Towards a multilevel model transformation engine

Leif Arne R. Johnsen

Master's thesis in Software Engineering at

Department of Computing, Mathematics and Physics,
Bergen University College

Department of Informatics,
University of Bergen

September 2017

Abstract

Domain specific modelling languages (DSML) are usually defined through fixed level meta modelling tools such as EMF. While this is sufficient for defining languages that has no overlap with other languages, the approach struggles to reuse overlapping parts of an existing language when defining a new language, especially when it comes to the definition of behaviour. Many domain specific languages have a significant overlap with eachother in terms of concepts and behaviour. Multilevel meta modelling is a promising approach to define a family of DSMLs.

In this thesis, we aim to define behaviour once on a higher level of abstraction, and reuse it on every DSML which share that behaviour. We use multilevel coupled transformations (MCMT) to define the behaviour, and we present a multilevel transformation engine capable of transforming MCMTs into traditional two-level rules which can be run by existing transformation engines.

Acknowledgements

My supervisor Adrian, thank you for being so patient and providing great guidance throughout this thesis. This work would not have been possible without your supervision.
Fernando, thanks for helping me understand multilevel matching and for the invaluable help with the algorithms.
My grandparents Eva and Sjur, thank you for inviting me over for a proper dinner once a week, and helping me out of though situations.
BSI football team, thank you for giving me something to look forward to at the end of day.
My best friend Tomas, thank you for all the long nights out we have experienced together.
Thanks to my family for being so patient and supportive during my time as a student, my parents Marit and Harald, and my brother Tommy.

Contents

List of figures

1. Introduction

In the beginning of software engineering, programs were written directly in machine code. Writing programs in machine code was a tedious and error prone process, and higher-level languages were developed to shield the developers from writing programs directly in machine code. Higher-level languages such as Fortran [1] are compiled into executable machine code automatically by a compiler, and work like an interface between the developer and the machine. Raising the abstraction level from machine code to fortran resulted in a huge swing in productivity.
Programming languages from different paradigms were created in an attempt to maximize software quality and minimize the cost of software [2]. Today the most popular programming languages follow the object-oriented paradigm and are refered to as object-oriented programming (OOP) languages [3]. OOP focuses on the data/model rather than the algorithms, and allows developers to create a class to represent similar concepts. Many objects can be instantiated from a class, and these objects inherit the attributes and methods defined in their class. This allows functionality to be reused among concepts, and changing behaviour is made easier as a change only needs to be made in the class and then the change is automatically reflected in the objects. Furthermore, a class can be reused from other OOP programs outside of the program it was defined. The ability to reuse classes in other applications has lead developers to create code libraries intended to be reused by many programs.

A software platform contains a collection of code libraries and is meant to provide the developers with base functionality when developing new applications. However, software platforms are becoming increasingly complex and difficult to use. Developers spend years mastering the libraries of a platform, and are usually only familiar with a subset of the functionality in the platform. Moreover, the complexity of the platform forces the developers to pay such close attention to the implementation details that they tunnel in on a specific part of the application and are unable to view the application as a whole. This leads to the developers creating duplicate code as they are unaware of the existing functionality in the application. Furthermore, it makes it hard for the developers to know which parts of the system are affected when making changes to a specific part of the system [4].

Model-driven software engineering (MDSE) is a software development discipline that emerged as an approach to tackle the ever-increasing complexity of software systems. MDSE use models as the central artifact in the development process. When changes occur, they are first made in the model, and then the other artifacts are updated to reflect these changes. MDSE considers models to be at a higher abstraction level than code, and code can be produced from models which allows developers to focus on concepts from the problem domain rather than the underlying algorithms and the solution domain.
Many studies support that the use of MDSE leads to an increase in productivity, quality, performance and more [5, 6]. However, the main advantage of MDSE is communication between all stakeholders in the project even those without a background in computer science. After all, poor communication between developers and domain experts is one of the main reasons that software projects fail [7].

Unfortunately, many people commonly associate MDSE with the Unified Modeling Language (UML) [8] which is a general purpose modeling language (GPML). UML was created as a means to model object oriented systems. However, stakeholders without a background in computer science struggle to understand models written in UML [9], and UML models are therefore not the optimal tool for the communication between developers and domain experts.

Instead of using general purpose languages, MDSE suggests creating domain specific modeling languages (DSML) through the use of metamodeling [10]. DSMLs are tailored to a specific domain and are developed with the involved stakeholders in mind. DSMLs use concepts directly from the domain and a syntax that is intuitive to the stakeholders which allow the stakeholders to understand domain models without additional learning effort. Furthermore, DSLs encapsulate all of the needed functionality within the language, so the developers only have to learn the language of the domain instead of searching through multiple libraries to find the needed functionality. On the other hand, code libraries usually contain many functions that the developer will never use in a specific domain, and at the same time not enough functions to cover the problems in the developers' domain. As a consequence, developers need to combine several libraries to cover the problems in their domain. Developers benefit from having all the needed functionality gathered in one place, as they do not have to search through several libraries and end up confused in the process which is often the case when functionality is scattered across many code libraries. Furthermore, DSLs has proven to be far more effective than GPLs when solving problems that they were designed to solve [11]. HTML and SQL are examples of how good DSLs can be although they are languages that solve problems of a specific technical space rather than a specific domain in the real world. However, DSLs of a specific domain are atleast equally effective as DSLs of a specific technical space, and the main difference between the two is that DSLs of technical spaces can be used to solve a broader specter of problems.

Businesses from most domains are moving on to the computer world because of digitalization therefore creating a DSL for each domain could benefit these businesses greatly. However, the process of creating a new DSL is time consuming and involves creating new editors, simulators and transformations which usually needs to be created from scratch. Furthermore, existing DSLs are difficult to change, and making changes to a language usually results in breaking the tools that supports that language such as editors, simulators etc [12]. The introduction of language workbenches has allieviated many of these issues, but currently they do not solve all of them.

An important observation is that there is a lot of overlap between some DSLs, and sometimes the difference between them is hardly noticeable [12]. In other words, some DSLs contains many of the same concepts, and these concepts usually have similar behaviour in all of these DSLs. We want to take advantage of this observation by creating a language at a higher abstraction level that contains many of the common concepts. We can then use the higher-level language as a base to create new DSLs. Moreover, we can define behaviour for the concepts in the higher-level language and let the DSLs that are created from the higher-level language inherit this behaviour. That way many DSLs can be created from the higher-level language, and we only need to define the common behaviour once.

In this thesis, we use the multilevel metamodeling tool MultEcore [13], to create a hierarchy of domain specific modeling languages. The hierarchy contains an arbitrary number of levels where every level is an instance of the levels above, and the highest-level language is defined at the topmost level in the hierarchy. Our objective is to create a tool that allows developers to define behaviour on types at a higher level, and apply this behaviour on direct and indirect instances of these types at a lower level.

2. Problem description

2.1. Problem statement

In the real world, similar concepts are grouped together under a common concept which in turn can be grouped under a more abstract concept. The result is a hierarchy of “things” which can be expanded by adding new abstraction levels. In other words, the hierarchy of the concepts in the real world has an unlimited number of abstraction levels which keeps growing as new concepts are identified. In this chapter, we look at how modelling approaches use a hierarchy with a fixed number of levels, and how that can cause problems when trying to mimic the hierarchy of the real world. Furthermore, we will see that behaviour is multilevel and can be expressed through model transformations. However, most model transformation tools are fixed-level and the model transformation tools which are multilevel have problems with precision which we will explain in chapter 4. We propose a solution to this problem and present two research questions that help determine the usefulness of our solution.

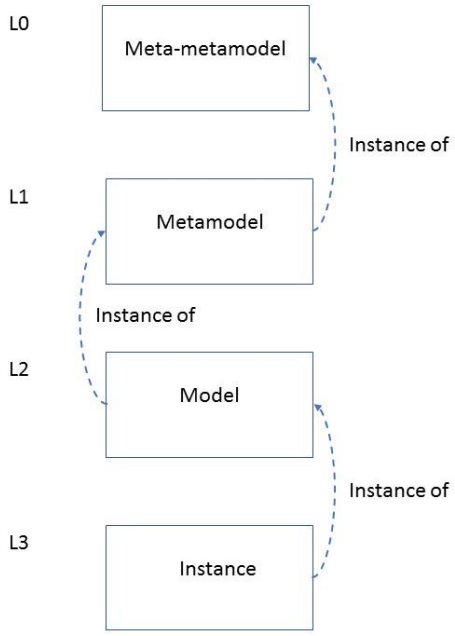

FIGURE 2.1 - LEVELS IN FIXED METAMODELING

Figure 2.1 shows the the structure of a fixed level modelling hierarchy. The meta-metamodel is the topmost level in the structure and is used to construct metamodels on the level below. We usually use MOF or Ecore as the meta-metamodel. The meta-metamodel contains generic concepts such as class and their relations to other classes. The meta-metamodel is fixed in the sense that it cannot be modified easily. The metamodel is used to construct models at the level below. Behaviour can be defined for model elements through model transformations. The model transformations are defined on the metamodel so that they can be applied to several models created from the metamodel. Here we only give a brief overview of the metamodeling structure in order to add some context to the problem description. We revisit this topic in the next chapter where we explain metamodeling in

depth.

The fixed-level approach works fine when only one metamodel is considered. The problem appears when we have an existing metamodel and are in the process of creating a new metamodel which have many concepts in common with our existing metamodel.

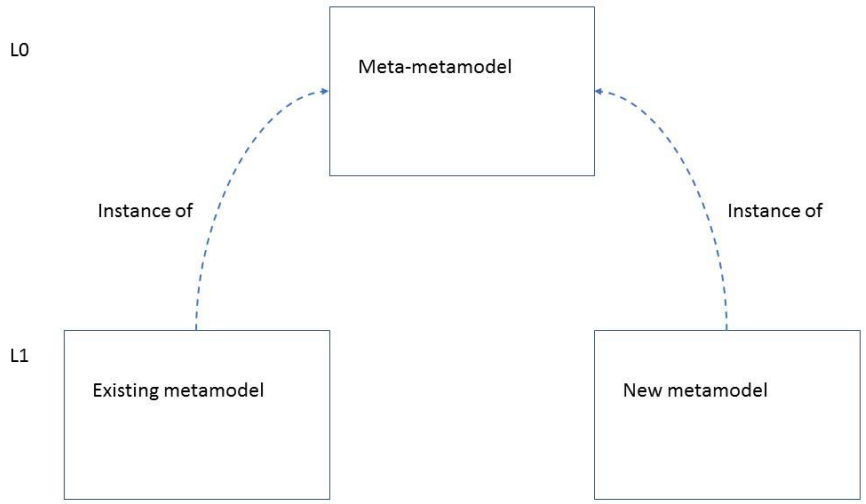

FIGURE 2.2 - PROBLEM SITUATION

Figure 2.2 shows the situation where problems might occur. The existing metamodel and the new metamodel are defined on the same level which means that they have no knowledge of eachother, and both metamodels can only use concepts that are defined in the meta-metamodel. Moreover, the behaviour that is defined for the existing metamodel has to be recreated for the new metamodel even though it might be identical. One might ask, what if the common concepts in the existing metamodel were moved up to the meta-metamodel? That could help reuse concepts in new languages, but the developer has no straightforward way to access the meta-metamodel which is considered off limits because it was not ment to be modified by developers. Furthermore, modifying the meta-metamodel can break other artifacts and metamodels that are dependent on the meta-metamodel.

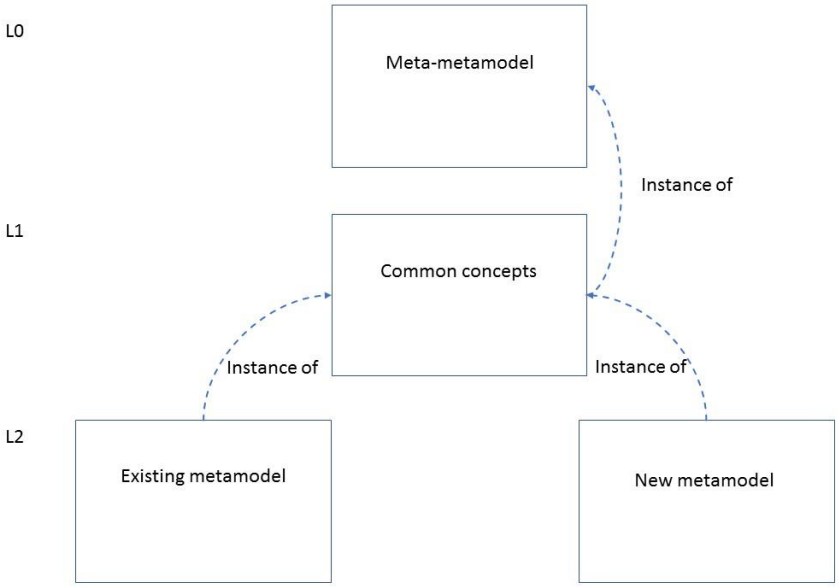

FIGURE 2.3 IDEAL SOLUTION

Figure 2.3 shows how we would like to solve the problem of reuse. The common concepts are defined on a new level between the meta-metamodel and the existing metamodel. That way the common concepts can be used in both the existing metamodel and facilitate the creation of new metamodels. Furthermore, the behaviour that is common to the existing metamodel and the new metamodel can be defined on the common concepts once and be applied to instances of both metamodels. However, inserting a new level in the hierarchy is not possible due to the limitation of fixed-level modelling because introducing a new meta-level would take up the space used by the models and their instances. We can avoid the limitations imposed by fixed-level modelling by turning to multilevel modelling. Multilevel modelling supports the solution displayed in Figure 2.3. Unfortunately, defining the behaviour of multilevel models is an issue due to the lack of multilevel transformation tools.
We aim to tackle this issue by developing a multilevel transformation engine which can transform multilevel models, and we aim to facilitate the re-use of behaviour between metamodels.

2.2. Research questions

A concrete example of the problem regarding re-usability between metamodels is given here.
The concepts **dog** and **cat** share the behavior **eat**. Currently, the behaviour **eat** needs to be defined separately for **cat** and **dog** even though the rules might look identical besides having different types. The reason is that current tools define behaviour which is tied to a specific type hence if a behaviour is defined for **dog** it is not applicable to **cat**. One way to re-use the behaviour is to define a new concept named **animal**. Make **cat** and **dog** inherit from **animal**, then define the behaviour eat on **animal**. Then the behaviour **eat** would be applicable to both **cat** and **dog**. The set back of this approach is that it requires **animal**, **dog** and **cat** to be defined in the same metamodel. This approach has some drawbacks: **Animal** is more abstract than **dog** and **cat**, and should be in a different layer of abstraction. Furthermore, it does not scale well because it requires all types to be in the same metamodel, and adding a new concrete animal like cow requires that the metamodel be updated along with the artifacts that are created for the metamodel such as editors, code-generators etc.

Frequently used concepts will get mixed with rarely used ones, leading to the pollution of the metamodel.
Fortunately, using multilevel modelling we can follow a similar approach to re-using behaviour that allows us to separate animal from dog and cat. The idea is similar to the inheritance approach, but instead of inheritance we use instance relations to determine which concepts are eligible for a given behaviour. We make **dog** and **cat** instances of **animal**, and asssign the behaviour **eat** to **animal** (See Figure 2.4). The former approach could be viewed as generalisation and the second approach as classification. The author of [14] compares the two approaches and marks that classification offers greater flexibility, but comes at the cost of precision and sanity checks.

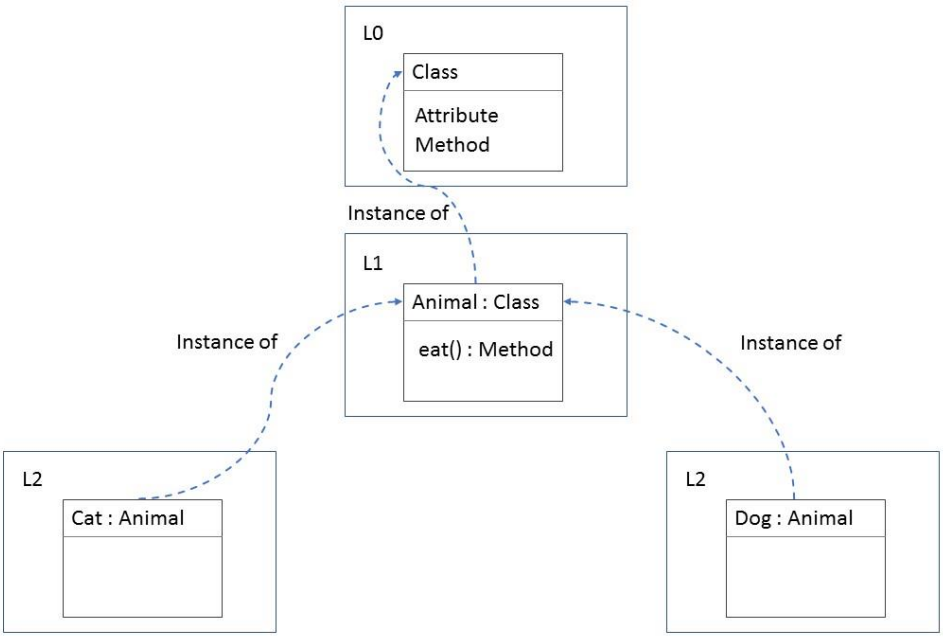

FIGURE 2.4 - SIMPLIFIED MODELS OF OUR SOLUTION

In order to execute the behaviour and determine which concepts can perform the behaviour, we need a multilevel execution engine because eating must be applicable to instances of dog and cat which are at level 3, but fixed-level engines are not able to recognize that instances of dog and cat are also instances of animal. Assuming, we have built a multilevel model transformation capable of running and matching the behaviour:

1. Can we re-use behaviour across different model languages?

2. Can we re-use behaviour defined for a higher-level language on refined lower-level versions of that language?

An example of the first question: We have an existing language which includes the concept of Dog, and we are creating a new language which includes the concept of Cat. We notice that some of the behaviour that is defined on Dog is the same as some of the behaviour that we are about to define on Cat. The behaviour is specifically defined on Dog as we follow the “you ain’t gonna need it” principle therefore we create a new concept Animal at a higher abstraction layer than Dog and Cat, and move the common behaviour from Dog up one level to Animal. We make Dog and Cat an

instance of Animal. Now the question asks if we can apply the behaviour on both Dog and Cat when Animal is defined in a separate metamodel from Cat and Dog.

The second question asks if we can apply the behaviour defined on Animal onto indirect instances of Animal. For example, dogs have many different breeds and some breeds have a specific behaviour that is different from other breeds, although all breeds have some behaviour in common. Now the question is: can we apply the behaviour defined on Animal onto some instances of Breed?

3. Metamodelling

In the introduction, we mentioned modeling languages. In this chapter, we look at the core components of a modelling language. We mainly discuss metamodeling which is a technique to define the abstract syntax of a modelling language. We discuss two approaches to metamodeling, the fixed-level approach and deep multilevel approach. We briefly introduce the concrete syntax and semantics of a modelling language. We end the chapter by displaying an example from Rossini [2] about modelling a component based web applications using the fixed-level and the multilevel approach to metamodeling. Lastly, we compare the two approaches.

3.1. Modeling language

A modelling language can be broken down to three core components: The abstract syntax, the concrete syntax, and the semantics [15]. The concrete syntax is not mandatory, but it is used to differentiate between the different concepts in the language aswell as making the concepts easier to understand.

The abstract syntax identifies the relevant concepts used to construct models in a particular domain. It does not distinguish between the appearance of the concepts and all concepts usually have the same notation. Since all the concepts have the same appearance, one can only distuingish between concepts by looking at the concepts name. This makes models written in the abstract syntax hard to read, and therefor they are not well suited for communication purposes.

However, the abstract syntax can be represented by a concrete syntax. Each concept in the abstract syntax can be mapped to a more concrete concept with a unique representation. A concrete syntax can be textual or graphical, and an abstract syntax can be mapped to a graphical concrete syntax and a textual concrete syntax. In fact, there are no limits to the number of concrete syntaxes that can be mapped to an abstract syntax. The point of the concrete syntax is to make the abstract syntax more intuitive and easier to understand, and that is dependent on the person who is reading the model. Therefor, many concrete syntaxes can be defined for a given abstract syntax and a reader can choose the concrete syntax that is most appealing to them.

5 + 6	Infix
(+ 5 6)	Prefix
(5 6 +)	Postix

TABLE 3-1 THREE CONCRETE SYNTAXES FOR THE SUM EXPRESSION

Table 3-1 shows three different syntaxes for expressing the same expression. All of the syntaxes express the same thing, and the meaning of each syntax should therefor only be defined once. The meaning is therefor only specified for the abstract syntax and is derived from the concrete syntax.

The idea of concrete syntax is to convey information as fast as possible, and therefor the graphical syntax is usually the best option as a picture says more than a 1000 word.

Graphical concrete syntaxes can be created using Sirius [16], and textual concrete syntaxes can be created using Xtext [17]. Sirius allows a concepts appearance to change depending on it's attributes. For example, the concept of a person may be displayed as a male or female depending on the

persons gender, and a child or an adult depending on the persons age which helps the reader gather more information quickly.

