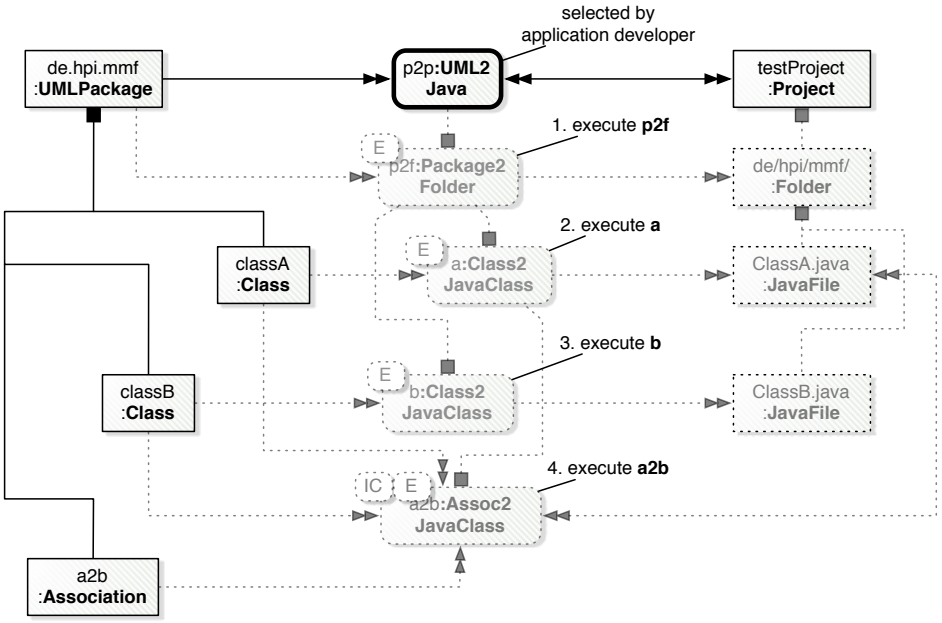

Figure 5.28.: Illustration of initial execution using the individual execution strategy

```
1  procedure getNextRelation(R*, R') : r ∈ R with R' are already executed relations
2    G_E := createSchedulingGraph(R); // with G_E = (V_E, E_E)
3    S := topologicallySort(G_E, R*); // with S is an ordered set
4    forall (r ∈ S)
5      if (r ∉ R')
6        return r;
7      endif
8    endforall
9    return ϵ;
10 endprocedure
```

Listing 5.4: Get next relation

This operation estimates the next relation from a set of relations, which are in a strict partial ordering. Therefore, it ensures that only relations from R^* are considered, which cannot be influenced by any other relation in R^* and that were not considered before. This is implemented by first creating a scheduling graph from all relations in R and then topologically sorting this graph starting from all relations in R^* (Line 2-3). Then, the resulting strictly partially ordered set S is traversed and the first relation $r \in S$ that is not in R' is returned. Thus, it is ensured that already (re-)executed relations will not be (re-)executed a second time. If the set R' is similar to the set of S, the operation will return ϵ, which means that no more relations have to be (re-)executed.

The scheduling graph is a directed graph consisting of relations, which are considered as vertices, and edges in between, which represent execution dependencies between relations. Three different kinds of execution dependencies between relations are considered, which are context composition, data-flow composition and indirect data-flow composition dependencies. Thus, if a directed edge between two vertices in a scheduling graph exists, it means that one of these dependencies exists in between them.

If two executable relation are in a context composition, they are also in an execution dependency (context composition dependency) because it should be ensured that higher-level relations are always executed before lower-level relations. If an executable relation is connected to an artifact via source & target or target parameter and another executable relation has the same artifact as source or source & target, these two relations are in a data-flow dependency.

An indirect data-flow dependency is similar to a data-flow dependency except that two executable relations must not directly be connected to a common artifact. Instead, the impact scope is used to decide whether two relations are in an indirect data-flow dependency.

Figure 5.29 shows a simple example of two indirect data-flow dependencies.

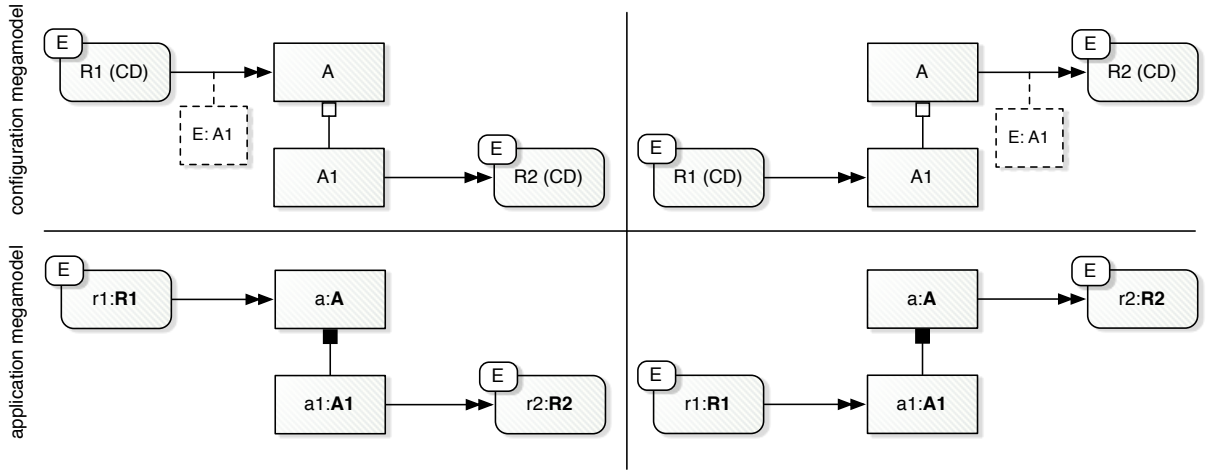

Figure 5.29.: Illustration of indirect data-flow dependencies

The configuration megamodels to the left and right show two scenarios in which the relations of type R2 may have an indirect data-flow dependency to relations of type R1. The scenario to the left shows a relation of type R2 that may have an indirect data-flow dependency to a relation of type R1 because R1 manipulates artifacts of type A1, which are source of relation of type R2.

The scenario to the right is slightly altered. The only difference is that relations of type R2 may have an indirect data-flow dependency to a relation of type R1 because R2 indirectly reads artifacts of type A1, which are directly manipulated by executable relations of type R1. The application megamodels below show such exemplary instances of these dependencies between instances of R1 and R2.

5.3.1 Definition (Scheduling Graph) A scheduling graph G_E is a directed graph (V_E, E_E) with $V_E = R$ is a finite set of relations, which act as vertices of the graph, and $E_E \subseteq V_E \times V_E$ is a finite set of edges without self-loops defined as $\forall (r_1, r_2) \in E_E : r_1 \neq r_2$. Furthermore, an edge $(r_1, r_2) \in E_E$ only exists if one of the following properties are satisfied:

- $\exists c_M = (A_B, P_B, R_B, P_C) \in C_M, r_C \in R_C, \exists r_B = (r_R, r') \in R_B : r_C \in sup_R(r_1) \wedge c_M = match_{R_C}(r_C) \wedge r_2 = r'$, which means that r_1 is directly composed into the context of r_2.
- $\exists a \in A, p_1, p_2 \in P : r_1 = connect_P(p_1) \wedge r_2 = connect_P(p_2) \wedge dir_{P_t}(type_P(p_1)) \in \{ST, T\} \wedge dir_{P_t}(type_P\ (p_2)) \in \{S, ST\} \wedge a \in val_P(p_1) \wedge a \in val_P(p_2)$, which means that r_1 is a direct predecessor of r_2 in a data-flow.
- $\exists a_1, a_2 \in A, p_1, p_2 \in P, i_S = (A'_t, p_t) \in I_S : r_1 = connect_P(p_1) \wedge r_2 = connect_P(p_2) \wedge a_1 \in val_P(p_1) \wedge a_2 \in val_P(p_2) \wedge dir_{P_t}(type_P(p_1)) \in \{ST, T\} \wedge dir_{P_t}(type_P(p_2)) \in \{S, ST\} \wedge type_P(p_1) = p_t \wedge type_A(a_2) \in A'_t \wedge a_2 \in subs^*_A(a_1)$, which means that r_1 is a direct predecessor of r_2 in an indirect data-flow.
- $\exists a_1, a_2 \in A, p_1, p_2 \in P, i_S = (A'_t, p_t) \in I_S : r_1 = connect_P(p_1) \wedge r_2 = connect_P(p_2) \wedge a_1 \in val_P(p_2) \wedge a_2 \in val_P(p_1) \wedge dir_{P_t}(type_P(p_1)) \in \{ST, T\} \wedge dir_{P_t}(type_P(p_2)) \in \{S, ST\} \wedge type_P(p_2) = p_t \wedge type_A(a_2) \in A'_t \wedge a_2 \in subs^*_A(a_1)$, which means that r_1 is a direct predecessor of r_2 in an indirect data-flow.

The scheduling graph, as shown in Definition 5.3.1, is a directed graph because execution dependencies always have a direction. This graph may contain cycles of size greater than one, which means that an application megamodel contains direct or indirect data-flow composition cycles. This does not influence the termination of the execution, because the scheduling graph is always topologically sorted, implemented by means of a depth-first search.

Nevertheless, it may be considered as a warning to the application and configuration developer because executing a cycle may result in overwriting previous changes. However, this is not the focus of this thesis.

5.3.3.1.1. Termination The termination of the individual execution strategies is not obvious because of the interleaving with the localization. The individual execution strategy terminates, if the main while-loop (Line 3-13 in Listing 5.3) has a fix-point. Such a fix-point always exists, if the number of relations that are created during execution is limited. If the number of new relations is limited, it holds that $R' = S$ in *getNextRelation* after a finite number of iterations. This implies that no more relations will be selected by *getNextRelation*.

That such a limit exists can be guaranteed by means of a scheduling type graph. The scheduling type graph is defined on the type layer (configuration megamodel) instead of on the instance layer (application megamodel). The vertices of this graph are relation types and the edges in between represent potential execution dependencies.

The scheduling type graph is formally defined as shown in Definition 5.3.2.

5.3.2 Definition (Scheduling Type Graph) A scheduling type graph G_{E_t} is a directed graph (V_{E_t}, E_{E_t}) with $V_{E_t} = R_t$ is a finite set of relation types, which act as vertices of the graph, and $E_{E_t} \subseteq V_{E_t} \times V_{E_t}$ is a finite set of edges. Furthermore, $\forall r_t \in V_{E_t} : mode_{R_t}(r_t) \in \{C, CD\} \wedge operation_{R_t}(r_t) \neq \epsilon$. Edges $(r_{t_1}, r_{t_2}) \in E_{E_t}$ only exist if one of the following conditions are satisfied:

- $\exists a_t \in A_t, p_{1_t}, p_{2_t} \in P_t : r_{1_t} = connect_{P_t}(p_{1_t}) \wedge r_{2_t} = connect_{P_t}(p_{2_t}) \wedge dir_{P_t}(p_{1_t}) \in \{ST, T\} \wedge dir_{P_t}(p_{2_t}) \in \{S, ST\} \wedge a_t = val_{P_t}(p_{1_t}) \wedge a_t = val_{P_t}(p_{2_t})$, which means that r_{1_t} is a direct predecessor of r_{2_t} in a potential data-flow.
- $\exists a_{1_t}, a_{2_t} \in A_t, p_{1_t}, p_{2_t} \in P_t, i_S = (A'_t, p'_t) \in I_S : r_{1_t} = connect_{P_t}(p_{1_t}) \wedge r_{2_t} = connect_{P_t}(p_{2_t}) \wedge a_{1_t} = val_{P_t}(p_{1_t}) \wedge a_{2_t} = val_{P_t}(p_{2_t}) \wedge dir_{P_t}(p_{1_t}) \in \{ST, T\} \wedge dir_{P_t}(p_{2_t}) \in \{S, ST\} \wedge p_{1_t} = p'_t \wedge a_{2_t} \in A'_t \wedge a_{2_t} \in subs^*_{A_t}(a_{1_t})$, which means that r_{1_t} is a direct predecessor of r_{2_t} in a potential indirect data-flow.
- $\exists a_{1_t}, a_{2_t} \in A_t, p_{1_t}, p_{2_t} \in P_t, i_S = (A'_t, p'_t) \in I_S : r_{1_t} = connect_{P_t}(p_{1_t}) \wedge r_{2_t} = connect_{P_t}(p_{2_t}) \wedge a_{1_t} = val_{P_t}(p_{2_t}) \wedge a_{2_t} = val_{P_t}(p_{1_t}) \wedge dir_{P_t}(p_{1_t}) \in \{ST, T\} \wedge dir_{P_t}(p_{2_t}) \in \{S, ST\} \wedge p_{2_t} = p'_t \wedge a_{2_t} \in A'_t \wedge a_{2_t} \in subs^*_{A_t}(a_{1_t})$, which means that r_{1_t} is a direct predecessor of r_2 in a potential indirect data-flow.

The scheduling type graph only considers relation types that are created automatically and that have a related execution operation. The edges encode potential execution dependencies, which can cause data-flows or indirect data-flows in application megamodels.

Generally, the number of relations that will be created when executing a relation is limited if the resulting scheduling type graph is acyclic. Thus, the execution is guaranteed to terminate. If a cycle in such a graph exists, the individual execution strategy may create infinitely many relations. That is because a cycle may result in infinitely long chains of executable relations coupled by means of data-flows or indirect data-flows. Nevertheless, a cycle does not directly imply that no fix-point exists. This depends on the actual constitution of instantiation conditions that may exclude such an infinite chain.

5.3.3.1.2. Correctness The individual execution strategy is correct if all relations that are directly or indirectly composed into the selected relation r_s are eventually (re-)executed if necessary. The individual execution strategy is correct because of the following reasons:

- For a given relation r_s, all relations that are directly or indirectly context composed into r_s are considered because *getNextRelation* for r_s will at least return all relations that are directly or indirectly context composed into r_s. Every relation that is considered for (re-)execution will be (re-)executed if it is in the set of relations for execution, which contains all relations that have been impacted by changes.
- If relations are created in the meantime, invoking the adapted localization will add all new relations to the set *getRelationsForExecution*. If these newly created relations are directly or indirectly composed into a relation r_s, they will be selected and executed in a subsequent iteration.

In case of cycles in the scheduling graph it may happen that a relation that has been (re-)executed becomes once again invalidated. These relations are not considered to be (re-)executed again automatically. However, this does not affect correctness because the application developer has to decide whether another application of the execution operation is necessary.

5.3.3.2. Complete Execution Strategy

The complete execution strategy is going to (re-)execute all executable relations as needed. Thus, it does not need any selection of relations made by an application developer upfront. The complete execution strategy is almost similar to the individual execution strategy. There are only minor differences, explained in the following.

```
1  procedure executeComplete()
2    R' := ∅; // set of already considered relations
3    while (true)
4      r := getNextRelation(getRelationsForExecution(), R'); // get next relation for execution
5      if (r = ϵ)
6        break;
7      else if (r ∈ getRelationsForExecution())
8        execute(r);
9        adaptedLocalization();
10       setRelationsForExecution(getRelationsForExecution() \ {r});
11     endif
12     R' := R' ∪ {r};
13   endwhile
14 endprocedure
```

Listing 5.5: Complete execution strategy

Listing 5.5 shows the complete execution strategy. This strategy does not take a selection of relations as a parameter. It chooses the next relation r by also invoking *getNextRelation* (Line 4). However, this strategy directly passes the set of relations to be executed as first parameter, because all of them should be executed. If a relation r exists and this relation is in *getRelationsForExecution*, the relation is executed as in the individual execution strategy.

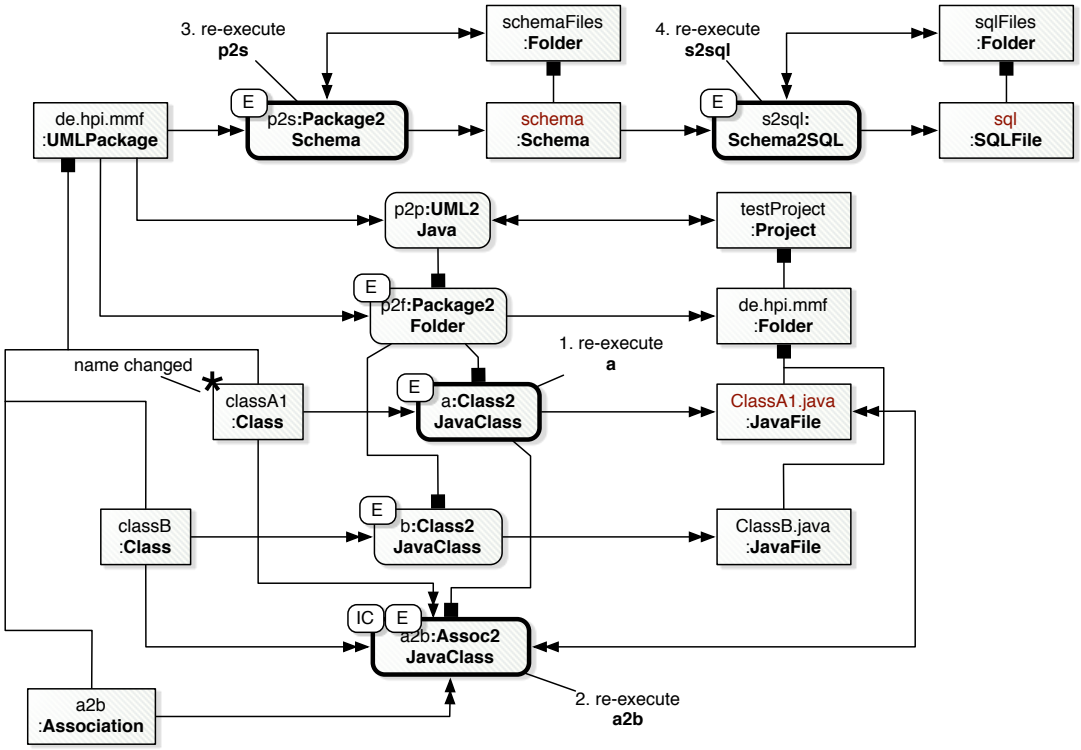

Figure 5.30.: Illustration of a change and subsequent execution using the complete execution strategy

Figure 5.30 shows a situation, which is taken from the example shown in Figure 5.24 and 5.25. The situation shows an application megamodel that was already completely executed but before the name of the artifact classA has been changed to classA1. Triggering the adapted localization does not change

anything and *getRelationsForExecution* contains a, a2b and p2s because a and a2b have the artifact classA1 as source and the source parameter of p2s is related to an impact scope that contains the type of classA1.

Applying the complete execution strategy first selects and (re-)executes a, which changes the name of the artifact ClassA.java to ClassA1.java. Subsequently, a2b is selected and (re-)executed. Then, p2s is going to be selected and executed, causing an update to the schema artifact. This update will cause the adapted localization to also add the relation s2sql to *getRelationsForExecution*. Thus, the last relation to be selected and executed is s2sql.

This is not the only sequence of how the complete execution strategy might (re-)execute the relations. Another alternative and legitimate sequence could be p2s, s2sql, a and a2b.

5.3.3.2.1. Termination The termination criterion of the complete execution strategy is similar to the termination criterion of the individual execution strategy. Thus, the absence of cycles in a scheduling type graph can guarantee termination.

5.3.3.2.2. Correctness The complete execution strategy is correct if all relations that have to be (re-)executed are eventually (re-)executed. This is satisfied by the complete execution strategy because it directly selects the next relation from the set of relations that have to be (re-)executed. In case of cycles in the scheduling graph it may happen that a relation that has been (re-)executed becomes invalidated again. This is considered as partial correctness and therefore an issue of how relations are composed.

5.4. Summary

In this chapter, the concept of the dynamic hierarchical megamodel has been extended to the concept of the executable and dynamic hierarchical megamodel. The executable and dynamic hierarchical megamodel provides extensions to relation types and relations. A relation type has been extended by means of an execution operation and has been specialized by means of a module.

An execution operation is an abstract representation of any model operation that is used to analyze or synthesize models. By means of an execution operation a relation of a certain type is considered as an application or an eventual application of a model operation. Thus, these relations are considered as being executable. A module is a specialization of a relation type. It provides encapsulation of relation types, as well as a means to specify data-flow compositions of model operations already in a configuration megamodel. The encapsulation can be used to restrict that relation types, defined in a module, cannot exist in the composition context of relation types outside the module. The specification of a data-flow is supported by means of a module pattern.

The specification of context compositions of model operations is already provided by the concepts from the previous chapter.

In order to support application developers in applying model operations, an execution mechanism has been introduced. The execution allows a user to (re-)apply model operations by (re-)executing relations in an application megamodel. Therefore, the execution mechanism provides two execution strategies. The first strategy allows the (re-)application of a single model operation, including the model operations that are directly or indirectly composed into the context of that model operation. The second strategy allows the (re-)application of all model operations which have changed.

By now, the approach has been explained by means of application examples from two case studies that have been presented in Section 2.2. Further application examples are shown in the next chapter.

6. Evaluation

Contents

* * *

So far, the approach has been applied to simple and comprehensible application examples from case studies introduced in Chapter 2, throughout the main chapters of this thesis. To achieve a better foundation for discussions about this approach, further evaluations will be discussed in this chapter (see Section 6.1).

Based on all examples and evaluations, Section 6.2 discusses how the initially stated goals are satisfied by the individual concepts of the shown approach. It is further discussed how these concepts concretely address the related challenges, followed by a critical discussion of the general applicability and generalizability is critically discussed.

6.1. Evaluations

This section provides two evaluations, both of which demonstrate the capabilities of the context composition concept of executable and dynamic hierarchical megamodels and its execution. They will demonstrate the capabilities of building complex model operations from fine-grained model operations by employing context composition, and showing the capabilities of extending complex model operations by employing context compositions.

6.1.1. Execution: Building Complex Model Operations

The purpose of this evaluation is to show that the introduced approach is able to build complex model operations from simple and fine-grained model operations by means of context composition, demonstrated using an example from the case study shown in Section 2.2.2. Therefore, an alternative implementation of the model transformation from UMLPackage artifacts to Schema artifacts is considered. This alternative implementation is based on fine-grained, loosely coupled and highly cohesive model transformations that are subsequently context composed to define a coherent model transformation.

This model transformation is nearly similar to the QVT model transformation (UML to RDBMS) used as an example in the QVT specification [124][1], except this transformation's task is to transform UMLPackage artifacts into Schema artifacts. Therefore, it transforms all UMLClass artifacts into Tables artifacts and Association and Attribute artifacts into corresponding Column artifacts.

It is implemented by means of six individual model transformations, which results in the definition of six relation types, shown as follows.

The first relation type Package2Schema defines the signature of a transformation that transforms a UMLPackage into a Schema (see Figure 6.1). This is a top-level relation type that is not defined in any

[1] It is not completely similar because the names of Key, ForeignKey and Column artifacts are not generated exactly in the same way. That is because parameter types cannot be related to primitive data types.

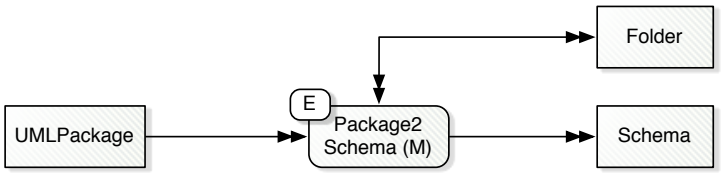

Figure 6.1.: Definition of the Package2Schema relation type

composition. The implementation of the model operation that is represented by the execution operation of this relation type is shown in Section C.3.2.1.

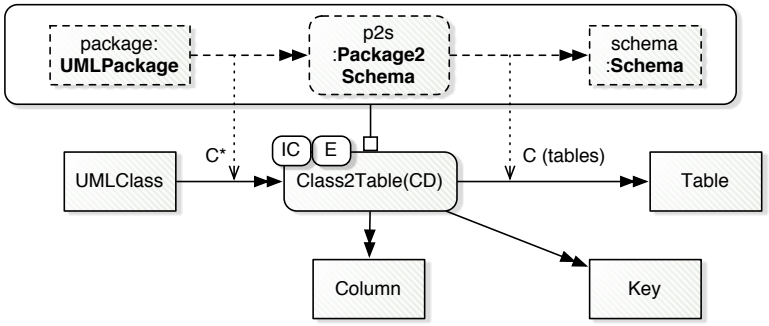

Figure 6.2.: Definition of the Class2Table relation type

The second relation type Class2Table is shown in Figure 6.2 and defines the signature of a model transformation, which transforms a persistent UMLClass artifact into a Table, a Key and a Column artifact. The implementation of the model operation that is represented by the execution operation of this relation type is shown in Section C.3.2.2.

The instantiation condition ensures that a UMLClass artifact related as source is indeed persistent, and is implemented using a simple OCL specification **context**: UMLClass **inv**: self.kind = "persistent".[2]

The relation type is context composed into a relation of type Package2Schema. Furthermore, the composition defines that a Class2Table relation exists for any UMLClass artifact within a Package artifact package. The Class2Table relation type has a Schema artifact connected as source & target because a Table artifact will be added to a Schema artifact that is similar to the Schema artifact schema.

The third relation type is called Assoc2FKey and is one of the complex relation types. This relation type is shown in Figure 6.3. It defines the signature of a model transformation that transforms an Association artifact into a ForeignKey and a Column artifact, whereby the ForeignKey artifact refers to a Key artifact. The relation type also takes two distinct UMLClass artifacts as source, which act as source and target of the Association artifact. An implementation of a model operation represented by the execution operation of this relation type is shown in Section C.3.2.3.

The instantiation condition for Assoc2FKey is shown in Figure 6.4. The only two conditions which need to be checked are that the UMLClass artifact, which is the value of the source parameter src, is indeed the source of a given Association artifact, and that the UMLClass artifact, which is the value of the source parameter tgt, is indeed the target of the Association artifact.

The context of a relation of type Assoc2FKey is defined by means of two relations (c2t1 and c2t2) of type Class2Table, which capture that a UMLClass artifact sc has been transformed into a Table artifact sct and that another UMLClass artifact dc has been transformed too. The parameter type connectors additionally define the relationships between the artifacts from the context and the artifacts of the relation that is composed.

The fourth relation type is called PrimitiveAttribute2Column and defines the signature of a simple model transformation that creates a Column from a given Attribute, whose type is a PrimitiveDataType artifact

[2]In this situation, OCL can be applied as instantiation condition because OCL can only process a single context.

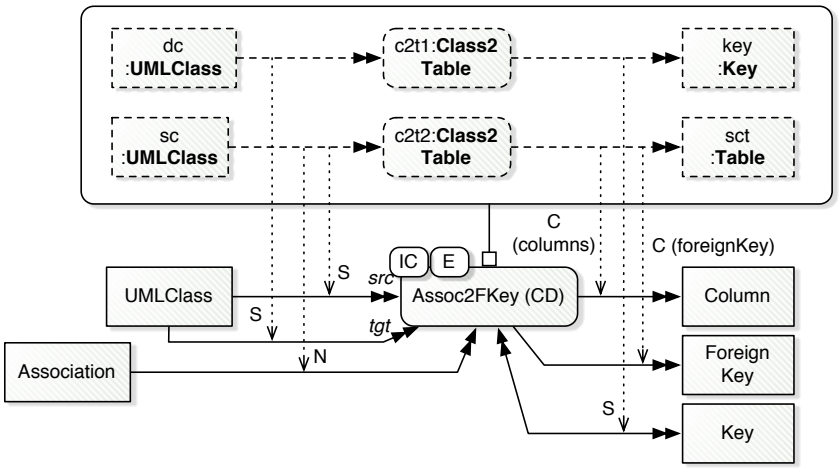

Figure 6.3.: Definition of the Assoc2FKey relation type

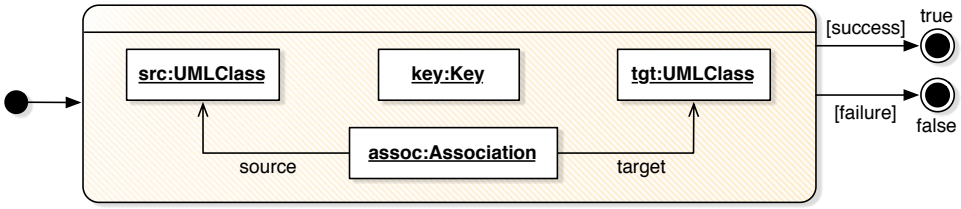

Figure 6.4.: Implementation of the instantiation condition of Assoc2FKey

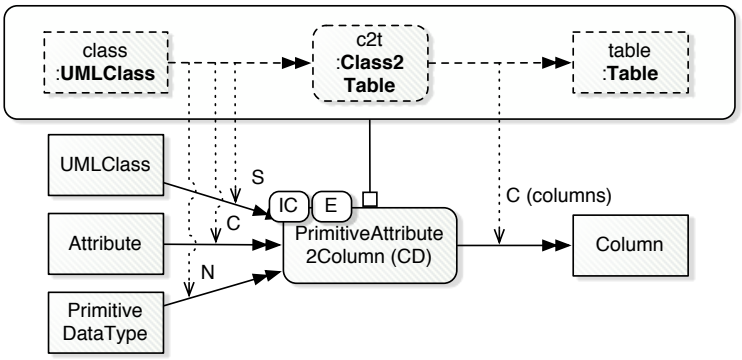

Figure 6.5.: Definition of the PrimitiveAttribute2Column relation type

(Figure 6.5). The implementation of the model transformation can be found in Section C.3.2.4 in detail. Instances of this relation type require passing a specific instantiation condition of that relation type. Thus, for each relation of this type connected UMLClass, Attribute and PrimitiveDataType artifacts must satisfy the following instantiation condition (Figure 6.6).

Relations of type PrimitiveAttribute2Column can only be instantiated into a specific context, which is a relation of type Class2Table. In that context, a UMLClass artifact connected to a relation PrimitiveAttribute2Column must be similar to the artifact class. The Attribute artifact must be a direct child of class and the PrimitiveDataType artifact is a neighbor of class. The resulting Column artifact is defined to be a child of the Table artifact table.

These relation types are currently not sufficient to do all necessary model transformations. Up to this point, Attribute artifacts, whose type is a PrimitiveDataType artifact, are not transformed if they are

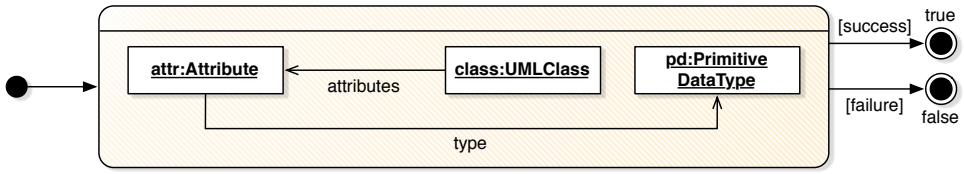

Figure 6.6.: Implementation of the instantiation condition of PrimitiveAttribute2Column

indirectly related to UMLClass artifacts that are transformed into Table artifacts. Indirectly related means two things. Firstly, Attribute artifacts that are contained by UMLClass artifacts which are themselves direct or indirect parents (super classes) of a transformed UMLClass artifact. Secondly, Attribute artifacts that are contained by UMLClass artifacts which are themselves types of other Attribute artifacts (complex data type), which are contained by UMLClass artifacts that are directly or indirectly related to UMLClass artifacts that are transformed into Table artifacts.

In order to also transform indirectly related Attribute artifacts, further relation types have to be defined. The relation type SuperClass, which is shown in Figure 6.7, is just used to build the composition context for the composition of other relation types, and does not provide an execution operation.

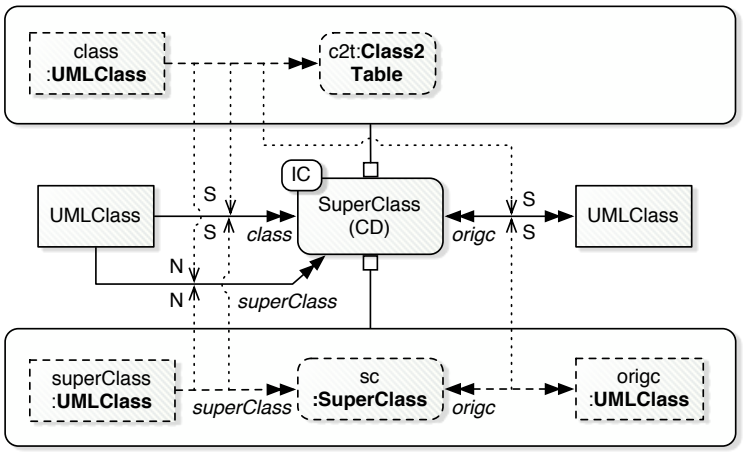

Figure 6.7.: Definition of the SuperClass relation type

Relations of that type indicate that two UMLClass artifacts are in a super class relationship. Thus, the UMLClass artifact which is the value of the source parameter superClass is the parent of the UMLClass artifact that is the value of the source parameter class. Because relations of this type can exist in a context of a relation of the same type, UMLClass artifacts are also identified that are indirect parents of other UMLClass artifacts, which is the value of the source & target parameter type origc.

Thus, for each Class2Table relation, a relation of type SuperClass is composed into a Class2Table relation for each UMLClass artifact that is a direct parent of the UMLClass artifact related to the Class2Table relation. Then, another SuperClass relation is composed into each SuperClass relation.

To ensure that UMLClass artifacts connected via class and superClass are indeed in a super class relationship, an instantiation condition is necessary as shown in Listing 6.1. The instantiation condition cannot be implemented using a Story diagram because the existing Story diagram interpreter requires isomorphism for all objects that are bound in a Story pattern.[3] This does not hold here because the UMLClass origc and clazz can be similar if composed into a relation of type Class2Table. Thus, the instantiation condition is implemented as an ordinary Java operation. The instantiation condition just checks if superClass is persistent and if superClass is a parent of clazz.

Based on this additional relation type, the previously shown relation type PrimitiveAttribute2Column can be reused to also transform Attribute artifacts into Column artifacts that are contained by UMLClass

[3]This means that two objects cannot be similar in a Story pattern.

```
1  public boolean superClassIC(UMLClass clazz, UMLClass superClass, UMLClass origc)
2  {
3    if (superClass.kind.equals('persistent'))
4      if (clazz.getParents().contains(superClass))
5        return true;
6    return false;
7  }
```

Listing 6.1: Implementation of the SuperClass instantiation condition

artifacts that are direct or indirect parents of a considered UMLClass artifact. This is obtained by providing another composition context for the relation type PrimitiveAttribute2Column as shown in Figure 6.8.

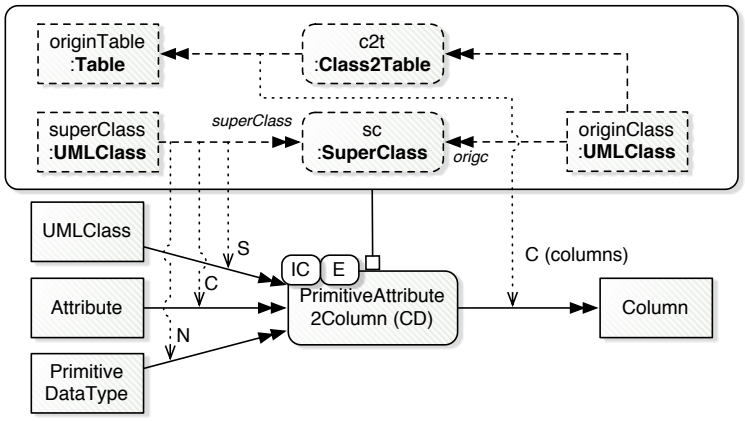

Figure 6.8.: Extension of the PrimitiveAttribute2Column relation type (A)

The context of a relation of type PrimitiveAttribute2Column is thus extended to be a relation of type Class2Table and a relation of type SuperClass, which both have the same UMLClass artifact connected (originClass) as source parameter. Based on such a context the Table artifact (originTable), created from the UMLClass artifact originClass, is used as a container for the Column artifact that is created by PrimitiveAttribute2Column.

If Attribute artifacts do not have PrimitiveDataType artifacts as type but other UMLClass artifacts, another relation type has to be introduced, which is called ComplexAttribute2Column (see Figure 6.9 and Figure 6.10). This relation type is just used to provide additional composition contexts to other relation types. Thus, it does not provide an execution operation.[4]

A relation of type ComplexAttribute2Column indicates that an Attribute artifact of a UMLClass artifact has another UMLClass artifact as type. Furthermore, a UMLClass artifact, which is connected via origc source parameter, is used to indicate the UMLClass artifact that was the starting point. In the case that the composition context is just a Class2Table relation, the UMLClass connected via origc is similar to the UMLClass artifact connected via attrc, which contains the Attribute artifact. A relation of type ComplexAttribute2Column can also exist in the composition context of a SuperClass relation and in the composition context of a ComplexAttribute2Column relation.

The instantiation condition of the relation type ComplexAttribute2Column is implemented as a simple Java operation, shown in Listing 6.2. This operation is satisfied if the type of a given Attribute is similar to the UMLClass connected via typec, if this UMLClass is defined to be persistent and if the UMLClass connected via attrc contains the Attribute attr.

Based on this new relation type ComplexAttribute2Column, another relation type composition is added to the already defined relation type PrimitiveAttribute2Column as shown in Figure 6.11.

The relation type composition defines a relation of type Class2Table and a relation of type ComplexAttribute2Column with both having the same UMLClass artifact connected (originClass) as source

[4]Figure 6.9 and 6.10 show a relation type but with three different relation type compositions. They are split into two Figures due to readability concerns.

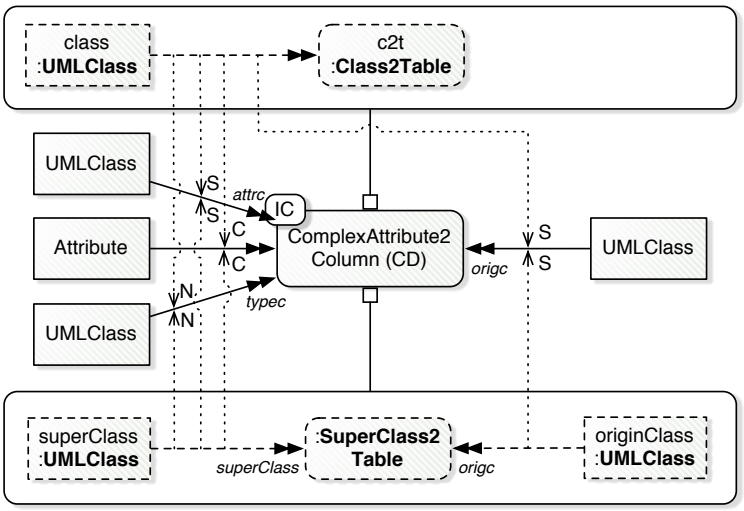

Figure 6.9.: Definition of the ComplexAttribute2Column relation type (A)

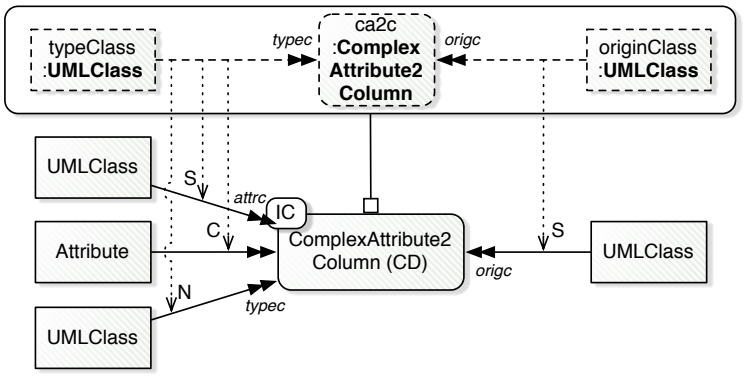

Figure 6.10.: Definition of the ComplexAttribute2Column relation type (B)

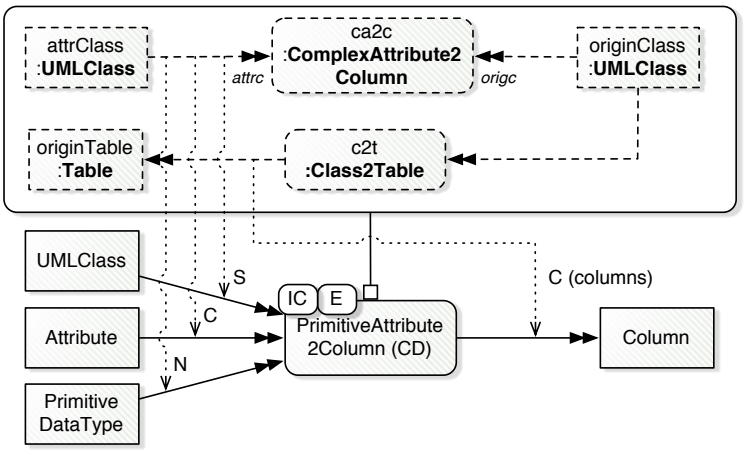

Figure 6.11.: Extension of the PrimitiveAttribute2Column relation type (B)

parameter. Based on such a composition context, the Table artifact (originTable), which is created from the UMLClass artifact originClass, is used as a container for the Column artifact that is created by Primi-

```
1  public boolean complexAttribute2ColumnIC(UMLClass attrc, UMLClass typec, UMLClass origc,
       Attribute attr)
2  {
3    if (attr.getType() == typec)
4      if (typec.kind.equals('persistent'))
5        if (attrc.getAttributes().contains(attr))
6          return true;
7    return false;
8  }
```

Listing 6.2: Implementation of the ComplexAttribute2Column instantiation condition

tiveAttribute2Column.

6.1.2. Execution: Extending Complex Model Operations

The purpose of this evaluation is to show that the introduced approach is able to extend legacy and complex model operations. This evaluation also shows that legacy model operations can be extended by model operation that are loosely coupled and highly cohesive. Furthermore, the legacy model operation is not impacted by the model operation that implements the extension.

This evaluation is applied to the case study example provided in Section 2.2.3. Subsequently, two different ways of realizing the extension are shown, using data-flow composition and context composition. It will also be explained how context composition is beneficial in such a scenario.

6.1.2.1. Extending via Data-Flow Composition

A first solution can be realized by employing the data-flow composition of the approach. Thus, a relation type is defined for the TGG model transformation from SysML models to AUTOSAR models, which is shown in Figure 6.12.

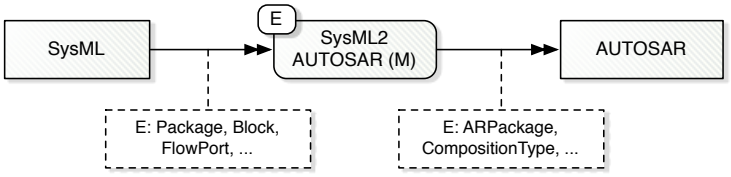

Figure 6.12.: Definition of the SysML2AUTOSAR relation type

The relation type SysML2AUTOSAR has a SysML artifact type as source and an AUTOSAR artifact type as target. The relation type has an execution operation that refers to the legacy TGG model transformation. The impact scope of the execution operation and the source parameter type is all artifact types that are direct or indirect children of the SysML artifact type except for the artifact type for requirements. The impact scope of the target parameter type is all artifact types that are direct or indirect children of the AUTOSAR artifact type, expect for artifact types coming from the AUTOSAR timing extension.[5]

Figure 6.13 shows the relation type of the extending model transformation called R2LTC, which transforms all timing requirements of a SysML model into timing constraints of an AUTOSAR model.

Therefore, a relation type R2LTC is created that has the same signature as the relation type SysML2 AUTOSAR. The difference is that R2LTC is connected to an AUTOSAR artifact type via the source & target parameter type, because it does not create an AUTOSAR model but only extends it. The impact scopes are also different because the new model transformation, which is represented by the execution operation of R2LTC, only processes artifacts of type Block, FlowPort and Requirement in order to manipulate artifacts of type CompositionType, or to create artifacts of types from the AUTOSAR timing extension.

[5]Due to the high number of artifact types, only a few are shown here.

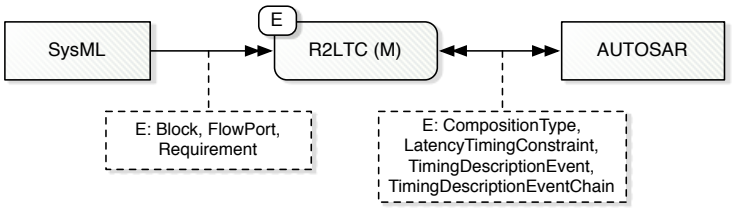

Figure 6.13.: Definition of the R2LTC relation type

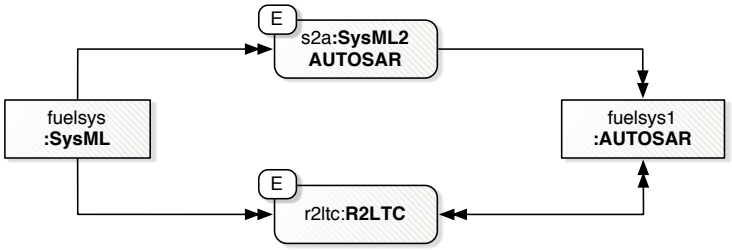

Figure 6.14.: Application megamodel of the extended SysML to AUTOSAR workflow

Figure 6.14 illustrates an application megamodel, which contains an instance of SysML2AUTOSAR (s2a) and an instance of R2LTC (r2ltc).

The major issue of this solution is the fact that the implementation of the model transformation related to the execution operation of R2LTC has a rather low cohesion because it has to transform all requirements from a SysML model instead of focusing on the transformation of a single requirement, and because it needs to find the right CompositionType into which a Requirement should be transformed. The implementation is also implicitly coupled to the legacy model transformation because it requires reasoning about how a Block, which contains instances of Requirement, relates to a CompositionType, which is the result of transforming a Block. Thus, the extending model transformation has redundant aspects from the legacy model transformation, which is considered as an implicit coupling and of a rather low cohesion.

6.1.2.2. Extending via Context Composition

A second solution is to realize the extension via a context composition. This solution will not require the implementation of the extending model transformation to be less cohesive and implicitly coupled to the legacy model transformation. Furthermore, two scenarios are shown that can be realized by using context composition.

6.1.2.2.1. Extending without Proprietary Traceability In the first scenario, it is assumed that the model transformation, which is going to be extended, only takes a SysML model as input and creates / updates an AUTOSAR model as output. Thus, the first relation type is similar to the one that is shown in Figure 6.12.

Because a Block contains requirements and a CompositionType contains latency timing constraints, a relation type called Block2CT, which just defines that a CompositionType belongs to a Block, is introduced, as shown in Figure 6.15. This relation type is defined to be composed into the context of a SysML2AUTOSAR relation. In this context composition, a Block is a direct or indirect child of the SysML model connected to SysML2AUTOSAR and the CompositionType is a direct or indirect child of the AUTOSAR model connected to SysML2AUTOSAR. The relation type Block2CT is related to an instantiation condition which checks whether a Block and a CompositionType artifact connected to a relation of type Block2CT indeed belong together. The implementation of the instantiation condition is a Story diagram and is shown in Figure 6.16.

The Story diagram is satisfied if a given Block and a given CompositionType have equivalent names.

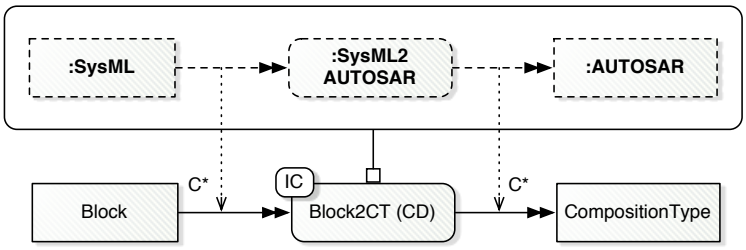

Figure 6.15.: Definition of the Block2CT relation type

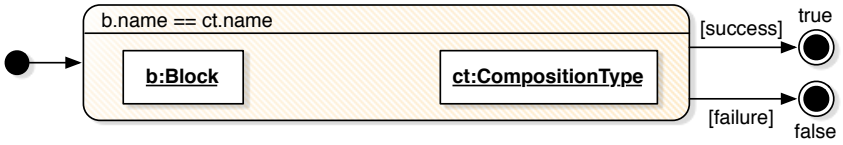

Figure 6.16.: Implementation of the instantiation condition of Block2CT using name equivalence

The legacy model transformation indeed creates CompositionTypes from Blocks and sets the name of the CompositionType to the name of the Block. Thus, this simple condition infers a dependency between a Block and a CompositionType by means of a simple heuristic. However, this simple condition assumes that the names of the Blocks in a SysML model are unique.

Now, the extending model transformation can be implemented for a single requirement that is transformed into a single latency timing constraint. Furthermore, this implementation does not need to include any aspects that reason about the dependency between a Block and a CompositionType. It is assumed Block artifacts somehow depend on CompositionType artifacts. This is realized by composing the relation type R2LTC into the context of a Block2CT relation as shown in Figure 6.17.

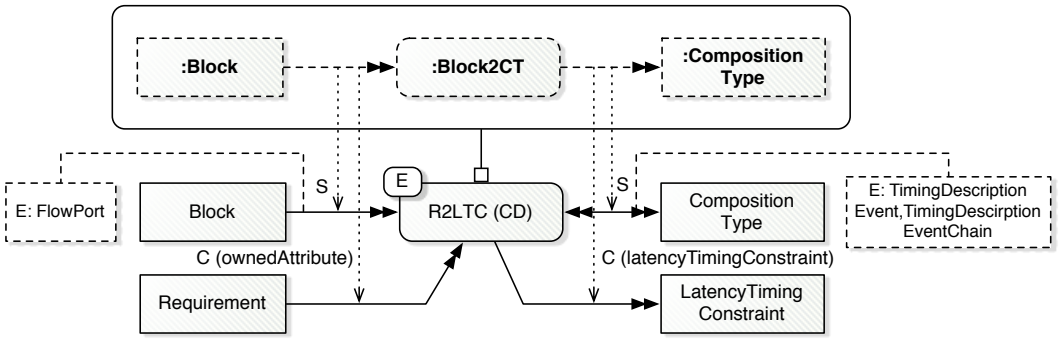

Figure 6.17.: Definition of the R2LTC relation type (fine-grained)

The new relation type takes a Block as source that is defined to be similar to the Block that is connected to Block2CT, and a CompositionType as source & target that is defined to be similar to the CompositionType of Block2CT. The latter artifact type is connected via source & target because the corresponding execution operation will manipulate the CompositonType. Furthermore, it takes a Requirement as source that is directly contained by the Block of Block2CT by means of an artifact context of type ownedAttribute. From this Requirement, the execution operation creates a LatencyTimingConstraint into CompositionType of Block2CT as container (latencyTimingConstraint), which is connected as target to R2LTC.

The execution operation of R2LTC has two impact scopes. The first impact scope is connected to the source parameter and is defined only for the artifact type FlowPort because the model operation processes FlowPort artifacts of a Block artifact that are indirectly referenced by the Requirement. The

second impact scope is connected to the source & target parameter and is defined over the artifact types TimingDescriptionEvent and TimingDescriptionEventChain because artifacts of these types will be additionally created and added to the CompositionType artifact and the LatencyTimingConstraint artifact, respectively.

6.1.2.2.2. Extending with Proprietary Traceability In certain situations inferring a relation between artifacts is not possible just by analyzing the artifacts (state), e.g., if the names of Block artifacts are not guaranteed to be unique. Instead, it might be necessary to directly analyze/monitor the behavior of some model operation to get insights about dependencies. Recent model operation technologies come with built-in traceability support, e.g., [36, 54, 82, 171, 68, 13, 86]; these technologies create traceability between processed artifacts as a by-product of their application.

In the following, it is shown how this information can be leveraged for the context composition by means of the same application example.

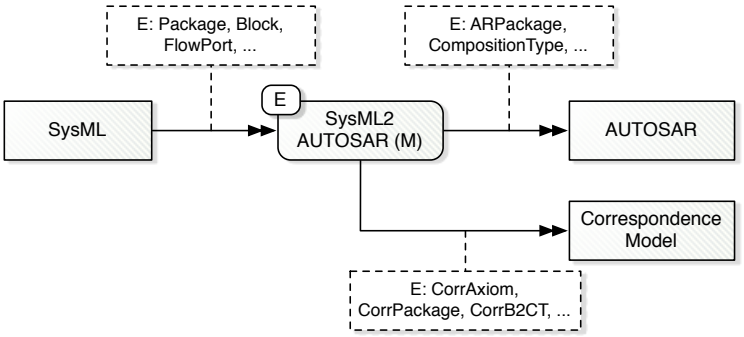

Figure 6.18.: Alternative definition of the SysML2AUTOSAR relation type

Figure 6.18 shows an alternative of the relation type Block2CT as previously presented in Figure 6.12, which also takes the creation of traceability (correspondence model) into account. The legacy model transformation is implemented by means of TGGs and provides a correspondence model when applying. The correspondence model contains correspondences between artifacts that were transformed. The adapted relation type makes this correspondence model explicit by adding it as target to it. The new parameter type is also connected to an impact scope that defines which artifact types are added to the correspondence model.

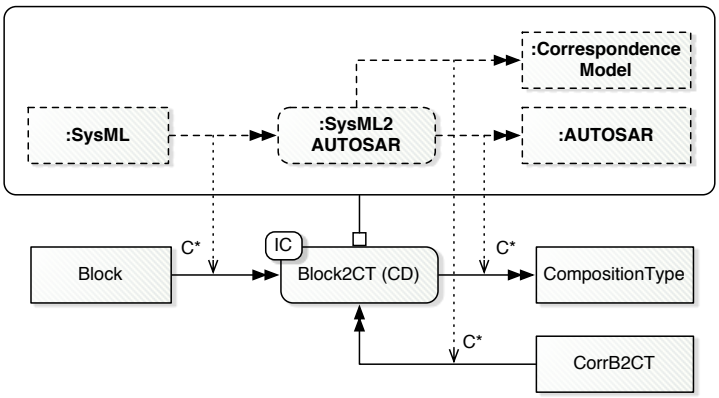

Figure 6.19.: Alternative Definition of the Block2CT relation type

Based on the adapted relation type SysML2AUTOSAR, the correspondence between a Block and a CompositionType can be identified by exploiting a specific correspondence in the CorrespondenceModel.

Thus, the relation type Block2CT is slightly extended and composed into the adapted relation type SysML2AUTOSAR as shown in Figure 6.19. The adapted relation type B2CT also takes a correspondence CorrB2CT as source, which is defined to be a direct or indirect child of the CorrespondenceModel of SysML2AUTOSAR.

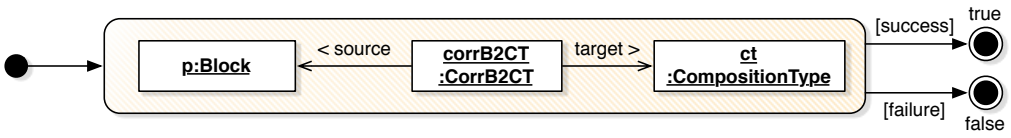

Figure 6.20.: Implementation of the instantiation condition of Block2CT using a correspondence

Now, the only detail that has to be checked is whether the artifact CorrB2CT indeed interconnects the given artifacts Block and CompositionType. This is obtained by implementing a simple instantiation condition as shown in Figure 6.20. Because of the loose coupling, the relation type R2LTC, which represents the signature of the extending model transformation (see Figure 6.17), the model operation it represents does not need to be adapted at all.

In this scenario, using the context composition is beneficial because the implementation of R2LTC is highly cohesive and loosely coupled. It is highly cohesive because it does not need to implement those aspects which are only necessary for finding appropriate pairs of Block and CompostitionType artifacts, which depends on the transformation of these artifacts. The implementation of R2LTC is loosely coupled for the same reason. It does not care about how a Block and a CompositionType actually relate to each other. It is only important that such pairs exist.

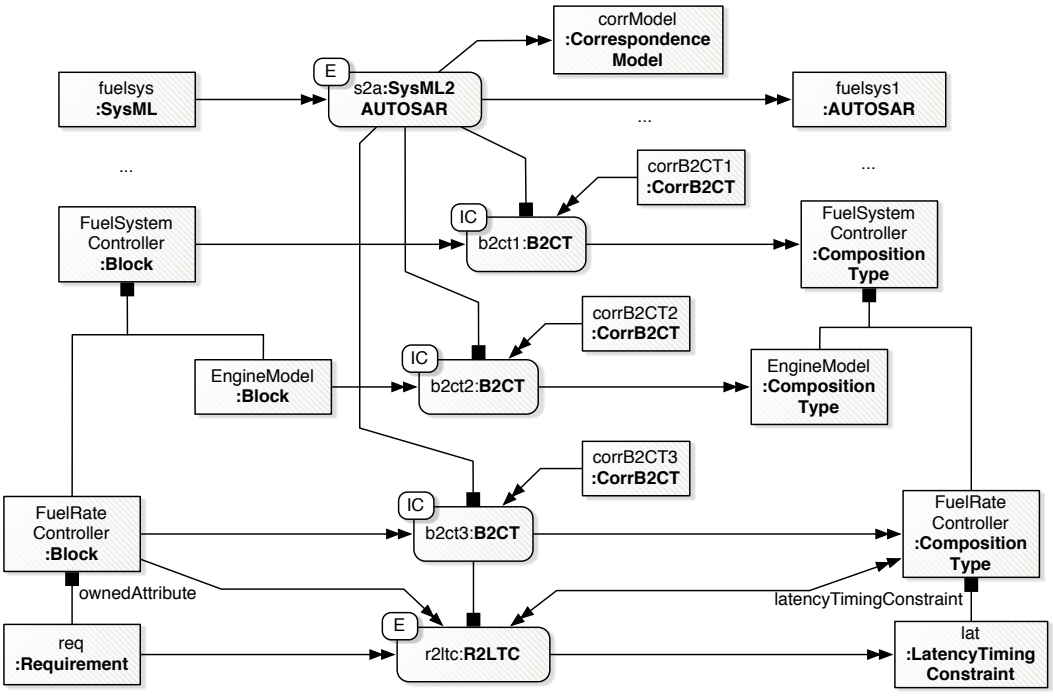

Figure 6.21.: Application megamodel after applying a SysML2AUTOSAR relation (s2a)

Figure 6.21 shows an application megamodel that results from executing the relation s2a of type SysML2AUTOSAR. The SysML and the resulting AUTOSAR example model is shown in Section C.3.1 as EMF tree.