The last component of a language is the semantics. The semantics specifies what a concept means and what should happen when a combination of concepts appear together. In modelling, one of the ways to specify the semantics of a language is through model transformations [18]. Formal semantics are necessary to make the concepts in the language unambiguous meaning that there are no room for miss-interpretation.

Figure 3.1 gives an overview of the components in a DSML.

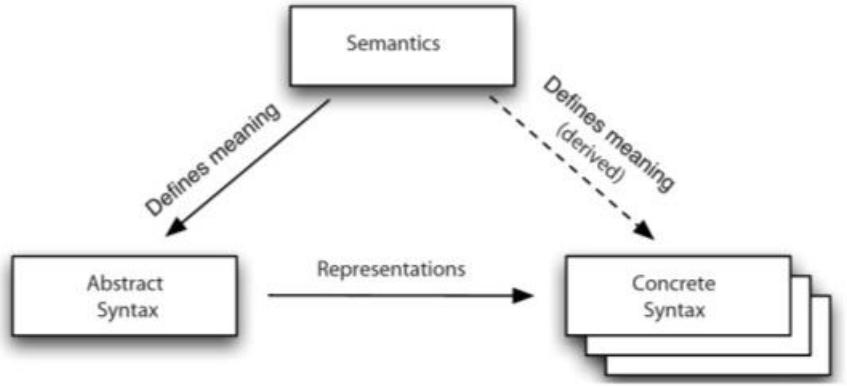

FIGURE 3.1 - ANATOMY OF A MODELING LANGUAGE [15]

We discuss metamodeling in the next section as a technique to define the abstract syntax of the language, and model transformation in the next chapter as a technique to specify the semantics of the language. We do not talk more about the concrete syntax as it is not relevant for this thesis. Furthermore, semantics can be divided into static and dynamic semantics. Static semantics refers to the structural properties of the model and dynamic semantics refers to the behaviour of the model. In this thesis, we use the words dynamic semantics and semantics interchangeably.

3.2. Metamodelling

In modeling languages, the abstract syntax is described by a metamodel. The process of creating metamodels is refered to as Metamodeling. A metamodel is a model that describes elements that can be used to create new models, additionally it provides a set of rules and constraints that dictates how elements are related to each other. The elements in the metamodel are referred to as types. Models can be produced by creating instances of the types in the metamodel hence the elements in the model are referred to as instances.

The metamodel dictates the set of valid models. For a model to be valid, it must conform to it's metamodel. A model is said to conform to it's metamodel if it is both typed by and follows the constraints placed on it's metamodel. A model is typed by it's metamodel if every element in the model is an instance of a type defined in the model's metamodel. Types in a metamodel may have cardinality and uniqueness constraints, referered to as structural constraints. Additional constraints may also be placed on the meta-model by using a constraint language such as OCL, these constraints

are referred to as attached constraints. A model is only considered valid if both structural and attached constraints are satisfied [19].

In MDSE, the abstract syntax of a language is described by a metamodel [15]. Models and metamodels are created in a hierarchical structure which often span many levels. We refer to the hierarchy as the model hierarchy, and a level in the model hierarchy as a metalevel. The most abstract metamodel is placed at the top of the model hierarchy. The relationship between two adjacent metalevels is that the upper metalevel acts as a metamodel, and the metalevel below acts as the model. The topmost metalevel is an exception to this rule, as it is reflexive meaning that it is defined on its own terms (conforms to itself). The pattern of model and metamodel can continue infinitely, and therefor the number of metalevels in the model hierarchy is arbitrary and can change over time.

However, tradionally the model hierarchy has been using a fixed size of three or four metalevels. A modeling hierarchy of fixed length means that it is not possible to insert new levels to account for changing requirements [20]. Additionally, the real world does not have a fixed number of abstraction levels, and using a fixed level hierarchy to represent the abstraction levels of the real world is sometimes not sufficient and requires complex workarounds.

Atkinson and Kühne has proposed deep metamodeling which is an approach that uses an arbitrary number of metalevels [21] and allows information to be carried across more than one level. The number of levels may change over time, depending on the requirements. A characteristic of deep metamodeling is that a model doesn't necessarily only conform to the metamodel on the level directly above itself, but may conform to several of the metamodels above. In addition, deep metamodeling does not have a fixed limitation on the number of metalevels in the model hierarchy.

In [20] the authors show an example of how accidental complexity may be introduced when restricted to only two metalevels, and how this could be avoided by introducing more metalevels. We will include the same example later in this chapter, but first we show the OMG's four-layer approach as an example of a fixed level approach, and MultEcore as an example of a deep metamodeling approach.

3.2.1. Object Management Group's (OMG) four-layer approach

The OMG defines an architecture with four levels. The top-most level is the meta object facility (MOF). The MOF was created by the OMG with a purpose to provide a type system to the CORBA architecture, and a set of interfaces through those types could be created, viewed, edited or deleted [22]. OMG's four-layer hierarchy is one of the most commonly used structures in practise. The eclipse modelling framework (EMF) adopts this construct, but utilises only three levels.

The MOF which is the meta-metamodel, is used to create metamodels. The metamodel works as a type system for a set of models, and every element in the model must be an instance of a type defined in the metamodel.

Figure 3.2 gives an overview of the OMG's four level architecture. We have until now used the opposite numbering of the levels in the hierarchy, and the reason will become clear later. The topmost metamodel is located at level **M3**, and is used to construct the modelling language on **M2**. In this case, **M2** is a model of **M3**, and every element in **M2** is an instance of a type defined in **M3**. **M3** is the metamodel of **M2**, and contains all the types that can be used to construct a language in

M2. In the same way, **M1** is a model of **M2**, and **M2** now acts as a metamodel to **M1**. One can see that **M2** is both a model and a metamodel. In the fixed-level architecture the whole modelling language is defined in the same level **M2**. Unfortunately, using only one level to describe a modelling language makes it difficult to re-use segments of an existing modelling language when creating a new modelling language. Attempts in reusing parts of an existing language in a fixed level environment often results in a complex relation between types and instances.

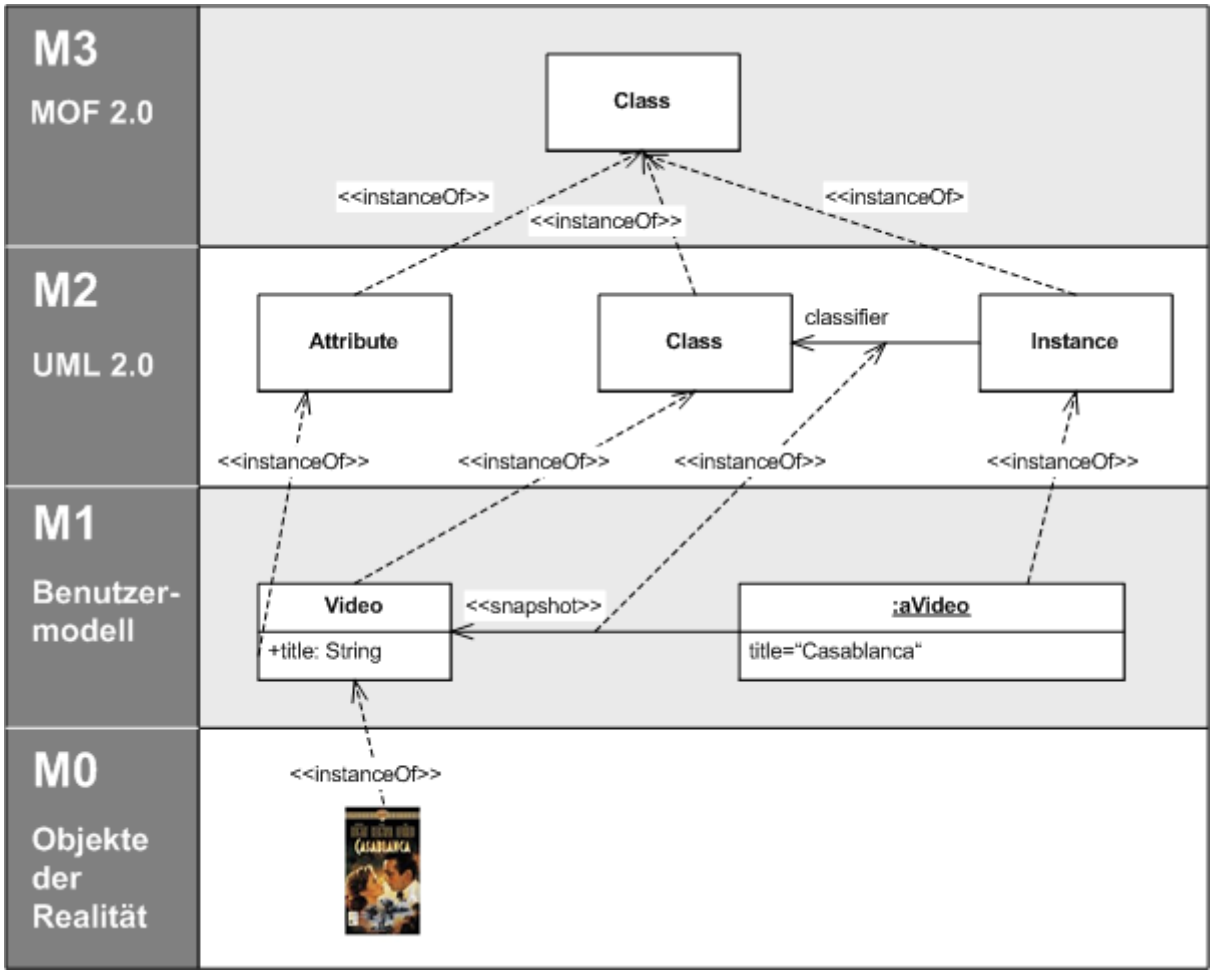

FIGURE 3.2 - OMG'S FOUR LAYER ARCHITECTURE [22]

3.2.2. Deep meta-modelling (DMM)

A consequence of the fixed metamodelings lack of flexibility is that people often start from scratch when creating a new language to avoid the added complexities of re-using existing languages. Starting from scratch is not ideal, as many domains uses similar concepts which could be re-used when creating new languages. The need for domain specific meta-modelling languages (DSMM) is discussed in [23]. Domain specific metamodeling languages are languages which contain concepts that are used in several domains, and they can be used as a starting point when creating a new language. DSMMs are created using many levels, and each level represents a language which is a refined version of the levels above. The topmost levels in the hierarchy contains the most abstract concepts which are applicable to the most domains. The process is the definition of a family of

languages, starting out with very general elements. The languages get more refined and specific on each level. The more general a language is, the easier it is to re-use. On the other hand, more specific languages are more beneficial, when they can be re-used. One can choose which level in the hierarchy to use as a starting point for creating a new language. However, two metalevels are not adequate to support the use of DSMMs [23]. We have turned to multilevel modelling which do not have a limitation on the number of metalevels in the hierarchy.

Multilevel modelling tools follow a similar pattern to the fixed level architecture, but the depth of the hierarchy may change over time as a result of levels being inserted or deleted from the hierarchy. Another difference between fixed and multilevel modelling tools is the mechanisms for instantiating types in a model. In fixed level modelling, shallow instantiation is commonly used. Shallow instantiation only considers two levels at a time, where one level is the model and the other is the metamodel. These two levels must be adjacent, and the model can only create instances of types which are defined in the metamodel. With shallow instantiation, it is not possible to instantiate a type two levels below the metalevel where the type was defined. Unless a copy of the type is instantiated on every level between where the type was defined and the level where the type is instantiated. Creating a copy of a type on every level is refered to as the replication of concepts problem. Shallow instantiation works fine with the fixed two-level approach, but does not scale well when used in a multilevel environment [24].

The preferred instantiation mechanism for multilevel modelling tools is deep instantiation. Deep instantiation is an instantiation mechanism where the system is aware of multiple modelling levels, and allows the user to create instances of types defined on more than one level above the current level. When deep instantiation is supported the metamodel is not restricted to only the level adjacently above the current level, but instead the metamodel is the union of all the levels above the current level. When we talk about fixed-level modelling tools we always assume shallow instantiation, and although some multilevel modelling tools only support shallow instatiation we wil assume that multilevel modelling tools support deep instantiation throughout this thesis.

One way to implement deep instantiation is through potency. Potency is an integer that is assigned to each concept/model element on every level in the hierarchy. The value assigned dictates how many levels below a concept may be instantiated. The potency value of an element is reduced by one everytime the element is instantiated.

In the four-layer approach, the top most level was labelled M3. When using multilevel modelling it makes more sense to let the top most level be M0 because it is not likely that another level is inserted on top of the top most level. If M0 is not used as the top most level then adding a new model at the bottom of the hierarchy would require that every model level in the stack is incremented by 1. This is also most likely the case therefore labelling the top most level as M0 makes the most sense, although traditionally the bottom most level has been labelled as M0. With deep metamodelling, a model at a given level does not always conform only to the level directly above itself. It can conform to several or even every level above. Deep meta-modelling offers greater flexibility, as it allows insertion of a new level inbetween meta-levels to cope with changing requirements.

Figure 3.3 shows an example of a multilevel model hierarchy with five levels. The top most level and the bottom most levels are not shown. The top most level is the ecore, and the bottom most level is the state of the model. The types are depicted as blue elipses, and the name of the instance is

depicted in the yellow square. The first level shown is named **generic_plant** and contains abstract concepts identified in product line systems. **Machine** is responsible for creating, assembling or de-assembling items which are represented by **Part**. **Machine** has a **creates** reference to **Part** which is necessary to specifcy which **Part** it should generate. **Container** is used to store items produced by the **Machine** or to store items that the **Machine** requires producing additional items. The two cases is recognized by the **out** and **in** relation respectively.

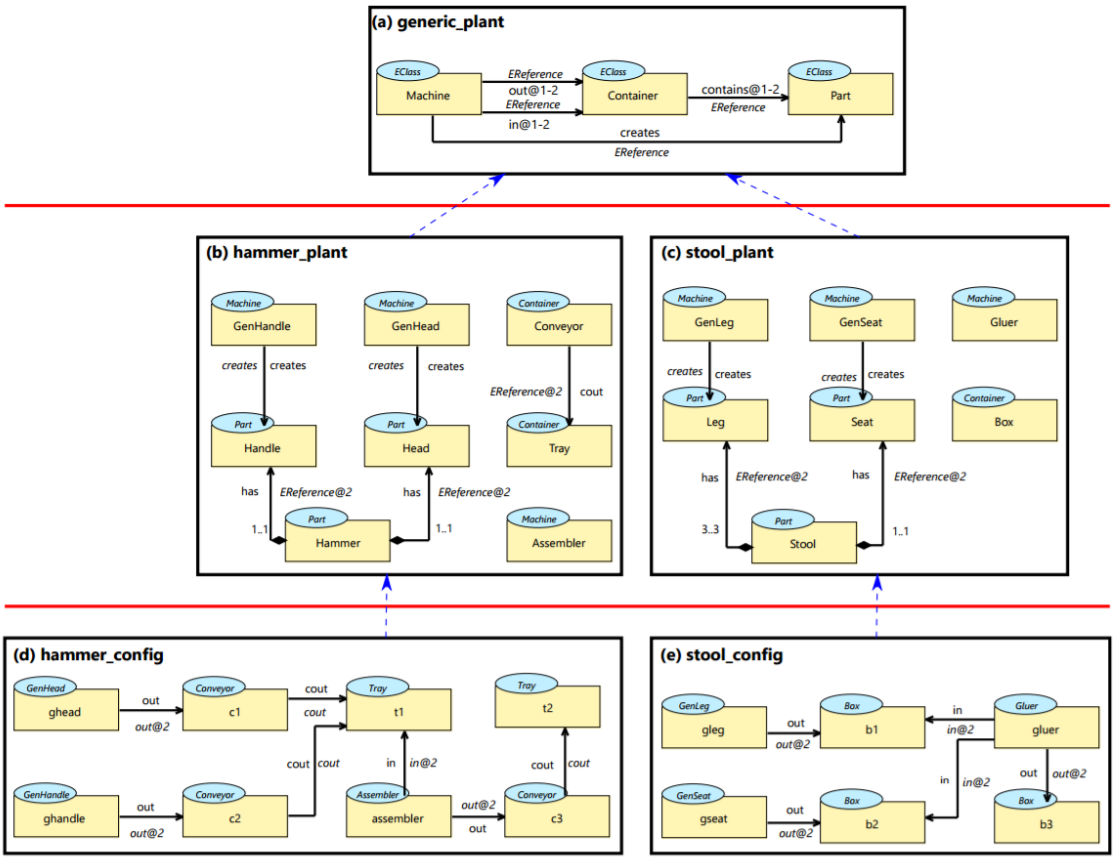

FIGURE 3.3 – DEEP METAMODELLING HIERARCHY EXAMPLE [25]

An arbitrary number of new levels may be created from **generic_plant**. In this example, two instances are created from **generic_plant** namely **hammer_plant** and **stool_plant** they are located on the second level in the hierarchy. These plants are refined versions of **generic_plant** and contains concepts for a specific plant. In **hammer_plant** there are three instances of **Machine**, **GenHandle** which generates **Handle**, **GenHead**, which generates **Head,** and **Assembler** which combines **Head** and **Handle** to produce a **Hammer**. There are two instances of **Container** namely, **Conveyor** and **Tray**. **Conveyor** is responsible for transporting a **Part** from a **Machine** to a **Tray**, and from a **Tray** to a **Machine**. **Tray** is responsible for storing items generated by the **Machine**. A new relation **cout** has been defined between **Tray** and **Conveyor**. The **cout** relation has not been defined in **generic_plant** which is possible because **hammer_plant** also conforms to **ecore**. Three instances of **Part** are defined, **Head**, **Handle**, and **Hammer** all mentioned previously. **Hammer** has a one to one ***has*** relation to both **Head** and **Handle** indicating that a **Head** and a **Handle** is required to create a **Hammer**. The **stool_plant** is similar to **hammer_plant** and will not be discussed much further here, other then it uses **Box** as the only transporter and **Gluer** has a similar role to the **Assembler** in **hammer_plant**.

The two models on the third level are **hammer_config** and **stool_config** which is the configuration of **hammer_plant** and **stool_plant** respectively. In the configurations, more components can be added to increase the production rate. Notice that the **out** relation between **ghead** and **c1** in **hammer_config** is not defined for **GenHead** and **Conveyor** in **hammer_plant**, but rather defined for **Machine** and **Container** in **generic_plant**. This is possible because **hammer_config** does not only conform to **hammer_plant** and **ecore**, but it conforms to **generic_plant** aswell.

The number of levels in the hierarchy is not fixed, and we may at any time create a new level anywhere in the hierarchy. This means that it is possible to create a model at the level above generic_plant or between generic_plant and hammer_plant.

3.2.3. Deep metamodeling vs fixed metamodeling

Earlier in this chapter we stated that the fixed-level approach can lead to accidental complexity, here we give an example of this. The example and the figures are taken from [2], they provide a more thorough explanation than we do here.

A DSL for developing component-based web applications is defined using metamodeling. Two approaches to metamodeling is used. First the DSL is created using the fixed-level approach then another version of the DSL is created using deep metamodeling. Both approaches must deal with the challenge of extending the DSL.

Figure 3.4 displays the language defined using a metamodel and a model from the OMG's 4-layer architecture. In the metamodel the following concepts are defined: ***Component*** represents component types and has an identifier to distinguish between component types. ***CInstance*** represents instances of component types having a name and a flag which indicates whether the instance should be displayed or hidden. ***Datalink*** is used to relate component types to eachother, ***dlinstance*** is an instance of ***datalink*** and is used to relate component instances to eachother. ***Type*** is a relation between ***Component*** and ***CInstance***, it specifies which component a **cinstance** is an instance of. The metamodel is used model many component-based web apps. The model included here is of a web app that shows the position of the professors' offices on a map. The model contains the concepts: ***Table*** and ***Map*** which are instances of Component, ***geopos*** is an instance of ***datalink*** and relates ***Table*** and ***Map*** together. ***UAMProfs*** and ***UAMCamp*** are instances of ***CInstance*** and they represent component instances of ***Table*** and ***Map*** respectively. The relation **offices** is an instance of **dlinstance** and specifies the relation between ***UAMProfs*** and ***UAMCamp***. The relations **profstype** and **camptype** are instances of the relation type in the metamodel, **profstype** specifies that ***UAMProfs*** is typed by ***Table*** and **camptype** specifies that ***UAMCamp*** is typed by **Map**.

The type-object relation between component types and instances is defined explicitly by the ***type*** relation in the metamodel. However, the type-object relation between ***datalink*** and ***dlinstance*** is implicit as no relation between ***datalink*** and ***dlinstance*** is specified in the metamodel. This may cause problems as we can not control that offices is an instance of **geopos**, and this instance relation could become ambigious if more relations between ***Table*** and ***Map*** are defined. Relations may be created between ***UAMProfs*** and ***UAMCamp*** even if no relation between ***Table*** and ***Map*** is defined. A reflexive relation from ***UAMProfs*** to ***UAMProfs*** could be defined even though table does not have a reflexive datalink relation. A constraint language such as OCL could be used to detect these errors by defining attached constraints on the elements in the metamodel. However, this may not be enough

to guide the correct instantiation of each datalink. Furthermore, defining the OCL constraints requires more work, and would still not work as good as a built-in type system would if the datalink types and instances would be defined on two separate meta-levels [2]. In addition, the built-in type system would not require any additional work, in constrast to the OCL approach.