6.2. Discussion

Now, the shown approach is discussed with respect to the initially stated goals, and the challenges addressed by these goals.

6.2.1. Capture Dependencies

The hierarchical megamodel uses relations to explicitly capture physical dependencies. A relation is considered as a directed relationship between a set of artifacts. The semantic of a physical dependency is declared as being a physical dependency type. This is explicitly captured by means of relation types. Thus, relations even have a semantic by explicitly defining that the type of a relation is a relation type.

In Chapter 3, the concept of relations and relation types was introduced by means of a case study that is shown in Section 2.2.1. In Chapter 3, the concept of relations and relation types was introduced by means of a case study that is shown in Section 2.2.1. In that chapter, it was also shown that physical dependencies can exist in the context of other physical dependencies and that this can be captured by means of a hierarchical megamodel using the context composition. For example, if a logical component in a solution architecture depends on a logical component in a reference architecture, a connector of the first logical component can depend on a connector of the latter logical component (cf. Figure 3.15).

By capturing physical dependencies by means of relations in application megamodels and capturing their types by means of relation types in configuration megamodels, the hierarchical megamodel allows application developers to create relations between heterogeneous artifacts. These relations can be used for visualization purposes, e.g., defining an explicit view on the hierarchical megamodel that focuses on showing the relationships between artifacts. The relations can also be employed for impact analysis or to navigate between heterogeneous artifacts (see [86]). Thus, additional tools can leverage the information that is provided by the application megamodel for different purposes.

Nevertheless, defining relations between artifacts only restricts capturing physical dependencies between physical artifacts that can be represented by the application megamodel. For example, the hierarchical megamodel is not able to define relations between attributes of classes, because attributes are not going to be represented by application megamodels.

6.2.2. Automatically Maintain Dependencies

The dynamic hierarchical megamodel introduces further concepts to enable the automated creation, update and deletion of relations in application megamodels by facilitating relation types provided by configuration developers in configuration megamodels. Thus, the dynamic hierarchical megamodel can be applied to solve traceability tasks, which require at least the automated creation of traceability links (relations).

In Chapter 4, the case study of Section 2.2.1 has been applied to show the automated maintenance. It has been shown that instantiation conditions are used to represent specific kinds of model operations that are employed to automatically reason about the validity of artifact contexts of relations. Furthermore, context composition of relations is enriched by providing a relation type composition specification. This specification allows for the definition of the composition context of a relation, and thus can be employed to automatically reason about the validity of composition contexts of relations. The actual automated maintenance has been realized by means of the localization, which was shown in two different versions (batch and incremental).

The dynamic hierarchical megamodel as a traceability approach is best applicable in the MDE domain because artifacts are primarily formal. In MDE, three kinds of dependencies can be covered between artifacts that are hard references, soft references and semantic connections (cf. [106]).

A hard reference is an explicit reference between software artifacts encoded in the language of the software artifacts. The approach translates these explicit references into relations of an application megamodel. A soft reference is an implicit reference between artifacts encoded by means of equivalence of certain attributes. Such soft references occur in heterogeneous MDE environments working with loosely coupled metamodels. The shown approach can make these dependencies explicit by automatically creating relations from them. A semantic connection is any overlap between artifacts, which basically mean the same thing but have different syntactical representations. Semantic connections are covered by the capability to integrate any technology or language for implementing instantiation conditions.

The shown approach can be considered as a retrospective traceability approach. From this view, it only reasons about the existence of physical dependencies by means of analyzing the structure and attributes of existing artifacts. Prospective and retrospective traceability approaches are complementary because both are disjunctive concerning their ability to maintain the existence of relations. Both have their advantages and disadvantages. In [107], Lucia et al. argue that those approaches should complement each other.

Furthermore, it is expected that all automatically created relations are of the highest precision because relation type composition specifications and instantiation conditions are precise methods which do not use any heuristics. However, it is assumed that a configuration developer, who implements the instantiation conditions, only defines precise and complete model operations for instantiation conditions. Whenever such a model operation is imprecise or incomplete, the precision of automatically created relations is questionable. Nevertheless, the precision or completeness cannot be validated because it always depends on the specific intention of each individual type of physical dependency.

A similar situation holds for the question about recall. Currently, it is assumed that all necessary relation types are formally underpinned by means of instantiation conditions and relation type composition specifications. However, the creation of relations might be missed if the instantiation conditions are specified imprecisely, or if whole relation types are missing.

Nevertheless, a configuration developer may be supported in implementing model operations for instantiation conditions. There are approaches that focus on the automated creation of mappings between metamodels [46]. These approaches may use heuristics as in information retrieval approaches. Because of applying heuristics, the established mappings underlie the same accuracy issues as the usual traceability approaches. However, employing such approaches can be used to automatically synthesize model operations for instantiation conditions or guide a configuration developer in specifying them.

6.2.3. Automatically (Re-)Apply Heterogeneous Model Operations

The application and re-application of model operations has been shown in Chapter 5 by using the executable and dynamic hierarchical megamodel. The shown approach does not explicitly distinguish between the application and the re-application of model operations because the execution only requires the existence of relations for (re-)applying model operations. The only differentiation is done in the *executeRelation* operation (see Listing 5.2) because when a relation is applied for the first time, all target artifacts must first be created by the approach.

The execution comes with two different execution strategies. The individual execution strategy only allows for the execution of those model operations that are directly or indirectly composed via context into the model operation triggered for execution. The complete execution strategy allows for the execution a set of model operations, e.g., all model operations that have been impacted by changes since last (re-)application or only a subset of them. The complete execution strategy applies all model operations that are in context compositions and that are data-flow compositions. Therefore, it automatically generates a schedule for (re-)applying all necessary model operations in a correct order. The execution has been explained by means of an example from the case study that is shown in Section 2.2.2.

Thus, by means of this approach application developers just need to select model operations (relations) from a list of impacted relations which should be (re-)applied. Generally, an application developer does not need to care about which model operations have to be applied (when and where). An application developer only needs to trigger the (re-)application. The model operations that can be selected are automatically estimated by an impact analysis.

Nevertheless, when making changes, which are scattered over diverse artifacts, conflicts may occur. These conflicts occur because changes from different sources may get propagated by applying model operations toward similar artifacts. In the worst case, this leads to overwriting changes, as well as syntactic and semantic inconsistencies within individual artifacts. This can be minimized by making changes locally and (re-)applying model operations frequently.

6.2.4. Specify and Apply Compositions of Heterogeneous Model Operations

The executable and dynamic hierarchical megamodel enables the integration of model operations. In combination with the ability to specify compositions between relation types, composition can also be

specified between model operations because model operations are represented by relation types and execution operations that are related to relation types.

In Chapter 5, it has been shown how two different kinds of compositions can be specified between model operations – by means of context compositions and by means of data-flow compositions.

The specification of a context composition is obtained by means of a relation type composition and a relation type composition specification. This was already introduced for relation types in Chapter 4.

A data-flow composition is explicitly specified by means of a module that was introduced in Section 5.2.1.4. A module is a specialization of a relation type, which provides two further concepts. A module supports encapsulation of a set of relation types such that other relation types outside the module are not allowed to use them in the composition context. Furthermore, a module supports specifying a pattern over relations and artifacts by means of a module specification. This pattern is employed to explicitly specify data-flow compositions of relation types within a module.

The case study, shown in Section 2.2.2, is used to explain how context compositions and data-flow compositions of model operations can be specified. In this chapter, the same case study is taken to also show how a complex model operation can be specified by means of compositions of fine-grained model operations (Section 6.1.1). The other case study, shown in Section 2.2.3, is evaluated to show a scenario of extending an existing model operation by means of a context composition.

Using this approach, model operations can be composed, via context or data-flow compositions, with other model operations. Using the context composition, the approach is able to build or extend existing model operations. Even complex compositions are possible because relation types can have multiple other relation types that are superiors in a relation type composition.

Because compositions are specified by means of relation types, model operations can even be in a context composition with arbitrary traceability information. For example, a relation type may encode a certain condition that should be used to trigger a model operation. This can be achieved by simply defining a context composition between the relation type, which encodes the condition, and the relation type, which represents the actual model operation. This also works the other way around. Traceability information can use applied model operations for their own context using the context composition. Thus, there is a new degree of freedom concerning the flexibility of compositions.

A model operation that is composed by means of this approach is loosely coupled and highly cohesive. This is true because everything that is required for the composition is part of the demonstrated approach and not part of the model operation. Thus, the actual model operation does not know any other model operation – neither the model operations, into which it is composed, nor the model operations that are composed into the model operation itself. Thus, the shown approach is completely transparent to model operations that are going to be used. This enables the composition of model operations as true black-boxes.

Furthermore, the condition of when a model operation should be applied is separated from the actual model operation implementation (instantiation condition versus execution operation). This also increases cohesion because the same model operation can be reused in different contexts by replacing the instantiation condition without changing the implementation of the model operation.

Nevertheless, the integration of model operations into the shown approach may influence the implementation of the model operations, but neither does it allow the integration of any arbitrary model operation. There are two reasons for this.

Firstly, a model operation must satisfy a set of requirements to be appropriately represented by an execution operation (see Definition 5.2.3). If a model operation does not satisfy these requirements, the integration may fail. These requirements are necessary to transfer the control from individual model operations to the approach that is responsible for the coordination of the model operations.

Secondly, the need for impact analysis influences what a relation type looks like, which impacts the implementation of a model operation. For example, Assoc2FKey (see Figure 6.3) does not necessarily require the source and target UMLClass artifacts because these artifacts can be traversed from the Association artifact within the implementation of the model operation. Nevertheless, they are required to correctly analyze the impact of changes because changing the name of one of these two UMLClass artifacts can change the name of the Column artifact or the ForeignKey artifact. However, this comes with a current shortcoming of the approach that the impact scope only works for direct or indirect subordinates of artifacts. If this shortcoming can be overcome, relation types can be specified more intuitively and thus, the implementations of fine-grained model operations may become more reusable.

6.3. Summary

In this chapter, two qualitative evaluations have been conducted by providing solutions for application scenarios taken from two case studies. In the first scenario, it has been shown that the approach can be used even to build complex context compositions by specifying a declarative model transformation. In the second scenario, it has been shown that the approach can also be employed to extend legacy model transformations using the context composition without coupling the extended model transformation to the extended model transformation. Furthermore, the satisfaction of the initially stated goals has been explained and critically discussed.

7. Related Work

Contents

* * *

The contribution of this thesis is a comprehensive model management approach build on top of a traceability approach.

The traceability approach supports capturing versatile dependencies between heterogeneous artifacts (models, documents, etc.) at any level of detail, including an automated maintenance of these dependencies in case of changes to artifacts. The comprehensive model management approach supports the specification and (re-)application of complex compositions of heterogeneous model operations at any level of detail. Through these means, the approach can be used to build complex compositions of all kinds of model operations. It can also extend existing model operations without either coupling their implementations to each other, or to some composition framework, which is considered as a true black-box.

In this chapter, related work in the domain of traceability and composition of model operations (model transformations and model management) is compared to the demonstrated approach. Because the complex composition of heterogeneous model operations is the major contribution of this thesis, the main focus is put on the second part of this chapter.

7.1. Traceability

In this section, the shown approach is compared to related work in the context of traceability. More specifically, the shown approach is first delimited against traceability approaches in the domain of MDE and subsequently delimited against traceability approaches in the domain of model management.

7.1.1. Traceability in Model-Driven Engineering

Many traceability approaches in the context of MDE are retrospective approaches, which only rely on information retrieval methods to automatically establish traceability links [16, 17, 40, 107, 45, 19]. These approaches focus only on the initial creation of traceability links and the attained quality of their outcome with regards to precision and recall.

In [122] Nguyen et al. introduce an approach called Software Concordance to manage versions of software documents and traceability links (relations) in between. Their focus is on the invalidation of traceability links whenever anchor (connected) software artifacts are created, deleted or updated. Each time a traceability link is invalidated, a new version is created. They employ a timestamp strategy to heuristically reason about the conformance of software artifacts connected to these invalidated traceability links.

In [112] they argue that relying on a timestamp strategy alone is insufficient. To accomodate this conclusion, the semantic of the underlying change should also be taken into account.

However, this thesis takes a different approach to the notion of maintenance, since it means reasoning about the existence of relations, and not about the effect of conformance of connected artifacts. If an

instantiation condition is triggered on a relation, the context of the impacted relation has changed in a way that the necessary condition for its existence may not hold anymore. In certain situations, the deletion of invalidated relations may be too restrictive.

In [81] Jirapanthong and Zisman have shown a comprehensive work on the establishment of automated traceability in the context of software product lines. It is a semi-formal method for establishing traceability links based on a combination of XML-based rules and LSI. However, they do not support an incremental maintenance of traceability links.

Ivkovic and Kontogiannis show another traceability approach toward automated establishment of traceability links (model dependencies) in [79], using a combination of heuristic and formal methods to establish traceability links (using type-based, spatial and text-based association rules). Their approach, however, does not provide the notion of incremental traceability maintenance.

Maletic et al. show an XML-based approach toward traceability in [111]. In their paper, the evolution of traceability in the domain of MDE is discussed. Their discussion is focussed on whether traceability links should evolve whenever software artifacts change. They do not give any insight into how to actually maintain traceability links in the case of changes.

Another idea toward a retrospective traceability approach is discussed in [9]. It advocates that traceability links should be maintained by means of formal operational semantics. The authors propose using the event-condition-action (ECA) method to realize the maintenance of traceability links. However, they do not show how these operational semantics look like, nor how they actually realize the maintenance of traceability links by means of ECA.

The prospective traceability approaches are incremental by nature, because they establish and de-establish traceability links by means of changes or change records.

Mäder et al. have shown in [108, 109, 110] a prospective approach to incrementally maintaining traceability links in the UML context. They employ rules used to specify development activities. In addition, they define in detail how elementary changes affect existing traceability links, and whether new traceability links have to be established. Thus, their notion of incremental traceability maintenance is similar to the one shown in this thesis. The only difference is that they do not consider the initial establishment of traceability links because their approach is not retrospective.

In [18] a multi-faceted traceability approach is discussed, where traceability links are used to connect development artifacts in different views. The approach proposes capturing traceability links as a side effect of development tasks. Traceability links may also be captured and classified by any kind of formally specified rules, which would at least enable the initial creation of traceability links. In a comparison to this approach, they mention subsequent validation activities, as proposed in a subsequent execution process. Further, the paper outlines requirements for a proposed traceability approach but no further elaboration.

Common prospective traceability approaches in the context of MDE are ones which automatically establish traceability links as by-products of the application of model transformations (e.g., TGGs) [36, 54, 68, 69, 82, 171, 86]. However, these approaches are restricted in establishing traceability links only in combination with the application of model transformations. Furthermore, whenever software artifacts change, the model transformations have to be completely re-applied.

Prospective approaches can be considered as complementary to retrospective approaches, but because no change information is available, they are not sufficient in the case where inferring traceability links is required.

7.1.2. Traceability in Model Management

Salay et al. show a traceability approach in the context of model management [143, 145, 146]. They use a formal method to establish traceability links (relationships) based on metamodel morphisms, which they call a macromodel. This macromodel is used to automatically establish traceability links. They guide the user to complete the models in such a way that traceability links are inferred. However, they do not consider further maintenance questions.

In [23] there is a clear separation of classical traceability (traceability-in-the-small) and traceability in model management (traceability-in-the-large). Classical traceability is considered as the ability to define weaving models which are used to relate model elements of different models. The approach should help to un-pollute traceability models in model management by putting traceability-related information into a megamodel. Therefore, the common megamodel is adapted by replacing simple traceability links

with traceability models that have source and target relationships to different models in the megamodel. In their particular case, a traceability model is a weaving model implementing traceability between models by defining mappings between model elements. Traceability information is further automatically established by weaving instantiated models into a traceability model. The approach of this thesis differs from this as it does not apply weaving as a technique to maintain traceability. Using weaving techniques is restricted regarding maintenance support for traceability links. If the source models change, the whole traceability model needs to be (re-)generated. Furthermore, there is no concept for applying subsequent validation techniques. In this thesis no strict separation between classical traceability and traceability in model management is attempted. The executable and dynamic hierarchical megamodel contains representations of models as well as model elements in combination and with explicitly captured and hierarchical dependencies.

7.2. Composition of Model Operations

In this section, related work is presented by classifying it into two different categories. The first category is comprised of model transformation approaches that provide the notion of decomposition of model transformations into smaller model transformation tasks (transformation rules or modules) and their subsequent composition. These are primarily declarative or hybrid model transformation approaches. The second category comprises dedicated approaches toward the composition of model operations (transformations), which are basically model management approaches or approaches that evolved in the context of model management.

7.2.1. Composition in Model Transformations

Composition in the context of model transformations is not directly related to the approach shown in this thesis. Nevertheless, under certain assumptions, they can be considered as an approach that can be employed toward model management. The discussed approaches are outlined concerning their composition abilities and their ability to integrate heterogeneous model operations as black-boxes.

GReAT is a graph rewriting and transformation language [7], consisting of a pattern specification language, a graph transformation language and a control flow language. The pattern specification language and the graph transformation language are employed to specify graph transformation rules. Each rule has a set of input and output ports that are mapped to objects in the graph pattern of a rule (input = initial match of the pattern and output = match after graph rewriting). By means of the input and output ports, rules can be chained in a specific control flow using the control flow language. The control flow language supports sequenced execution, parallel execution and nesting of rules (compound rule). Furthermore, the control flow language supports the specification of conditional branching.

This has several similarities to the approach shown in this thesis. A rule can be considered as a relation type with input ports similar to source parameter types, and output ports similar to target parameter types. A compound rule is, to some extent, similar to the specification of a module. Thus, GReAT supports the specification and execution of sophisticated data-flow compositions, but only for a specific pattern specification and transformation language. GReAT also does not provide a concept similar to that of context composition. Finally, GReAT can only process UML based models whereas the shown approach supports the processing of any kind of artifact that can be represented by an application megamodel, even folders, files, images, etc.

Tefkat is a declarative model transformation language that has been developed in the context of QVT [104]. In Tefkat a model transformation is a named entity with parameters defining the input and the output of the model transformation. The specification of a model transformation further contains any number of class definitions, rules, pattern definitions and template definitions. A class is used for tracking purposes (traceability) and specifies the class of a relationship between source and target elements of a model. A rule is the major concept of a model transformation, which consists of a source and a target constraint. The source constraint defines when and where a rule matches, and the target constraint defines the characteristic of the target for a match. Pattern and template definitions can be employed to parameterize source and target constraints if they should be used in multiple rules.

Tefkat provides an explicit and internal concept to compose individual rules into a coherent model transformation. Therefore, a rule explicitly requests the existence of a certain class (using the LINKS

keyword), which can be created by rules explicitly (using the LINKING keyword). Thus, Tefkat provides context composition of rules by means of traceability information.

Compared to the shown approach, a rule can be compared to a relation type. The source constraint has similarities to an instantiation condition, while the target constraint has similarities to an execution operation. The context composition of the shown approach is also realized by means of traceability (relations that are instances of relation types). Nevertheless, the context composition is defined to be explicit but external. Thus, the implementation of an instantiation condition and an execution operation does not need to directly process (create/interpret) a relation. Furthermore, Tefkat does not support the integration of external model operations. Thus, it is not suitable for composing heterogeneous model operations as true black-boxes.

TGGs were first introduced in [147] to cover the issue of data integration of graph-like structures that occur because of employing various software development tools. The primary concept of TGGs is to employ an explicit specification of a correspondence between a left-hand side and a right-hand side. Nowadays, TGGs have been applied in MDE to realize model-to-model transformations and synchronizations [68, 69]. The correspondence between artifacts on the left-hand side and artifacts on the right-hand side is specified by means of an axiom and a set of correspondences. The axiom is always the top-level correspondence that is used to initiate the model transformation. A correspondence represents an $n : m$ relationship between artifacts. It can depend on the existence of a set of superior correspondences. Thus, the whole model transformation / synchronization is defined as a set of (dependent) correspondences that must be interpreted in a certain order.

The idea of correspondences is similar to the idea of relation (types). As correspondences can depend on a set of correspondences, a relation (type) can depend on the existence of a set of relation (types) (context composition). However, the major difference is that TGGs do not consider the integration of heterogeneous model operations as shown in this approach. Thus, the capabilities of TGGs are restricted to transforming or synchronizing a set of models.

In [83, 84, 85, 101] a hybrid model transformation called ATL is shown. An ATL model transformation consists of a set of transformation rules, which can be either declarative (matched rule) or imperative (called rule). A matched rule consists of a source pattern, a guard on the source pattern and a target pattern, which is used to manipulate the target model. The guard is specified by means of a boolean OCL expression. A called ruled is a procedure that can be explicitly invoked by other rules with a given set of parameters. It can be implemented by means of a target pattern (source pattern is given by the parameters) or by calling a native Java operation. If a called rule is implemented by means of a target pattern, it can further provide an action block that may provide an additional control flow.

Considering ATL as a dedicated composition language is possible with certain restrictions. A matched rule can implicitly call another transformation rule by implicitly triggering a resolution mechanism when executing the target pattern. Other transformation rules are automatically invoked when they are able to provide the required context for applying the target pattern. This can be considered as an instance of a context composition. In ATL transformation rules can explicitly call a called ruled, which can also be employed to specify context compositions.[1] A data-flow composition can be specified by invoking another called rule on elements in the target pattern of the transformation rule that is invoking the called rule.

Nevertheless, ATL as a dedicated composition language has certain deficiencies. Firstly, transformation rules that are composed within ATL are always assumed to be model transformations that are going to create a certain target. Thus, ATL does not support the composition of transformation rules into an arbitrary traceability context but only composition into the context of another model transformation. Secondly, a matched rule always has only one element in the source pattern, which does not allow reasoning about a set of independent models within a single matched or called rule. Thirdly, the guard is expressed by means of OCL. Thus, reasoning about the validity of the source pattern (artifact context) is only possible on artifacts that can be parsed via OCL. The shown approach supports any language or technology (instantiation condition), which makes it able to express guards even on artifacts that are not compatible to OCL.

In [93, 97] the model transformation language ETL is presented. It is a hybrid model-to-model transformation language, which can transform a set of input models into a set of output models. An ETL transformation is defined in a module, which has a pre- and a post-condition and consists of a set of

[1] ATL supports rule inheritance. However, this is not considered as a possible composition technique in this thesis because it requires white-box model operations.

transformation rules. A transformation rule further has a single source parameter and a set of target parameters. A transformation rule has a body and an optional guard, both of which are expressed via EOL. The body defines the actual transformation while a guard is a mixture of a condition and a statement like the body. Executing an ETL module will result in applying all non-lazy transformation rules (greedy). When applying a transformation rule, the body of the rule may invoke the application of other (lazy) transformation rules by explicitly invoking the equivalent(s) operation on a source parameter and for a set of transformation rules, which requires an implicit traceability feature of ETL.

Thus, ETL supports context composition for lazy transformation rules, which is similar to called rules in ATL. Nevertheless, employing ETL to compose model operations in a model management context is not sufficient due to the following reasons. ETL does provide the specification of control flows in the body of a transformation rule but it does not allow for specifying data-flow compositions between transformation rules because transformation rules can only operate within artifacts of the superior module and the equivalent(s) operation can only be invoked on source artifacts. However, to specify a data-flow it is required to call equivalent on a target artifact. ETL also only supports EOL as the language to express the body and the guard of a transformation rule. In addition, because the composition of transformation rules is explicitly specified within the body, exchanging the language for specifying the body would implicitly require that other languages than EOL have to use the equivalent(s) operation to specify compositions. This, however, would violate the true black-box principle.

In [173] a declarative model transformation approach has been presented that is built atop of AMW and ATL. Their approach provides a syntactic as well as semantic extension to specify mappings by extending AMW. Because AMW does not come with an execution semantic, their approach provides an execution semantic by generating ATL model transformations from their extended AMW mappings. Their approach provides an additional concept called mapping operator (MOp) to specify reusable mappings by providing explicit composition concepts. An MOp is a mapping between metamodel elements and can contain further MOps. The composition between MOps is explicitly specified by means of context passing (exchange traceability information).

Generally, the idea of an MOp partly overlaps with the idea of a module and a relation type. An MOp has a signature that is similar to the parameters of a relation type. A set of relation types can be composed via context composition, an idea which is similar to the composition via context passing between MOps.

Nevertheless, the composition between MOps requires that the composed MOp can process a trace of the composition target MOp. This would require opening the implementation of the actual model operation because it has to internally process the trace information. Furthermore, their approach does not support the integration of heterogeneous model operations. It rather executes mappings for transformation purposes.

QVT is a standardized specification of a model transformation language proposed by the OMG [124]. QVT is a model transformation specification strongly reliant on the use of OCL. Basically, QVT provides the specification of an imperative model transformation language (QVT operational mappings) and the specification of a declarative model transformation language (QVT relations). QVT can be considered as a specification of a hybrid model transformation language because it allows for the invocation of imperative constructs in QVT relations. QVT also supports integrating heterogeneous model operations as black-boxes, which are considered as implementations of a relation in QVT, using the QVT black-box. QVT relations support context compositions and data-flow compositions by means of QVT operational mappings.

In combination with the black-box facility of QVT, it could be employed as a dedicated composition language for heterogeneous model operations. The context composition of model operations could be realized by means of when and where clauses, that could be used to directly invoke other model operations. The invocation does not pass traceability information directly, but rather parameters that are coming from the source and target of the invoking model operation.

However, beside the fact that currently no implementation of a combination of QVT relations and a QVT black-box exists, none of the clauses implement a composition semantic that is suitable for the presented extension scenario (see Section 6.1.2). The where clause implies that the success of the invoking model operation relies on the success of the invoked model operation. In the shown scenario, it is expected that the extended model operation can be applied successfully even if the invoked model operation cannot be applied successfully. For example, consider the case that a SysML block contains an as-yet syntactically incorrect requirement. In this case, the extending model operation would fail.

However, this should not imply that the extended model operation fails. The when clause is principally the contrary of the where clause. It implies that, e.g., applying R2LTC would invoke B2CT. However, an AUTOSAR model might contain a high amount of requirements. Thus, needing to apply R2LTC on each requirement manually makes the context composition cumbersome.

7.2.2. Composition in Model Management

The composition of model operations in the context of model management are approaches that are dedicated to the composition of heterogeneous model operations (transformations), but which are not primarily considered as a model transformation approach themselves.

In [77] the command line tool make is explained that is used to arrange the application of various command line tools based on dependencies between processed files. The problem of make is that its granularity is restricted to the file level. Dependencies can be automatically derived by using patterns. However, these patterns are restricted to analyze similarity of file names. E.g., any file *.h and *.c will be compiled into *.o using the gcc compiler.

In [78] the build tool Ant is explained in detail. Ant can be considered as being similar to make but in the context of Java-based software development. Thus, Ant allows for the integration of arbitrary Java operations, called tasks, that implement a certain interface. Furthermore, Ant is an XML-based language with a clear semantic able to be executed by Ant interpreters. In comparison to make, Ant uses "virtual" source and targets that may depend on each other, instead of building dependencies based on file names. Thus, their way of specifying a data-flow is more flexible than in case of make. However, Ant does not support the context composition of tasks (model operations). Furthermore, Ant is rather a low-level specification language for workflows or model transformation chains, with a limited analysis support.

In [14] Bézivin et al. have explained the integration of the idea of global model management by means of megamodels into an Eclipse-based environment (AMMA platform). The megamodel that is employed in AMMA has a similar foundation to the megamodel that is employed in this thesis. It captures artifacts as abstract representations of artifacts that are involved in software development and the dependencies inbetween them, which represent the application of ATL model transformations. The megamodel has further concepts to also express chains of ATL model transformations. Their execution is realized by creating Ant scripts from these specified chains. Thus, their approach is also limited to the specification and execution of coarse-grained model transformations. Furthermore, they only provide ATL as the supported model transformation technology.