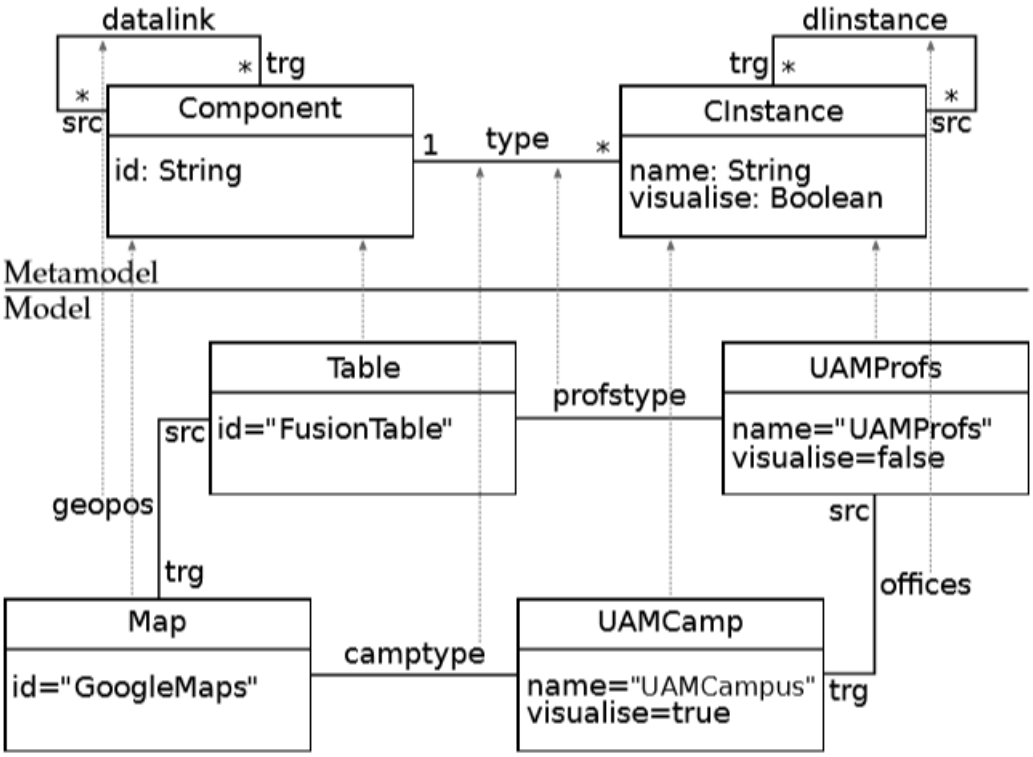

FIGURE 3.4 - A FIXED-LEVEL DSL FOR COMPONENT BASED WEB APPLICATIONS

In the complete version of the language, component types may have features. These features need to be correctly instantiated in the instance of the component type. This addition makes the models even more complex, Figure 3.5 displays the complete DSL. ***Feature*** and ***Slot*** are added to the metamodel. ***Feature*** represents the attribute of a component type, and ***Slot*** represents the value of a ***Feature***. In the model, the feature ***Scroll*** is added to the component type ***Map*** and it represents the ability to zoom in on the map. Defining the class ***UAMScroll*** and relating it to ***Scroll*** and ***UAMCamp*** has to be done manually. Furthermore, the system does not know that the the value of ***UAMSCroll*** needs to be a ***Boolean***, and this check needs to be done manually. A tool that emulates the existence of two meta-levels within the same level must be built to automatically perform conformance tests and relate instances to their apropriate types otherwise the language quickly becomes too complex to use/maintain [2].

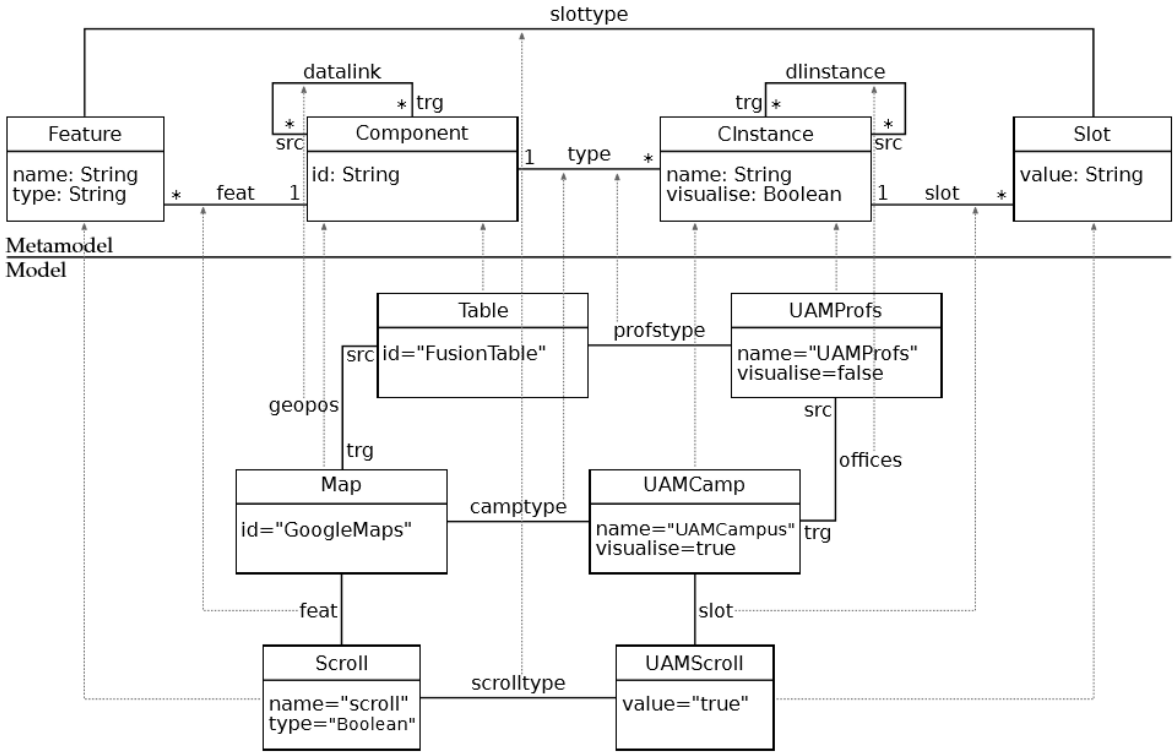

FIGURE 3.5 - EXTENDED VERSION OF THE FIXED-LEVEL DSL, ADDING FEATURES TO COMPONENT

Now we show a solution which uses deep metamodeling with three levels, and how this yields a much simpler DSL with the benefits of built in conformance check between types and instances.

The solution of the initial DSL without the added features is shown in Figure 3.6. The topmost level M1 contains: ***Component*** which has the attributes ***id***, ***name*** and ***visualise***. ***Datalink*** which is the relation between components. Note that the @ sign denotes potency which we mentioned earlier in this chapter. The middle level M2 contains: ***Map*** and ***Table*** which are instances of ***Component***, ***geopos*** which is an instance of ***datalink*** and it relates ***Table*** and ***Map***. Lastly the bottom most model M3 contains: ***UAMCamp*** which is an instance of ***Map***, ***UAMProfs*** which is an instance of ***Table*** and offices which is an instance of geopos and relates ***UAMProfs*** to ***UAMCamp***.

The DSL in Figure 3.6 uses fewer concepts than the DSL in Figure 3.4. Furthermore, the type-instance relation between ***geopos*** and ***offices*** is unambigious and the system recognizes the conformance. The concept of component instance is not needed as the system recognizes that ***UAMCamp*** and ***UAMProfs*** are indirect instances of Component.

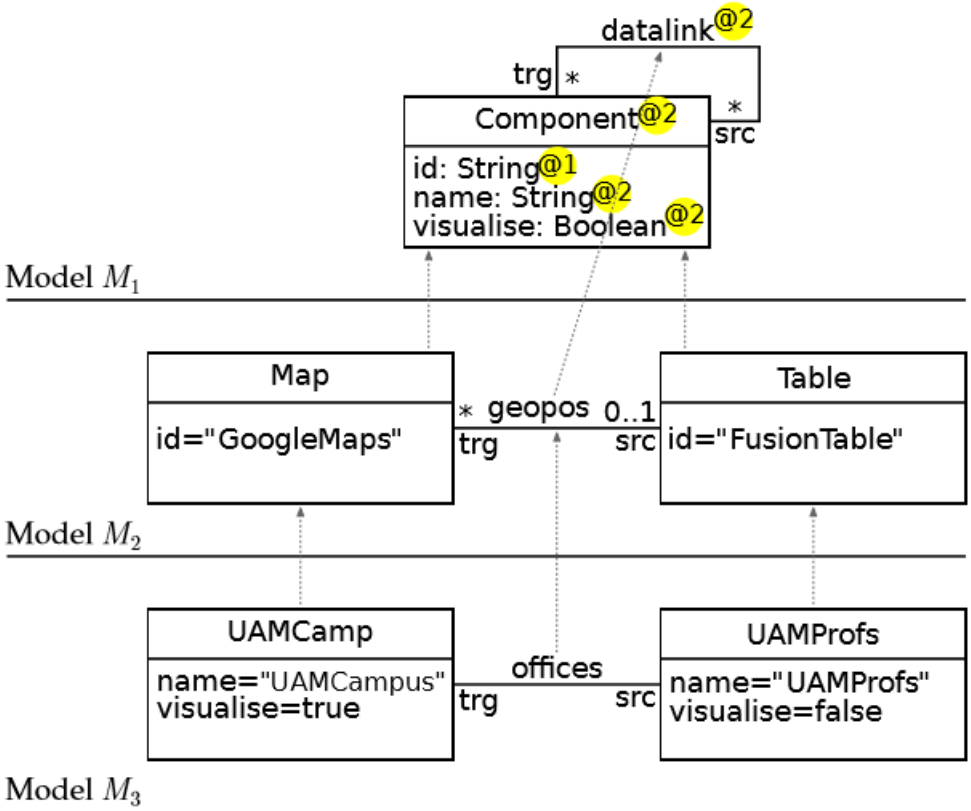

FIGURE 3.6 - A DSL FOR COMPONENT BASED APPLICATIONS, CREATED USING DEEP MULTILEVEL MODELING

Figure 3.7 displays how features can be added to ***Component*** when using deep metamodeling. Like before the ***scroll*** feature is to be added to ***Map***. The problem is solved easily by linguistic extension which allows the user to add new elements; to instances; that was not defined in their types metamodel. In Figure 3.7, ***scroll*** is not defined for ***Component***, and is directly added to ***Map*** which is an instance of ***Component***. It's allowed because ***Map*** is an instance of ***Class*** in addition to an instance of ***Component***, therefor any native datatype that can be added to class, may also be added to ***Map***. This is very useful because it allows us to refine instances at a lower level with details that could not be foreseen when designing the high-level language. In addition to the hierarchy being much simpler than the one in Figure 3.5, this solution also has two other advantages over the fixed-level solution: Firstly, the linguistic extension allows the system to check that the value of ***scroll*** is a Boolean by performing an automatic conformance check. Secondly, when instantiating an object all its attributes are also instantiated by the system. With the fixed-level approach this had to be done either manually or a tool had to be created that emulated 2 levels within the same level. However, such a tool would be coupled to a specific metamodel and a new tool would have to be developed when creating a new DSL, in contrast the deep approach does this for free.

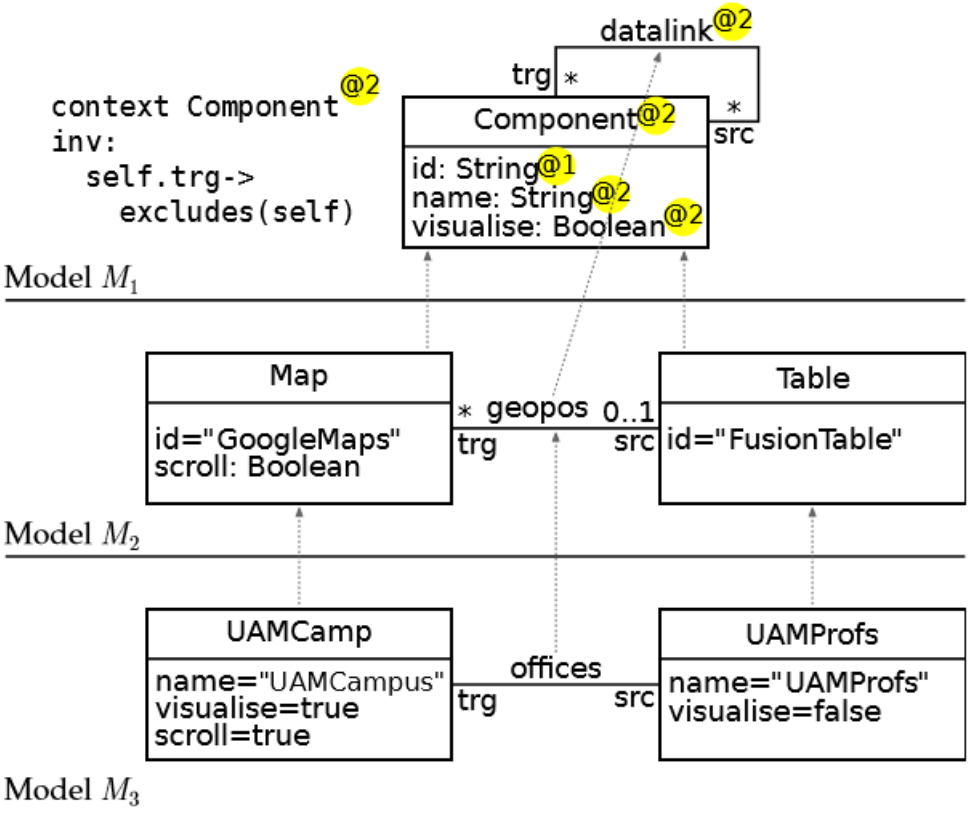

FIGURE 3.7 - EXTENDING THE MULTILEVEL DSL, ADDING FEATURES TO COMPONENT

3.2.4. Example of re-use challenge in Fixed-level modeling

In the previous example, we showed that deep metamodeling produced simpler models and were more resilient to change than fixed-level metamodeling. However, the example only included the case where a single application were modeled therefore we could not compare the re-usability between the two approaches. We illustrate this in the next example by describing the PLS for hammer generation using fixed-level metamodeling.

Figure 3.8 displays a 2-level metamodel of the **hammer_plant** in Figure 3.3. The metamodel combines **generic_plant** and **hammer_plant** from Figure 3.3. The instance relations are modeled as inheritance. The metamodel works fine when only **hammer_plant** needs to be described.

However, the approach fail to reuse **generic_plant** when developing languages for other production line systems such as **stool_plant**.

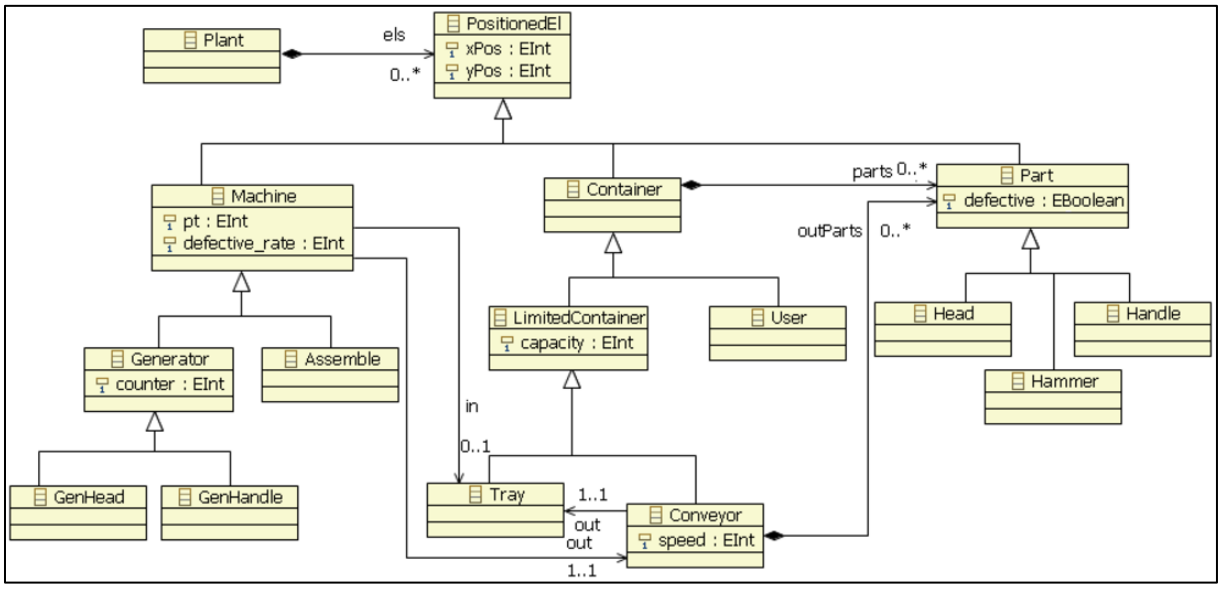

FIGURE 3.8 - HAMMER PRODUCTION DESCRIBED BY A 2-LEVEL METAMODEL [26]

The pattern in Figure 3.9 needs to be created from scratch when defining a fixed-level metamodel for **stool_plant** and many other production line systems. Moreover, changes to **generic_plant** has to be made in each metamodel because there is no correspondence between the implementations of **generic_plant.**

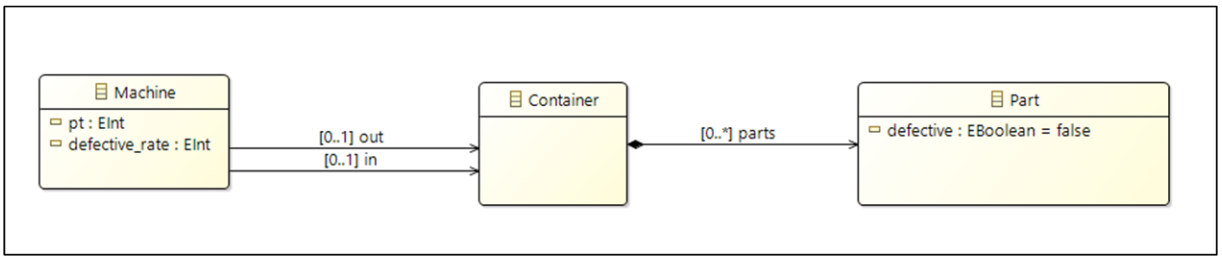

FIGURE 3.9 - REPEATED PATTERN IN PRODUCTION LINE SYSTEMS

4. Model transformations

In the previous chapter, we introduced metamodeling as a way to define the abstract syntax of a modelling language. Furthermore, we introduced two approaches to create metamodels, namely the fixed-level approach and the multi-level approach. We provided an example of each approach and we argued that the fixed-level approach was not adequate to support a domain specific metamodeling framework. Moreover, we showed an example where the fixed-level approach led to accidental complexity, and the multi-level approach yielded much simpler metamodels.

In this chapter, we discuss how model transformations can be used to define the dynamic semantics of a modelling language. We explain what model transformations are and the different approaches to performing model transformations. We look at different classifications of model transformations, and the differences between transformation approaches to fixed-level models and multilevel models. Lastly, we discuss the theory of graph transformations because that is the approach we used for our transformation engine.

4.1. Model transformations in general

Model transformations have many applications in MDE and is considered one of the key techniques. They can be used to refine models by adding details to higher-level models. They can also be used to translate a model written in one language to the corresponding model written in another language. For example, a UML model can be transformed into an ER model. Model transformations have many more applications in MDE. In this thesis, we use model transformations as a means to execute models by altering their states.

The following is a general definition of model transformation given by [27]:
A transformation is the automatic generation of a target model from a source model, according to a transformation definition. A transformation definition is a set of transformation rules that together describe how a model in the source language can be transformed into a model in the target language.

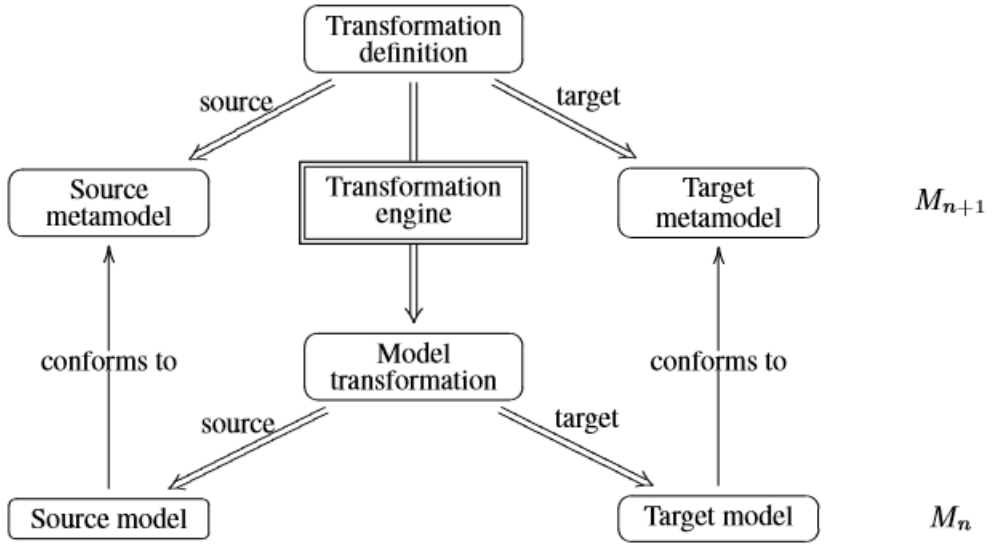

FIGURE 4.1 - MODEL TRANSFORMATION OVERVIEW [19]

In a model transformation, a target model is produced from a source model. It is not necessarily a one to one relation (although it's most common). Several source models could be combined into a target model, or a source model could be split into several target models. A transformation definition defines how a source model should be transformed into a target model. The transformation definition contains a set of transformation rules. A transformation rule contains a source pattern and a target pattern. The source pattern is referred to as the left-hand side (LHS) and the target pattern is referred to as the right-hand side (RHS). The target pattern is produced if a match of the source pattern is found. A pattern is a set of elements (could also be the empty set) and their relations. The transformation definition is written in a transformation language. The model transformations are defined on the types, and executed on instances of these types (Figure 4.1). Therefore, a link to the source and targets metamodel is necessary so that the transformation engine can know about the types of the models to be transformed, and thereby verify that the resulting target models are valid.

Model transformations are performed in one of two ways: Either the target model could be the result of modifying the input model or the target model could be created from scratch. The former case is refered to as in-place transformations and it means that the input and the output model are the same model. The latter case is refered to as out-place transformations. Furthermore, model transformations are either endogenous or exogenous. Endogenous transformations are transformations where the input and output model conform to the same metamodel. Exogenous transformations on the other hand are transformations where the input and output model conform to different metamodels.

Like programming languages, model transformations follow a paradigm. Declarative, imperative and graph transformations are the most commonly used paradigms in model transformations. Declarative languages focus on what should be transformed from what without specifying how it should be transformed. The order of execution is neglected, and an output pattern is generated from an input pattern. The style is similar to that of a logical programming language such as prolog. Imperative languages focus on how and when a model should be transformed. The order of execution is important and changing the sequence that the commands are executed in might change the result of the transformation. Imperative languages ressembles object oriented languages such as

Java, and might be easier for a developer to learn and use than declarative languages. Graph transformation languages use nodes to represent concepts, and edges between nodes to express relations between concepts.

As with metamodeling tools, model transformation tools can be limited to transform fixed-level models or they can support the transformations of multilevel models. Since fixed-level modelling tools have been around the longest, most model transformation tools only support fixed-level model transformations. In the next sections, we discuss the two approaches fixed and multilevel transformations, and we will see that both approaches are similar in many aspects.

4.2. Model transformation with fixed meta levels

We have discussed fixed-level metamodelling in the previous chapter. Fixed-level model transformations work on models that are defined by such metamodeling frameworks. Most fixed-level transformation tools are defined for the standard 4-level architecture. We will use atlas transformation language (ATL) as an example of a fixed-level transformation tool. ATL is a hybrid language which means that it supports both the declarative and imperative style of programming. The creators of ATL encourages the declarative style, but the imperative part is included to deal with more complex problems [28]. ATL performs out-place transformations by default, but supports in-place transformations by using the refining keyword.

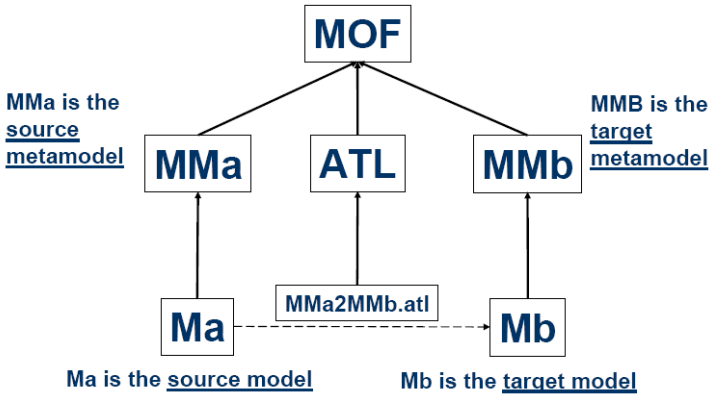

FIGURE 4.2 –THE ATL RULE DEFINITION OVERVIEW [28]

In ATL it is necessary to specify the source and target metamodel, so that the transformation engine may ensure the validity of the models. This also enriches the editor with code suggestions and error detection, along with other useful features. Figure 4.2 illustrates an example of the ATL transformation context. The **MOF** is on level 0, **MMa** is on level 1 and **Ma** is on level 2. Adjacent elements are on the same level. There is a source and a target model, **Ma** and **Mb** respectively. **MMa** and **MMb** contains all the types that the source and target model could use respectively. **MMa**, **MMb**, **Ma**, and **MMa2MMb.atl** must be defined before running a transformation. **Mb** is generated by running the transformation and does not exist until the transformation is performed.

```
rule GenerateHead {
    from
        gh : Hammer_Plant!GenHead
    to
        h : Hammer_Plant!Head(
            ),
        genhe : Hammer_Plant!GenHead(
            creates <- h
        )
}
```

FIGURE 4.3 SAMPLE ATL RULE, GENERATING A HEAD FROM HEAD GENERATOR

Figure 4.3 illustrates a transformation rule in ATL. The rule matches every **GenHead** instances of **hammer_plant** and creates a new head object that is connected to the matched **GenHead** instance. The **from** and **to** keyword refers to the left-hand side and the right-hand side of the rule respectively. Variable definition is done in the **from** and **to** part of the rule. It follows the pattern of $variable : Metamodel!Type$. The instances to be matched are specified on the left-hand side of the rule, and these instances are read only meaning that they can not be modified. The matched instances are only used to set the values of the created instances in the right-hand side of the rule. A matched instance is deleted after a transformation has been performed, and it is necessary to make a copy of the matched instance in the right-hand side of the rule if one does not wish to delete it. In the figure, a copy of **GenHead** is made to preserve the matching instance **gh**. Moreover, an instance **h** of **Head** is created and assigned as the target of the **creates** releation where **genhe** is the source.

On the right-hand side of the rule, a set of bindings are defined after the variable declaration. A binding connects a feature of the target model with a feature of the source model. The bindings are specified using the **<-** operator which means that the feature on the left side of the arrow is initialized by the expression on the right side of the arrow.

Notice the variable declaration in ATL, they follow as mentioned the pattern $Metamodel!Type$ where the element to be transformed must be a direct instance of the given type which means that a transformation is tied to a specific type of a specific metamodel.

This example uses ATL, however most fixed-level transformation languages follow a similar structure. For example, Figure 4.2 could be said to be an instance of Figure 4.1 and many other transformation languages follow the pattern in Figure 4.1.

4.3. Multilevel model transformations

Multilevel transformations are defined on multilevel model hierarchies. They can be used to define and share behaviour that is common to a group of concepts by defining the behaviour on an abstract representation of the group. The abstract concept would be defined on an upper level in the

hierarchy, and the behaviour could be applied at an arbitrary level below which could be the level directly below or several levels below.

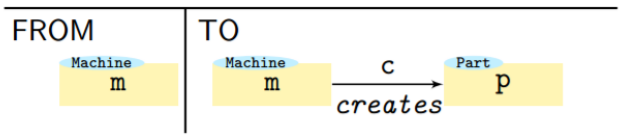

FIGURE 4.4 - MULTILEVEL MODEL TRANSFORMATION RULE: CREATE PART

For example, Figure 4.4 displays how a create part rule is defined for **Machine** and **Part** in **generic_plant** (Figure 3.3). The rule can be applied on **GenHead** and **GenHandle** in the **hammer_plant** and **GenLeg** and **GenSeat** in the **stool_plant**, see Figure 3.3(b) and Figure 3.3(c). Multilevel transformations rules work fine when the model to be transformed contains the pattern that is required by the rule and the types used by the rule are all located in the same metalevel [25]. Otherwise the rules become too generic and imprecise. For instance, machine instances would create all parts. **GenHead** would create **Hammer** and **Handle**. **Assembler** which is not supposed to create any parts would create **Handle**, **Head** etc...

The approach could be viewed as a way to loosen up the strictness of types imposed by fixed level model transformations [25]. However, with it's side effects and limitations, we deem it unfit for our purpose. Instead we turn to multilevel coupled transformations which is another form of multilevel transformations that aim to accomplish the same thing as multilevel transformations without the side effects.

4.4. Multilevel coupled model transformations (MCMT)

MCMT [25] is an approach to model transformations that transforms the model together with its metamodel. The metamodel here is not necessarily one metalevel, it may be several ones. The metamodel consist of all the metalevels above the model to be transformed, and could be thought of as the union of the metalevels above, granted that the potency allows it. The MCMT rules share a similar construct to fixed level rules, but MCMT rules have a meta block in adition to the source and target block. The meta block allows one to put constraints on the types and to define a pattern that the types must satisfy to be eligible for a transformation. The meta block dictates over the types in the source block and in the target block, and one could say that it works as a type system when writing the rules.

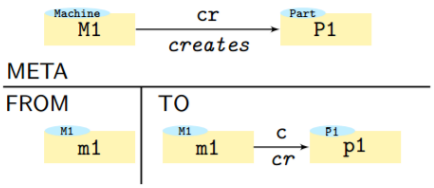

FIGURE 4.5 - MCMT RULE: CREATE PART

We use the same example from earlier namely create part. This time the rule is defined using MCMT, Figure 4.5 shows what the MCMT version look like. The metablock contains a variable **M1** of type **Machine**, a variable **P1** of type **Part** and a relation of type **creates** between the two. The metablock

indicates that a valid match of **M1** needs to have a relation of type **creates** to an instance of **Part**. If **M1** is matched first then **P1** needs to be the exact same type that is related to the match of **M1** by the creates relation. For example, if **M1** is matched to **GenHead**, then GenHeads **creates** relation is looked up and the target of this relation is **Head** and therefor **P1** must be assigned to **Head**. This makes sure that create part rules are only produced for instances of machine that has a creates relation to an instance of part. Thereby rules are created that generate the correct part for the right machine, and rules are not produced for machines that are not supposed to create parts such as assembler. As mentioned before the metablock does not require all its types to be in the same metalevel and each type could in theory be defined in a separate level. That elimates one of the limitations of the multilevel transformations mentioned earlier. The other one being related to case distinction, which is tackled by defining a pattern in the meta block that the types must satisfy. This means that it is possible to perform multilevel transformations without the limitations and side effects mentioned earlier.

While the fixed-level approach was too strict, and the multilevel approach too loose, MCMT could be considered to be somewhere inbetween those two approaches.

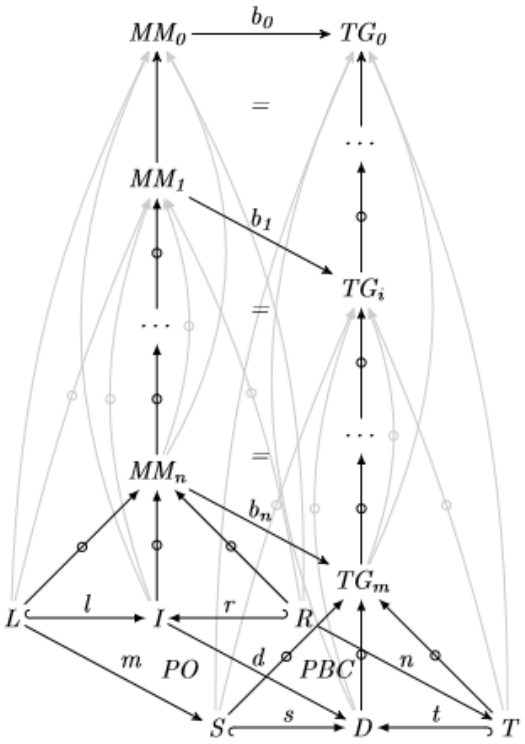

FIGURE 4.6 – MULTILEVEL COUPLED MODEL TRANSFORMATIONS [29]

Figure 4.6 gives an overview of the elements involved in an MCMT, and how they are structured. The structure could be viewed as two flat trees **M** and **TG** where each node in the trees represent a level in a hierarchy. The tree on the left side **MM** represents the typing hierarchy defined in the meta part of the rule. **TG**, the tree on the right side represents the multi level model hierarchy where the actual types are defined. The nodes in **MM** contains variables which are only partly typed meaning their concrete types are not known before the nodes in **MM** are mapped to nodes in **TG**. Nodes in **MM** is bound to nodes in **TG** by the relation **b**. For example, $\mathbf{MM_0}$ is bound to $\mathbf{TG_0}$ by the relation $\mathbf{b_0}$ in the figure. All levels in **MM** is required to be bound to a level in **TG** for a rule to be valid. The bottom part of the left tree contains the left and right-hand side of the rule depicted in the figure as **L** and **R** respectively. The interface I contains the union of **L** and **R** which are the elements to be preserved.

The elements in **L**, **I** and **R** are all typed by concepts defined in the levels of **MM**. **S** represents the model we wish to transform and **T** is the resulting model of performing the MCMT to **S**.

4.5. Graph transformations

For the underlying transformation engine, we have used Groove which is a tool that performs transformations on Graphs. Graph transformation uses graphs to represent the model and the rules. Rules include a left-hand side and a right-hand side, LHS and RHS respectively [30]. The left-hand side of a rule must be matched to the host graph for the rule to be applicable. When a rule is applied, the left-hand side is matched in the host graph, and then this match is replaced by the right-hand side which results in the target graph, this step is referred to as graph-rewriting.

We use the algebraic approach to graph transformations. An approach that is based on pushout constructions. Pushouts specifies how the target graph is produced from applying a rule to the host graph. There are two ways to perform a pushout. Namely, the single-pushout (SPO) and the double-pushout (DPO). The double-pushout is the most used in practice [31]. However, both approaches have their unique advantages. We will briefly explain both approaches and then make a short comparison.

The double-pushout approach performs graph rewriting in two steps. In the first step elements are deleted and in the second step new elements are added. DPO has a condition called the gluing condition which must be satisfied for a transformation to be applicable. The gluing condition consists of two other conditions, namely the identification and the dangling edge condition. The identification condition specifies that elements to be deleted by the rule must be mapped injectively to the host graph. For example, if two pattern elements **u** and **v** should be deleted by a rule then **u** and **v** must not be mapped to the same element in the host graph. The dangling edge condition specifies that if a node **u** is deleted by a rule, then all edges adjacent to **u** must also be deleted.

Figure 4.7 illustrates how the target graph **H** is derived by applying a rule (**L,K,R**) to the host graph **G**. The figure contains the elements:

- **L** describes the elements that must be matched by the host graph, for a rule to be applicable
- **R** describes the elements that are added to the host graph when applying the rule
- **K** describes the elements that are in both **L** and **R**. All elements in **K** exists before and after the rule is applied
- **G** is the host graph
- **H** is the target graph
- Elements in **L** that are not in **K** is deleted from **G** when a rule is applied which results in the temporary graph **D**.
- **D** is a temporary graph that holds the result of the first pushout.
- Elements in **R** that are not in **K** is added to **D** when performing the second pushout which results in the target graph **H**
- After rule completion, **H** becomes the new host graph, instead of **G**

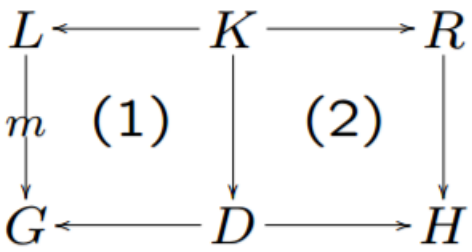

FIGURE 4.7 - THE DOUBLE-PUSHOUT APPROACH

Furthermore, here is a quick explaination of what happens during DPO. Firstly, a match of **L** needs to be found in **G**, let **m** be such a match. Secondly, if **m** statisfies the gluing condition, then the context graph $D \coloneqq G \backslash (m(L) \backslash m(K))$ is constructed by deleting the elements in **m** that is not in **K**. Lastly the target graph $H \coloneqq D \cup R \backslash K$ is constructed by adding the elements in **R** that is not in **K**.

Single-pushout performs graph rewriting in a single step, hence the name single-pushout. Figure 4.8 illustrates the process: Let **mL** be a match of **L** in **G**, then all elements in **mL** that are not in **R** is deleted from **G** and the elements in **R** is added to **G** to produce $H \coloneqq G \backslash (L \backslash R) \cup R \backslash L$

In other words: **L** is mapped to elements in **G** by the morphism **mL,** which must be complete for a rule to be applicable. Only the elements in **G** that are mapped by **L** is modified by the rule. Furthermore, **r** is a partial mapping between **L** and **R** that describes the application context. The application context is made up of elements preserved by the transformation, and it plays a similar part as the interface graph in DPO [32]. Let **v** be an element in **L** then there are two options; either **v** has a morphism in **r** that points to a corresponding element in **R** in which case **mL(v)** is preserved, otherwise **mL(v)** is deleted because it has no morphism in **r** to an element in **R**. The elements in **R** that are not mapped by an element in **L** need to be added to the graph. Finally, the addition and deletion are performed in the same step.

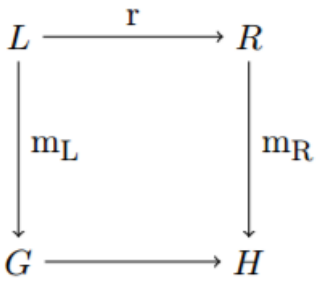

FIGURE 4.8 - THE SINGLE-PUSHOUT APPROACH [33]

There are two main differences between SPO and DPO:

1. SPO does rewriting in one step, meaning that the target graph is directly derived from the host graph.
2. SPO does not have a gluing condition which allows it to run rules that DPO would not therefore it is considered more powerful than DPO. However, it is also more dangerous than DPO because SPO does not have a gluing condition hence SPO transformations may invalidate the graph.

5. Design and implementation

This chapter is structured into several parts which starts by presenting our alternatives and explaining our design choice before introducing the structure of our framework. The framework is composed of several components. We describe the components of the framework in their own section. Lastly, we explain the functionalities of the framework and the responsibilies of each component.

5.1. Design options

We were faced with two options when implementing the multilevel transformation engine. The first option was to create an engine from scratch. An engine capable of interpreting multilevel models at run-time. This ment to implement many of the features which were already supported by existing transformation engines.

The second option was to re-use an existing transformation engine. Unfortunately, the existing engines available are not capable of running MCMT rules. It is therefor necessary to reduce the MCMT rules into 2-level rules. The process of reducing MCMT rules into 2-level rules are referred to as flattening of the rules. The flattening process will be discussed further later in this chapter, but for now we will mention that it happens in a preprocessing step. That is, we need to generate new rules from our MCMT rules. In that sense, this option resembles code-generation while the first option is similar to that of model interpretation.

We decided early on to go with the second option because the first option was presumably too much work for a master's thesis. Not much time was spent investigating the pros and cons of the two options, because of the workload of the first option. Furthermore, both options were capable of solving our goal. It is still worth to mention few of the many benefits of the first option. The approach is more elegant, no code generation is needed, and a component can be removed from the framework because no external engine is needed. The fact that the multilevel engine does not rely on an external engine makes the engine more flexible and gives the developer more control. These benefits and more lead us to believe that the first option should be implemented in the future, while the implementation in this thesis could work as a proof of concept.

5.2. Framework overview

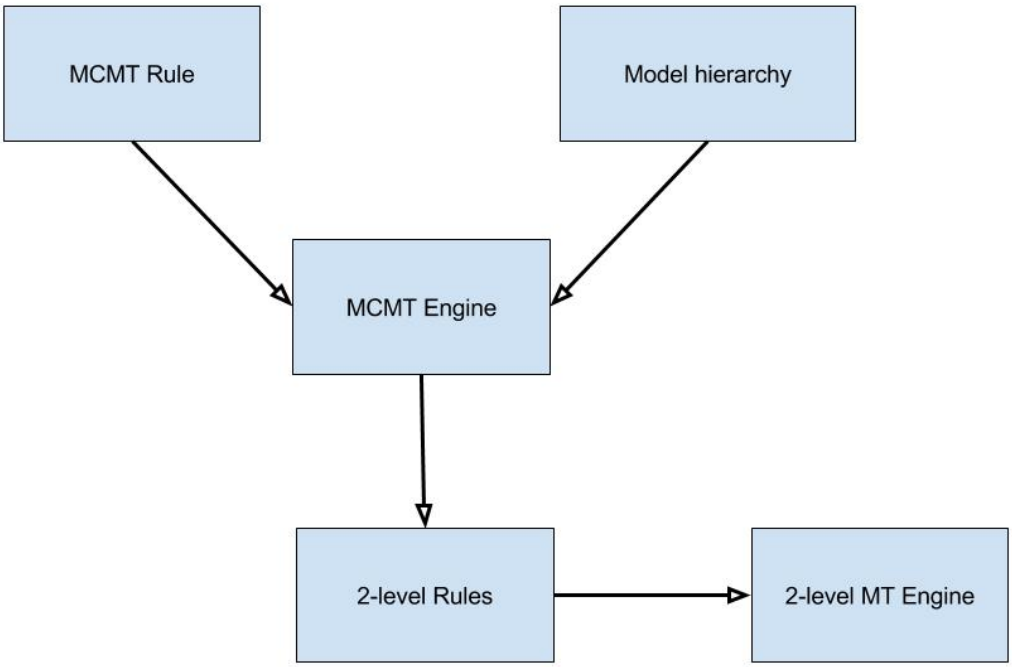

FIGURE 5.1 - OVERVIEW OF MCMT FRAMEWORK

We execute multilevel coupled model transformations through an MCMT framework. Figure 5.1 shows the components of our MCMT framework. The **MCMT Engine** is the central component and takes as input an **MCMT Rule** together with a **Model hierarchy**. The **MCMT Engine** matches the **MCMT Rule** with the **Model hierarchy** and generates a **2-level Rule** for each valid match. We describe each component in the framework by illustrating what they look like and explain what role they have in the matching and generation process. Furthermore, we explain how the matching and rule generation process is performed.