In [93, 95] the Epsilon model management framework is presented. Epsilon comes with a set of model operation technologies (E* languages) for model comparison, transformation, merge, etc. The model management framework comes with a set of Ant tasks for each E* language. Thus, workflows can be specified by explicitly specifying Ant scripts under usage of the appropriate tasks for Epsilon. Because of that, the capabilities of composing model operations are similar to the capabilities of Ant.

In [160, 161, 162, 164] a sophisticated approach toward composing heterogeneous model transformations called UniTI is presented. UniTI comes with a language used to specify how model transformations behave independently of the actual implementation, which is called a unified transformation representation (UTR). Based on this language, UniTI provides a facility for specifying compositions. In their UTR language, a distinction is made between transformation specifications and transformation executions (explicit instances of specifications).

A transformation specification has similarities with a relation type, whereas a transformation execution has similarities with a relation. A transformation specification is specialized by means of a composite transformation specification, which is a model transformation chain. A model transformation chain consists of a set of model transformation executions, which define applications of model transformations between actual models. Thus, the composite transformation specification can be employed to specify a data-flow composition. However, their approach does not provide the notion of context composition of heterogeneous model operations.

In [125] a modeling framework is presented for composing heterogeneous model transformations. The approach provides an extendible metamodel for model transformation types, whose basic concept is the generic transformation as a representation of a model transformation. A generic transformation has a set of source and target metamodels that are processed by the underlying model. Furthermore, a generic transformation is specialized by means of a complex transformation, which can be either a sequential

transformation or a parallel transformation. In both cases, the model transformation refers to a set of generic transformations that are composed. The approach proposes to specify model transformations by means of UML activity diagrams. Every activity in the UML activity diagram represents an individual model transformation. Input and output objects are used to specify the input and output of the model transformation. The composition of model transformations is specified by means of object flows between activity nodes in an UML activity diagram. Thus, the approach supports building hierarchical compositions of model transformations (using the nesting capabilities of UML activity diagrams). However, the way of composing transformations is restricted to data-flows.

MWE is a dedicated XML-based language that is employed to specify chains/workflows of heterogeneous model operations [129]. MWE is part of the openArchitectureWare (oAW)[2] framework. In MWE, workflows are called generator workflows, and executing a workflow is called a generator process. A generator workflow consists of a set of workflow components, which can be implemented by any heterogeneous model operation. Within a generator workflow specification, workflow components are explicitly invoked by passing parameters that can be statically specified or that are generated by invoking other workflow components. Thus, MWE can be employed to build and execute data-flow compositions of heterogeneous model operations. Nevertheless, it cannot be employed to build context compositions of fine-grained and heterogeneous model operations.

In [92] an approach called MDA control center (MCC) is introduced, enabling the specification of networks of model transformations. Therefore, MCC provides the notion executable units, which are the basic building blocks in its transformation environment architecture. An executable unit can be a creator, a transformer or a finisher. A creator is used as a pre-processor to create an abstract syntax from a given concrete syntax while a finisher is the counterpart of a creator. A transformer is any tool that implements the mapping (transformation) from a set of artifacts to another set of artifacts (abstract syntax graphs). Furthermore, MCC provides three kinds of compositions of transformers, which are sequence, parallel and choice. However, all of these compositions are data-flow compositions that are used to specify a network of executable units. The choice composition allows explicitly specifying conditions in order to provide a more complex data-flow. Nevertheless, MCC cannot be applied to compose fine-granular model operations as in case of context compositions.

In [170] an approach is shown focussing on chaining model transformations by passing models between each other. It proposes to specify a metamodel that fits into a specific schema for describing compositions of model transformations. From these kinds of metamodels, the validity of the transformation composition as well as Ant scripts for execution can be generated. Nevertheless, the example metamodel only supports the specification of data-flow compositions of black-box model transformations.

In [12] Aldazabal et al. propose specifying the composition of model transformations explicitly and externally by means of business process execution language (BPEL)[3] models. Therefore, a BPEL engine is employed for the orchestration of modeling services (model operations) by interpreting BPEL models. The composition that can be specified in BPEL models is similar to the specification of a sophisticated data-flow including conditional branches. However, the approach does not support fine-grained context compositions or heterogeneous model operations.

In [76] an approach called transformation composition (TraCo) has been presented. The focus of their approach is not only the composition of black-box model transformations but also a safe composition. Basically, the composition that is provided by TraCo is, to some extent, similar to the composition as provided by MWE, MCC and UniTI. They introduce a component specification, which is the signature of a model transformation. A component specification has a set of port specifications with a certain direction, and an implementation, which is the actual implementation of a model transformation. A port specification can be related to a set of constraints. Furthermore, a component specification has a set of constraints, pre- and post conditions. A constraint can be a specified by means of OCL or Java. The composition of model transformations is specified by means of component instances, port instances and connectors between port instances.

The presented metamodels for the specification of components and their composition is similar to the notion of relation types and relations. A relation type can be considered as a component specification, with a parameter type as a port specification, and the pre-condition is the instantiation condition. The execution operation is similar to the implementation provider. A relation has similarities to a component instance with a parameter considered as port instance. However, the approach supports the specification

[2] http://www.eclipse.org/workinggroups/oaw/
[3] http://bpel.xml.org/

of explicit and external data-flow compositions (sequential or parallel) only because it uses overlapping input and output models as concepts for the composition. Such data-flow compositions can be specified in the shown approach by means of modules.

In [99] an approach toward the composition of graph transformations is presented. They defined the concept of a model transformation unit,as a set of input model types, a set of actions, a control condition and a set of output model types. The control condition declares how actions are executed when executing a model transformation unit. A model transformation unit can be considered, to some extent, as a declarative model transformation with each action as a model transformation rule that is explicitly and externally controlled by the control condition. However, the individual actions are specified in some specific syntax and thus only these specific actions can be composed in a declarative style. Based on the definition of a model transformation unit, their approach further introduces the concept of sequential and parallel composition for model transformation units. Nevertheless, this kind of composition is considered as an explicit and external data-flow composition of model transformation units. They do not consider the composition of black-box model transformations.

In [22] a tool integration framework is introduced. Their approach aims at the integration of various tools into a common tool chain, which is employed to automate or guide certain development steps in the tool chain. The tool chain is specified by means of a high-level, domain-specific language for process modeling. Individual activities in the process can be automated by using graph transformation techniques and the interaction between these activities is specified by means of contracts. A contract can be formally specified by means of graph patterns, which can be used to automatically validate individual activities in the process. Generally, this approach can be considered as a dedicated approach toward the composition of graph transformations. The composition is explicitly and externally specified by means of a process model, which is considered as a data-flow composition because individual activities share models via connectors.

In [140] Rivera et al. have introduced a graphical and executable language for the orchestration of ATL model transformations called Wires*. Their approach should enable the compositional specification and execution of complex model transformation chains. Wires* is a data-flow based process in which a set of input models is processed by a chain of ATL model transformations until a set of output models is produced. Their approach is somehow similar to the approach shown in [125]. However, this approach further supports additional control flow concepts like conditional branches and loops. Nevertheless, Wires* does not support the composition of fine-grained and heterogeneous model operations as shown in this thesis. It is further restricted to the application of ATL.

All previously considered composition approaches support the explicit and external composition of model transformations by means of a data-flow. Nearly all of them support the composition of heterogeneous model transformations as true black-boxes, without needing to adapt the implementation of the composed model transformations for sake of the composition. However, none of these approaches support the composition of more fine-grained and heterogeneous model operations. There are a few dedicated composition approaches that, to some extent, support context composition of heterogeneous model operations at any level of detail, explained below.

Cuadrado et al. have shown one of them in [43]. In their approach potentially heterogeneous model transformations are grouped in so called phases. The composition of these phases is obtained in two different modes. Firstly, independent model transformations are composed. Secondly, model transformations that depend on the outcome of previously applied model transformations are composed, which the authors call refinement. They use implicitly created traceability information that can be queried by model transformations through a function they provide to explicitly resolve the traceability information. Their approach supports explicit but internal context composition. Thus, the authors assume that model transformations have to implement the provided traceability function to reason about the additional context provided by traceability, which violates the true black-box principle. Furthermore, their approach does not support an automated (re-)application of model operations as shown in this thesis.

Vanhooff et al. presented another approach in [163, 165]. The authors use a global traceability graph, which contains traceability information that can be created by individual model transformations. This traceability information is employed as a context for the composition of model transformations. Nevertheless as in Cuadrado et al.'s approach [43], Vanhooff et al. expect that model transformations need to explicitly interpret traceability information and thus also violate the black-box principle. They further do not show how to execute these compositions at all.

In contrast to these two approaches, the context composition of this approach does not pass relations (traceability) directly to the model transformation, but rather unpacks the information beforehand. In this way, it provides the required context in the form of artifacts (artifact context) alone. Thus, this approach provides data-flow and context compositions of heterogeneous model operations that are considered as true black-boxes.

7.3. Summary

The contributions of this thesis are a traceability approach in the context of MDE, and a comprehensive model management approach that builds on top of the demonstrated traceability approach. In this chapter, it has been shown that there is currently no other approach that combines traceability and model management as shown in this thesis.

Concerning traceability in the context of MDE, the shown approach is considered as a retrospective traceability approach that is used to incrementally create and maintain precise traceability links (relations). Nevertheless, it can only be considered as a complement to prospective traceability approaches in this context. In the context of model management, there is currently no other approach that supports an incremental and automated maintenance of relations at any level of detail.

However, the major contribution of this thesis is the composition of heterogeneous model operations at any level of detail without violating the black-box principle. Therefore, related work has been studied in two different domains.

The first domain is about dedicated model transformation approaches, which explicitly support the notion of decomposition and composition of individual modules or rules. These are basically declarative or hybrid model transformation approaches, all of which provide the notion of context composition of transformation modules or rules. Nevertheless, these approaches either do not support the composition of heterogeneous model operations, violate the black-box principle or do not provide a context composition semantic as required in this thesis.

The second domain is concerned with dedicated composition approaches. These are considered as approaches in the domain of model management, because their primary purpose is to support model management. Most of these approaches support the composition of heterogeneous model operations considered as black-boxes. However, these approaches are restricted to data-flow compositions. Thus, only coarse-grained model operations can be composed. Only two approaches exist that support context compositions, but they violate the black-box principle because the composition has to be obtained internally within the actual model operation.

To the best of my knowledge, there is currently no approach that provides the composition of heterogeneous model operations, which are considered as true black-boxes, at any level of detail. Furthermore, this thesis also provides an approach to automatically (re-)apply heterogeneous model operations that are defined to be in such compositions.

8. Conclusions and Future Work

Contents

* * *

In this chapter, the shown approach is concluded. Furthermore, current technical and conceptual limitations, as well as further research challenges, are outlined.

8.1. Conclusions

In the introduction of this thesis, a typical application scenario for software development has been shown, which is being the development of multiple software systems for different customers in related domains. In addition, instead of considering a classical software development methodology (programming), MDE is considered as the major methodology for developing software systems.

Before MDE can be applied to actually develop a software system, it has to be configured properly. Therefore, an MDE configuration is provided by a configuration developer. An application developer subsequently applies this MDE configuration to provide an MDE application, which basically contains all artifacts that define the actual software system. Applying MDE in such an application scenario is challenging due to various reasons. In this thesis, four particular challenges were addressed, that are common to approaches in the domain of traceability and model management.

The more DSMLs are required to develop a software system, the more complicated an MDE application may get because the number of artifacts increases. Furthermore, artifacts in MDE applications do not exist in isolation because they all describe the same software system but from different perspectives and from different levels of abstraction. Thus, MDE applications get more and more confusing and application developers may lose the overview of why certain artifacts exist and how they relate to each other. Therefore, the first challenge for MDE is to support the comprehensibility of MDE applications by understanding why artifacts exist and how they interrelate.

Because software systems are usually developed iteratively and requirements to the software system may change, the artifacts in MDE applications are subject to continuous change. The development environment has to support making changes and not making it a insuperable barrier. This is challenging for at least two reasons. Firstly, changing artifacts may impact other artifacts because they somehow interrelate. Secondly, changing artifacts may impact the application of model operations because the changed artifacts could be a source of model operations. In both cases, inconsistencies could arise, if changes are not properly reacted to.

Due to the increased number of artifacts in MDE applications and the heterogeneity of the kinds of artifacts, there is an increasing amount of heterogeneous model operations that have to be managed. Thus, application developers have to deal with various technologies at the same time. Furthermore, they have to (re-)apply model operations in the right context and at the right time, e.g., when changes impact already applied model operations. This is challenging because of the increased heterogeneity of model operations and the increased number of model operations. This is even more tedious when model operations are decomposed into smaller and more reusable units, which is caused by the fourth challenge.

The fourth challenge is considered as the major challenge of this thesis and concerns the need to provide reusable and adaptable model operations in order to make the process of setting up MDE configurations more efficient. This is extremely challenging for MDE because various heterogeneous DSMLs are required

to appropriately specify a software system, which implies that there is no common model operation, like a compiler, that can transform any specification into executable code. A configuration developer has to provide a collection of heterogeneous model operations that are required for analysis or synthesis purposes.

To address these challenges four goals, identified in the beginning of this thesis, are used as requirements for the thesis's concepts.

To ease the understandability of MDE applications, it should be possible to capture all kinds of dependencies between different kinds of artifacts. These captured dependencies must have an interpretable semantic such that application developers are able to reason about captured dependencies and are able to apply subsequent techniques on top, e.g., navigation support, impact analysis, etc.

As soon as application developers make changes to artifacts, new dependencies may have to be captured or existing dependencies may become invalidated. To counteract the deterioration of captured dependencies, the second goal was to automatically maintain the existence of dependencies. Making changes to artifacts may also require applying model operations or re-applying already applied model operations because the output of a model operation may no longer be consistent with the source. Thus, the third goal has been defined as supporting application developers by automatically (re-)applying model operations in the case of changes that impact the validity of model operations.

To support reusable and adaptable model operations by decomposition, the fourth and major goal has been defined as specifying and applying compositions of such model operations. So as to not decrease the reusability and adaptability of these model operations, the approach for composing these model operations must ensure that they remain loosely coupled and highly cohesive. At the same time, the composition approach must be expressive enough to be able to build complex model operations from these fine-grained and heterogeneous model operations.

All these goals have been addressed by an approach called executable and dynamic hierarchical megamodels, which is an extension of the classical notion of megamodels from Bézivin et al. The shown approach is a combination of a traceability and a model management and has been developed in three consecutive steps, which were shown in Chapter 3, 4 and 5. The foundation of this approach is the hierarchical megamodel (see Chapter 3), which is an extension of the megamodel introduced by Bézivin et al. It consists of representations of artifacts in MDE applications and dependencies inbetween them. In addition, it supports a hierarchical representation of artifacts and dependencies, which is leveraged by the subsequent extension.

The approach can be considered as a traceability approach because it not only supports capturing dependencies between artifacts by means of relations but it also allows for the automatic maintenance of their existence. This extension is called a dynamic hierarchical megamodel and has been introduced in Chapter 4. Due to the hierarchical megamodel, dependencies at any level of detail can be automatically captured and maintained.

For automated maintenance, two new concepts has been introduced in the dynamic hierarchical megamodel – relation type composition specifications and instantiation conditions. The relation type composition specifications are a formal definition of the composition context of a relation. This is used to automatically reason about the validity of a composition context. The instantiation condition is an abstract representation of a model operation that is used to reason about the validity of the artifact context of a relation. In combination, they can be used to reason whether a relation should exist in a certain context or not.

This is operationalized by the localization shown in the same chapter. The localization uses these concepts to automatically update existing relations, creating new relations or deleting those relations which are no longer valid. Two conceptual implementations of the localization have been shown, which are a batch and an incremental localization strategy.

The approach can be considered as a model management approach because it allows for the uniform automatic management of the (re-)application of heterogeneous model operations. Therefore, an extension of the dynamic hierarchical megamodel has been introduced in Chapter 5 called executable and dynamic hierarchical megamodel. Therefore, relations are optionally treated as executable relations, which are relations that represent the application or potential application of a model operation. This is obtained by introducing the concept of execution operations, which are abstract representations of any kind of model operation.

The automated (re-)application of all kinds of model operations is realized by means of execution, shown in the same chapter. It automatically analyzes which relations have to be (re-)applied. Based on

that analysis, the execution automatically executes all necessary relations in a correct order.

In addition to the basic model management capabilities of the approach, specifying complex compositions of heterogeneous model operations at any level of detail is also possible. The approach supports specifying two different kinds of compositions – data-flow compositions and context compositions. A data-flow composition basically means specifying that model operations are composed by means of their input and output, which is useful to build chains of rather monolithic model operations. Specifying data-flow compositions is realized by means of a concept called modules, also introduced in Chapter 5.

A context composition means specifying that model operations are composed by means of an explicit composition dependency, which uses other model operations as explicit context. Context compositions are especially useful if model operations at different levels of detail can be composed in a declarative way, which is known from declarative model transformation approaches. Technically, the specification of context compositions does not require any new concept. It is already realized by means of relation type composition specifications as introduced in Chapter 4.

In Chapter 6, the capabilities of the context composition has been demonstrated based on two application examples from two different case studies.

In the first example, it has been shown that the context composition can be used to build complex model operations. This is realized by the following facts. A relation can be specified to exist in multiple alternative contexts. Each context can be specified as a pattern of a set of relations and artifacts inbetween them. Parameter type connectors are used to specify the binding between artifacts in the artifact context of a relation and the artifacts in the composition context of a relation. In comparison to declarative model transformation approaches, the shown approach does not rely on a specific technology to actually realize the model operations because model operations are treated to be completely decoupled from the approach.

In the second example, it has been shown that the context composition can be used to extend existing legacy model operations by even using different technologies. The positive effect of extending existing model operations in this way is that the extended model operations are loosely coupled and highly cohesive. They are loosely coupled because they do not know anything about the model operation they extend. They are highly cohesive because they do not have to implement any navigation concerns which are only required to navigate from high level artifacts to the actual place where the operation should be applied. This is completely done by the context composition.

Another benefit of the shown approach comes by the combination of traceability and model management. The approach allows for the context composition of a model operation not only into the context of other model operations but also into any kind of traceability information (relation). Thus there is more flexibility in the actual composition of model operations.

Finally, the actual model operations are considered as true black-boxes because the approach does not require that they have to implement any composition concerns. The model operations are also decoupled from the composition framework and further do not have to implement composition concerns, which would otherwise decrease cohesion.

8.2. Future Work

The shown approach is only a piece in the puzzle toward comprehensive traceability and model management in the context of MDE. At this stage, the shown approach has certain limitations, which are technical as well as conceptual. Furthermore, there are several subsequent challenges that need to be addressed by model management approaches like these.

8.2.1. Technical Limitations

The interoperability of prototypical implementation is currently restricted to work within Eclipse only. However, even tools beyond Eclipse might have to be integrated into an MDE configuration. In [30], this is addressed by using a common meta-metamodel and in [12] interoperability is increased by requesting models, metamodels and model operations from a service-oriented tool integration framework called ModelBus. Another approach is using adapters to automatically transform model between different environments (see [66]). As future work, an integration of those approaches might be preferable.

The shown concepts of the approach have not yet been completely implemented, specifically, the shown concepts of the relation type composition and specification. The latest prototype only implements

a simple relation type composition, which only allows for the context composition of a relation type into one other relation type. Also the module concept and the parameter type multiplicities have no implementation yet. Thus, the implementation of the model management framework has to be completed such that other research on model management could use the implemented model management framework as foundation for further prototypical implementations.

The concrete syntax that is used to explain the concepts is not yet implemented. The implemented concepts are currently represented in tree-based editors only. To support configuration developers, the implementation should further comprise an analysis that checks whether certain context compositions are possible. This could be necessary because parameter type connectors may be specified incorrectly, e.g., the mode is not supported between the type of the parameter role and the parameter type.

8.2.2. Conceptual Limitations

In a project seminar, a student has implemented a feature such that execution operations have additional parameter types which have primitive data types as values. However, these are only used as additional configuration parameters when applying a model operation related to the execution operation. Unfortunately, this feature is currently not reflected in the conceptual elaboration as shown in this thesis. Furthermore, parameter types that are connected to relation types are always considered to have an artifact type as value and parameters that are connected to relations to have an artifact as value. However as mentioned in the evaluation (see Section 6.1.1), it may be necessary to provide primitive data types like strings, integers, etc. as values for parameter types and parameters. Thereby, model operations could also exchange information that is not directly represented by artifacts.

The condition for applying an execution operation is currently similar to the condition of instantiating a relation type (instantiation condition). In certain situations it may be useful to separate between an instantiation condition and a set of pre-conditions that only specify whether the execution will be successful or not. Thus, the existence of an executable relation does not necessarily need to mean that the execution will provide the expected results but it could be applied if certain changes will be made. Instead of pre-conditions, post-conditions could also be employed. A post-condition could specify a condition that defines the required output of executing a relation.

The module concept is introduced too briefly. It should be considered in more detail to elaborate the actual possibilities of this concept. Furthermore, the semantics of modules should be directly added into the localization as well as into the execution so as not to rely on a transformation into relation types with similar semantics.

The relation type composition specifications could be further improved by not only specifying a pattern over relations and artifacts but also over relation compositions. This would enable reasoning about the validity of composition contexts that include relations, which already exist in a specific context composition.

8.2.3. Research Challenges

A challenge of the current approach is the uncertainty about artifacts that were created as result of applying a model operation, which does not exist anymore. Currently, these artifacts remain. But should they remain or should they be removed if a relation is going to be removed that executes the model operation? It could make sense to add a concept that marks artifacts as automatically removed if their originator has been removed. But what is the condition of marking an artifact to be automatically removed?

This goes hand in hand with another challenge to model management approaches, which is changeability of artifacts. For example, should an artifact that is the target of a uni-directional model operation be changeable by an application developer? This could be necessary in scenarios where artifacts that were partially generated have to be refined manually. The hierarchical megamodel might provide means to analyze whether manual changes can be overridden by re-applying a model operation because it allows capturing fine-grained relations between source and target artifacts. Furthermore, conflicts that result from changes and potential re-application of model operations should be analyzed. For example, the potential overriding of manual changes, or the propagation of several changes from different sources toward similar artifacts.

Recently, we have started working on a distributed version control system based on executable and dynamic hierarchical megamodels [27]. Version control systems have various benefits, e.g., providing multiple versions of artifacts or supporting collaborative development in a team of application developers. A version control system based on executable and dynamic hierarchical megamodels allows not only version artifacts but also relations in between them. Thus, common version control operations like commit or checkout exploit relations to commit or checkout transitive closures of artifacts. In the master's thesis of Thomas Beyhl [27], it has also been discussed if a pessimistic approach by using locking can be applied to avoid scenarios where merging artifacts is required. The locking operation uses relations to also lock artifacts that are in the transitive closure of artifacts that have been locked. Because the megamodel is hierarchical, fine-grained locking can be supported, which potentially reduces the number of artifacts that have to be locked and thus the sequentialization of the development process that may come with locking. However, this is still ongoing work.

Currently, it is assumed that artifacts are always consistent if changes are propagated by applying model operations. However, propagating changes may still result in structural as well as semantic consistency violations. That is because the semantic of model operations and consistency of artifacts do not conform to each other or because the sources of a model operation were inconsistent or incomplete before application. Because megamodels also implement a macroscopic view on software development, they are suitable for also incorporating the notion of global consistency. Thus megamodels could ease the application of heterogeneous consistency checks in a homogeneous way. The megamodel could also be used to provide warnings if model operations are going to process inconsistent artifacts, or analyze whether the (re-)application of model operations will result in new inconsistencies, which could indicate that a model operation does not provide the expected results.

Another ongoing challenge in the domain of model management using megamodels is the integration of software development processes and the automation of software development using chains or workflows of model operations as shown in this thesis. An issue of current model management approaches is that in order to provide automation, they are focused on technical realization. However, not all operations in MDE can be automated. Furthermore, certain chains or workflows might look different in different phases of a software development process. Thus, it could make sense to integrate a software development process into model management to provide further control about which operations are available or which operations should be applied when and where.

In [168] and [169], we have investigated how model management with megamodels can support current research in the context of models at runtime. In [60] it has been mentioned that MDE is not only beneficent to developing software systems but also to maintaining already deployed software systems. In the latter case, runtime models are used to manage software systems at runtime. These models do not exist in isolation but rather have inherent dependencies. Thus, traceability and model management could be applied also for models at runtime. In this domain, megamodels could be employed to help structure and maintain runtime models.

Appendix

A. Implementation

This chapter is designated to show the design of the implementation of the recent prototype of the model management framework theoretically presented in this thesis. Figure A.1 shows the prototypical integration of the model management framework in the context of Eclipse and EMF.

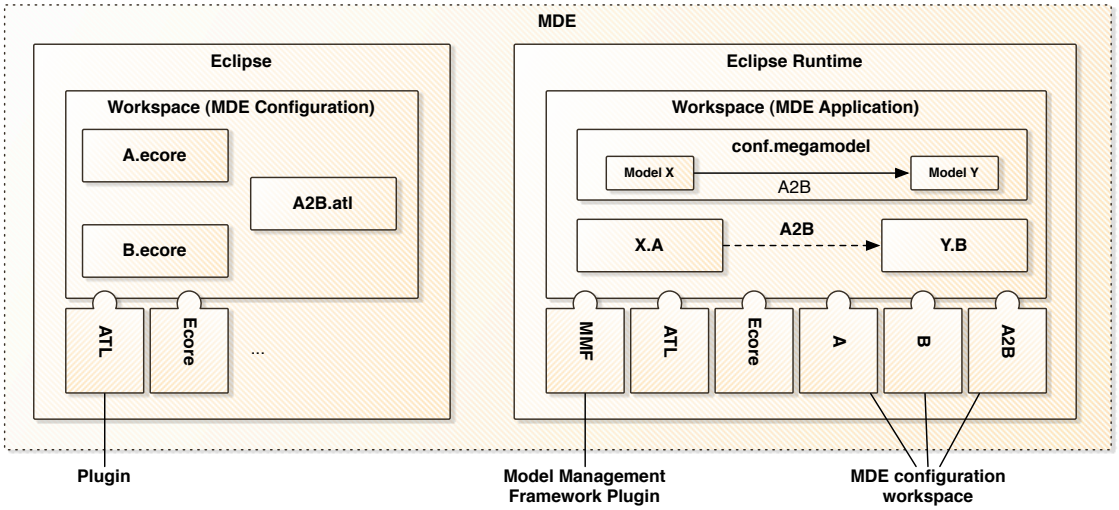

Figure A.1.: Prototypical integration into Eclipse

The figure illustrates two Eclipse instances. The first Eclipse instance, shown to the left, is used to configure MDE for subsequent application. It contains a workspace, which is the actual MDE configuration. This workspace contains EMF metamodels (A.ecore and B.ecore) and model operations. For example, A2B.atl is the implementation of an ATL model transformation, which takes a model of type A as source and a model of type B as target. Eclipse uses a plugin mechanism to integrate metamodels and model operation technologies.

In this thesis, the only relevant Plugins are those that provide the capability to define metamodels and those that provide capabilities to implement model operations (e.g., ATL or Story diagram). The metamodels, as well as the model operations, are defined so that they can be used as Plugins in a second Eclipse runtime instance.

The second Eclipse instance, shown to the right, is used to apply MDE – based on a previously defined MDE configuration. The MDE configuration from the first Eclipse instance is integrated by integrating the Plugins of the metamodels and model operations. Thus, the workspace of the MDE configuration is available as a set of Plugins that can be accessed from within the second Eclipse instance. The workspace of the second Eclipse instance is considered as an MDE application, which contains models (X.A and Y.B) that are instances of the metamodel from the MDE configuration. Furthermore, the implemented model operations available as Plugins can be applied between models in the workspace.

The model management framework is also integrated as a plugin into the second Eclipse instance. It manages information from the MDE configuration by communicating with the Plugins from the MDE configuration. Furthermore, the framework manages the workspace of the MDE application by abstractly representing it and capturing all kinds of dependencies between the models in the workspace, e.g., the application of model operations like A2B between X.A and Y.B.