5.3. DSL for writing MCMT rules

The DSL for writing MCMT rules was created using Xtext [17] which is a framework for creating languages. The standward way of Xtext is to first define the concrete syntax, and then Xtext infers the abstract syntax from the concrete syntax. In other words, the user defines a concrete syntax in a file which is then used to construct an EMF metamodel. The constructed metamodel is used to generate a textual editor for the language. The textual editor provides features such as syntax highlighting, code completion, error detection and more.

Using EMF metamodels as the abstract syntax opens the possibilities of using graphical frameworks such as Sirius to create a graphical syntax for the language.

The purpose of the DSL is to facilitate the creation of MCMT rules by improving the structure of the syntax and making the syntax more concise. Thereby making the transformations easier to read, write and learn.

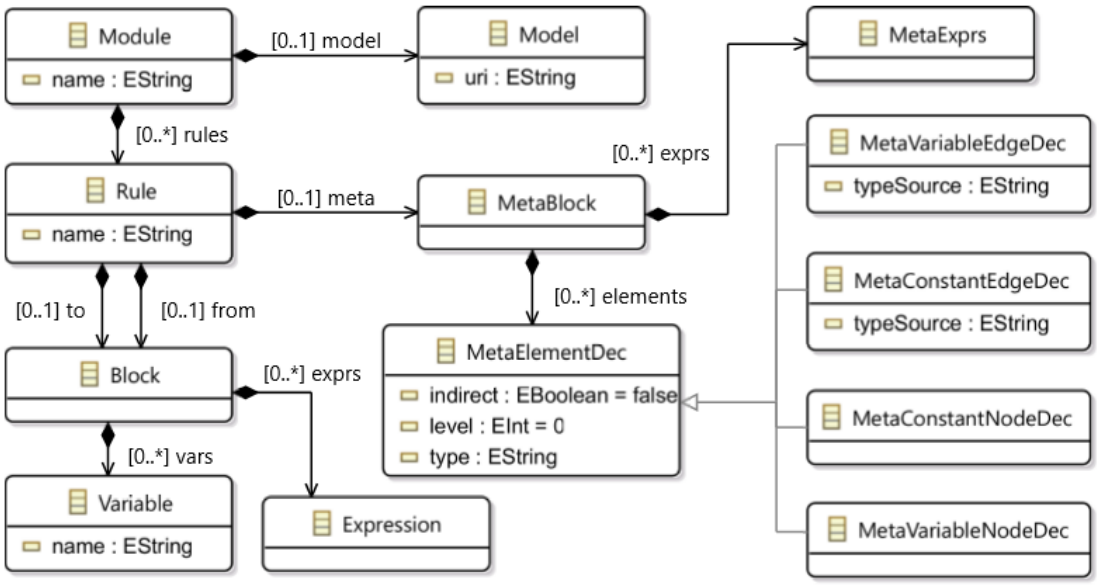

FIGURE 5.2 - FRAGMENT OF THE METAMODEL DEFINING THE ABSTRACT SYNTAX OF THE DSL [25]

Figure 5.3 illustrates how a rule is structured. A rule always starts with the keyword RULE followed by the name of the rule. It contains three blocks: META, FROM and TO, named in lowercase in the textual concrete syntax. The META block may not be empty and must contain a valid pattern. The FROM and TO blocks can only use variables of types which are defined in the META block, they may also be empty. In the METABLOCK there are MetaElementDecl and MetaExprs. MetaElements include nodes and edges which can be variables or constants. Constants are used as specific types to use with the FROM and TO block in order to reduce the number of matches found in the proliferation process. Constants are specified by supplying the dollar sign $ as a suffix to the type. The model where the MetaElement's type is defined must be specified, it's done so by suffixing the name of the MetaElement with ***mm*** and followed by the level of the model where the MetaElements type is defined. The level starts at 0, representing the top most model, and is increased by one for each level that is visited. The level doesn't necessarily represent the exact level in the actual model hierarchy, for instance there might be models in the actual hierarchy which are above the model declared as level 0 in the rule. There might also be models which are between 0 and 1. This is allowed, because if we were to lock these rules to their exact rule locations, it would limit flexibilitiy, in addition introducing new levels in the actual hierarchy would break the rules. Note also that the ecore is presumed to be at level 0 and doesn't need to be included explicitly. The MetaExprs is the other construct used in the Meta block. It's mainly used to assign values. For example, to specify the source and target of an edge. MetaExprs are also used to place additional constraints on an element for example that a value must be between 5 and 10. The expression also helps limit the number of rules proliferated. The more constraints, the less rules are generated because there will be less candidates which fits the requirements.

The META block is used to find matches in the proliferation process, and also works as a type system for the body part (FROM and TO block) of the rule. The body part is used for the generation of fixed-level rules that should be executed by the underlying transformation engine.

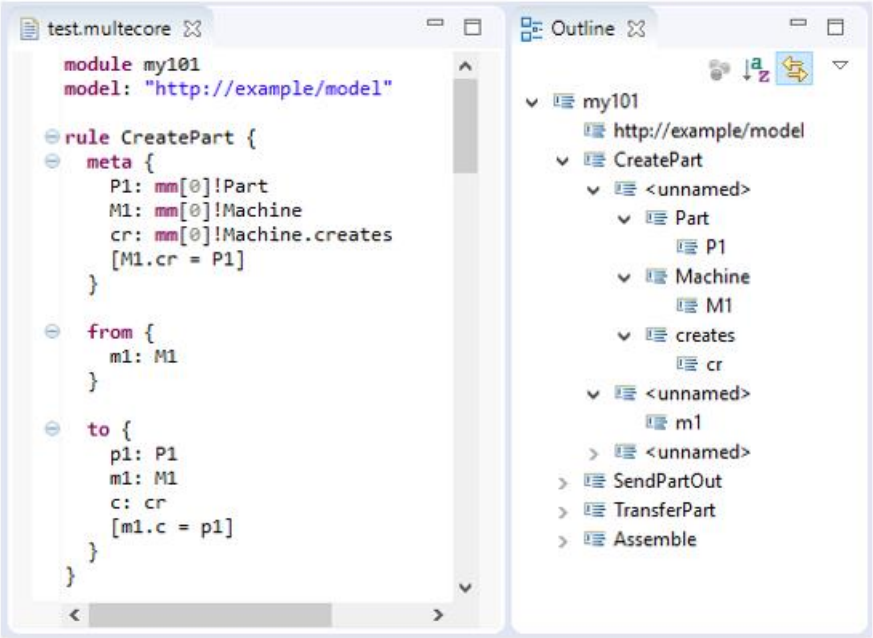

FIGURE 5.3 - THE DSL EDITOR BEING USED TO DEFINE THE CREATEPART RULE [25]

The DSL is optional when defining multilevel transformations. The alternative to using the DSL is to use a Java API, and write the transformation rules programmatically. Either approach will produce a java code representation of the rules because the engine can only work with rules that are defined in Java.
The rules defined in the DSL are translated into Java code before they are executed. One could say that the DSL works as a façade over the engine, and it is therefore possible to exclude the DSL and write the rules directly in the engine. However, the DSL provides so many useful features as well as much more concise and readable syntax that it becomes somewhat mandatory when using the framework for larger projects.
We defined the rules directly in Java when developing the engine because the rules we used to test the functionalitiy of the engine were simple, and it was straightforward to define them using objects in Java.

5.4. MultEcore

We use MultEcore to create multilevel models to transform with our Engine. MultEcore is a metamodeling tool that extends EMF with unlimited number of abstraction levels. In EMF, a metamodel is defined in an Ecore file and instances of a metamodel is specified in an xmi file. However, it is not possible to create new instances from the xmi file so therefor the model hierarchy ends after a model is created in xmi. MultEcore allows the user to transform the xmi file back into an

Ecore file, and produce a new model from the produced Ecore file. The elements in the xmi file are transformed into EClasses in the produced Ecore file, and their actual type is placed as a type annotation within their new EClass. The transformations are semi automatic in the sense that the user must click a button to transform an xmi file into an ecore file.

Other tools to create multilevel models such as Melanee [34] and Metadepth [35] could have been used instead of MultEcore. We did not investigate which metamodelling tool was best suited however we believe that any metamodeling tool that supports deep metamodeling would be suitable. Changing the modelling tool from MultEcore to Melanee would require that a generator from Melanee models to the engines representation of the model hierarchy is built. The user could then choose if he wanted to transform MultEcore or Melanee models, and overtime many modelling tools could be supported. The translation of Melanee models to the engine representation is refered to as tool chaining or bridging.

The multilevel hierarchy is one of the inputs that we use when determining the proliferated rules. The other input is the multilevel rule. Changes to either one of the inputs could lead to an increased or decreased number of proliferated rules.

We call it a model hierarchy because several new models can be created from one level onto the level below which creates a tree like structure with many branches. The engine is only working with one path in the hierarchy when matching multilevel rules. One path in the hierarchy is when there is only one model in each level. In other words, we only consider one model from each level even though in realitiy there might exist many more models per level.

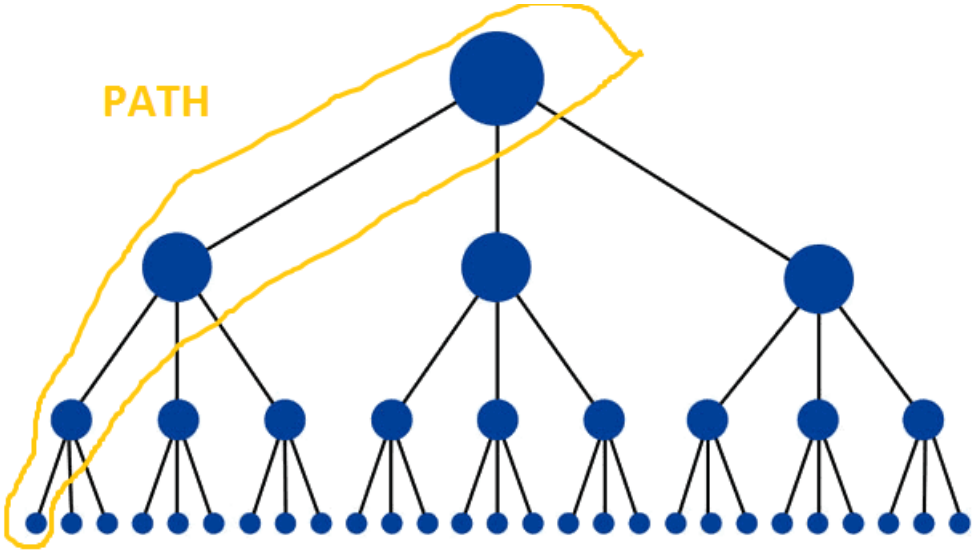

FIGURE 5.4 - EXAMPLE OF A PATH IN THE MODEL HIERARCHY

Using a different path for the matching requires that a new path is specified and that the engine runs with the new path. There are some reasons for only considering one path at a time.
It shortens the search space because only one model per level needs to be visited. The alternative being that *m* visits are needed per level where *m* is the number of branches. This means that the number of visists are reduced from m^n to *n*.
Running the matching for every path in the hierarchy will produce a lot of redundant rules which will never be matched to the model we are working with at the current time. These redundant rules lead to needless complexity.

5.5. The underyling transformation engine

Our multilevel transformation engine produces a set of two-level rules out of an MCMT, and we need a tool to execute the two-level rules that our engine produces. We decided to use an existing tool to run our produced rules. However, there are over a 100 transformation tools available [36] and not every tool is suited as an underlying engine to a multilevel transformation framework. We made a comparison of some of the transformation tools out there to find a tool that could be used as our underlying engine.

5.5.1. Comparison of transformation tools

Transformation tools are often created to solve specific problems which means that the correct transformation tool to use depends on the problem you are trying to solve. Many studies on comparing model transformation tools have been done in the past. Some studies laying out the features of every transformation tools other studies comparing transformation tools for a given purpose. However, no studies have been done on which transformation tool is best suited as an underlying engine to a multilevel transformation tool.
Here we do a literature review of studies done on comparison of model transformation tools in general, and apply it to determine which transformation tool is best suited as an underlying engine to our multilevel transformation engine.

5.5.1.1. Criterias

The first question to ask when determining the correct transformation tool to use is: What should be transformed into what? [37] The question refers to what is the source of the transformation and what is the target. Should models be transformed into other models (M2M) or should models be transformed into text (M2T) or should text be transformed into models (T2M). In our case, we want to transform models into other models and therefore we only consider M2M tools.

Transformation tool often have many features that they support such as rule scheduling, rule application control, traceability, interoperability etc. No transformation tools support every feature, and we need to determine which features we need, and pick a transformation tool that supports most of those features.
Since the transformation tool only acts as an underlying engine, many of the features typically supported in transformation tools become redundant because the underlying engine is hidden from the users. That means most features relating to the usage of the tool such as the editor, and re-usability techniques(RUT).
RUT refers to the re-use of transformation rules which can be done through rule composition, generic types and higher order transformations. The multilevel engine provides an editor, and re-usability of rules are supported through instance relations and therefore it is not needed in the underlying engine. There are more features that loses their importance, but we will not mention those here. Instead, we focus on the features that are important for the underlying engine.

We focus on endogenous transformation which is transformations where the source and target models conform to the same language. When transformations are endogenous it is usually beneficial to perform in-place transformations. In-place transformation means to produce the target model by modifying the source model. The other option is to produce the target model from scratch which is

referred to out-place transformations. In our case, we want our underlying engine to support in-place transformations.

We use MultEcore models which are EMF models and therefore our underlying engine should be able to transform EMF models directly, or provide mechanisms to automatically import EMF models and transform them into their representation, and mechanisms for exporting the tools native representation into the equivalent EMF model.

The underlying engine should be extensible which means that we can integrate the underlying engine with other tools, and add new functionality to the underlying engine. Furthermore, we should be able to execute the rules programmatically through an API.

The underlying engine should be a plugin to eclipse because eclipse plugins tend to be easily extendible and it is beneficial to keep everything on the same platform. However, standalone tools with open source code that runs on Java could be imported to the eclipse project.

The domain application of the tool should be general, and the underlying engine should be able to solve any problem. The underlying engine should support the CRUD operations meaning that it should be able to create, read, update and delete elements from a model. Furthermore, there should be support for logical constructs such as if statements and there should be mechanisms for looping or recursion.

The engine should support rule priority which means that we can decide which rule should be run first when two rules are applicable to the model at the same time. In other words, there should be a mechanism to control the order of which rules are executed when the order of execution is important.

The tool should provide documentation on how to use the tool, and provide examples. It's also preferred if a tutorial is included. The tool should be easy to use, although it does not really matter for the user as the syntax of the tool is hidden from them, but it's nice to work with a tool that is easy to use when mapping the MCMT rules to the underlying engine.

Transformation languages have different mechanism for applying the transformations. The mechanisms used depends on the paradigm the transformation tool follows. Declarative approaches focus on what should be transformed into what, with no regards to the order of execution. The order is non-deterministic which means that the engine transforms the models by applying the rules in a random order. Rules created in a declarative language are more compact and easier to maintain because the order of execution is implicit.
Imperative languages on the other hand, focuses on how a transformation should be performed and when. The multilevel transformation engine follows a declarative style, and therefore the underlying engine should also be of declarative nature. The declarative style is commonly used in transformation tools in the modelling community, and it seems most promising in theory [37].

Declarative tools are desired, but graph transformation tools are very similar to declarative tools and overlaps in many aspects therefore we rate graph transformation tools equally with declarative tools.

In summary, we are looking for a tool that:

- Supports EMF models
- Performs in-place transformations
- Declarative approach
- Eclipse plugin
- General purpose
- Supports explicit rule-scheduling
- Easily extendible
- Called through API
- Well documented

There is no way to determine which tool is the best without implementing the solution using each tool, but that would take far too much time. We will have to take an educated guess based on the criteria we have selected. The tool of choice may not contain all of the features we would like, but it will act as a filter to limit the number of transformation tools that we will further investigate.

5.5.1.2. Candidates

In [36] they do a study on model transformation tools. They give an overview of a total of 65 model transformation tools where the features of each tool is laid out. We use their information to determine four transformation tools which we will investigate further. We cross-reference our criteria against the features of each transformation tool included in their study, and pick the tools that support the most criterias.
The reason for picking four tools is that we do not have time to go into depth about every tool, and the information is not accurate enough to pick a winner straight away. Instead, we went somewhere in between where we picked some candidates which seem like they could get the job done, and then we tested each tool to find out which tool felt best.

TEFKAT

Tefkat is a declarative transformation tool that has a simple SQL like syntax. The tool is an eclipse plugin that performs transformations directly on EMF models. Tefkat is specifically designed to write re-usable transformations that operates on high level domain concepts [38].
Tefkat only supports out-place transformations, and does not have an API to programmatically execute the transformation rules.

- Declarative
- General tool
- Plugin for eclipse
- No extensibility support
- Documented with tutorial and examples.
- The tool transforms EMF models directly without any intermediate representation
- The tool performs out-place transformations
- Rule scheduling is implicit

HENSHIN

Henshin is an eclipse plugin that performs graph transformations on EMF models. Henshin transformation rules can be executed by using an interpreter wizard or the interpreter can be called programmatically by an API. The transformations can be performed in-place or out-place depending on the needs of the user. In the case of in-place transformations, Henshin supports state space exploration which is an overview of all the intermediate states that are used to transform the source model into the target model. In the case of out-place transformations, Henshin can transform an arbitrary number of source models into an arbitrary number of target models.

- Graph based
- General tool
- The tool is an eclipse plugin
- The tool supported extensibility
- The tools transforms EMF models directly without any intermediate representation
- The rules can be invoked through an API
- Supports both in-place and out-place transformations
- The rule scheduling is explicit

ATL

ATL is transformation tool developed by AtlanMod as a plugin to eclipse. The tool is a hybrid between declarative and imperative where both styles can be used although the declarative style is encouraged when possible. It supports in-place and out-place transformations and the transformations are performed on EMF models. The transformations can be executed programmatically through an API which we can use to extend the tool. The order of rule execution is nondeterministic, but the user can manipulate the order of execution by using Lazy rules. Lazy rules are not matched by the engine, but rather are called from other rules that have been matched by the engine. The tool is well documented and provides several examples in addition to a large user base.

- Hybrid style - mix of declarative and imperative
- General tool
- Performs transformations directly on EMF models
- Out-place and in-place transformations
- Plugin to eclipse
- The transformations can be called programmatically through an API.
- The tool is extensible.
- Supports both implicit and explicit rule scheduling

Groove

Groove is a graph transformation tool specifically designed to explore state spaces. The tool performs in-place transformations on a graph representation of the models. This means that EMF models can not be transformed directly in Groove, and must be converted into a graph representation before they can be transformed. Groove supports functionality that does this transformation automatically. Groove runs as a standalone that runs on the JVM virtual machine. However, Groove is open source which can be downloaded and included in a Java project hence the

rules can be called programmatically.

- Graph based
- General tool
- Uses an intermediate graph representation of the model to perform transformations on.
- Standalone
- Extensible
- EMF models import/export mechanisms
- All forms of rule scheduling including rule priority

5.5.1.3. Overview of the candidates

	TEFKAT	*HENSHIN*	*ATL*	*GROOVE*
Paradigm	Declarative	Graph	Hybrid	Graph
Purpose	General	General	General	General
Model representation	XMI/EMF	XMI/EMF	XMI/EMF	GXL/Graph
Execution environment	Eclipse	Eclipse	Eclipse	Standalone
Extensible	No	Yes	Yes	Yes
Transformation type	Out-place	In-place and out-place	In-place and out-place	In-place
Documentation	Sufficient	Yes	Yes	Yes
Rule scheduling	Implicit	Explicit	Implicit and explicit	Implicit and explicit

5.5.2. Groove as the underlying engine

We chose to use Groove as the underlying engine because it was the candidate that we preferred. In addition, Groove was very easy to learn and the rules were easy to generate due to the simple syntax used in Groove.

The syntax uses nodes and edges in a similar way that we do in our engine. Figure 5.5 displays the syntax of a groove rule. The rule is defined in a GXL file which is basically the same as an XML file, and it is possible to produce a GXL file by using an XML generator. The syntax always starts with an XML declaration, followed by a gxl tag. A graph is defined inside of a gxl tag, and may contain attributes, nodes and edges. The attributes of a graph specify its properties. An example would be the priority of a rule graph, another one would be RHS is NAC which means that a rule may only be applied once. A node has an ID and nothing else, the attributes of a node is specified by using an edge from the node, to itself and attach the attribute to that edge. An edge has a source and target node which must be specified by using their ID, an edge contains an attribute that hold information.

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<gxl xmlns="http://www.gupro.de/GXL/gxl-1.0.dtd">
<graph edgeids="false" edgemode="directed" id="GenPart1" role="rule">
  <attr name="priority">
            <string>1</string>
        </attr>
<node id="m1"/>
<edge from="m1" to="m1">
<attr name="label">
<string>type:GenHandle</string>
</attr>
</edge>
<edge from="m1" to="p1">
<attr name="label">
<string>new:creates</string>
</attr>
</edge>
<node id="p1"/>
<edge from="p1" to="p1">
<attr name="label">
<string>type:Handle</string>
</attr>
</edge>
<edge from="p1" to="p1">
<attr name="label">
<string>new:</string>
</attr>
</edge>
</graph>
</gxl>
```

FIGURE 5.5 TEXTUAL SYNTAX OF CREATE HANDLE IN GROOVE

Every element in groove is either a node or an edge. Groove uses a host graph (Figure 5.6) which is a representation of the model, and a set of rule graphs which are matched against the host graph and transforms it if a match is found. The rule graph consists of three different graphs, the left-hand side (LHS), the right-hand side (RHS) and the negative application conditions (NACs). They appear to be in the same graph in the view editor, but they are stored in separate containers in the background.

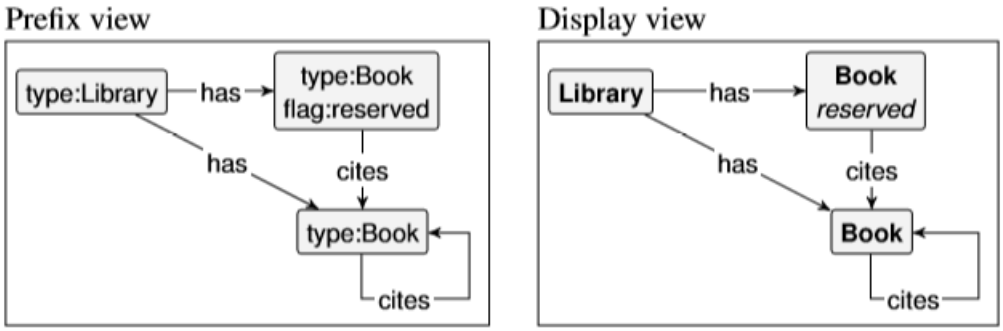

FIGURE 5.6 - HOST GRAPH IN GROOVE

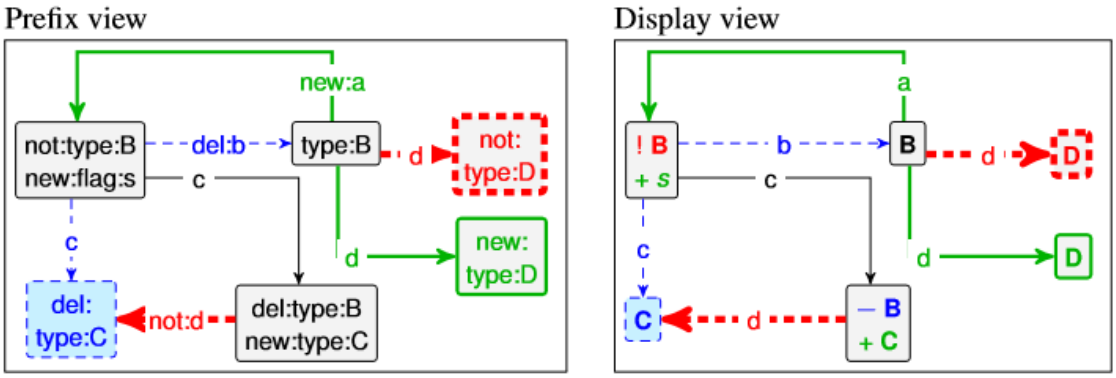

FIGURE 5.7 - BASIC RULE GRAPH ELEMENTS IN GROOVE [39]

Figure 5.7 shows how the three graphs are seperated by color coding. There are four basic elements in rule graphs.

Readers are elements which appear in both the LHS and the RHS. They are used to match the host graph, but they don't modify the host graph in anyway. These elements are represented by the color black and unlike the other elements the readers don't need a keyword. The reader element is the default element, so if no keyword is specified then the element becomes a reader.

Creators are elements which appear only in the RHS of the rule. They are not used for matching, but are instead added to the host graph when applying the rule. They are represented by green in the rule graph, and are declared as a creator by prefixing the keyword ***new***.

Erasers are elements which appear only in the LHS of the rule. They are used to match the host graph and are removed from the host graph when a rule is applied. They are represented by blue in the rule graph, and are declared as an eraser by prefixing the keyword ***del***.

Embargoes are elements which appear in the NAC, and are not allowed to appear in the LHS. They are used as a guard to filter out unwanted matches. They are represented by blue in the rule graph and are declared by the prefix ***not***.

There are more advanced concepts in Groove for example counting and merging, but we will ignore these in this thesis. The reason is that the goal of this thesis is to show that transformation rules can be defined on generic types and be reused on instances of these types. It should not make a difference if the rule is simple or complex, the principle is the same and should work in both cases.

Therefore, if we succeed in reusing the rules for simple cases then we should be able to reuse them in more complex cases. That is not certain though, so it could be investigated in further work.

One of the drawbacks with Groove is that it uses its own graph representation for models, and doesn't support emf models directly therefore we need to transform our emf models into Groove graphs before performing the transformations. Then the Groove graph needs to be transformed back into an emf model, after the transformations are completed.

Other transformation tools could have been used as the engine. Changing Groove with another transformation tool would require that we make a new generator from our flattened rules to the new transformation tool. We could still use Groove by using the Groove generator, but the new generator gives us another option and over time we could potentially have many transformation tools that could run the flattened rules. It is only a question as to how many bridges we build from the flattened rules to different transformation tools. The initial flattened rules would not change from changing the underlying engine, only the way they are transformed.

5.6. Multilevel execution engine

The multilevel execution engine is the core of the MCMT framework. The engine is implemented in java, and connects all the other components in the framework. The engine matches an MCMT rule against a model hierarchy and produces a flattened rule for every valid match. In other words, the engine transforms an input rule into a set of new rules which is the same concept as that of higher order transformations [40] (HOTs). The number of produced rules are dependent on both the MCMT rule and the model hierarchy. Making changes to either one could change the produced rules. The equation:
$MCMT\ Rule + Model\ Hierarchy + Engine \rightarrow (0..*)\ Flattened\ Rules$
gives a good overview of what the engine does. Flattened rules are 2-level versions of the MCMT rule, and is ment to be executed by a 2-level execution engine.

The engine has two main responsibilities: Firstly, find all the matches of the MCMT rule in the model hierarchy. Secondly, generate a flattened rule for each match found.

5.6.1. Components inside the Engine

Here we will present the components of the engine. Each component has an internal representation within the engine aswell as being an external component with an external representation. The external representation is mapped to its corresponding internal representation. The mapping is commonly referred to as tool chaining and could be viewed as a bridge from one tool to another.

Model hierarchy

The model hierarchy acts as an in-memory representation of a multilevel model hierarchy. The hierarchy is represented by a list of metalevels. A metalevel contains a list of nodes. A node contains a name, a type and a list of edges which represents relations to other nodes. Edges have a source and target node, a name and a type. The model hierarchy is used to match the transformation rules.

The objects should be populated by parsing a MultEcore hierarchy file, but we did not implement that part yet, so we are using hardcoded objects to represent the model hierarchy.

Currently a node only represents a concept, the attributes of a concept is omitted for simplicity. One way to represent a concept's attributes would be to add a list of attributes to a node.

MCMT rule

The class used to represent a transformation rule contains a meta part and a body. The meta part contains a list of variables. Variables have a name, a type, and a link to the metamodel that defines its type. Variables also contain a list of connections which represents relations from itself to other variables. A connection has a source and target which are variables, they also have a name and a type.

Variables and connections are similar to nodes and edges in the model hierarchy, and could be presented by the same classes. We chose to use different names to distinguish between the two because variables and connections are to be matched with nodes and edges respectively.

The body of a rule contains a From part and a To part which represents the left hand side and right han side respectively. Both sides contain body variables and body connections, which are typed by variables and connections in the meta part. The type of the elements in the body is replaced when a match is found.

Pattern hierarchy

A pattern hierarchy contains a list of pattern levels, the number of pattern levels and their contents are inferred by the number of model levels used in the meta part of the rule. A pattern level is similar to the meta part of a rule, but instead of having all variables in the same level they are scattered across multiple pattern levels. The union of pattern levels are equal to the metapart of the rule which they were inferred from. All variables and connections that are typed by the the same metalevel are placed in the same pattern level.
The pattern hierarchy is matched against the model hierarchy where every level of the pattern hierarchy must match a level in the model hierarchy. We will go into depth about how this is done later in the chapter.

5.7. Matching

In the engine, matching is the process of finding every way that an MCMT rule matches a model hierarchy. A pattern hierarchy is inferred from an MCMT rule, and is matched against the model hierarchy. Two different forms of matching are used, an inner matching and an outer matching. Both the pattern hierarchy and model hierarchy may have several levels. The inner matching determines if a pattern level matches a given model level. The outer matching iterates over each pattern level and attempts to find a match with a model level using the inner matching.

5.7.1. Graph Pattern Matching

Matching is a process to determine if a transformation rule can be applied to a model. The process is usually performed by finding a pattern of the left-hand side of the rule in a model. A rule may be applied if such a pattern exists in a model.

Finding patterns in graphs means finding a homomorphic or isomorphic image of an input graph (pattern) in another graph (target) [41]. Because of this, graph pattern matching is also known as the subgraph isomorphism problem. Subgraph isomorphism problem is known to be NP-complete [42] so no efficient perfect solution exists. Some algorithms are efficient, but comes at the cost of the matching result, meaning the matches are not complete. Because we want to find all the matches, this is not an option for us and we don't care too much about efficiency at this stage. Two algorithms are the most used ones for finding exact subgraphs. The tree search based Vf2 and Ullmanns algorithm, they are both using depth first search. The difference between the two is that Vf2 tries to build up matches from scratch while Ullmanns algorithm starts out with all matches as potential candidates and tries to prune the unfit candidates in each iteration until only fitting candidates remain. Candidate nodes are mapped to variables in the pattern graph if they have the same type, and will become a match for a variable if they have the same structure.

We have adopted the algorithm used in [43], the author has created an algorithm for subgraph pattern matching on large scaled graphs based on dual simulation. The algorithm is easily extendible and is meant to be used as a backbone for a query processing engine for a graph database.

5.7.1.1. Dual Simulation

Dual simulation is one of the approaches to pattern matching that does not produce exact matches, the algorithm runs in quadratic time and rapidly eliminates many of the candidates which would have been eliminated by Ullmanns algorithm [43]. We use dual simulation in conjunction with Ullmanns algorithm to prune out bad candidates early on. Removing bad candidates early on means that the search tree will have less nodes, and removing a bad node higher up in the tree is more efficient because the number of bad results that are spawned from the node is multiplied by **n** on every new level. We run dual simulation on the matches as a preprocessing step, and on each iteration of the search tree.

Dual simulation is a pruning method, identified by the author of [43]. Given the pattern graph $Q(V_q, E_q)$, the target graph $G(V, E)$ and the morphism $\Phi(V_q) \rightarrow V$ from Q to G. It checks that the two conditions hold:

$$\forall(u, u') \in Eq, \forall v \in \Phi(u), \exists v' \in \Phi(u') s.t. (v, v') \in E$$

Which means that if **v** is matched to a pattern node **u**, then all childrens of **u** in **Q** must be be matched by childrens of **v** in **G**. Children here means target nodes of an edge with the parent being the source node.

$$\forall(u, u') \in E_q, \forall v' \in \Phi(u'), \exists v \in \Phi(u) s.t. (v, v') \in E$$

The second condition is the other way around and states that if the pattern node **u'** has a parent **u**, then the target node **v'** must have a parent node **v** which is a match for **u**.

Checking only the first condition, is refered to as simple simulation. Dual simulation is used to filter out bad matches early on, and thereby reducing the search path.

FIGURE 5.8 - A RELATION BETWEEN TWO VARIABLES A AND B IN THE PATTERN

Figure 5.8 illustrates two adjacent variables **A** and **B** in the pattern graph, where **A** is the parent and **B** is the child. Finding a match for **A** and **B** in a target graph requires that the match for **A** is adjacent to the match for **B**, otherwise the structure is violated. Moreover, the match for **A** must be the parent and the match for **B** must be the child. Let φ(A) be the candidates for variable **A**, and φ(B) the candidates for **B**. Then simple simulation checks that each candidate $a \in \varphi(A)$ has an adjacent child node which is a candidate for **B**. In other words, there exists an edge $e(a, b)$ in the target graph where $b \in \varphi(B)$. Dual simulation works the other way around, and is checked when evaluating the candidates of the child of a relation. We continue from the example of simple simulation, and the candidates of **B** are being evaluated. Dual simulation requires that a candidate for **B** is adjacent to a parent node that is a candidate for **A**. An edge $e(a, b)$ must exist in the target graph for each candidate $b \in \varphi(B)$. The condition is the same in both cases, but with simple simulation it is only checked when evaluating the candidates of the parent variable (**A** in this case). Dual simulation checks this condition when visting both the child and parent variable. However, it is not necessary to check this condition explicitly when visiting the child variable because it is possible to remove bad candidates of the child variable when visiting the parent variable. The bad candidates are removed by constructing a new set of candidates for the child variable. The new set only contains children of the set of candidates for the parent variable and therefor every candidate of the child variable has a parent in the set of candidates for the parent variable.

```
1: procedure DualSim(G, Q, Φ):
2:     changed ← true
3:     while changed do
4:         changed ← false
5:         for u ← V_q do
6:             for u' ← Q.adj(u) do
7:                 Φ'(u') ← ∅
8:                 for v ← Φ(u) do
9:                     Φ_v(u') ← G.adj(v) ∩ Φ(u')
10:                    if Φ_v(u') = ∅ then
11:                        remove v from Φ(u)
12:                        if Φ(u) = ∅ then
13:                            return empty Φ
14:                        end if
15:                        changed ← true
16:                    end if
17:                    Φ'(u') ← Φ'(u') ∪ Φ_v(u')
18:                end for
19:                if Φ'(u') = ∅ then
20:                    return empty Φ
21:                end if
22:                if Φ'(u') is smaller than Φ(u') then
23:                    changed ← true
24:                end if
25:                Φ(u') = Φ(u') ∩ Φ'(u')
26:            end for
27:        end for
28:    end while
29:    return Φ
30: end procedure
```

FIGURE 5.9 - PSEUDOCODE FOR DUALSIMULATION TAKEN FROM [43]

In Figure 5.9 the algorithm takes the target graph **G**, the pattern graph **Q** and the mapping from pattern nodes to target nodes **φ.** The mapping is done by having a pattern node point to a list of possible target nodes. A loop iterates over every pattern node **u**, and a nested loop iterates over every children **u'** of **u**. On each iteration of child **u'** a new list of matches **φ'** are created from the empty set. Then each target node which matches **u** is iterated in a third forloop, during this forloop the conditions discussed earlier are checked. The first condidition for simple simulation is checked on line 9, and the second condition is ensured in line 17, by constructing a new set of matches **φ'(u')** which only consists of nodes with a parent in **φ(u)**.

5.7.1.2. Subgraph Matching

After the initial matching, a variable might be matched to several types. For instance, a

HeadGenerator and a HandleGenerator are both a valid match for a variable of type Machine. However only one type is allowed to be mapped to a given variable, this means that the variable for machine needs to create atleast two new rules. One where it's mapped to HandleGenerator and one where it's mapped to HeadGenerator. This algorithm makes sure that every combination of variables assigned types are generated. It does so by creating a new copy of the matches and replacing the list of matches for the variable at the current depth with a singleton list containing only one of the matched nodes. Then the algorithm continues to the variable at the next depth until it has reached every variable, and you end up with a singleton list for each variable. A complete match is found when all variables have been assigned to a single node. A search could provide many such matches, these matches are returned in a list. One could think of the process as the process of generating every possible permutation of the legal assignments of the variables. The process is illustrated in Figure 5.11.

```
1:  procedure FindMatches(G, Q)
2:      matches ← ∅
3:      Φ0 ←FeasibleMatches(G, Q)
4:      Φ0 ←DualSim(G, Q, Φ0)
5:      Search(G, Q, Φ0, 0)
6:
7:      procedure Search(G, Q, Φ, depth)
8:          if depth = Q.size then
9:              matches ← matches ∪ Φ
10:         else
11:             for v ← Φ(depth) do
12:                 if v ∉ Φ(0) ∪ ... ∪ Φ(depth − 1) then
13:                     Φ′ ← copy of Φ
14:                     Φ′(depth) ← {v}
15:                     Φ′ ←DualSim(G, Q, Φ′)
16:                     if Φ′ is not empty then
17:                         Search(G, Q, Φ′, depth + 1)
18:                     end if
19:                 end if
20:             end for
21:         end if
22:     end procedure
23:
24:     return matches
25: end procedure
```

FIGURE 5.10 - THE ALGORITHM USED FOR FINDING ISOMORPHIC MATCHES [43]

The algorithm in Figure 5.10 illustrates how this matching works. The algorithm starts with finding a set of candidate nodes for each variable (line 3). Variables are nodes in the pattern graph. For a node in the target graph to be a valid candidate for a variable, it needs to be an instance of the same type as the variable. For example, let M be a variable of type Machine, then Assembler of type Machine is a valid candidate for M. The initial search for candidates is purely based on labels therefor the initial

set of candidates can be quite large. The candidate nodes are checked for structural constraints in line 4 so unfit candidates are discarded. Structural constraints are the constraints mentioned in section 5.7.1.1. Now we have a map from a variable to a list of candidates, the depth of the map is the number of variables. The length of the list for each variable depends on the number of valid candidates for the given variable. The end goal of the search is to end up with a list of maps where each map contains all the variables, and each variable point to a singleton list with only one valid candidate. The search begins on line 5, and takes as arguments a target graph, a pattern graph, the map of matches and the starting depth (index starting at 0). The search algorithm starts at line 11. It iterates through all the candidates for the variable at the current depth. It makes sure that the candidate node is not already matched to another variable (line 12). If the candidiaite is not matched to another variable then it makes a copy of the map of matches, and replaces the candidate list for the variable at the current depth with the candidate node from line 11. Then it runs the dual simulation pruning on the new map. If the map is not empty after running dual simulation, then recursively continue with the new map to the next depth. If the algorithm makes it to the depth equal to the number of variables then each variable has found a match, and therefore the current map is added to the list of matches.

The algorithm is not the exact same as the one we are using because we have modified it to work with multilevel model transformations. We display this algorithm here because the principle is the same, but this algorithm is simpler and therefore illustrates the idea better. The changes we've had to make to the algorithm is that we have made edge labels part of the matching. This makes the algorithm abit clunkier than the original. We've also had to build a new algorithm on top of this one which uses this algorithm to search for matches in every level of the hierarchy. We'll get back to that later in this chapter.

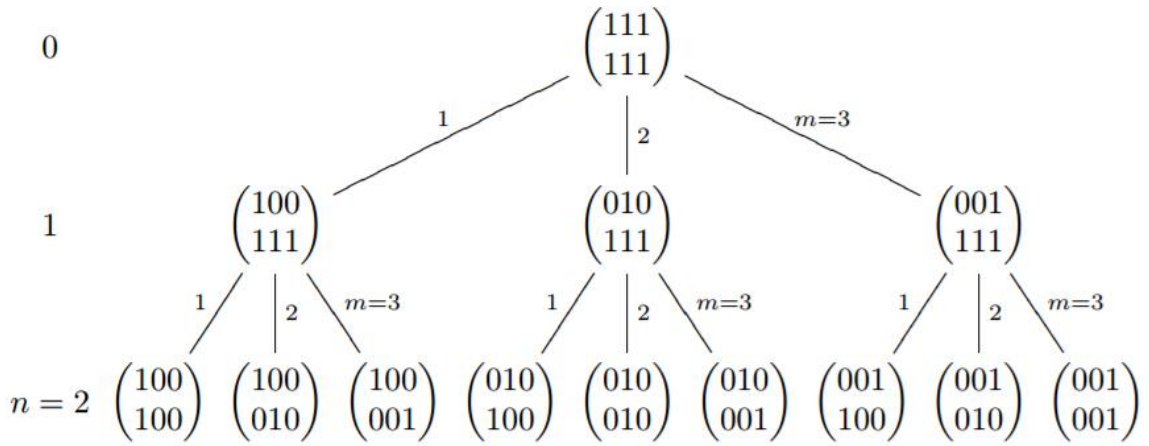

FIGURE 5.11 - TREE SEARCH FOR VARIABLE ASSIGNMENT [41]

As mentioned earlier, a variable can only be assigned to one specific type at a time. Figure 5.11 illustrates how variables are assigned to only one type. For each node(variable) in the pattern there exists a row in the matrix, and the number of rows is n which is also equal to the depth of the tree. For each node (type) in the target graph there is a column. If a type matches a variable then the value 1 is placed in that variables row in the types column otherwise it has the value 0. The initial candidates are shown in level 0, they are represented by a nXm matrix where all the columns have a

value of 1. A row which has more than one column with the value 1 means that the variable has multiple matches. The point of the search is to make sure that each variable only has one matching type. This means that there must be exactly one column with the value 1 in each row. This is achieved by iterating over each collumn in a row, and on each column with a value of 1 we make a copy of the current matrix, but with the current column as the only match. Then visit the next row of the newly made copy and repeat the process, continue until we have reached the end of the matrix (tree depth).

Note that the process illustrated here is very simplified, it shows two variables which is assigned to the same type. Furthermore, this illustration uses matrixes, but in our implementation, we use a map of variables pointing to lists of varied sizes. These lists do not hold Boolean values, but rather uses node objects. The process of variable assignment is the same with both approaches, and the simplified version was used for illustration purposes.

5.7.2. Multilevel Matching

The algorithm presented earlier in Figure 5.10 is only able to find a match of one graph in another graph. The algorithm only uses one level however with multilevel matching both the source graph and the pattern graph can have multiple levels. One way to view the multilevel matching would be to imagine a pattern tree that only has one node in each level and the nodes in the tree are graphs. The pattern tree should be matched to a target tree (the model hierarchy) by finding a matching node in the target tree for each node in the pattern tree.

We can reuse the algorithm in Figure 5.10 to match a node in the pattern tree with a node in the target tree without making significant changes to the algorithm. Because the pattern and the model on a given level are both graphs, we can take the pattern on a given level and match it with a model in each level and if we get a match then we can store the the model level as a match for the pattern level. The algorithm in Figure 5.12 takes advantage of this. Note that graphMatch is same as the algorithm in Figure 5.10, but with a few modifications.

Algorithm 1 Matching algorithm