The high-level architecture of the model management framework is shown as a UML component diagram in Figure A.2. The framework consists of four components: – the model management core, the artifact manager, the relation manager, and the core dispatcher. The components localization and

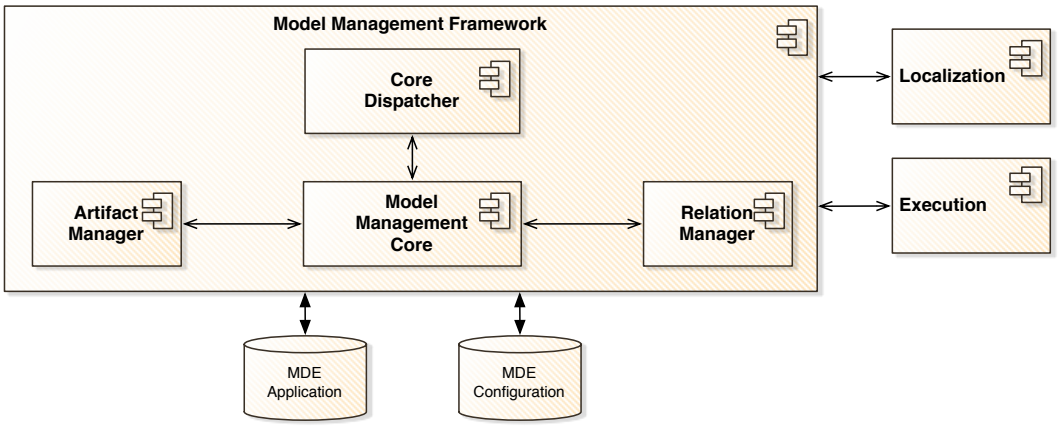

Figure A.2.: High-level architecture of the model management framework

execution are implemented as extensions to the framework and represent the functionality as shown in Chapter 4 and 5, respectively.

A.1. Metamodel Extensions

The metamodels that were already shown in the main chapters are basically the same as used in the implementation. The only difference is that implementation specific concepts have not been shown. These implementation specific concepts are now added to the specific concepts of the previously shown metamodels.

A.1.1. Model Operation Extension

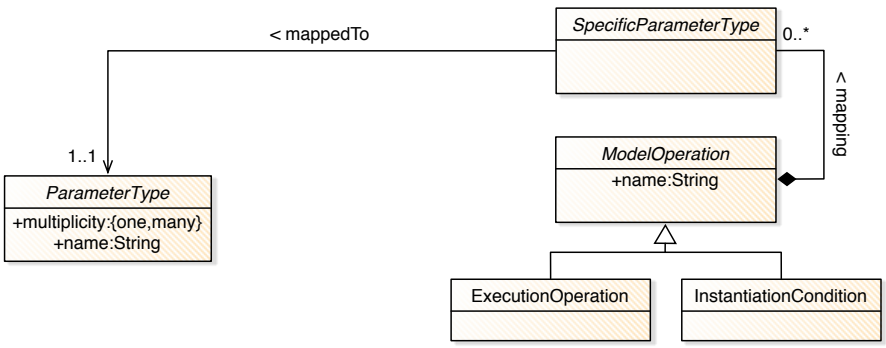

Figure A.3.: Metamodel extension for the integration of model operations

The model operation (ModelOperation) is an abstract concept that is extended in the implementation metamodel. The extension is required because the model operation is not sufficient for actually applying represented model operations. It is insufficient because parameters that are related to a relation have to be mapped to parameters that are specific for the implementation of the model operations. This is shown in Figure A.3.

Therefore, a model operation additionally has a set of specific parameter types (SpecificParameterType), which is an abstract concept needing to be further specialized.[1] A specific parameter type can be considered as a mapping from a technology-specific parameter, which is used to apply the related model

[1] For each model operation technology a specific extension is required. Two examples are shown in Section A.2.4.

operation, and a parameter type, which is connected to a relation type that is the signature of the model operation. This information is sufficient to apply heterogeneous model operations based on a given relation that is an instance of such a relation type.

A.1.2. Change Events

In the main chapters, it has already been mentioned that change events may exist that can be employed to facilitate incremental operations. These change events are implemented in the framework as shown in the metamodel of Figure A.4.

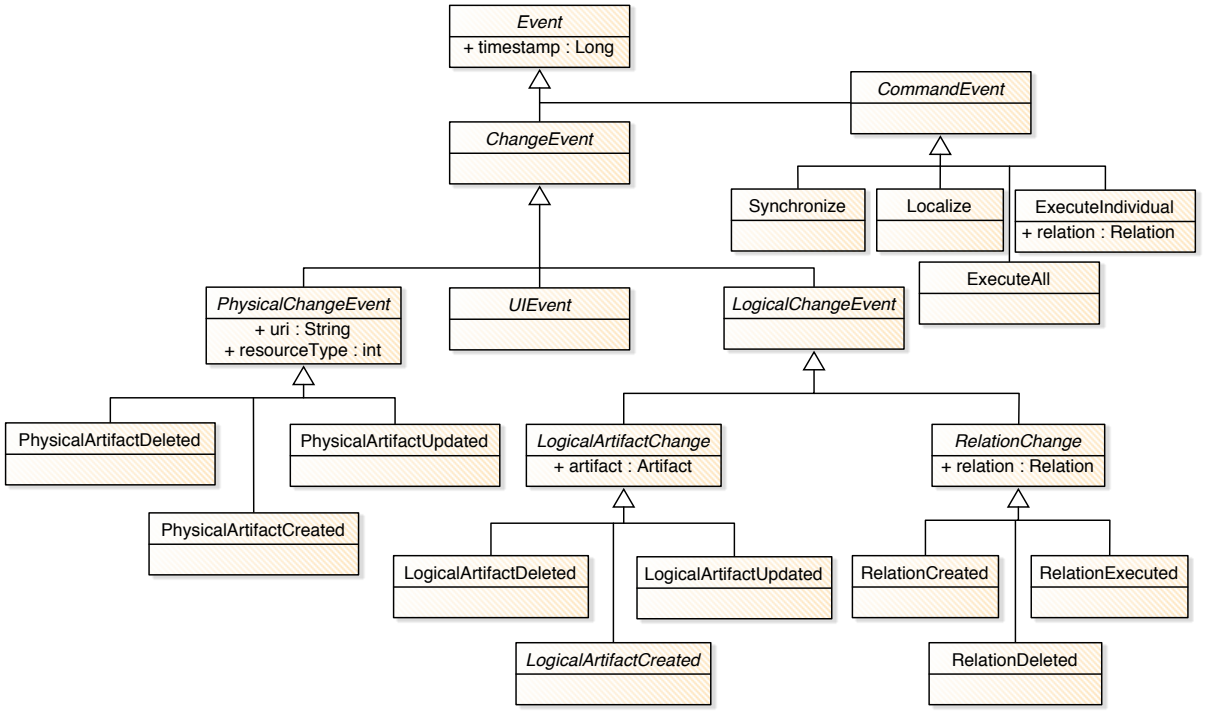

Figure A.4.: Metamodel extension for change events

The most abstract concept is an event (Event), which provides a timestamp that reflects the occurrence of an event. All other events are specializations of Event. Events are further classified into change events (ChangeEvent) and command events (CommandEvent).

A command event is any event explicitly created by some tool or by a user through a user interface. These events are used to communicate with the model management framework. For example, the synchronize event (Synchronize) is a command that triggers the synchronization process, localize (Localize) triggers the localization, execute individual (ExecuteIndividual) triggers the individual execution strategy (see Section 5.3.3.1), and execute all (ExecuteAll) triggers the complete execution strategy (see Section 5.3.3.2).[2]

A change event is any event that reflects some changes. Change events are further specialized by user interface events (UIEvent), physical change events (PhysicalChangeEvent) and logical change events (LogicalChangeEvent). A user interface event is any event that describes an event that is coming from a user interface, e.g., opening or closing an editor, etc. A physical change event indicates the deletion (PhysicalArtifactDeleted), the creation (PhysicalArtifactCreated), or the update (PhysicalArtifactUpdated) of a physical artifact.[3] A logical change event indicates a change to an artifact or a relation in an

[2]The current implementation of this prototype does not yet support command events. The synchronization, localization and execution are currently triggered by a user interface directly.

[3]Changes are only recorded for physical artifact but not for physical artifact type. This is sufficient because the synchronization of a configuration megamodel is user-driven and not change-driven.

application megamodel. Thus, LogicalArtifactChange and RelationChangeEvent further subclass LogicalChangeEvent. The relation executed event (RelationExecuted) is an event that is created when a the model operation related to a relation is (re-)applied.

A.2. Framework Design

Based on the metamodel extensions, the designs of the four major components of the model management framework (see Figure A.2) are shown.

A.2.1. Model Management Core

The model management core component is designed as shown in Figure A.5.[4] The basic concepts of the design are the core manager (CoreManager) and the megamodel handler (MegamodelHandler).

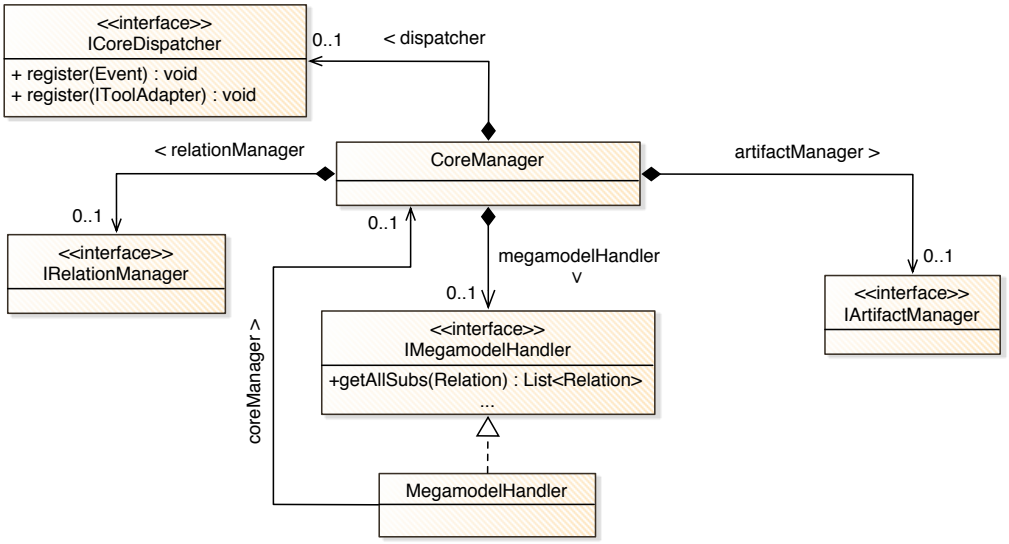

Figure A.5.: Design of the model management core component

The core manager is a central player in the model management core and does not have a specific obligation other than building the backbone of the model management framework. Thus, it holds a reference to a core dispatcher, an artifact manager and a relation manger. The megamodel handler is used to decouple an application and a configuration megamodel from other components that require access to them. Therefore, it provides a set of access operations, e.g., $getAllSubs(Relation)$.

A.2.2. Core Dispatcher

The core dispatcher component is responsible for managing additional tools, which manipulate the application megamodel, e.g., the localization and the execution is implemented by means of tools managed by the core dispatcher. The internal design of the core dispatcher component is shown in Figure A.6.

The core dispatcher (CoreDispatcher) implements an observer pattern (cf. [62]). The core dispatcher manages a tool adapter set (IToolAdapter), which can be registered via the $register(IToolAdapter)$ operation. The core dispatcher receives events from change providers (ChangeProvider), which use the $register(Event)$ operation for this purpose. Thus, in mapping the core dispatcher to the observer pattern, a tool adapter is an observer while a change provider is a subject.

The core dispatcher works as follows. Each time a set of events is registered, command events are added to a queue change events that are directly propagated to registered tool adapters. This should enable individual tool adapters to pre-process change events for further application. After propagating

[4]The shown classes do not reveal all available operations and attributes due to readability.

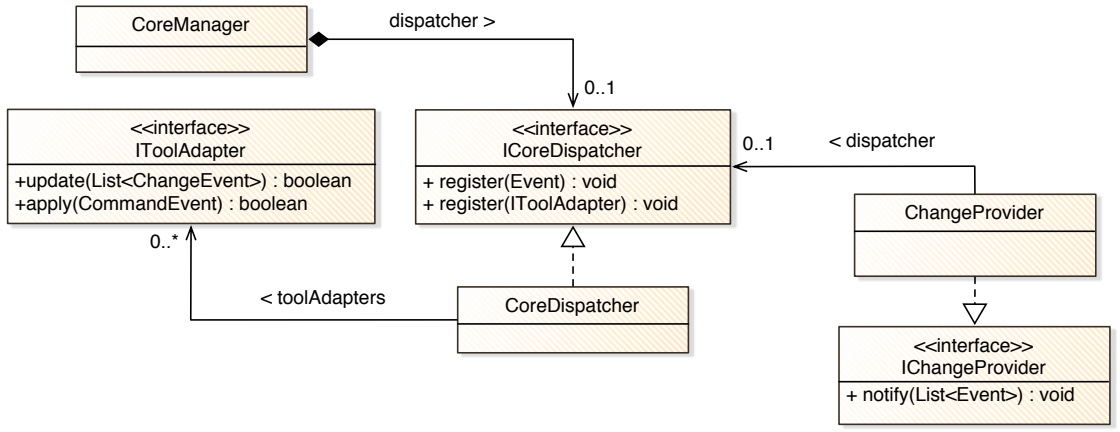

Figure A.6.: Design of the core dispatcher component

all change events to registered tool adapters, the next command from the queue is taken and send to all tool adapters by using an *apply* operation. This is repeated until the queue is empty. Thus, the actual functionality of tool adapters is triggered by explicit command events.

A.2.3. Artifact Manager

The artifact manager component consists of an artifact manager (ArtifactManager) and a set of artifact adapters (ArtifactAdapter). The design of this component is shown in Figure A.7.

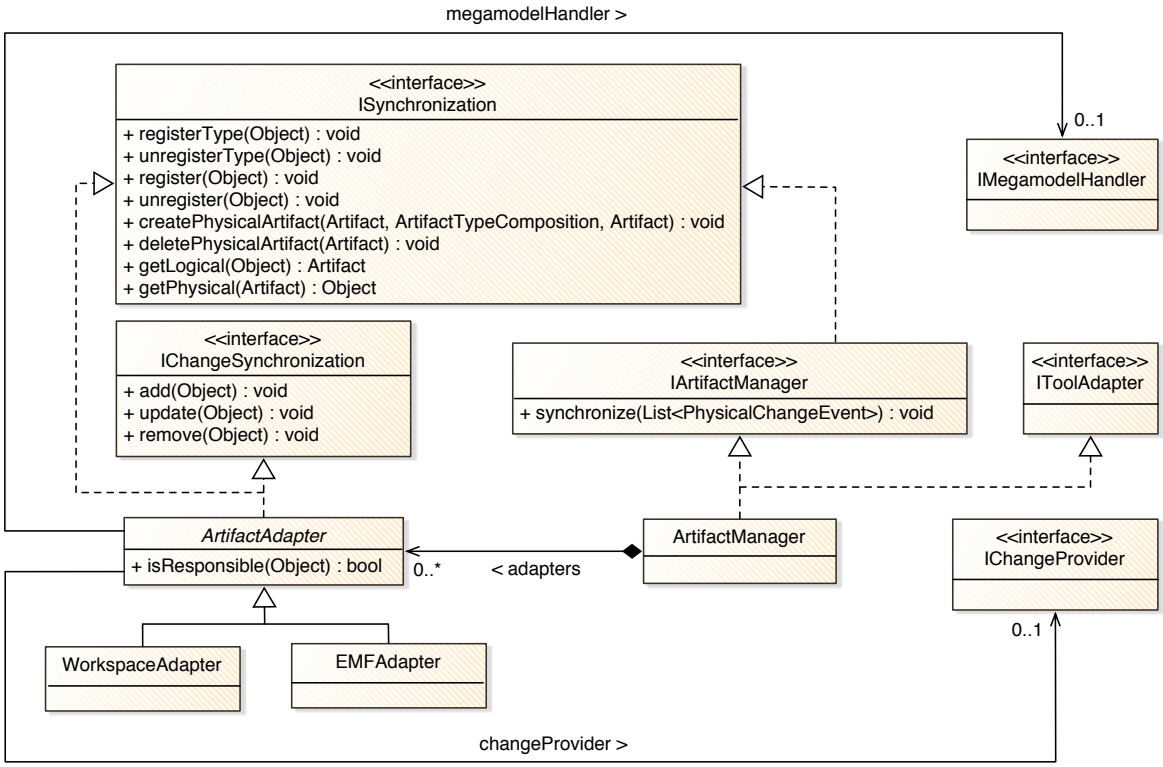

Figure A.7.: Design of the artifact manager component

The artifact manager is responsible for managing a set of artifact adapters. It implements an interface

(IArtifactManager) that can be used to trigger the synchronization process as generically explained in Section 3.3. The interface ISynchronization provides all operations that are necessary to realize the user-driven synchronization operations. It further provides operations that may be queried by the core manager that can only be implemented by a specific artifact adapter.

The interface IChangeSynchronization provides the other operations that are required to implement change-driven synchronization operations. The change-driven synchronization of the artifact manager is realized by means of implementing the artifact manager as a tool adapter. Thus, it buffers physical change events coming from the core dispatcher. The actual synchronization is triggered upon the arrival of a synchronize command event.

The artifact adapter is an abstract concept that has to be specialized. Two specializations are already provided – are a workspace adapter (WorkspaceAdater) and an EMF adapter (EMFAdapter). Both adapters implement the synchronization process but for specific kinds of physical artifacts and physical artifact types. An artifact adapter is always related to a megamodel handler and a change provider, using them to propagate change events to the dispatcher or to access the megamodel.

A.2.3.1. EMF Adapter

The EMF adapter is responsible for creating and synchronizing EMF metamodels and models. Because artifact and artifact type are too abstract to represent such specific physical artifacts and physical artifact types, they have to be specialized to appropriately represent them.

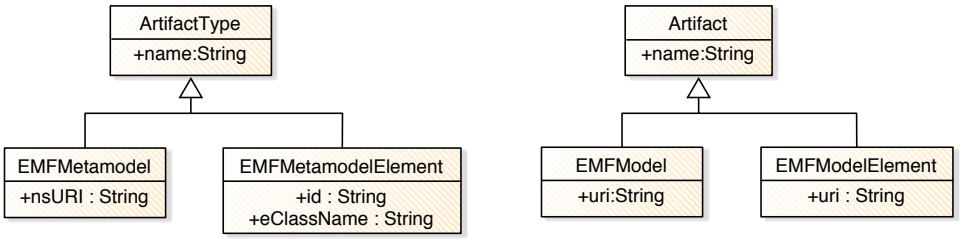

Figure A.8.: Metamodel extension for EMF metamodels and models

Figure A.8 shows the required metamodel extension for EMF metamodels and models. A metamodel is considered as a physical artifact type and is therefore represented by a specialized artifact type called EMFMetamodel. The representation relationship is encoded by means of the namespace URI of the metamodel (nsURI attribute). The metamodel elements are represented by another specialization of artifact type called EMFMetamodelElement. The representation relationship is defined by an identifier of the element (id attribute) and the name of the element (eClassName attribute).[5] A model is considered as a physical artifact and is thus represented by a specialized artifact called EMFModel. The representation relationship between this artifact and its physical counterpart is encoded by a URI (uri attribute), which is the location in the workspace. A model contains model elements, which are represented by a specialized artifact called EMFModelElement. The representation relationship is also encoded by means of a URI (uri attribute). However, in this case the URI defines a unique identifier within the model.[6]

The EMF adapter is also responsible for setting the type/instance relationship between EMFModel and EMFMetamodel and between EMFModelElement and EMFMetamodelElement.

A.2.3.2. Workspace Adapter

The workspace adapter is responsible for creating and synchronizing artifacts and artifact types which are abstract representation of a workspace including projects, folders and files. Considering the workspace in Eclipse as a set of physical artifacts is slightly different because the workspace is not a model with an explicit metamodel that it instantiates. Thus, the workspace has no explicitly available physical artifact

[5] Only EClass instances are represented by EMFMetamodelElement because only these elements can be explicitly instantiated by model elements.

[6] This requires that an EMF model always provide unique identifiers for its elements. This is obtained by using XMI resources with turning the unique identifier feature on.

type. Nevertheless, this is emulated by representing a virtual metamodel in a configuration megamodel. For representing a workspace, the virtual metamodel is the extension that is shown in Figure A.9.

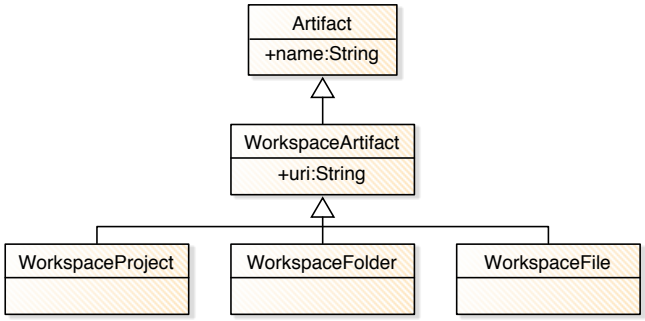

Figure A.9.: Metamodel extension for representing Eclipse workspace artifacts

The specialized artifact that represents any project in a workspace is called WorkspaceProject, the specialized artifact that represents any folder in a workspace is called WorkspaceFolder, and the specialized artifact that represents any file in a workspace is called WorkspaceFile. All of them directly inherit from WorkspaceArtifact, which provides the attribute uri that is used to encode the representation relationship to any workspace artifact. Instances of WorkspaceProject, WorkspaceFolder and WorkspaceFile in an application megamodel are always related to instances of EMFMetamodelElement in a configuration megamodel via the type reference. These instances of EMFMetamodelElement represent the metamodel elements WorkspaceProject, WorkspaceFolder and WorkspaceFile, respectively.

A.2.4. Relation Manager

The relation manager component has a strucutural design similar to the artifact manager component but has a different obligation. The relation manager component is responsible for providing basic operations on relations, which are currently the operations required by localization as shown in Section 4.3.1 and the execute relation operation as shown in Listing 5.2. Figure A.10 shows the internal design of the relation manager component.

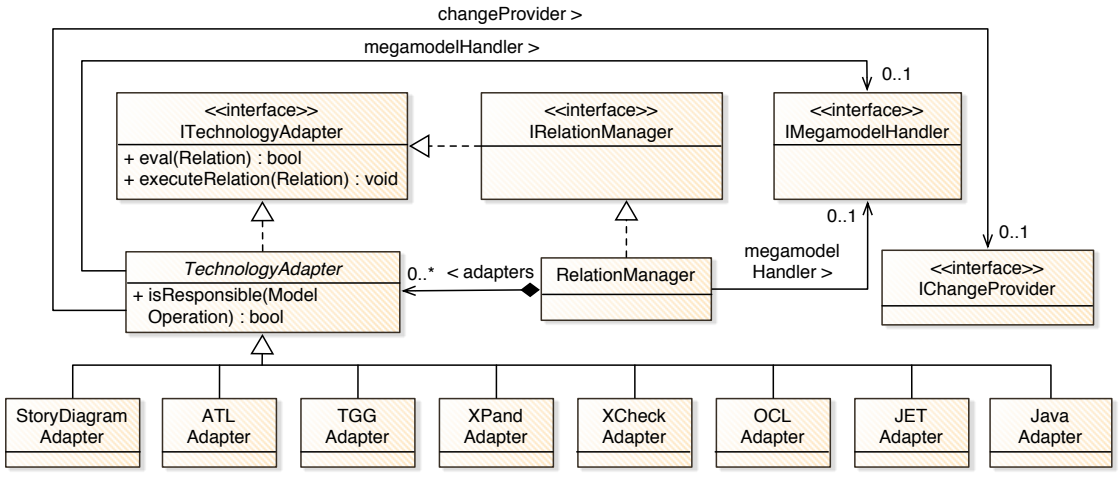

Figure A.10.: Design of the relation manager component

The central concepts of the relation manager component are the relation manager (RelationManager) and technology adapters (TechnologyAdapter). The relation manager is used by the model management core and provides all the previously mentioned operations. Therefore, it has access to a megamodel handler. If required, the relation manager can access a set of technology adapters.

A technology adapter is used as facade between the actual model operation technology and the model management framework. Currently, a technology adapter is responsible for applying instantiation conditions (*eval*) and execution operations (*execute*). Thus, if a relation should be (re-)applied it can just invoke the *execute* operation of a specific technology adapter responsible for the execution operation. A technology adapter has access to a megamodel handler and to a change provider to access the configuration or application megamodel and to create change events, e.g., executing a relation.

The concept technology adapter is only abstract. It has to be specialized for any model operation technology that should be integrated into the model management framework. Therefore, each specific technology adapter has to implement the *isResponsible*, *eval* and *execute* operation. The implementations of two technology adapters are explained in the following.

A.2.4.1. Story Diagram Adapter

The first technology adapter is implemented for using Story diagrams as instantiation conditions and as execution operations. Therefore, the metamodel of the executable and dynamic hierarchical megamodel has to be slightly extended as shown in Figure A.11.

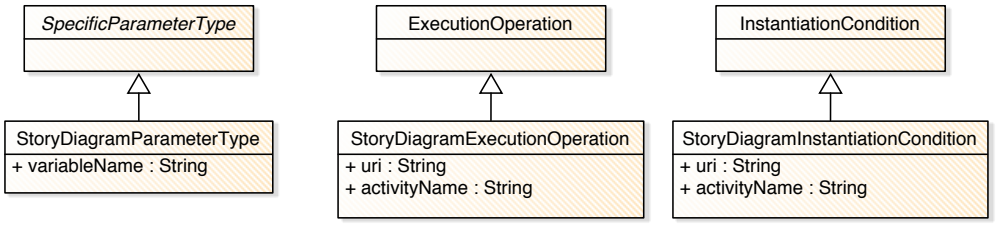

Figure A.11.: Metamodel extension for the integration of Story Diagrams

Because Story diagrams can be used for instantiation conditions and for execution operations, a specialization of InstantiationCondition (StoryDiagramInstantiationCondition) and a specialization of ExecutionOperation (StoryDiagramExecutionOperation) are provided. Both have an attribute uri and an attribute activityName. The URI is used to locate the actual implementation of the Story diagram while the activity name defines which activity in of the Story diagram is the one that should be applied.[7]

Furthermore, a specialization of SpecificParameterType is provided for Story diagrams called StoryDiagramParameterType. It provides the attribute variableName, which defines the name of a bound Story pattern object that is passed as parameter when executing the Story diagram via the provided API.

A.2.4.2. Java Adapter

The second technology adapter is implemented for using Java operations as instantiation conditions and as execution operations.

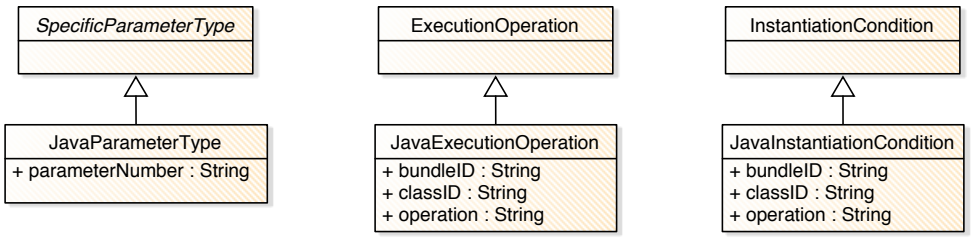

Figure A.12.: Metamodel extension for the integration of Java operations

Figure A.12 shows the necessary extensions to the megamodel for the Java adapter. The JavaExecutionOperation and the JavaInstantiationCondition have three attributes bundleID, classID and operation,

[7] A Story diagram can provide multiple activities at the same time with different functionalities.

which are required to locate the correct Java operation for invocation. Java operations must be provided as a Java operation in a Java class bundled into an Eclipse plugin. Thus, the bundle identifier is the name of the Eclipse plugin that contains the required class. The class identifier is the fully qualified name of the class that contains the required Java operation and operation is the name of the required Java operation. The Java adapter invokes Java operations by using Java reflection.

A java parameter type JavaParameterType is employed as specialization of the SpecificParameterType. A java parameter type has an attribute parameterNumber that is used to define the position of a relation parameter in the parameter list of a Java operation.

A.2.5. Localization and Execution

The integration of the localization and the execution component is realized by means of implementing them as tool adapters, shown in Figure A.13.

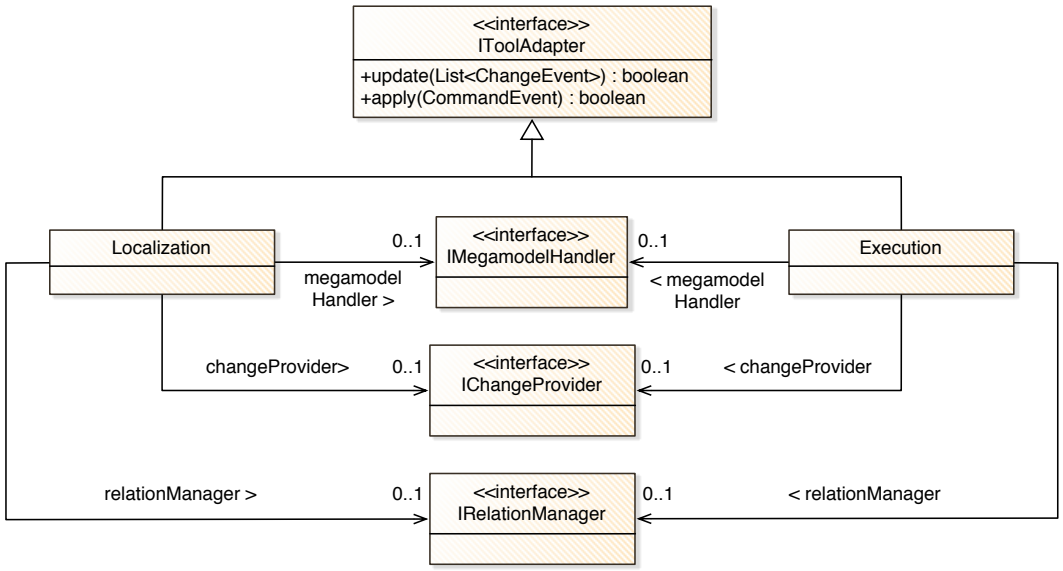

Figure A.13.: Design of the localization and execution components

The central concept of the localization is the class Localization. It is related to a megamodel handler, a change provider and a relation manager. Thus, it can access an application megamodel, a configuration megamodel and basic operations for maintaining and executing relations. The execution (Execution) is integrated in the same way and therefore looks pretty similar.

The *update* operation of the localization is implemented such that incoming change events are only buffered. If the *apply* operation is triggered with a localize command as parameter, the localization starts automatically maintaining the relations in an application megamodel based on the given set of changes in the buffer. Furthermore, a localize command should also have a synchronize command as predecessor to ensure that all changes were synchronized. Thus, localize always comes with a previous synchronize. The *update* operation of the execution is implemented such that incoming changes events are not buffered but rather used to continuously update a list of relations that have been impacted by these changes.

If the *apply* operation is triggered with an execute individual command, the *executeIndividual* operation, which is shown in Listing 5.3, is invoked. However, the implementation is different because each time the execution is triggered only one relation is applied. Additionally, the localization is not directly triggered but instead is triggered by means of the core dispatcher. Thus, after executing the relation new command events are created to trigger another round for executing the next relations. This is done until no more relations have to be executed.

If the *apply* operation is triggered with an execute all command, the *executeComplete* operation, which is shown in Listing 5.5, is invoked. However, this operation is also implemented differently as was done with the *executeIndividual* operation.

An execute individual or an execute all command always requires a preceding localize command to be published, which itself requires a preceding synchronization.

B. Additional Basics

B.1. Meta-Metamodel of EMF (Ecore)

The meta-metamodel of EMF is called Ecore, which has its roots in MOF (see [123]). It is compatible to EMOF, which is a subset of MOF, meaning that EMF metamodels and models are basically similar to EMOF metamodels and models. Figure B.1 shows the meta-metamodel of EMF that is also the meta-metamodel to which all metamodels that are shown in this thesis conform.

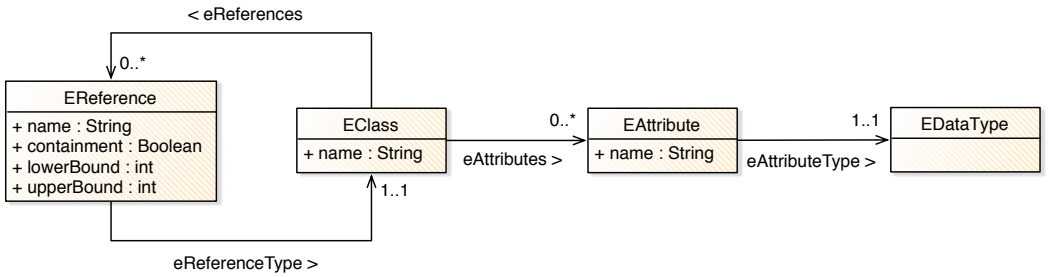

Figure B.1.: A simplified subset of Ecore [158]

The four primary concepts of Ecore are EClass, EReference, EAttribute and EDataType. EClass is used to represent a class in an EMF metamodel, which has a name, optional references and optional attributes. EReference is used to represent one end of an association between two classes. A reference has a name, a containment flag, lower and upper bounds (multiplicity) and a type (eReferenceType) that represents the target class. EAttribute represents an attribute of a class and has a name as well as a type (EDataType). EDataType represents the type of an attribute. A data type can be a primitive like int, float or an object type, which is another class.

B.2. Story Diagrams

Story diagrams were initially introduced in [58]. A Story diagram is a combination of a UML activity diagram and graph transformation rules. A first implementation of Story diagrams was provided in the Fujaba CASE tool [91]. Recently, an implementation has been provided in the context of Eclipse and EMF [64, 65]. This implementation comprises an EMF metamodel for the specification of Story diagrams, a graphical editor and an interpreter for executing Story diagrams. To better understand the examples that are shown in this thesis, Figure B.2 shows an abstract example of Story diagram using its concrete syntax.

A Story diagram consists of a single start activity, a set of Story patterns, and a set of final nodes. A Story pattern contains a graph transformation rule, which has a left-hand side (LHS) and a right-hand side (RHS). The LHS defines a pattern over metamodel elements that have to be matched, and the RHS defines the modification that will be applied to a match of a LHS.

An LHS as well as a RHS consist of Story pattern objects (SPO) and Story pattern links (SPL) in between. The type of an SPO is an element of a metamodel (e.g., A, B, C, and D). Each SPO must have a unique name in a Story diagram. An SPO can be considered as being bound or unbound. Bound means that a model element of a compatible type is assumed to be already mapped (bound) to the SPO, while unbound means the opposite.

SPOs and SPLs in the RHS of a Story pattern are further annotated with a ++ or a − −. SPOs and SPLs that are annotated with a ++ will be created to a match of the LHS while SPOs and SPLs that are annotated with a − − will be removed from a match of the LHS.

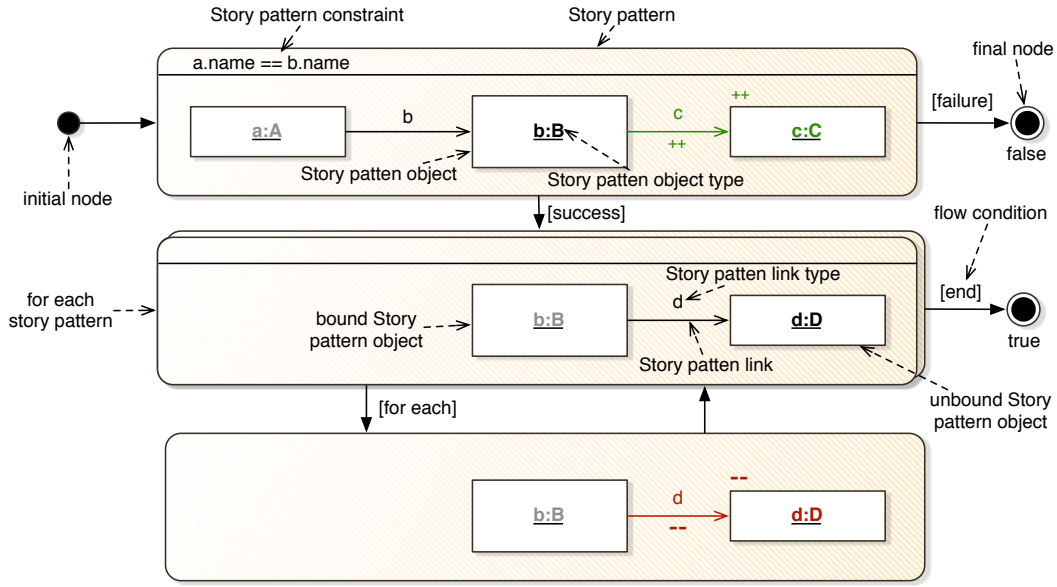

Figure B.2.: Example of Story diagram shown as concrete syntax

A Story pattern can have an additional constraint expressed in OCL. This constraint can be specified over all SPOs in the Story pattern. A Story pattern can also be defined as for each. The semantic of a for each Story pattern is defined as follows. For each match of the LHS, the next Story pattern is visited with the given match using the [for each] edge. This is repeated until no more matches can be found. In this case, the Story pattern is left via the [end] edge.

The Story diagram of Figure B.2 just means that for a given model element a of type A, a model element b of type B is searched. If the names of a matched a and b are similar, a model element of type C is added to b using a reference of type c. If no such match is found for the first Story pattern, the Story diagram terminates and returns false. Else, the second Story pattern is visited using the match of the first Story pattern. The second and third Story pattern define that each model element of type D, which is connected to the match b, is removed. If all model elements of type D have been removed, the Story diagram terminates and returns true.

B.3. Graphs and Graph Operations

Graphs are used several times in this thesis to apply known algorithms from graph theory. The graphs that are employed in this thesis are defined as shown in Definition B.3.1.

B.3.1 Definition (Graph) A graph G is a tuple (V, E) with V is a finite set of vertices and $E \subseteq (V \times V)$ is a finite set of edges connecting two vertices.

Graphs are distinguished between undirected and directed graphs. A graph is called to be undirected if all edges in E have no direction. A graph is a directed graph if all edges $(v, v') \in E$ have a direction, e.g., v is the source and v' is the target of (v, v').

B.3.1. Depth-First Search

The depth-first search algorithm is a common algorithm to traverse a directed graph, which does not necessarily need to be acyclic. The algorithm that is used in this thesis is outlined in Listing B.1.

The parameters of the DFS algorithm are a graph G and a set of starting vertices V'. Each vertex $v' \in V$ is not connected to any other vertex in V'. The result of the algorithm is a tuple (F, B) where $F \subseteq V$ is a set of vertices whose order is defined visiting the vertices the first time, and $B \subseteq V$ is a set

```
1  procedure DFS(G, V') : (F, B)
2    G = (V, E) //graph that is traversed
3    V' ⊆ V // set of vertices for starting DFS with all vertices in v' in V' are not connected
4
5    F := {∅}; // set of forward vertices
6    B := {∅}; // set of backward vertices
7
8    forall (v' ∈ V')
9      if (v' ∉ F)
10       visitDFS(G, v', F, B);
11
12   return (F, B);
13 endprocedure
```

Listing B.1: DFS algorithm

of vertices whose order is defined by leaving visited vertices because no more successors can be visited (backtracking).

The algorithm starts with iterating over all vertices $v' \in V'$. For each of these vertices, it is checked whether the vertex was previously visited ($v' \notin F$). If not, $visitDFS(G, v', F, B)$ is invoked that is shown in Listing B.2.

```
1  procedure visitDFS(G, v, F, B)
2    if (v ∉ F)
3      F := F ∪ {v}; // add v to the end of F
4      forall ((v, v') ∈ E)
5        if (v' ∉ F)
6          visitDFS(G, v', F, B);
7        endif
8      endforall
9      B := B ∪ {v}; // add v to the end of B
10   endif
11 endprocedure
```

Listing B.2: DFS algorithm: recursive visit operation

The $visitDFS$ operation starts from a single vertex v' that is not yet visited. Thus, it is first added to the end of F. Then, for all successors v' of v, if they are not yet added to F, the $visitDFS$ operation is recursively invoked. If all successors of v are traversed recursively, the vertex v is added to the end of B before backtracking starts.

B.3.2. Topological Sort

Sorting a graph topologically means bringing all the graph's vertices into a linear ordering. This ordering depends on the edges, requiring that the graph to be topologically sorted is acyclic and directed. A set of vertices is topologically sorted if, for any vertex, it holds that it only depends on preceding vertices in the set, but not on succeeding vertices in the set. Computing such a set of topologically sorted vertices is show in Listing B.3.

```
1  procedure topologicalSort(G, V') : S
2    G = (V, E) // graph that is traversed
3    V' ⊆ V; // set that are used as starting point for sorting
4
5    (F, B) := DFS(G, V');
6    return reverse(B);
7  endprocedure
```

Listing B.3: Topological sort algorithm

The algorithm takes a DAG G and a set of vertices $V' \subseteq V$ as input. V' is the set that is used as the starting point for a topological sort of G. The algorithm that is used in this thesis exploits the previously shown DFS algorithm by just reversing the order of the backward vertices set B.

C. Additional Details

C.1. Hierarchical Megamodels

C.1.1. Formal Definitions

C.1.1 Definition (Artifact Type Hierarchy Functions) Given a configuration megamodel M_C, different types of hierarchy relationships between individual artifact types can exist. These different hierarchy relationships are formally defined by using the following mapping functions.

- To ease estimating a set of artifact types that are directly or indirectly subordinate to an artifact type $a_t \in A_t$ four additional functions are employed.
  - $subs_{A_t} : A_t \rightarrow \mathcal{P}(A_t)$ with $\forall a'_t \in subs_{A_t}(a_t), \exists a_{C_t} \in A_{C_t} : a_t = sup_{A_{C_t}}(a_{C_t}) \wedge a'_t \in sub_{A_{C_t}}(a_{C_t})$ provides a set of artifact types that are subordinate to a given artifact type $a_t \in A_t$.
  - $subs_{A_t,A_{C_t}} : A_t \times A_{C_t} \rightarrow \mathcal{P}(A_t)$ with $\forall a'_t \in subs_{A_t,A_{C_t}}(a_t, a_{C_t}) : a_t = sup_{A_{C_t}}(a_{C_t}) \wedge a'_t \in sub_{A_{C_t}}(a_{C_t})$ provides a set of artifact types that are subordinate to a given artifact type $a_t \in A_t$ via an artifact type composition $a_{C_t} \in A_{C_t}$.
  - $subs^*_{A_t} : A_t \rightarrow \mathcal{P}(A_t)$ with $\forall a'_t \in subs^*_{A_t}(a_t) : a'_t \in subs_{A_t}(a_t) \vee \exists a''_t \in subs_{A_t}(a_t) : a'_t \in subs^*_{A_t}(a''_t)$ provides a set of artifact types that are directly or indirectly subordinate to a given artifact type $a_t \in A_t$.
  - $subs^*_{A_t,A_{C_t}} : A_t \times A_{C_t} \rightarrow \mathcal{P}(A_t)$ with $\forall a'_t \in subs^*_{A_t,A_{C_t}}(a_t, a_{C_t}) : a'_t \in subs_{A_t,A_{C_t}}(a_t, a_{C_t}) \vee \exists a''_t \in subs_{A_t,A_{C_t}}(a_t, a_{C_t}) : a'_t \in subs^*_{A_t,A_{C_t}}(a''_t, a_{C_t})$ provides a set of artifact types that are directly or indirectly subordinate to a given artifact type $a_t \in A_t$.
- To ease estimating a set of artifact types that are directly or indirectly superior to an artifact type $a_t \in A_t$ four additional functions are employed.
  - $sups_{A_t} : A_t \rightarrow \mathcal{P}(A_t)$ with $\forall a'_t \in sups_{A_t}(a_t), \exists a_{C_t} \in A_{C_t} : a_t \in sub_{A_{C_t}}(a_{C_t}) \wedge a'_t = sup_{A_{C_t}}(a_{C_t})$ provides a set of artifact types that are superior to a given artifact type $a_t \in A_t$.
  - $sups_{A_t,A_{C_t}} : A_t \times A_{C_t} \rightarrow \mathcal{P}(A_t)$ with $\forall a'_t \in sups_{A_t,A_{C_t}}(a_t, a_{C_t}) : a_t \in sub_{A_{C_t}}(a_{C_t}) \wedge a'_t = sup_{A_{C_t}}(a_{C_t})$ provides a set of artifact types that are superior to a given artifact type $a_t \in A_t$.
  - $sups^*_{A_t} : A_t \rightarrow \mathcal{P}(A_t)$ with $\forall a'_t \in sups^*_{A_t}(a_t) : a'_t \in sups_{A_t}(a_t) \vee \exists a''_t \in sups_{A_t}(a_t) : a'_t \in sups^*_{A_t}(a''_t)$ provides a set of artifacts types that are directly or indirectly superior to a given artifact type $a_t \in A_t$.
  - $sups^*_{A_t,A_{C_t}} : A_t \times A_{C_t} \rightarrow \mathcal{P}(A_t)$ with $\forall a'_t \in sups^*_{A_t,A_{C_t}}(a_t, a_{C_t}) : a'_t \in sups_{A_t,A_{C_t}}(a_t, a_{C_t}) \vee \exists a''_t \in sups_{A_t,A_{C_t}}(a_t, a_{C_t}) : a'_t \in sups^*_{A_t,A_{C_t}}(a''_t, a_{C_t})$ provides a set of artifacts types that are directly or indirectly superior to a given artifact type $a_t \in A_t$.
- An artifact type $a'_t \in A_t$ is defined to be a neighbor of another artifact type $a_t \in A_t$ if instances of both can be contained by instances of an artifact type $a''_t \in A_t$. This relationship is further defined by two functions:
  - $sibs_{A_t} : A_t \rightarrow \mathcal{P}_{A_t}$ with $\forall a'_t \in sibs_{A_t}(a_t) : a'_t \neq a_t \wedge \exists a''_t \in sups_{A_t}(a_t) : a'_t \in subs_{A_t}(a''_t)$ maps an artifact type a_t to a set of artifact types a'_t that are defined to be siblings of a_t.
  - $sibs_{A_t,A_{C_t}} : A_t \times A_{C_t} \rightarrow \mathcal{P}(A_t)$ with $\forall a'_t \in sibs_{A_t,A_{C_t}}(a_t, a_{C_t}) : a'_t \neq a_t \wedge \exists a''_t \in sups_{A_t}(a_t) : a'_t \in subs_{A_t,A_{C_t}}(a''_t, a_{C_t})$ maps an artifact type a_t to a set of artifact types a'_t that are siblings of a_t connected via a given artifact type composition a_{C_t}.

C.1.2 Definition (Artifact Hierarchy Functions) Given an application megamodel M_A, different types of hierarchy relationships between individual artifacts can exist. These different hierarchy relationships are formally defined by using the following mapping functions.

- An artifact $a' \in A$ that is directly or indirectly contained by another artifact $a \in A$ is defined to be a child of a. This relationship is further defined by four mapping functions:
  - $subs_A : A \rightarrow \mathcal{P}(A)$ maps an artifact a to a set of artifacts a' that are directly subordinate to a, with $\forall a' \in subs_A(a), \exists a_C \in A_C : a \neq a' \wedge a_C \in sub_A(a) \wedge a' \in sub_{A_C}(a_C)$.
  - $subs_{A,A_{C_t}} : A \times A_{C_t} \rightarrow \mathcal{P}(A)$ maps an artifact a to a set of artifacts a' that are directly subordinate to a by considering artifact compositions of type a_{C_t} only, with $\forall a' \in subs_{A,A_{C_t}}(a, a_{C_t}), \exists a_C \in A_C : a \neq a' \wedge a_{C_t} = type_{A_C}(a_C) \wedge a_C \in sub_A(a) \wedge a' \in sub_{A_C}(a_C)$.
  - $subs^*_A : A \rightarrow \mathcal{P}(A)$ maps an artifact a to a set of artifacts a' that are directly or indirectly subordinate to a, with $\forall a' \in subs^*_A(a) : a' \in subs_A(a) \vee \exists a'' \in subs_A(a) : a' \in subs^*_A(a'')$.
  - $subs^*_{A,A_{C_t}} : A \times A_{C_t} \rightarrow \mathcal{P}(A)$ maps an artifact a to a set of artifacts a' that are directly or indirectly subordinate to a by considering artifact compositions of type a_{C_t} only, with $\forall a' \in subs^*_{A,A_{C_t}}(a, a_{C_t}) : a' \in subs_{A,A_{C_t}}(a) \vee \exists a'' \in subs_{A,A_{C_t}}(a, a_{C_t}) : a' \in subs^*_{A,A_{C_t}}(a'', a_{C_t})$.
- An artifact $a' \in A$ that is directly or indirectly a container of another artifact $a \in A$ is defined to be a parent of a. This relationship is further defined by four mapping functions:
  - $sups_A : A \rightarrow A \cup \{\epsilon\}$ maps an artifact a to another artifact a' if a' is directly superior to a, and to ϵ if a is not contained by another artifact, with $\forall a' \in A : a' = sups_A(a) \Rightarrow a \neq a' \wedge \exists a_C \in A_C : a' = sup_{A_C}(a_C) \wedge a = sub_{A_C}(a_C)$.
  - $sups_{A,A_{C_t}} : A \times A_{C_t} \rightarrow A \cup \{\epsilon\}$ maps an artifact a to another artifact a' if a' is directly superior to a by considering artifact compositions of type a_{C_t} only, and to ϵ if a is not subordinate to another artifact, with $\forall a' \in A : a' = sups_{A,A_{C_t}}(a, a_{C_t}) \Rightarrow a \neq a' \wedge \exists a_C \in A_C : a_{C_t} = type_{A_C}(a_C) \wedge a' = sup_{A_C}(a_C) \wedge a = sub_{A_C}(a_C)$.
  - $sups^*_A : A \rightarrow \mathcal{P}(A)$ maps an artifact a to a set of artifacts a' that act as directly or indirectly superior to a, with $\forall a' \in sups^*_A(a) : a' = sups_A\ (a) \vee \exists a'' = sups_A(a) : a' \in sups^*_A(a'')$.
  - $sups^*_{A,A_{C_t}} : A \times A_{C_t} \rightarrow \mathcal{P}(A)$ maps an artifact a to a set of artifacts a' that act as directly or indirectly superior to a by considering artifact compositions of type a_{C_t} only, with $\forall a' \in sups^*_{A,A_{C_t}}(a, a_{C_t}) : a' = sups_{A,A_{C_t}}\ (a, a_{C_t}) \vee \exists a'' = sups_{A,A_{C_t}}(a, a_{C_t}) : a' \in sups^*_{A,A_{C_t}}(a'', a_{C_t})$.
- An artifact $a' \in A$ is defined to be a neighbor of another artifact $a \in A$ if both are not contained by any other artifact or if both are contained by the same artifact $a'' \in A$. This relationship is further defined by two mapping functions:
  - $sibs_A : A \rightarrow \mathcal{P}(A)$ maps an artifact a to a set of artifacts a' that are neighbors of a, with $\forall a' \in sibs_A(a) : a' \neq a \wedge sups_A(a) = sups_A(a')$.
  - $sibs_{A,A_{C_t}} : A \times A_{C_t} \rightarrow \mathcal{P}(A)$ maps an artifact a to a set of artifacts a' that are neighbors of a by only considering artifact compositions of type a_{C_t}, with $\forall a' \in sibs_{A,A_{C_t}}(a, a_{C_t}) : a' \neq a \wedge sups_{A,A_{C_t}}(a, a_{C_t}) = sups_{A,A_{C_t}}(a', a_{C_t})$.

C.2. Executable and Dynamic Hierarchical Megamodels

C.2.1. Directions of Relation Types

Relation types can implement different directions. This depends on their connected parameter types. This further implies that the model operations implementing the execution operation conform to the direction of the execution operation. The different types of directions are shown and explained in the following.

C.2.1.1. Uni-Directional

A relation type is considered to be uni-directional, if it is only connected to source or target parameter types. Figure C.1 shows five different uni-directional relation types relevant to execution operations, to give an idea about dependencies that can be defined by means of uni-directional relation types.

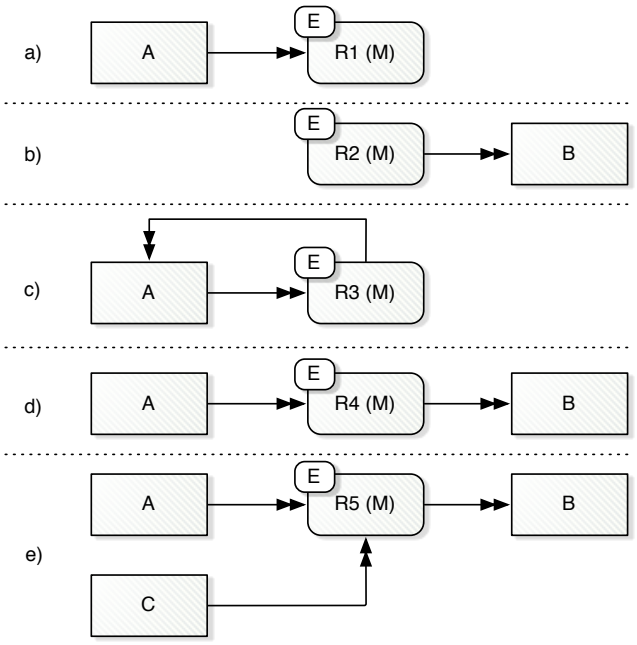

Figure C.1.: Uni-directional relation types

R1 may be the interface of a model operation that ascribes a certain annotation to an artifact type A, e.g., that some condition on an artifact of that type holds. R2 could represent a dependency that reflects the application of a factory, which creates an artifact type B. R3 is a relation type that might define a relationship indicating that an artifact of type A was copied into another artifact of the same type. The relation type R4 defines a relationship that reflects the application of model transformations, which takes an artifact type A as input and creates an artifact type B as output. R5 shows a relation type that may define a relationship that reflects the application of model merges, which take two artifact types A and C as input, creating an artifact type B as merged output.

C.2.1.2. Bi-Directional

A relation type is considered to be bi-directional, if it is only connected to source & target parameter types. Figure C.2 shows two examples of bi-directional relation types.

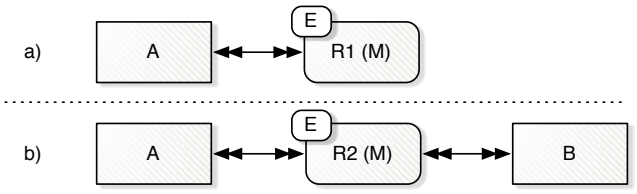

Figure C.2.: Bi-directional relation types

The first relation type R1, could be the representation of applying in-place model transformations (e.g., model refactoring). It takes an artifact type A as source and target at the same time. A second example of a bi-directional relation type is R2. R2 could reflect the application of model synchronizations. The

model synchronization keeps artifact types A and B in sync. Thus, both are source and target of R2 at the same time.

C.2.1.3. Hybrid-Directional

A relation type is considered to be hybrid-directional, if it is connected to at least one source & target parameter type and to at least one source or target parameter type. In Figure C.3 two exemplary hybrid-directional relation types are shown.

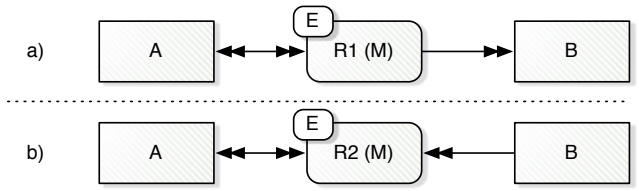

Figure C.3.: Hybrid-directional relation types

R1 could represent the application of an in-place model transformation on artifact type A that additionally provides an artifact type B, which could be a log file or model showing details about the performed model transformation. A slightly different situation is shown with relation type R2. R2 could reflect a hybrid-directional relation type that represents the applications of parameterized in-place model transformations. In this case, the parameter is encoded as artifact type B, which is an additional source of the relation type.

C.3. Evaluation Details

C.3.1. Extending Complex Model Operations

This section provides additional details of Section 6.1.2. Figure C.4 shows the abstract syntax of theSysML model from the application example (left) and the AUTOSAR model (right), which results from executing an extended SysML to AUTOSAR model transformation defined by the relation type SysML2AUTOSAR (see Figure 6.15).

C.3.2. Building Complex Model Operations

This section provides additional Listings, which show implementations of model transformations from Section 6.1.1 in Java code. These implementations are represented by the execution operations related to the relation types Package2Schema, Class2Table, Assoc2FKey and PrimitiveAttribute2Column. Each of these implementations is briefly explained in the following.

C.3.2.1. Implementation of Package2Schema

The implementation of the model operation represented by the execution operation of Package2Schema is shown in Listing C.1.

The implementation is very simple and just updates the name of the given Schema (schema) to the name of the given UMLPackage (pkg). The name of schema is only set if the current name is not set or different. The Schema does not need to be created. The model management framework is doing this when executing a relation of that type for the first time (see Listing 5.2). However, the given Schema is added to the given Folder by setting the URI of the underlying resource to the URI of the Folder and setting the name of the resource to the name of the UMLPackage with a *'.schema'* suffix.

C.3.2.2. Implementation of Class2Table

The implementation of the model operation represented by the execution operation of Class2Table (see Figure 6.2) is shown in Listing C.2.