```
1: procedure MATCH(MM, TG, mmLevel, tgLevel, matches)
2:     if mmLevel = MM.size then
3:         return true                                   ▷ End of pattern reached
4:     end if
5:     found ← false
6:     while tgLevel < TG.size do                        ▷ Every level in hierarchy
7:         maps ← graphMatch(MM[mmLevel], TG[tgLevel], matches[matches.size − 1])
8:         for all m ∈ maps do
9:             size ← matches.size
10:            if size > 0 and mmLevel > 0 then
11:                currentMatch ← matches[size − 1]
12:                matches[size − 1] ← currentMatch ∪ m
13:                found ← found or match(MM, TG, mmLevel + 1, tgLevel + 1, matches)
14:                matches[matches.size] ← currentMatch
15:            else
16:                matches[size] ← m
17:                found ← found or match(MM, TG, mmLevel + 1, tgLevel + 1, matches)
18:            end if
19:        end for
20:        tgLevel ← tgLevel + 1
21:    end while
22:    if mmLevel > 0 then
23:        matches[matches.size − 1] ← ∅                 ▷ Remove incomplete match
24:    end if
25:    return found
26: end procedure
```

FIGURE 5.12 - MULTILEVEL MATCHING ALGORITHM [25]

The algorithm for multilevel matching takes five arguments: The pattern hierarchy (MM), the model hierarcy (TG), the index of the current pattern level being matched which starts at 0 and increases by 1 if a match for the current pattern is found, the index of the current model level being matched which starts at 0 and is incremented everytime a level has been visited, and lastly the set of matches found so far which is initially empty and is used to save progress between recursive calls.

The base case of the algorithm is when the index of the pattern level is equal to the number of pattern levels in the rule. This indicates that atleast one complete match has been found, and true is returned.

First, we set found to false (line 5), found is a Boolean value that indicates whether or not we have found a complete match. Found is modified in line 13 and 17. The statement says that if found is true or the match function returns true, then we set the value of found to true. Once found is set to true, it will never go back to being false.

The search starts inside the while loop on line 6, the while loop iterates until it has reached the end of the model hierarchy, this means that all model levels have been visited.

The function in line 7 finds all the ways that the two current levels can be matched against eachother, and stores each match in a list of maps. If the two levels can't be matched then maps will be empty, and no further actions will be taken except for increasing the model level. If the list of maps is not empty then each map will be iterated in a for loop. The variable ***size*** is the number of matches found already, this includes partial matches. We check if we are adding to a partial match or

starting a new match in line 10. If the pattern level is greater than 0 and the size of matches is greater than 0 then we are adding to a partial match.

If we are adding to a partial match then we take the current map in maps and append it to the last element in the matches list. Then we increase the pattern level and the model level, and try to match these. We add currentMatch to matches (line 14), this will be removed immediately (line 22) and will not be used for anything else. The addition is done post visit, and will not be included to any recursive call.

Whenever the pattern level is 0, we must insert the current map as the only element in matches. Instead of the route on line 10, we pick the route on line 15.

There is a check at line 22 that finds incomplete matches and removes them in line 23, the reason that the check works, is that if it's a complete match, then the function will never get further than line 3, so complete matches will never encounter that check. This check will trigger based on the pattern level, this could lead to complete matches being removed. Therefor we need to temporarily add partial matches at the end of the list (line 14), so that these partial matches are removed instead of complete matches found in previous calls.

5.7.2.1. Determining the number of pattern levels in the rule

We have previously mentioned that an MCMT rule contains a META part and a BODY part both of which play a different role in the proliferation process. We will now talk about the role of the META part of the MCMT rule. The metapart of the rule is used to define the variable types used in the bodypart of the rule. The variable types have a metatype, and their actual type in the hierarchy is unknown. We use the metapart of the rule to construct a pattern hierarchy based on how many levels are used to define the variables in the metapart of the rule. We use an algorithm to decide the number of pattern levels that should be created and an algorithm that places the variables in their correct pattern level. The algorithm to decide how many pattern levels to create is straightforward:

- Create an empty set of patternlevels
- Visit each element of the metapart of the rule
- For each visit: If the element is a constant then add elements metalevel url suffixed by constant. If the element is a variable then add the elements metalevel url suffixed by constant and then add the elements metalevel url suffixed by variable.
- Lastly count the size of the set

This works because sets do not add duplicates and therfore if two variables are typed by the same level then only one level will be added to the set. The name of the metalevel is suffixed by whether the visited element being a variable or constant because constants should be matched to the level they were defined in and variables should be matched with a level somewhere below the level it was defined in. In otherwords, if a variable is typed by a metalevel and a constant is typed by the same metalevel then two pattern levels are required. Note that; the metatypes of variables are considered constants, and patternlevels are added for these constants to check that the MCMT rules are typed by the model hierarchy. Therefore, metavariables always span two pattern levels: One pattern level for the variable and one for the metatype.

The second algorithm to place the meta elemenets in the correct pattern level is as follow:

- Continue from the set of pattern levels that was created in the previous step

- Visit each element of the metapart of the rule
- For each visit: If it is a constant then look up the patternlevel at elements url + constant and then put the element in this patternlevel. If it is a variable then look up the the pattern level with the name elements url + constant and then add the metatype of the variable as a constant to this level, and then look up the patternlevel at elements url + variable and put the element in this patternevel.

5.7.2.2. Replacing the variables with actual types

The variables in the body part of the rule is initially typed over meta-variables, these meta-variables are just placeholders for actual types. Now we need to swap these meta-variables with actual types. As a result of the matching we get a list which contains all the valid matches, and for each of those matches we need to produce a new rule. This means that we need to iterate through the list of valid matches, and for each valid match we need to make a copy of the rule and then replace the variables in the copy rule with the types in the match. Meta-variables are looked up in the current match to find it's corresponding type, the meta-variable is then replaced by that type. Once every meta-variable is replaced by an actual type, we get a new proliferated rule.

5.8. From Java objects to Groove rules

Todo the code generation a simple XML writer is used. It works well because groove uses gxl which is no different than XML, and can be written to by using an XML writer. With Groove, there are only nodes and edges. They are matched by their label, so we need to map our variables to nodes and edges with corresponding labels. We also need to make sure that the variables in the from part only is in the left-hand side, in groove they use the keyword *del*. The variables in the to part is only in the right-hand side, in groove they use the keyword *new*, and the variables which are both in the from and to part need no keyword. We need to look at the rule to determine if an element is a creator, reader or eraser.

The algorithm to determine the keyword of an element is straight forward. First go through the elements in the from part, check if it is contained in the to part and if it is then mark it as a reader, if it's not then mark it as an eraser. Then visit the elements in the to part, if the element is marked then it means that the element occurs in both the left-hand side and the right-hand side hence it's a reader. If it's not marked then it means that it only occurs in the right-hand side and is therefore a creator.

6. Demonstration

In the previous chapter, we described a multilevel transformation engine that took as input a multilevel modelling stack and a multilevel coupled transformation rule and produced a collection of flattened rules. The engine is available at http://prosjekt.hib.no/ict/multecore where instructions are provided.

In this chapter, we demonstrate how the engine performs this process. We will use the production line system that was introduced in Figure 3.3. The hammer production system is matched with an MCMT rule for generating parts and then the stool_plant is matched with the same MCMT rule. We intend to illustrate that an MCMT rule can be reused within the same system, and across similar systems.

We repeat the hierarchy from Figure 3.3 because the hierarchy is central to the matching process. Looking at the hierarchy together with the MCMT rule should give the reader a better understanding of the flattening process.

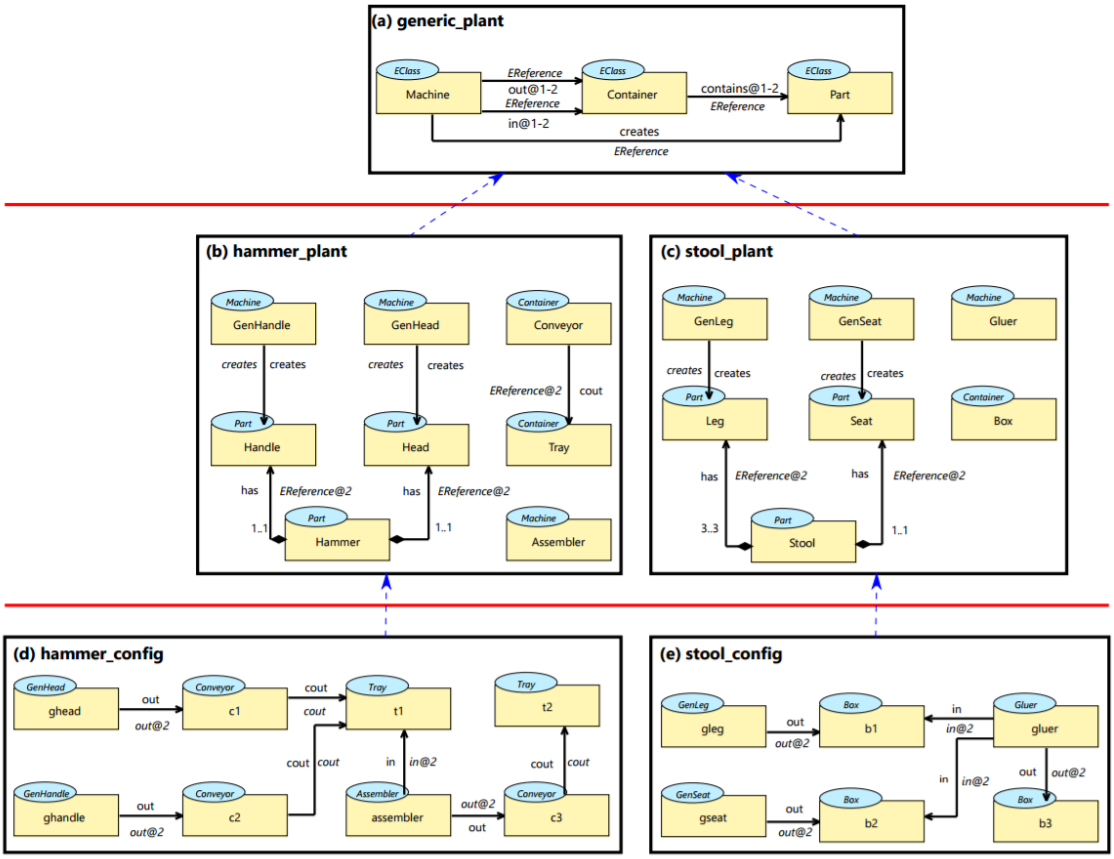

FIGURE 6.1 - MULTILEVEL PLS, INTRODUCED IN FIGURE 3.3

The hierarchy illustrates two systems. However, only one system is matched at a time. A system is specified by a chain of metalevels. A chain of metalevels means that there is one model in each metalevel, and that each model conforms to the models above. For example, **generic_plant** ← **hammer_plant** ← **hammer_config** constitutes a chain, whereas **generic_plant** ← **hammer_plant** ← **stool_config** does not because **stool_config** does not conform to **hammer_plant**.

6.1. MCMT rule

As a starting point of the demonstration, we use an MCMT rule which should be transformed into proliferated rules.

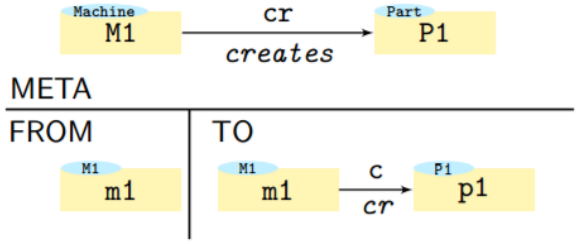

FIGURE 6.2 - THE MCMT RULE TO BE MATCHED WITH THE PLS

Figure 6.2 illustrates the rule we use to demonstrate the flattening process. The rule was explained earlier in Figure 4.5. Here, we add some details to that explaination. The meta part of the rule contains the pattern that should be matched to the hierarchy. The create_part rule has two pattern levels as illustrated in Figure 6.3. The first pattern level P_0 contains the types of the rule, and must be matched to a metalevel in order to confirm that the MCMT rule conforms to a metalevel. The second level P_1 contains the variables that should be matched to instances of the types in P_0.

Note that the number of pattern levels in a MCMT rule can be much greater than 2. A rule may also only have one pattern level in the special case where the meta part only contains constants that are defined in the same metalevel; in which case, the rule is the same as a two-level rule.

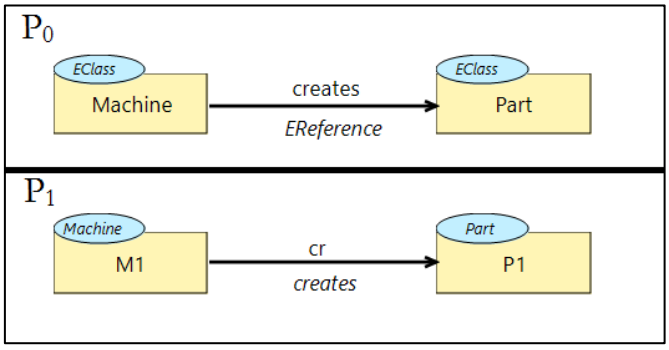

FIGURE 6.3 - PATTERN LEVELS IN THE CREATE_PART RULE

After the pattern hierarchy in Figure 6.3 has been constructed, the levels of the pattern hierarchy are matched against the levels of the model hierarchy in Figure 6.1. Starting with the topmost pattern level **P_0**. The only valid match for **P_0** is **generic_plant** therefor a morphism from **P_0** to **generic_plant** is made, and a list of ways that the elements in **P_0** can be mapped to elements in **generic_plant.** In this case, only one mapping is possible because every element in **P_0** is a constant. After a match is found for **P_0**, the next pattern level **P_1** is attempted to be matched. The matching of **P_1** is different from the matching of **P_0** because **P_1** has variables whereas **P_0** only has constants. The main difference is that **P_0** could only be matched in one-way whereas **P_1** could potentially be matched to the same metalevel in several seperate ways. The other difference being that constants are matched by the name of the elements, whereas variables are matched by the types.

Matching **P_1** to **hammer_plant** starts out by setting every instance of Machine in **hammer_plant** as candidates to **M1**, every instance of **Part** as candidates to **P1**, and every instance of **creates** as

candidates to **cr**. The initial matching is only based on the label of the type, and does not consider structure. Performing the initial matching of $\mathbf{P_1}$ to **hammer_plant** results in the mapping in Table 6-1.

M1 : Machine	GenHandle, GenHead, Assembler
P1 : Part	Head, Handle, Hammer
cr : creates	(GenHandle → Handle), (GenHead → Head)

TABLE 6-1 - INITIAL MAPPING OF VARIABLES TO THEIR POTENTIAL MATCH

The initial matching sets **Assembler** as a valid match for **M1** even though **Assembler** has no creates relation, these invalid matches are pruned by performing the pruning techniques discussed in 5.7.1.1. Pruning the mapping in Table 6-1 produces the mapping in Table 6-2.

M1 : Machine	GenHandle, GenHead
P1 : Part	Head, Handle
cr : creates	(GenHandle → Handle), (GenHead → Head)

TABLE 6-2 - MAPPINGS OF VARIABLES AFTER PRUNING

The mappings in Table 6-2 are not valid because the variables are not mapped injectively. The last part of matching $\mathbf{P_1}$ to **hammer_plant** involves creating tables where the variables are mapped injectively. For example, **M1** creates two new tables; one where **M1** is mapped to **GenHandle** and one where it is mapped to **GenHead**. The two tables produced are shown in Table 6-3 and Table 6-4.

M1 : Machine	GenHandle
P1 : Part	Head, Handle
cr : creates	(GenHandle → Handle), (GenHead → Head)

TABLE 6-3 - M1 MAPPED TO GENHANDLE

M1 : Machine	GenHead
P1 : Part	Head, Handle
cr : creates	(GenHandle → Handle), (GenHead → Head)

TABLE 6-4 - M1 MAPPED TO GENHEAD

After the two new tables are produced, they are pruned to eliminate candidates that are no longer valid. The pruning on table Table 6-3 eliminates **Head** as a candidate to **Part** because **GenHead** is no longer a candidate for **M1**, aswell as the create relation between the two. The result of the pruning is shown in Table 6-5. The table shows an injective mapping that is valid, and may be used to generate a two-level rule. The other valid match is found by running the pruning on the mapping in Table 6-4.

M1 : Machine	GenHandle
P1 : Part	Handle
cr : creates	(GenHandle → Handle)

TABLE 6-5 - RESULT OF PRUNING THE MAPPING IN TABLE 6-3

6.2. Proliferated rules

Successfully matching the **META** part of the rule with the model hierarchy results in a list of matches, where one element in the list looks like Table 6-5. A two-level rule must be generated from each element in the list, these rules are refered to as proliferated rules. The **BODY** part of the rule is the key to autogenerating the two-level rules. The **BODY** of a multilevel rule is typed by the variables in the **META** part of the rule, which means that a two-level rule can be constructed by replacing the

variable types in the **BODY** with their matched type. For example, generating a two-level rule from the match in Table 6-5 is done by replacing **M1** with **GenHandle**, **P1** with **Handle**, and **cr** with **creates**. The proliferated rule corresponding to Table 6-5 is shown in Figure 6.4. The other proliferated rule from matching **create_part** to the hammer chain is shown in Figure 6.5.

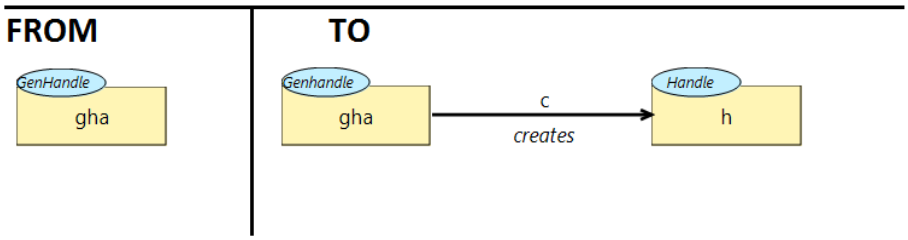

FIGURE 6.4 - GENHANDLE AUTOGENERATED TWO-LEVEL RULE

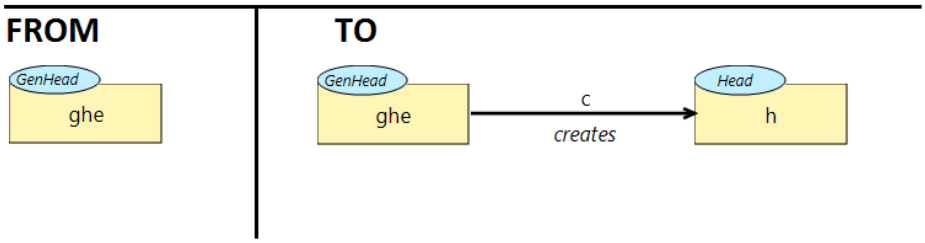

FIGURE 6.5 - GENHEAD AUTOGENERATED TWO-LEVEL RULE

The proliferation of **create_part** resulted in two rules; **GenHandle** and **GenHead**; when matched with the hammer chain. The MCMT rule can be matched with other branches of the hierarchy. Matching **create_part** with the stool chain in Figure 6.1 results in the two rules shown in Figure 6.6 and Figure 6.7.