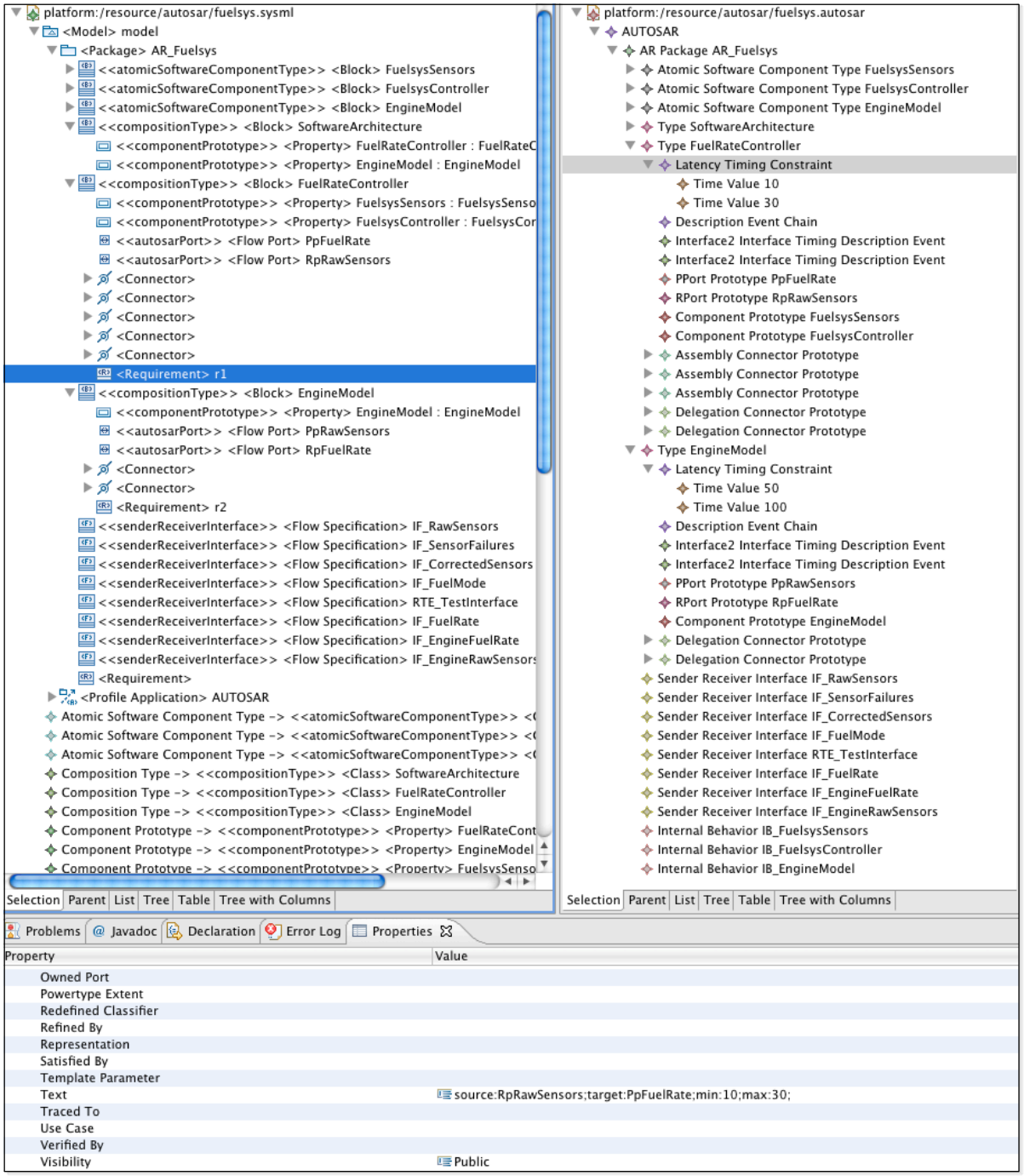

Figure C.4.: SysML example models (shown as abstract syntax by means of EMF tree views)

The parameters of the operation are all physical artifacts that are represented by artifacts connected to a relation of type Class2Table. Executing such a relation for the first time creates a Column, a Table and a Key. Executing such a relation any further will update these physical artifacts appropriately.

Thus, the operation updates the necessary attributes of table, column and key first. Subsequently, it adds column and key to table because table contains those two physical artifacts. The model management framework cannot achieve this because table is not defined in the composition context of this relation.

Figure C.5 shows two abstract syntaxes of a UMLModel (left) and a Schema (right). The bold-lined object classA already exists and is used as parameter for invoking class2Table. In addition, the dashed-

```
1  public void package2Schema(UMLPackage pkg, Folder folder, Schema schema)
2  {
3    //update schema resource
4    if (schema.getResource().getURI() != folder.getURI())
5      schema.getResource().setURI(folder.getURI());
6    if (!schema.getResource().getName().equals(pkg.getName() +'.schema'))
7      schema.getResource().setName(pkg.getName() +'.schema');
8
9    //update schema name
10   if (!schema.getName().equals(pkg.getName()))
11     schema.setName(pkg.getName());
12 }
```

Listing C.1: Implementation of the Package2Schema execution operation

```
1  public void class2Table(UMLClass clazz, Table table, Key key, Column column)
2  {
3    //update table name
4    if (!table.getName().equals(clazz.name))
5      table.setName(clazz.getName());
6    //update column name and type
7    if (!column.getName().equals(clazz.getName() +'_tid'))
8      column.setName(clazz.getName() +'_tid');
9    if (column.getType().equals('NUMBER'))
10     column.setType('NUMBER');
11   //update key name
12   if (!key.getName().equals(clazz.getName() +'_pk'))
13     key.setName(clazz.getName() +'_pk');
14   //add column to table if required
15   if (!table.getColumns().contains(column))
16     table.getColumns().add(column);
17   //add key to table if required
18   if (!table.getKey()! = key)
19     table.setKey(key);
20 }
```

Listing C.2: Implementation of the Class2Table execution operation

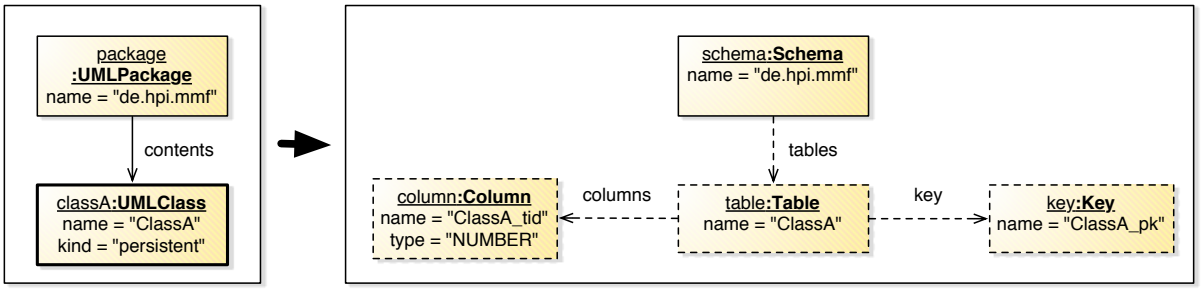

Figure C.5.: Effect of executing a relation of type Class2Table (shown as abstract syntax)

lined objects column, table and key are created/updated, as well as the parameters to class2Table. The Schema to the right shows the effect after applying the operation class2Table.

C.3.2.3. Implementation of Assoc2FKey

The implementation of the model operation represented by the execution operation of Assoc2FKey (see Figure 6.3) is shown in Listing C.3.

The operation is responsible for transforming a single Association into a Key, a ForeignKey and a Column. The operation first updates the attributes of the given ForeignKey (fkey) and Column (column). Then, it sets the refers to reference between fkey and the Key (pk). The other references are not set because the model management framework automatically sets them.

Figure C.6 shows the result of executing a relation of type Assoc2FKey for an Association a2b. When executing a relation of that type, the *executeRelation* operation will automatically create a ForeignKey (fKey) and a Column (column) and add them to a Table (tableA) that relates to the source UMLClass

```
1  public void assoc2FKey(UMLClass src, UMLClass tgt, Association assoc, Key pk, ForeignKey fkey,
       Column column)
2  {
3    //update foreign key name
4    String fKeyName := src.getName() +'_'+ assoc.getName() +'_'+ tgt.getName();
5    if (!fkey.getName().equals(fKeyName))
6      fkey.setName(fKeyName);
7    //update column name
8    String columnName := fKeyName +'_tid';
9    if (!column.getName().equals(columnName))
10     column.setName(columnName);
11   //update column type
12   if (column.getType().equals('NUMBER'))
13     column.setType('NUMBER');
14   //set refers to reference if required
15   if (!fkey.getRefersTo() != key)
16     fkey.setRefersTo(key);
17 }
```

Listing C.3: Implementation of the Association2ForeignKey execution operation

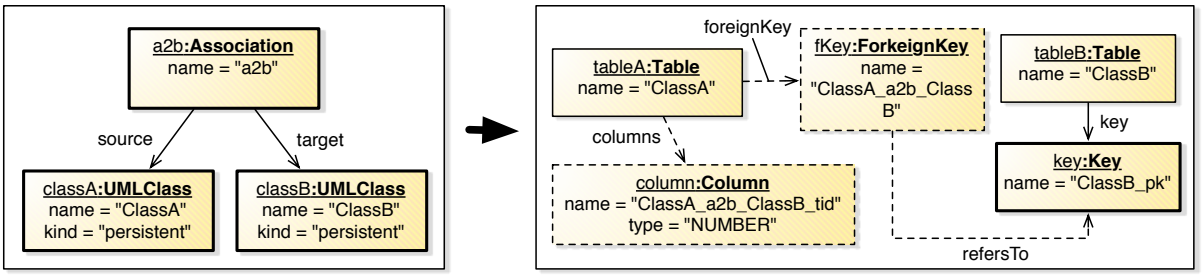

Figure C.6.: Effect of executing a relation of type Assoc2FKey (shown as abstract syntax)

(classA) of the given Association (a2b). Furthermore, the ForeignKey (fKey) is set to refer to the Key (pk) of the Table (tableA) that was transformed from the UMLClass (classB) which is the target of the Association (a2b).

C.3.2.4. Implementation of PrimitiveAttribute2Column

The implementation of the model operation represented by the execution operation of PrimitiveAttribute2Column (see Figure 6.5) is shown in Listing C.4.

```
1  public void primitiveAttribute2Column(UMLClass clazz, Attribute attr, PrimitiveDataType pd,
       Column column)
2  {
3    //update column name
4    String columnName := clazz.getName() +'_'+ attr.getName();
5    if (column.getName() != columnName)
6      column.setName(columnName);
7    //update column type
8    if (!column.getType().equals(primitiveTypeToSQLType(pd.getName())))
9      column.setType(primitiveTypeToSQLType(pd.getName()));
10 }
```

Listing C.4: Implementation of the PrimitiveAttribute2Column execution operation

The operation creates/updates a Column (column) based on a given Attribute (attr), whose type is a PrimitiveDataType (pd). Therefore, the operation needs to update the name and type of the Column. The Column is automatically added to a Table because a Table is in the composition context of a relation of that type. To set the correct type of the Column, a helper operation is invoked called *primitiveTypeToSQLType*, which is shown in Listing C.5.

The helper operation returns 'NUMBER' if the given PrimitiveDataType (primitiveType) is 'INTEGER' and it returns 'VARCHAR' if it is 'STRING'. Figure C.7 shows the result of executing a relation of type

```
1 public String primitiveTypeToSQLType(String primitiveType)
2 {
3   if (primitiveType.equals('INTEGER'))
4     return 'NUMBER';
5   if (primitiveType.equals('STRING'))
6     return 'VARCHAR';
7   return '';
8 }
```

Listing C.5: Helper operation to estimate an SQL type of a primitive data type

PrimitiveAttribute2ColumnA.

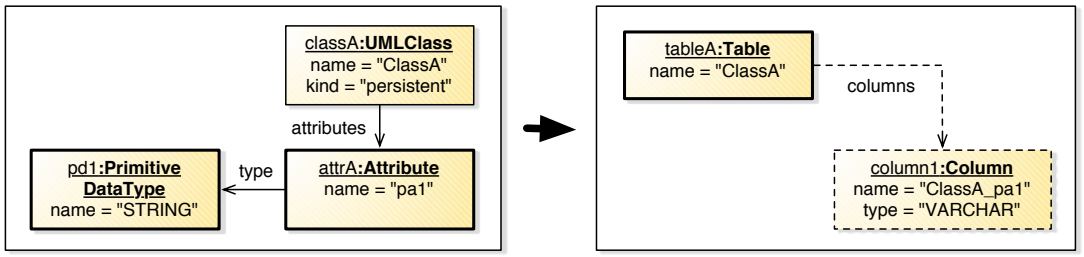

Figure C.7.: Effect of executing a relation of type PrimitiveAttribute2ColumnA (shown as abstract syntax)

Bibliography

[1] *Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools.*

[2] *New Oxford American Dictionary.* Oxford University Press, Inc., 2nd edition, 2005.

[3] Eclipse modeling framework project (EMF); http://www.eclipse.org/modeling/emf/?project=emf, September 2011.

[4] EMF compare project; http://wiki.eclipse.org/index.php/emfcompare, September 2011.

[5] JET project; http://www.eclipse.org/modeling/m2t/?project=jet#jet, September 2011.

[6] openarchitectureware (oAW); http://www.openarchitectureware.org/, September 2011.

[7] A. Agrawal, G. Karsai, and F. Shi. Graph transformations on domain-specific models. Technical Report TN 37203, Institute for Software Integrated Systems, Vanderbilt University, Nashville, 2003.

[8] N. Aizenbud-Reshef, B. T. Nolan, J. Rubin, and Y. Shaham-Gafni. Model traceability. *IBM Syst. J.*, 45(3):515–526, 2006.

[9] N. Aizenbud-Reshef, R. F. Paige, J. Rubin, Y. Shaham-Gafni, and D. S. Kolovos. Operational semantics for traceability. In *in Proc. Workshop on Traceability, ECMDA'05, Nurnberg, Germany*, November 2005.

[10] D. Akehurst and S. Kent. A relational approach to defining transformations in a metamodel. In J.-M. Jézéquel, H. Hussmann, and S. Cook, editors, *UML 2002 - The Unified Modeling Language*, volume 2460 of *Lecture Notes in Computer Science*, pages 155–178. Springer Berlin / Heidelberg, 2002.

[11] D. H. Akehurst, W. G. Howells, and K. D. McDonald-Maier. Kent model transformation language. In *Proc. of Model Transformations in Practice Workshop, MoDELS Conference, Montego Bay, Jamaica*, 2005.

[12] A. Aldazabal, T. Baily, F. Nanclares, A. Sadovykh, C. Hein, and T. Ritter. Automated model driven development processes. In *Proc. of the ECMDA workshop on Model Driven Tool and Process Integration*. Fraunhofer IRB Verlag, Stuttgart, 2008.

[13] M. Aleksy, T. Hildenbrand, C. Obergfell, and M. Schwind. A pragmatic approach to traceability in model-driven development. In A. Heinzl, H.-J. Appelrath, and E. J. Sinz, editors, *PRIMIUM*, volume 328 of *CEUR Workshop Proc.* CEUR-WS.org, 2008.

[14] F. Allilaire, J. Bézivin, H. Brunelière, and F. Jouault. Global Model Management In Eclipse GMT/AM3. In *Proc. of the Eclipse Technology eXchange workshop (eTX) at ECOOP'06*, 2006.

[15] B. Amar, H. Leblanc, and B. Coulette. A traceability engine dedicated to model transformation for software engineering. In J. Oldevik, G. Olsen, and T. Neple, editors, *ECMDA Traceability Workshop (ECMDA-TW)*, pages 7–16, 2008.

[16] G. Antoniol, G. Canfora, G. Casazza, and A. De Lucia. Maintaining traceability links during object-oriented software evolution. *Softw. Pract. Exper.*, 31:331–355, April 2001.

[17] G. Antoniol, G. Canfora, G. Casazza, A. De Lucia, and E. Merlo. Recovering traceability links between code and documentation. *IEEE Trans. Softw. Eng.*, 28:970–983, October 2002.

[18] H. U. Asuncion. Towards Practical Software Traceability. In *ICSE Companion '08: Companion of the 30th international conference on Software engineering*, pages 1023–1026, New York, NY, USA, 2008. ACM.

[19] H. U. Asuncion, A. U. Asuncion, and R. N. Taylor. Software traceability with topic modeling. In *Proc. of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE '10, pages 95–104, New York, NY, USA, 2010. ACM.

[20] H. U. Asuncion, F. François, and R. N. Taylor. An end-to-end industrial software traceability tool. In *ESEC-FSE '07: Proc. of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 115–124, New York, NY, USA, 2007. ACM.

[21] AUTOSAR GbR. *AUTOSAR - Specification of Timing Extensions*, 2009.

[22] A. Balogh, G. Bergmann, G. Csertán, L. Gönczy, A. Horváth, I. Majzik, A. Pataricza, B. Polgár, I. Ráth, D. Varró, and G. Varró. Graph transformations and model-driven engineering. chapter Workflow-driven tool integration using model transformations, pages 224–248. Springer-Verlag, Berlin, Heidelberg, 2010.

[23] M. Barbero, M. D. D. Fabro, and J. Bézivin. Traceability and Provenance Issues in Global Model Management. In J. Oldevik, T. Neple, and G. Olsen, editors, *3rd ECMDA Workshop on Traceability, Haifa (Israel)*, 2007.

[24] M. Barbero, F. Jouault, and J. Bézivin. Model driven management of complex systems: Implementing the macroscope's vision. In *ECBS '08: Proc. of the 15th Annual IEEE International Conference and Workshop on the Engineering of Computer Based Systems*, pages 277–286, Washington, DC, USA, 2008. IEEE Computer Society.

[25] P. A. Bernstein. Applying model management to classical meta data problems. In *Proc. of the Conf. on Innovative Database Research (CIDR)*, 2003.

[26] P. A. Bernstein, A. Y. Halevy, and R. A. Pottinger. A vision for management of complex models. *SIGMOD Rec.*, 29(4):55–63, 2000.

[27] T. Beyhl. Versionsbasiertes model management auf basis von verteilten megamodellen. Master's thesis, Hasso Plattner Institute at the University of Potsdam, 2011.

[28] J. Bézivin. On the unification power of models. *Software and System Modeling*, 4(2):171–188, 2005.

[29] J. Bézivin, S. Gérard, P.-A. Muller, and L. Rioux. MDA components: Challenges and Opportunities. In *Proc. of First International Workshop on Metamodelling for MDA*, pages 23 – 41, York, UK, November 2003.

[30] J. Bézivin, G. Hillairet, F. Jouault, I. Kurtev, and W. Piers. Bridging the MS/DSL Tools and the Eclipse Modeling Framework. In *OOPSLA Int. Workshop on Software Factories*, 2005.

[31] J. Bézivin, F. Jouault, P. Rosenthal, and P. Valduriez. *Modeling in the Large and Modeling in the Small*, volume 3599/2005 of *Lecture Notes in Computer Science*, chapter 3, pages 33–46. Springer Berlin / Heidelberg, August 2005.

[32] J. Bézivin, F. Jouault, and P. Valduriez. On the need for megamodels. In *Proc. of the OOPSLA/GPCE: Best Practices for Model-Driven Software Development workshop, 19th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2004.

[33] R. W. Blanning. Issues in the design of relational model management systems. In *Proc. of the May 16-19, 1983, national computer conference*, AFIPS '83, pages 395–401, New York, NY, USA, 1983. ACM.

[34] R. W. Blanning. An entity-relationship approach to model management. *Decision Support Systems*, 2(1):65 – 72, 1986.

[35] R. W. Blanning. A relational theory of model management. In *Proc. of the NATO Advanced Study Institute on Decision support systems: theory and application*, pages 19–53, London, UK, 1987. Springer-Verlag.

[36] A. Boronat, J. Ã. Carsã, and I. Ramos. Automatic support for traceability in a generic model management framework. 2005.

[37] A.-M. Chang, C. W. Holsapple, and A. B. Whinston. Model management issues and directions. *Decision Support Systems*, 9(1):19 – 37, 1993.

[38] A. Cicchetti, D. D. Ruscio, R. Eramo, and A. Pierantonio. Automating Co-evolution in Model-Driven Engineering. In *Proc. of the 2008 12th International IEEE Enterprise Distributed Object Computing Conference*, pages 222–231, Washington, DC, USA, 2008. IEEE Computer Society.

[39] J. Cleland-Huang, B. Berenbach, S. Clark, R. Settimi, and E. Romanova. Best practices for automated traceability. *Computer*, 40(6):27–35, 2007.

[40] J. Cleland-Huang, C. K. Chang, and M. Christensen. Event-Based Traceability for Managing Evolutionary Change. *IEEE Transactions on Software Engineering*, 29:796–810, 2003.

[41] S. Cook, G. Jones, S. Kent, and A. C. Wills. *Domain-Specific Development with Visual Studio DSL Tools*. Addison-Wesley Professional, 2007.

[42] M. Costa and A. R. da Silva. RT-MDD Framework – A Practical Approach. In J. Oldevik, G. K. Olsen, and T. Neple, editors, *ECMDA-TW'07: ECMDA Traceability Workshop, Haifa (Israel)*, pages 17–26. SINTEF, June 2007.

[43] J. Cuadrado and J. Molina. Modularization of model transformations through a phasing mechanism. *Software and Systems Modeling*, 8(3):325–345, 2009.

[44] K. Czarnecki and S. Helsen. Feature-based survey of model transformation approaches. *IBM Syst. J.*, 45(3):621–645, 2006.

[45] A. De Lucia, R. Oliveto, and G. Tortora. Adams re-trace: traceability link recovery via latent semantic indexing. In *ICSE '08: Proc. of the 30th international conference on Software engineering*, pages 839–842, New York, NY, USA, 2008. ACM.

[46] M. D. Del Fabro and P. Valduriez. Semi-automatic model integration using matching transformations and weaving models. In *Proc. of the 2007 ACM symposium on Applied computing*, SAC '07, pages 963–970, New York, NY, USA, 2007. ACM.

[47] D. Di Ruscio, L. Iovino, and A. Pierantonio. What is needed for managing co-evolution in MDE? In *Proc. of the 2nd International Workshop on Model Comparison in Practice*, IWMCP '11, pages 30–38, New York, NY, USA, 2011. ACM.

[48] R. Dömges and K. Pohl. Adapting Traceability To Project-SP. *Communications of the ACM*, 41(12):54–62, December 1998.

[49] N. Drivalos, D. S. Kolovos, R. F. Paige, and K. Fernandes. Engineering a DSL for Software Traceability. In *First International Conference, SLE 2008, Toulouse, France*, volume 5452/2009 of *Lecture Notes in Computer Science (LNCS)*, pages 151–167. Springer-Verlag, 29-30 September 2009.

[50] N. Drivalos, R. F. Paige, K. Fernandes, and D. S. Kolovos. Towards rigorously defined model-to-model traceability. In *Proc. 4th Workshop on Traceability, ECMDA'08, Berlin, Germany*, June 2008.

[51] A. Egyed. A scenario-driven approach to traceability. In *ICSE '01: Proc. of the 23rd International Conference on Software Engineering*, pages 123–132, Washington, DC, USA, 2001. IEEE Computer Society.

[52] A. Egyed and P. Grünbacher. Automating requirements traceability: Beyond the record & replay paradigm. In *ASE '02: Proc. of the 17th IEEE international conference on Automated software engineering*, page 163, Washington, DC, USA, 2002. IEEE Computer Society.

[53] A. Etien, A. Muller, T. Legrand, and X. Blanc. Combining independent model transformations. In *Proc. of the 2010 ACM Symposium on Applied Computing*, SAC '10, pages 2237–2243, New York, NY, USA, 2010. ACM.

[54] J. R. Falleri, M. Huchard, and C. Nebut. Towards a traceability framework for model transformations in kermeta. In T. Neple, J. Oldevik, and J. Aagedal, editors, *ECMDA-TW'06: ECMDA Traceability Workshop, Bilbao (Spain)*. SINTEF, 2006.

[55] J.-M. Favre. Foundations of Model (Driven) (Reverse) Engineering – Episode I: Story of The Fidus Papyrus and the Solarus. In *Proc. of Dagstuhl Seminar on Model Driven Reverse Engineering*, 2004.

[56] J.-M. Favre. Megamodelling and etymology. In *Proc. of Dagstuhl Seminar on Transformation Techniques in Software Engineering*, volume 05161, 2005.

[57] J.-M. Favre and T. Nguyen. Towards a Megamodel to Model Software Evolution Through Transformations. *Electr. Notes Theor. Comput. Sci.*, 127(3):59–74, 2005.

[58] T. Fischer, J. Niere, L. Torunski, and A. Zündorf. Story Diagrams: A New Graph Rewrite Language Based on the Unified Modeling Language and Java. In *TAGT'98: Selected papers from the 6th International Workshop on Theory and Application of Graph Transformations*, pages 296–309, London, UK, 2000. Springer-Verlag.

[59] W. B. Frakes and R. Baeza-Yates. *Information Retrieval: Data Structures and Algorithms*. Prentice-Hall, Englewood Cliffs, NJ, 1992.

[60] R. France and B. Rumpe. Model-driven development of complex software: A research roadmap. In *FOSE '07: 2007 Future of Software Engineering*, pages 37–54, Washington, DC, USA, 2007. IEEE Computer Society.

[61] D. D. Gajski, S. Abdi, A. Gerstlauer, and G. Schirner. *Embedded System Design: Modeling, Synthesis and Verification*. Springer Publishing Company, Incorporated, 1st edition, 2009.

[62] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns - Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional Computing Series, 32 edition, 2005.

[63] A. Gavras, M. Belaunde, L. Ferreira Pires, and J. Andrade Almeida. Towards an mda-based development methodology for distributed applications. In M. van Sinderen and L. Ferreira Pires, editors, *1st European Workshop on Model-Driven Architecture with Emphasis on Industrial Applications (MDA-IA 2004)*, pages 71–81, Enschede, 2004. Centre for Telematics and Information Technology, University of Twente.

[64] H. Giese, S. Hildebrandt, and **A. Seibel**. Feature Report: Modeling and Interpreting EMF-based Story Diagrams. In *Proc. of the 7th International Fujaba Days*, 2009.

[65] H. Giese, S. Hildebrandt, and **A. Seibel**. Improved Flexibility and Scalability by Interpreting Story Diagrams. In T. Magaria, J. Padberg, and G. Taentzer, editors, *Proc. of the Eighth International Workshop on Graph Transformation and Visual Modeling Techniques (GT-VMT 2009)*, 2009.

[66] H. Giese, S. Neumann, and S. Hildebrandt. Model Synchronization at Work: Keeping SysML and AUTOSAR Models Consistent. In G. Engels, C. Lewerentz, W. Schäfer, A. Schürr, and B. Westfechtel, editors, *Graph Transformations and Model Driven Enginering - Essays Dedicated to Manfred Nagl on the Occasion of his 65th Birthday*, volume 5765 of *LNCS*, pages 555–579. Springer Berlin / Heidelberg, 2010.

[67] H. Giese, M. Tichy, S. Burmester, W. Schäfer, and S. Flake. Towards the compositional verification of real-time uml designs. In *Proc. of the 9th European software engineering conference held jointly with 11th ACM SIGSOFT international symposium on Foundations of software engineering*, ESEC/FSE-11, pages 38–47, New York, NY, USA, 2003. ACM.

[68] H. Giese and R. Wagner. Incremental Model Synchronization with Triple Graph Grammars. In O. Nierstrasz, J. Whittle, D. Harel, and G. Reggio, editors, *Proc. of the 9th International Conference on Model Driven Engineering Languages and Systems (MoDELS), Genova, Italy*, volume 4199 of *Lecture Notes in Computer Science (LNCS)*, pages 543–557. Springer Verlag, October 2006.

[69] H. Giese and R. Wagner. From model transformation to incremental bidirectional model synchronization. *Software and Systems Modeling*, 8(1), 3 2009.

[70] O. C. Z. Gotel and C. W. Finkelstein. An analysis of the requirements traceability problem. pages 94–101, 1994.

[71] B. Grammel. Towards a generic traceability framework for model-driven software engineering. *Transformation*, pages 44–47, 2009.

[72] R. Hebig. An Approach to Integrating Model Management and Software Development Processes,. In *Doctoral Symposium at MODELS 2011*, 2011.

[73] R. Hebig and H. Giese. MDE Settings in SAP. A Descriptive Field Study. Technical report, Hasso-Plattner Institut at University of Potsdam, 2011.

[74] R. Hebig, **A. Seibel**, and H. Giese. On the Unification of Megamodels. In *Proc. of the 4th International Workshop on Multi Paradigm Modeling (MPM'10) at the 13th IEEE/ACM International Conference on Model Driven Engineering Languages and Systems (MoDELS 2010), Oslo, Norway*, 3 October 2010.

[75] F. Heidenreich, J. Johannes, M. Seifert, and C. Wende. Closing the gap between modelling and java. In M. van den Brand, D. Gaševic, and J. Gray, editors, *Software Language Engineering*, volume 5969 of *Lecture Notes in Computer Science*, pages 374–383. Springer Berlin / Heidelberg, 2010.

[76] F. Heidenreich, J. Kopcsek, and U. Aßmann. Safe composition of transformations. In *Proc. of the Third international conference on Theory and practice of model transformations*, ICMT'10, pages 108–122, Berlin, Heidelberg, 2010. Springer-Verlag.

[77] H. Herold. *make: Das Profitool zur automatischen Generierung von Programmen.* Addison-Wesley, Bonn, 3. edition, 2003.

[78] S. Holzner, D. Nehren, and B. Galbraith. *Ant: The Definitive Guide.* O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2005.

[79] I. Ivkovic and K. Kontogiannis. Towards automatic establishment of model dependencies using formal concept analysis. *International Journal of Software Engineering and Knowledge Engineering*, 16(4):499–522, 2006.

[80] H.-Y. Jiang, T. N. Nguyen, I.-X. Chen, H. Jaygarl, and C. K. Chang. Incremental latent semantic indexing for automatic traceability link evolution management. In *Proc. of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, ASE '08, pages 59–68, Washington, DC, USA, 2008. IEEE Computer Society.

[81] W. Jirapanthong and A. Zisman. Xtraque: traceability for product line systems. *Software and Systems Modeling*, 8:117–144, 2009. 10.1007/s10270-007-0066-8.

[82] F. Jouault. Loosely coupled traceability for atl. In *Proc. of the European Conference on Model Driven Architecture (ECMDA) workshop on traceability*, pages 29–37, 2005.

[83] F. Jouault, F. Allilaire, J. Bézivin, and I. Kurtev. Atl: A model transformation tool. *Science of Computer Programming*, 72(1-2):31 – 39, 2008. Special Issue on Second issue of experimental software and toolkits (EST).

[84] F. Jouault, F. Allilaire, J. Bézivin, I. Kurtev, and P. Valduriez. Atl: a qvt-like transformation language. In *Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*, OOPSLA '06, pages 719–720, New York, NY, USA, 2006. ACM.

[85] F. Jouault and I. Kurtev. Transforming models with atl. In *Satellite Events at the MoDELS 2005 Conference*, volume 3844 of *Lecture Notes in Computer Science*, pages 128–138, Berlin, 2006. Springer Verlag.

[86] F. Jouault, B. Vanhooff, H. Bruneliere, G. Doux, Y. Berbers, and J. Bezivin. Inter-dsl traceability and navigability support by combining megamodeling and model weaving. In *Proc. of Special Track on the Coordination Models, Languages and Applications at the 25th Symposium On Applied Computing (SAC 2010), Sierre, Switzerland, March 22-26*, 2010.

[87] A. Kalnins, J. Barzdins, and E. Celms. Model transformation language mola. In U. Aßmann, M. Aksit, and A. Rensink, editors, *Model Driven Architecture*, volume 3599 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, 2005.

[88] R. Kammal. *EMBEDDED SYSTEMS : Architecture, Programming and Design.* Number 0070667640. Tata McGraw-Hill Publishing, second edition, 2008.

[89] S. Kämper. On the appropriateness of petri nets in model building and simulation. *Syst. Anal. Model. Simul.*, 8:689–714, August 1991.

[90] D. Kensche, C. Quix, X. Li, and Y. Li. GeRoMeSuite: a system for holistic generic model management. In *VLDB '07: Proc. of the 33rd international conference on Very large data bases*, pages 1322–1325, Vienna, Austria, 2007. VLDB Endowment.

[91] F. Klein, U. Nickel, J. Niere, and A. Zündorf. From UML to Java And Back Again. Technical Report tr-ri-00-216, University of Paderborn, Paderborn, Germany, September 1999.

[92] A. Kleppe. Mcc: A model transformation environment. In *Model Driven Architecture – Foundations and Applications: Second European Conference, ECMDA-FA 2006, Bilbao, Spain*, LNCS, pages 173–187. Springer Berlin / Heidelberg, 2006.

[93] D. Kolovos, L. Rose, and R. Paige. *The epsilon Book*, 2011.

[94] D. S. Kolovos. Establishing correspondences between models with the epsilon comparison language. In *Proc. of the 5th European Conference on Model Driven Architecture - Foundations and Applications*, ECMDA-FA '09, pages 146–157, Berlin, Heidelberg, 2009. Springer-Verlag.

[95] D. S. Kolovos, R. Paige, and F. Polack. A Framework for Composing Modular and Interoperable Model Management Tasks. In *MDTPI workshop, EC-MDA, Berlin, Germany*, June 2008.

[96] D. S. Kolovos, R. F. Paige, and F. Polack. On Demand Merging of Traceability Links with Models. In T. Neple, J. Oldevik, and J. Aagedal, editors, *ECMDA-TW'06: ECMDA Traceability Workshop, Bilbao (Spain)*. SINTEF, 2006.

[97] D. S. Kolovos, R. F. Paige, and F. A. Polack. The epsilon transformation language. In *Proc. of the 1st international conference on Theory and Practice of Model Transformations*, ICMT '08, pages 46–60, Berlin, Heidelberg, 2008. Springer-Verlag.

[98] D. S. Kolovos, R. F. Paige, and F. A. C. Polack. On the evolution of ocl for capturing structural constraints in modelling languages. In *Rigorous Methods for Software Construction and Analysis, Essays Dedicated to Egon Börger on the Occasion of his 60th Birthday*, volume 5115 of *Lecture Notes in Computer Science (LNCS)*, pages 204–218. Springer, 2009.

[99] H.-J. Kreowski, S. Kuske, and C. von Totth. Stepping from graph transformation units to model transformation units. *ECEASST*, 30, 2010.

[100] I. Kurtev. State of the Art of QVT: A Model Transformation Language Standard. In *Applications of Graph Transformations with Industrial Relevance: Third International Symposium, AGTIVE 2007, Kassel, Germany,*, pages 377–393, Berlin, Heidelberg, 10-12 October 2007. Springer-Verlag.

[101] I. Kurtev, K. van den Berg, and F. Jouault. Evaluation of rule-based modularization in model transformation languages illustrated with ATL. In *SAC '06: Proc. of the 2006 ACM symposium on Applied computing*, pages 1202–1209, New York, NY, USA, 2006. ACM Press.

[102] P. Lago, H. Muccini, and H. Van Vliet. A Scoped Approach to Traceability Management. *The Journal of Systems and Software*, (82):168–182, 2009.

[103] J. Lara and H. Vangheluwe. Atom3: A tool for multi-formalism and meta-modelling. In R.-D. Kutsche and H. Weber, editors, *Fundamental Approaches to Software Engineering*, volume 2306 of *Lecture Notes in Computer Science*, pages 174–188. Springer Berlin / Heidelberg, 2002.

[104] M. Lawley and J. Steel. Practical Declarative Model Transformation with Tefkat. In *MoDELS Satellite Events*, volume 3844 of *LNCS*, pages 139–150. Springer, 2005.

[105] A. E. Limón and J. Garbajosa. The need for a unifying traceability scheme. In J. Oldevik and J. Aagedal, editors, *ECMDA-TW'05: ECMDA Traceability Workshop, Nürnberg (Germany)*, pages 47–56. SINTEF, Novermber 2005.

[106] H. Lochmann and A. Hessellund. An integrated view on modeling with multiple domain-specific languages. In *Proc. of the IASTED International Conference Software Engineering (SE 2009)*, pages 1–10. ACTA Press, February 2009.

[107] A. D. Lucia, F. Fasano, R. Oliveto, and G. Tortora. Recovering traceability links in software artifact management systems using information retrieval methods. *ACM Trans. Softw. Eng. Methodol.*, 16(4):13, 2007.

[108] P. Mäder, O. Gotel, and I. Philippow. Enabling automated traceability maintenance by recognizing development activities applied to models. In *23rd IEEE/ACM International Conference on Automated Software Engineering*, pages 49–58. IEEE, 2008.

[109] P. Mäder, O. Gotel, and I. Philippow. Rule-based maintenance of post-requirements traceability relations. In *Proc. of the 2008 16th IEEE International Requirements Engineering Conference*, pages 23–32, Washington, DC, USA, 2008. IEEE Computer Society.

[110] P. Mäder, O. Gotel, and I. Philippow. Enabling Automated Traceability Maintenance Through the Upkeep of Traceability Relations. In *Proc. 5th European Conference on Model-Driven Architecture Foundations and Applications (ECMDA2009) – LNCS5562*, pages 174–189, Enschede, Netherlands, June 2009.

[111] J. I. Maletic, M. L. Collard, and B. Simoes. An xml based approach to support the evolution of model-to-model traceability links. In *Proc. of the 3rd international workshop on Traceability in emerging forms of software engineering*, TEFSE '05, pages 67–72, New York, NY, USA, 2005. ACM.

[112] J. I. Maletic, E. V. Munson, A. Marcus, and T. N. Nguyen. Using a hypertext model for traceability link conformance analysis. In *Proc. of the 2nd Int. Workshop on Traceability in Emerging Forms of Software Engineering (TEFSE)*, pages 47–54, 2003.

[113] D. D. F. Marcos, B. Jean, J. Frédéric, B. Erwan, and G. Guillaume. Amw: A generic model weaver. In *Premières Journées sur l'Ingénierie Dirigée par les Modèles*, 2005.

[114] F. Marschall and P. Braun. Model transformations for the mda with botl. In *Proc. of the Workshop on Model Driven Architecture: Foundations and Applications, Enschede, The Netherlands*, pages 25—36, 2003.

[115] S. Melnik. *Generic Model Management: Concepts And Algorithms.* Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2004.

[116] S. Melnik. Model management: First steps and beyond. In G. Vossen, F. Leymann, P. C. Lockemann, and W. Stucky, editors, *Datenbanksysteme in Business, Technologie und Web, 11. Fachtagung des GI-Fachbereichs "Datenbanken und Informationssysteme" (DBIS), Karlsruhe, 2.-4. März 2005*, volume 65 of *LNI*, pages 455–464. GI, 2005.

[117] T. Mens and P. V. Gorp. A taxonomy of model transformation. *Electronic Notes in Theoretical Computer Science*, 152:125–142, March 2006.

[118] P. Mohagheghi, V. Dehlen, and T. Neple. Definitions and approaches to model quality in model-based software development - a review of literature. *Inf. Softw. Technol.*, 51:1646–1669, December 2009.

[119] P.-A. Muller, F. Fleurey, and J.-M. Jézéquel. Weaving executability into object-oriented meta-languages. In S. K. L. Briand, editor, *Proc. of MODELS/UML'2005*, LNCS, pages 264–278, Montego Bay, Jamaica, 2005. Springer.

[120] L. Naslavsky, T. A. Alspaugh, D. J. Richardson, and H. Ziv. Using scenarios to support traceability. In *Proc. of the 3rd international workshop on Traceability in emerging forms of software engineering*, TEFSE '05, pages 25–30, New York, NY, USA, 2005. ACM.

[121] S. Neumann and **A. Seibel**. Toward Mega Models for Maintaining Timing Properties of Automotive Systems. In *Proc. of the 3rd International Workshop on Model-Based Architecting and Construction of Embedded Systems (ACES-MB 2010) at the 13th IEEE/ACM International Conference on Model Driven Engineering Languages and Systems (MoDELS 2010), Oslo, Norway*, October 2010.

[122] T. Nguyen, S. C. Gupta, and E. V. Munson. Versioned hypermedia can improve software document management. In *Proc. of the thirteenth ACM conference on Hypertext and hypermedia*, HYPERTEXT '02, pages 192–193, New York, NY, USA, 2002. ACM.

[123] Object Management Group. Meta Object Facility (MOF) Core Specification. http://www.omg.org/spec/MOF/2.0/PDF/.

[124] Object Management Group. *MOF 2.0 QVT 1.0 Specification*, 2008.

[125] J. Oldevik. Transformation composition modelling framework. In L. Kutvonen and N. Alonistioti, editors, *Distributed Applications and Interoperable Systems, 5th IFIP WG 6.1 International Conference, DAIS 2005, Athens, Greece, June 15-17, 2005*, volume 3543 of *Lecture Notes in Computer Science*, pages 108–114. Springer, 2005.

[126] G. K. Olsen, J. Aagedal, and J. Oldevik. Aspects of reusable model transformations. In A. Kleppe, editor, *First European Workshop on Composition of Model Transformations, CMT 2006*, number TR-CTIT-06-80, pages 21–26, Enschede, 2006. Centre for Telematics and Information Technology, University of Twente.

[127] G. K. Olsen and J. Oldevik. Scenarios of Traceability in Model to Text Transformations. In *Model Driven Architecture- Foundations and Applications, Third European Conference, ECMDA-FA 2007, Haifa, Israel*, pages 144–156, 2007.

[128] OMG. UML 2.0 Superstructure Specification, Object Management Group, Version 2.0, formal/05-07-04, 2005.

[129] openArchitectureWare. The modeling workflow engine, http://www.eclipse.org/ modeling/emft/?project=mwe, 2011.

[130] openArchitectureWare (oAW). Xpand project; http://www.eclipse.org/modeling/m2t/?project=xpand, September 2011.

[131] openArchitectureWare (oAW). Xtend project; http://www.eclipse.org/xtext/#xtend2, September 2011.

[132] openArchitectureWare (oAW). Xtext project; http://www.eclipse.org/xtext/, September 2011.

[133] L. Orman. Flexible management of computational models. *Decis. Support Syst.*, 2:225–234, September 1986.

[134] R. F. Paige, G. K. Olsen, D. S. Kolovos, S. Zschaler, and C. Power. Building Model-Driven Engineering Traceability Classifications. In *ECMDA-TW'08: Proc. of 4th Workshop on Traceability, Berlin, Germany*. SINTEF, 9-12 June 2008.

[135] D. Perovich, M. C. Bastarrica, and C. Rojas. Model-Driven Approach to Software Architecture Design. In *Proc. of ICSE Workshop on Sharing and Reusing Architectural Knowledge (SHARK '09)*, Washington, DC, USA, 2009. IEEE Computer Society.

[136] G. Pietrek and J. Trompeter, editors. *Modellgetriebene Softwareentwicklung: MDA und MDSD in der Praxis*. Entwickler.Press, Frankfurt am Main, 2007.

[137] F. Pinheiro. Requirements traceability. In J. Leite and J. Doorn, editors, *Perspectives on Software Requirements*, chapter 5. Springer Berlin / Heidelberg, 2003.

[138] J. Richardson and J. Green. Traceability through automatic program generation. 2003.

[139] M. Richters and M. Gogolla. Ocl: Syntax, semantics, and tools. In *Object Modeling with the OCL, The Rationale behind the Object Constraint Language*, pages 42–68, London, UK, UK, 2002. Springer-Verlag.

[140] J. E. Rivera, D. Ruiz-Gonzalez, F. Lopez-Romero, J. Bautista, and A. Vallecillo. Orchestrating ATL model transformations. In F. Jouault, editor, *Proc. of MtATL 2009: 1st International Workshop on Model Transformation with ATL*, pages 34–46, Nantes, France, July 2009.

[141] L. Rose, R. Paige, D. Kolovos, and F. Polack. The epsilon generation language. In I. Schieferdecker and A. Hartman, editors, *Model Driven Architecture – Foundations and Applications*, volume 5095 of *Lecture Notes in Computer Science*, pages 1–16. Springer Berlin / Heidelberg, 2008.

[142] J. Rothenberg. *The nature of modeling*, pages 75–92. John Wiley & Sons, Inc., New York, NY, USA, 1989.

[143] R. Salay. Towards a Formal Framework for Multimodeling in Software Engineering. In *Proc. of the Doctoral Symposium at the ACM/IEEE 10th International Conference on Model-Driven Engineering Languages and Systems*, volume Vol-26, Nashville (TN), USA, October 2007.

[144] R. Salay, M. Chechik, S. Easterbrook, Z. Diskin, P. McCormick, S. Nejati, M. Sabetzadeh, and P. Viriyakattiyaporn. An Eclipse-based tool framework for software model management. In *eclipse '07: Proc. of the 2007 HSLA workshop on eclipse technology eXchange*, pages 55–59, New York, NY, USA, 2007. ACM.

[145] R. Salay, J. Mylopoulos, and S. Easterbrook. Managing models through macromodeling. In *Proc. of ASE*, 2008.

[146] R. Salay, J. Mylopoulos, and S. Easterbrook. Using macromodels to manage collections of related models. In *CAiSE*, pages 141–155, 2009.

[147] A. Schürr. Specification of graph translators with triple graph grammars. In E. Mayr, G. Schmidt, and G. Tinhofer, editors, *Graph-Theoretic Concepts in Computer Science*, volume 903 of *Lecture Notes in Computer Science*, pages 151–163. Springer Berlin / Heidelberg, 1995.

[148] H. Schwarz, J. Ebert, and A. Winter. Graph-based traceability: a comprehensive approach. *Software and Systems Modeling*, 9:473–492, 2010. 10.1007/s10270-009-0141-4.

[149] **A. Seibel**. From software traceability to global model management and back again. In *15th European Conference on Software Maintenance and Reengineering (CSMR'11), Doctoral Symposium, Oldenburg, Germany*, 3 2011.

[150] **A. Seibel**, R. Hebig, and H. Giese. *Software and Systems Traceability*, chapter Traceability in Model-Driven Engineering: Efficient and Scalable Traceability Maintenance. Springer London, 2012. to be published.

[151] **A. Seibel**, R. Hebig, S. Neumann, and H. Giese. A dedicated language for context composition and execution of true black-box model transformations. In *4th International Conference on Software Language Engineering (SLE 2011) , Braga, Portugal*, 7 2011. to be published.

[152] **A. Seibel**, S. Neumann, and H. Giese. Dynamic hierarchical mega models: comprehensive traceability and its efficient maintenance. *Software and Systems Modeling*, 9:493–528, September 2010.

[153] Y. Shaham-Gafni and A. Hartman. Modelware traceability: A survey (draft). Technical report, IBM Haifa Research Lab, 2005.

[154] H. A. Simon. The architecture of complexity. *Proc. of the American Philosophical Society*, 106(6):467–482, 1962.

[155] H. M. Sneed. *Software-Qualitätssicherung für kommerzielle Anwendungssysteme*. Verlagsgesellschaft Rudolf Müller, 1983.

[156] S. Spaccapietra, P. Atzeni, F. Fages, M.-S. Hacid, M. Kifer, J. Mylopoulos, B. Pernici, P. Shvaiko, J. Trujillo, I. Zaihrayeu, D. Kensche, C. Quix, M. Chatti, and M. Jarke. *GeRoMe : A Generic Role Based Metamodel for Model Management*, volume 4380, pages 82–117. Springer Berlin / Heidelberg, 2007.

[157] G. Spanoudakis and A. Zisman. *Software Traceability: A Roadmap*, pages 395–428. Handbook of Software Engineering & Knowledge Engineering: Recent Advances. World Scientific Publishing Company, 3 edition, 2005.

[158] D. Steinberg, F. Budinsky, M. Paternostro, and E. Merks. *EMF - Eclipse Modeling Framework Second Edition*. Number ISBN 978-0-321-33188-5. Addison-Wesley, 2nd edition, 2009.

[159] G. Taentzer. Agg: A graph transformation environment for modeling and validation of software. In J. Pfaltz, M. Nagl, and B. Böhlen, editors, *Applications of Graph Transformations with Industrial Relevance*, volume 3062 of *Lecture Notes in Computer Science*, pages 446–453. Springer Berlin / Heidelberg, 2004.

[160] B. Vanhooff, D. Ayed, and Y. Berbers. A framework for transformation chain development processes. In A. Kleppe, editor, *First European Workshop on Composition of Model Transformations, CMT 2006*, number TR-CTIT-06-80, pages 3–8, Enschede, 2006. Centre for Telematics and Information Technology, University of Twente.

[161] B. Vanhooff, D. Ayed, S. Van Baelen, W. Joosen, and Y. Berbers. Uniti: A unified transformation infrastructure. In *Model Driven Engineering Languages and Systems: 10th International Conference, MoDELS 2007, Nashville, USA*, LNCS, pages 31–45. Springer Berlin / Heidelberg, 2007.

[162] B. Vanhooff and Y. Berbers. Breaking up the transformation chain. In *Proc. of the Best Practices for Model-Driven Software Development at OOPSLA 2005, San Diego, California, USA*, San Diego, California, USA, 2005.

[163] B. Vanhooff and Y. Berbers. Supporting modular transformation units with precise transformation traceability metadata. In *ECMDA-TW: Traceability Workshop, at European Conference on Model Driven Architecture, Nürnberg, Germany*, November 2005.

[164] B. Vanhooff, S. Van Baelen, A. Hovsepyan, W. Joosen, and Y. Berbers. Towards a transformation chain modeling language. In *Embedded Computer Systems: Architectures, Modeling, and Simulation – 6th International Workshop, SAMOS 2006, Samos, Greece*, LNCS, pages 39–48. Springer Berlin / Heidelberg, 2006.

[165] B. Vanhooff, S. Van Baelen, W. Joosen, and Y. Berbers. Traceability as input for model transformations. In *Third ECMDA traceability workshop (ECMDA-TW) 2007 proceedings*, pages 37–46. SINTEF, June 2007.

[166] D. Varró, G. Varró, and A. Pataricza. Designing the automatic transformation of visual languages. *Science of Computer Programming*, 44(2):205 – 227, 2002.

[167] A. Vignaga and M. C. Bastarrica. Verification of Megamodel Manipulations Involving Weaving Models. Technical Report TR/DCC-2009-9, Universidad de Chile, Departamento de Ciencias de la Computaciòn, October 2009.

[168] T. Vogel, **A. Seibel**, and H. Giese. Toward Megamodels at Runtime. In N. Bencomo, G. Blair, F. Fleurey, and C. Jeanneret, editors, *Proc. of the 5th International Workshop on Models@run.time at the 13th IEEE/ACM International Conference on Model Driven Engineering Languages and Systems (MoDELS 2010), Oslo, Norway*, volume 641 of *CEUR Workshop Proc.*, pages 13–24. CEUR-WS.org, October 2010. (best paper).

[169] T. Vogel, **A. Seibel**, and H. Giese. The Role of Models and Megamodels at Runtime. In J. Dingel and A. Solberg, editors, *Models in Software Engineering, Workshops and Symposia at MODELS 2010, Oslo, Norway, October 3-8, 2010, Reports and Revised Selected Papers*, volume 6627 of *Lecture Notes in Computer Science (LNCS)*, pages 224–238. Springer-Verlag, May 2011.

[170] D. Wagelaar. Blackbox composition of model transformations using domain-specific modelling languages. In A. Kleppe, editor, *First European Workshop on Composition of Model Transformations, CMT 2006*, pages 15–19, Enschede, 2006. Centre for Telematics and Information Technology, University of Twente.

[171] S. Walderhaug, U. Johansen, E. Stav, and J. Aagedal. Towards a Generic Solution for Traceability in MDD. In T. Neple, J. Oldevik, and J. Aagedal, editors, *ECMDA-TW'06: ECMDA Traceability Workshop, Bilbao (Spain)*. SINTEF, 2006.

[172] E. Willink. Umlx: A graphical transformation language for mda. In *Proc. of the 18th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, Anaheim, CA*, pages 13—24, 2003.

[173] M. Wimmer, G. Kappel, A. Kusel, W. Retschitzegger, J. Schoenboeck, and W. Schwinger. Surviving the heterogeneity jungle with composite mapping operators. In L. Tratt and M. Gogolla, editors, *Theory and Practice of Model Transformations*, volume 6142 of *Lecture Notes in Computer Science*, pages 260–275. Springer Berlin / Heidelberg, 2010.

[174] S. Winkler and J. von Pilgrim. A survey of traceability in requirements engineering and model-driven development. *Software and Systems Modeling*, 9:529–565, 2010. 10.1007/s10270-009-0145-0.

[175] Z. Xing and E. Stroulia. Umldiff: an algorithm for object-oriented design differencing. In *Proc. of the 20th IEEE/ACM international Conference on Automated software engineering*, ASE '05, pages 54–65, New York, NY, USA, 2005. ACM.

[176] E. Yourdon and L. L. Constantine. *Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1st edition, 1979.

[177] A. Zündorf. Story driven modeling: a practical guide to model driven software development. In *ICSE'05: Proc. of the 27th international conference on Software engineering*, pages 714–715, New York, NY, USA, 2005. ACM Press.

List of Figures

Listings

Selected Publications

[149] **A. Seibel**. *From Software Traceability to Global Model Management and Back Again.* In 15th European Conference on Software Maintenance and Reengineering (CSMR'11), Doctoral Symposium, Oldenburg, Germany, 3 2011.

In this publication a coarse overview about the approach that has been shown in this thesis is given. It further shows that traceability and (global) model management are closely related and that a model management approach can be employed on top of a traceability approach.

[152] **A. Seibel**, S. Neumann, and H. Giese. *Dynamic hierarchical mega models: comprehensive traceability and its efficient maintenance.* Software and Systems Modeling, 9:493-528, September 2010.

This publication describes an approach that can be considered as a less detailed but nearly complete version of this thesis. The publication tends to focus on the traceability aspect by introducing the localization concept. But, it also introduces the concept of execution and a raw description of that concept. In that early version, the execution only supported the execution of coarse-grained model operations that are defined in data-flow compositions.

[150] **A. Seibel**, R. Hebig, and H. Giese. *Software and Systems Traceability, chapter Traceability in Model-Driven Engineering: Efficient and Scalable Traceability Maintenance.* Springer London, 2012.

This publication is an extension of the publication [152] by providing an improved traceability approach based on the notion of dynamic hierarchical megamodels. It already described a batch and an incremental localization. However, it does not focus on execution because of the traceability context.

[151] **A. Seibel**, R. Hebig, S. Neumann, and H. Giese. *A dedicated language for context composition and execution of true black-box model transformations.* In 4th International Conference on Software Language Engineering (SLE 2011), Braga, Portugal, 7 2011.

This publication is an extension of the publication [152]. It primarily focuses on the execution concept. In that publication the specification of context and data-flow compositions of heterogeneous model operations has been introduced. Furthermore, it is shown how those compositions can be automatically executed in a correct order. Nevertheless, that approach does not support as complex context compositions as shown in this thesis.

[74] R. Hebig, **A. Seibel** and H. Giese. *On the Unification of Megamodels.* In Proc. of the 4th International Workshop on Multi Paradigm Modeling (MPM'10) at the 13th IEEE/ACM International Conference on Model Driven Engineering Languages and Systems (MoDELS 2010), Oslo, Norway, 10 2010.

This publication provides a survey of recent model management approaches that use the notion of megamodels. It further provide a definition of the core idea of a megamodel that is the consensus of all considered megamodel notions.

[169] T. Vogel, **A. Seibel**, and H. Giese. *The Role of Models and Megamodels at Runtime.* In Juergen Dingel and Arnor Solberg, ed., Models in Software Engineering, Workshops and Symposia at MODELS 2010, Oslo, Norway, October 3-8, 2010, Reports and Revised Selected Papers, vol. 6627 of Lecture Notes in Computer Science (LNCS), pages 224-238. Springer-Verlag, 5 2011.

This publication proposes and discusses the idea of megamodels at runtime. It explains categories of models that can be employed to manage the configuration of a running software system. Based on these categories of models it is motivated that megamodels can be employed to manage a collection of configuration models at runtime. Furthermore, simple examples are shown by means of using megamodels.

[168] T. Vogel, **A. Seibel**, and H. Giese. *Toward Megamodels at Runtime*. In Nelly Bencomo and Gordon Blair and Franck Fleurey and Cedric Jeanneret, ed., Proc. of the 5th International Workshop on Models@run.time at the 13th IEEE/ACM International Conference on Model Driven Engineering Languages and Systems (MoDELS 2010), Oslo, Norway, volume 641 of CEUR Workshop Proc., pages 13-24, 10 2010. CEUR-WS.org. (best paper)

This publication is a preceding version of the publication [169].

[121] S. Neumann and **A. Seibel**. *Toward Mega Models for Maintaining Timing Properties of Automotive Systems*. In Proc. of the 3rd International Workshop on Model-Based Architecting and Construction of Embedded Systems (ACES-MB 2010) at the 13th IEEE/ACM International Conference on Model Driven Engineering Languages and Systems (MoDELS 2010), Oslo, Norway, 10 2010.

This publication outlines a scenario that uses megamodels to support developing automotive software systems with AUTOSAR. It is specifically shown for a scenario that requires the automated maintenance of timing properties.

[64] H. Giese, S. Hildebrandt, and **A. Seibel**. *Feature Report: Modeling and Interpreting EMF-based Story Diagrams*. In Proc. of the 7th International Fujaba Days, 2009.

This publication introduces an implementation of Story diagrams into the context of Eclipse and EMF. It presents the graphical editing of Story diagrams as well as a first version of an interpreter for Story diagrams in EMF.

[65] H. Giese, S. Hildebrandt, and **A. Seibel**. *Improved Flexibility and Scalability by Interpreting Story Diagrams*. In Tiziana Magaria and Julia Padberg and Gabriele Taentzer, ed., Proc. of the Eighth International Workshop on Graph Transformation and Visual Modeling Techniques (GT-VMT 2009), volume 18 2009. Electronic Communications of the EASST.

This publication shows an update of [64]. It provides new features and analyses scalability concerns of the interpreter.