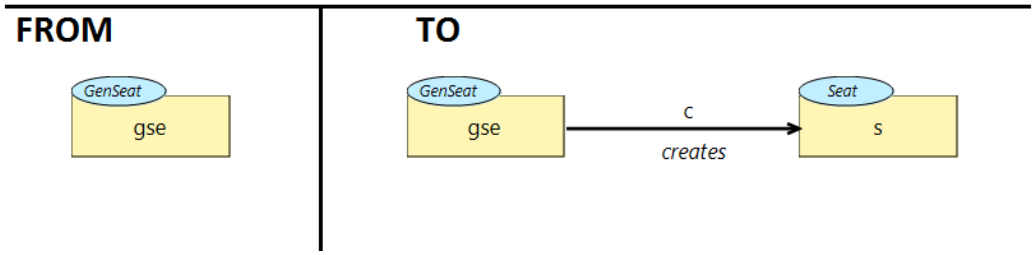

FIGURE 6.6 - GENSEAT AUTOGENERATED TWO-LEVEL RULE

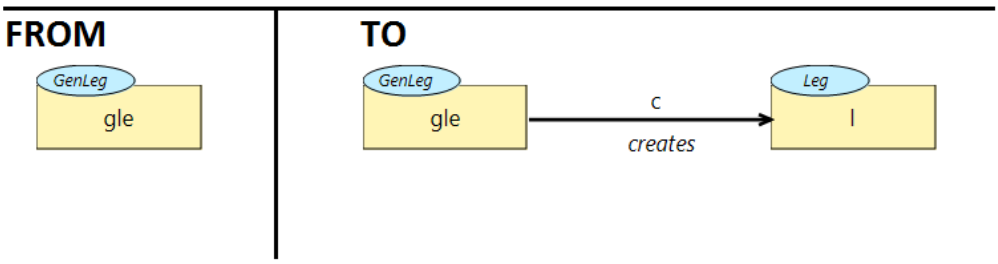

FIGURE 6.7 - GENLEG AUTOGENERATED TWO-LEVEL RULE

In this chapter, we demonstrated how the MCMT rule **create_part** was matched to a model hierarchy, and how a two-level rule was generated from each match of **create_part**. Furthermore, the results showed that a MCMT rule could be matched to many elements in the same system and many elements in different systems as long as they conform to the same typing chain that the MCMT rule is typed over. The re-use of behaviour is achieved through instance inheritance of behaviour which means that the instances should have the the same behaviour that is defined for their type in addition to their own behaviour.

6.3. From MCMT rule to Groove execution

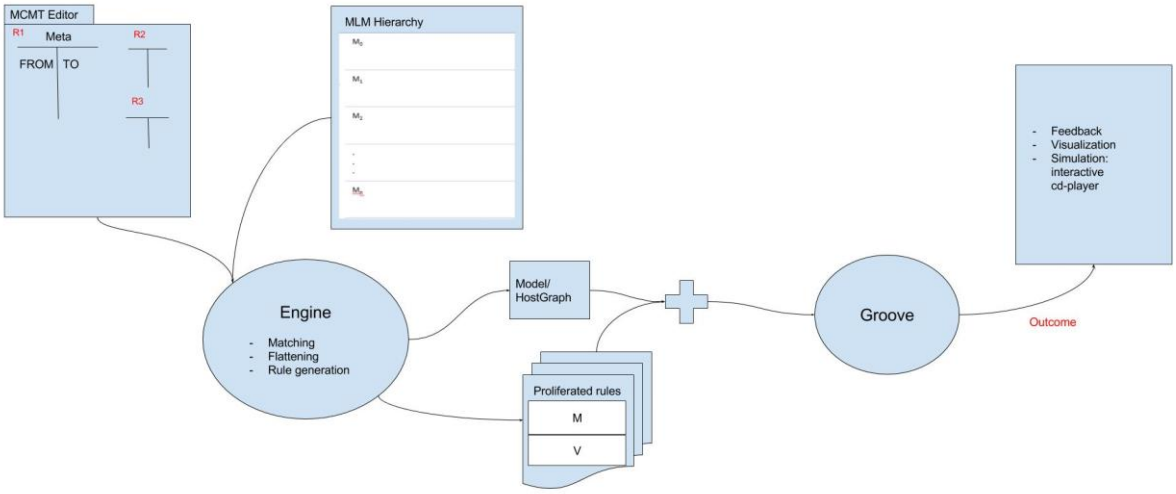

FIGURE 6.8 - FRAMEWORK COMPONENT OVERVIEW

The process starts by writing MCMT rules in the MCMT editor. The editor in the figure has three rules which all follow the same structure with a META, FROM and TO part. Each of these rules are transformed into a set of proliferated rules one at a time. The MCMT rules are matched against the MLM hierarchy which may contain an arbitrary number of metalevels. Only one metalevel is transformed by the rule which is usually the bottom most level and this metalevel is not part of the MCMT matching process. In fact, only the metalevels above the metalevel to be transformed is part of the matching. We refer to the metalevel to be transformed as the model. An arbitrary number of proliferated rules may be generated from a single MCMT rule. Proliferated rules are the rules that

transform the model (metalevel to be transformed). They are executed by a 2-level engine which in our case is Groove. However, any 2-level engine could be used and Groove is easily replacable by modifying the rule generator. We need to translate the model into a hostgraph because the model conforms to a MultEcore metamodel whereas the hostgraph conforms to Grooves metamodel. For example, the model in Figure 6.9 must be transformed into the hostgraph in Figure 6.10 before Groove can apply any proliferated rules on it.

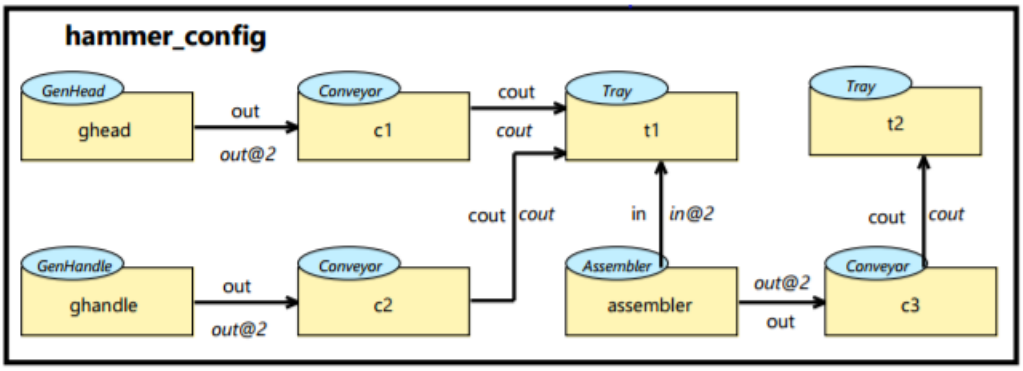

FIGURE 6.9 – CONCRETE SYNTAX OF HAMMER CONFIG IN MULTECORE

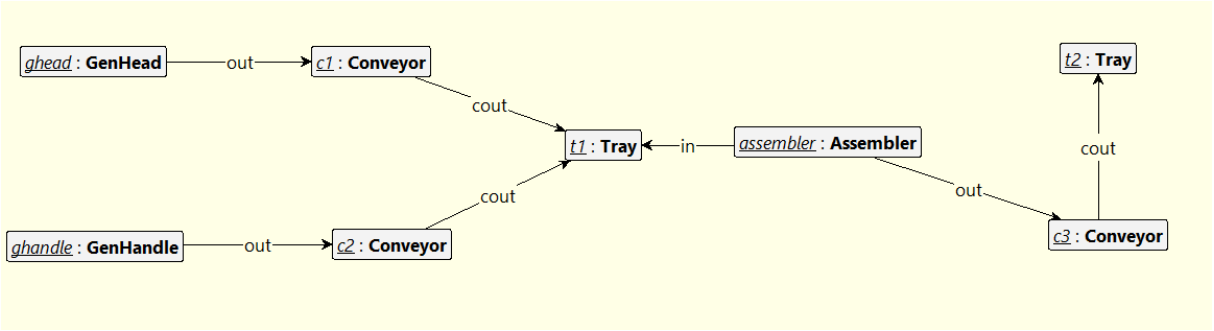

FIGURE 6.10 HAMMER CONFIG AS HOSTGRAPH IN GROOVE

The hostgraph together with the proliferated rules are given to Groove which then transforms the model based on the proliferated rules. The result of applying the proliferated rules onto the hostgraph is a modified hostgraph therefor the last step is to transform the modified hostgraph back into an XMI MultEcore model.

The matching in Groove is similar to our way of matching a single pattern level to a metalevel. In Groove, the left-hand side of the rule is the pattern and the hostgraph is the metalevel. A rule may be applied if its LHS matches the hostgraph. The order of execution is nondeterministic when many rules are applicable to the hostgraph. In other words, the developer has no control over the order that the rules are applied. Groove has two mechanics to work around this: Firstly, the simulator allows the developer to choose which rule he wants to execute. Secondly, assigning rule priority to the rule graphs. Rule priority is an integer value that indicates which rule should be applied first. The rule with the highest value is executed first, but if two rules have an equally highest value then the order between them is nondeterministic. We currently do not support priority in our MCMT rules,

but it should be straight forward to implement. One way would be to include a keyword for prioritiy in the DSL that assigns a value for a given MCMT rule and then carry this value over to the proliferated rules generated by the MCMT rule.

The engine would usually execute all proliferated rules and then return the result, but often times we would like to observe the the steps that produced the final result. That is, we would execute one rule at a time and instantly get feedback by viewing the updated model. That way, we could detect at which step an error occurs. The simulation can be interactive or like a cd-player. The interactive approach lets the developer choose which rule should be applied, and then the result of applying the rule is presented to him. The cd-player approach lets the developer decide when the next rule should be applied, but it does not let him control which rule should be applied. The cd-player approach gives him a more realistic picture because the order is non-deterministic.

As we mentioned earlier, the hostgraph is a graph representation of the model. A hostgraph in Groove could be typed over a typegraph. Typegraphs are optional, but when a typegraph is active it works as a metagraph to the hostgraphs and rulegraphs in Groove. An active typegraph constrains the hostgraph and rulegraphs to only use types that are defined in the typegraph. One could say that the relation between the hostgraph and the typegraph is parallel to that of a model and its metamodel. However, the typegraph is optional whereas the metamodel is not. Furthermore, many typegraphs can be active at the same time in which case the allowed types in the hostgraph is the union of the active typegraphs.

The typegraphs make sure that target hostgraphs from applying rules to a source hostgraph are valid. This is done by enforcing that the source hostgraph and the rulegraphs conform to the typegraph. In otherwords, the source hostgraph is valid to begin with and no rule produces a node or edge that is not allowed by the typegraph, or deletes an element that is mandatory in the typegraph. Moreover, creation of edges between nodes only happens if the metatypes of these nodes have an edge between them.

We do not need to use typegraphs in Groove to ensure that we produce valid target models. We already know that our source model is conforming to the model hierarchy, and we know that the proliferated rules are generated by an MCMT rule which in turn is typed by the model hierarchy. The proliferated rules are generated from types in the model hierarchy, and must therefor be typed by the model hierarchy. We can encode pre-conditions in the rules to make sure that constraints are not violated when applying the generated rules to our models. The pre-conditions can be encoded in the generated rules automatically by looking up the constraints of the types in the hierarchy and transforming those constraints into pre-conditions. Generating pre-conditions automatically would require a considerable amount of work. Alternatively, the constraints can be manually added to the MCMT rules. The drawback of that approach is that the same constraints might have to be specified in multiple MCMT rules, and if the constraint changes then multiple MCMT rules needs to be updated.

However, if we want to use a typegraph in Groove to control the validity of our target models then we would need to translate our model hierarchy into a typegraph. Figure 6.11 illustrates the use of typegraps in Groove. The typegraph is not multilevel so we can not map the hierarchy directly into a typegraph. We must flatten the model hierarchy before performing the translation. We would do the flattening by creating an empty metamodel and then add a package for each level in the hierarchy. Translating the flattened metamodel into a typegraph is done in the same way as a model is transformed into a hostgraph, except that we must do the translation for each package in the metamodel and then activate all of the generated typegraphs.

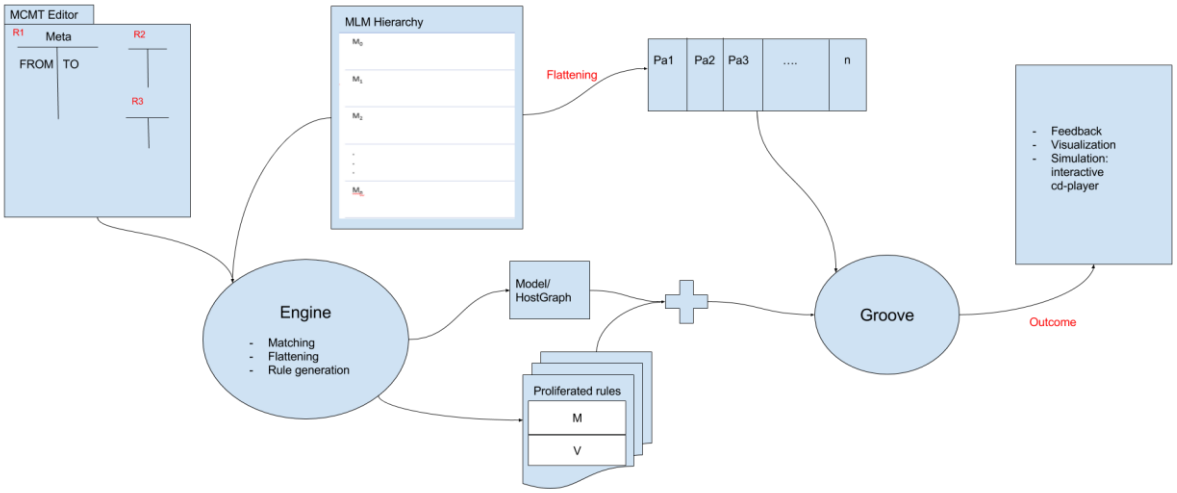

FIGURE 6.11 - OVERVIEW OF ENGINE WITH FLATTENING OF MODEL HIERARCHY APPROACH

We went with the approach to check the conformance of the proliferated rules in the preprocessing step and omit the flattening of the model hierarchy. Our reasoning is that this could allow us to detect/remove bad rules before they are sent to Groove. Moreover, the translation of the model hierarchy into the typegraph might have to be repeated when using a different engine. Which means that we would have to modify the translater everytime we used a different engine. In contrast, performing the conformance check on the proliferated rules is independent on the underlying transformation engine, and therefor replacing Groove with another engine would require no additional changes. Lastly, we have all the information from the model hierarchy in the preprocessing step, whereas some information could be lost when translating the model hierarchy into a typegraph hence we have more control in the preprocessing step.

7. Conclusion

In this chapter, we summarise our achievements and shortcommings. We give suggestions to what can be done in the future to improve our work.

7.1. Summary

In this thesis, we have created a multilevel model transformation engine that takes as input a model hierarchy, and a multilevel coupled transformation (MCMT) rule and performs higher order transformations to transform the MCMT rule into a several two-level transformation rules. We generate our MCMT into Groove rules, but the relation between the flattened rules and Groove is independent, and we may replace Groove with another engine by creating a bridge between the proliferated rules and the new engine. The bridge would be a code generator that translates proliferated rules into the transformation language used by the engine of our choice. The prolifered rules are equivalent to all the ways a multilevel model can be matched by an MCMT rule when using a true multilevel engine that supports direct manipulation of multilevel models (no flattening). The MCMT rules can indeed be reused by different modelling languages as one can change to a different branch of the model hierarchy and match it against the MCMT rule without making any changes to the MCMT rule. Matching the new branch to the unchanged MCMT rule produces new two-level rules that is applicable to models of the new branch. That answers the first research question which asks if we can reuse behaviour across different modeling languages, and the answer is yes.

However, we cannot answer the second research question yet because the engine does not keep track of the whole typing chain and therefore does not match variables of the MCMT rules to indirect types of the variables. For example, given the typing chain {**EClass**, **Consumables**, **Food**, **Meat**, **Cow**} Then a MCMT rule has a variable **X** of type **Consumables** then only **Food** can be matched to the rule, but in reality, **Meat** and **Cow** should also be matched by the engine.
This is a limitation of the engine which is already solved in theory. One solution is to keep track of the types typing chain, and then see if the typing chain of the element we are visiting contains the type of the current variable we are trying to match. For example, if we are trying to find a match for variable **X** of type **Consumables**, and we are matching it against **Meat**, then we look up the typing chain of **Meat** and see if **Consumables** is contained in this typing chain.
Another solution is to recursively visit the type of the element we are visiting until we reach the type EClass or find the type we are looking for. On each visit, we compare the type of the variable with the type of the element we are visiting.
The first approach is preferable as it does not require one to traverse model levels when matching one level to another, and it is straightforward to implement.

The main contribution of this thesis is the work on multi level matching. That is matching a hierarchy specified in the rule against a model hierarchy. A valid match is found if all levels in the rule hierarchy matches a level in the model hierarchy. Ullmann's algorithm is used to determine if a level in the rule hierarchy matches a level in the model hierarchy. The process is repeated until every level in the rule hierarchy has been matched against every level in the model hierarchy. At which point, the algorithm has found every valid match for the two hierarchies. Multilevel matching is a vital part of multilevel model transformations, and the work done in this thesis could be useful to anyone developing a multilevel transformation engine. In particular, the matching algorithm could be used as the basis

when developing a new multilevel transformation engine that directly manipulates multilevel models.

7.2. Further work

Add support for more complex transformation such as *for all* and *exists* statements and arithmetic operations. Currently only simple transformations are supported such as create, read, update and delete. Additionally, the engine does not consider the attributes of a class and potency is not supported. The reason for that was to keep the engine simple while focusing on the algorithms, but the engine should be updated to support these features.

Currently the produced groove rules are imported to Groove manually, and then run inside of Groove. This should be done automatically executing the generated rules through an API to Groove. Moreover, the MultEcore models to transform are recreated inside of Groove as hostgraphs manually, but this should be done automatically. The reverse transformation of hostgraphs into MultEcore models should also be done automatically after the rules have been applied to the hostgraph.
Future work includes making the entire process fully automatic which means that groove rules and hostgraphs are hidden from the user. Thereby, the user only needs to focus on creating MCMTs and MultEcore models.

In the aftermath Groove was not as great as initially thought out. Mostly because it uses a native representation of the model that it performs the transformations on. The ideal scenario would be that Groove transformed EMF models directly. Furthermore, the import and export functionalities in Groove are glitchy and we were better off creating our own translation.

If we had to do it over again we would probably have used Tefkat or Henshin as the underlying engine. The reason is that they perform transformations directly on EMF models, and that is very beneficial because then we don’t have to transform our EMF models into a temporary model to perform transformations on and then transform it back each time we want to do model transformations. Furthermore, Tefkat and Henshin are plugin to eclipse and therefore they are easier to integrate into our project. That said, we believe that Groove would have been the better choice if it supported direct transformation of EMF models. We did not realize the importance of this feature in the start. One drawback of the EMF tools is that the multilevel hierarchy needs to be transformed into a flat metamodel containing all the types. However, this only needs to be done once every time the model hierarchy is changed. That is, we can transform many models without regenerating the flattened model hierarchy if no changes have been made. Groove on the other hand, needed to transform the model into another format each time we wanted to transform the model. Further work could include trying to use the multilevel engine with a different underlying engine.

References

[1] J. Backus, “THE HISTORY OF FORTRAN I, II, AND III,” *IEEE Annals of the History of Computing 1(1),* vol. 1, no. 1, pp. 21-37, 1979.

[2] A. Rossini, Diagram Predicate Framework meets Model Versioning and Deep Metamodelling, Bergen: University of Bergen, 2011.

[3] Eugene Kindler, Ivan Krivy, “Object-oriented simulation of systems with sophisticated control,” *International Journal of General Systems,* vol. 40, no. 3, pp. 313-343, 2011.

[4] D. C. Schmidt, “Guest Editor's Introduction: Model-Driven Engineering,” *Computer,* vol. 39, no. 2, pp. 25-31, 2006.

[5] Parastoo Mohagheghi, Vegard Dehlen, “Where Is the Proof? - A Review of Experiences from Applying MDE in Industry,” *European Conference on Model Driven Architecture - Foundations and Applications,* vol. 5095, pp. 432-443, 2008.

[6] J. Whittle, J. E. Hutchinson, M. Rouncefield, “The state of practice in model-driven engineering,” *IEEE Software,* vol. 31, no. 3, pp. 79-85, 2014.

[7] Parastoo Mohagheghi, Magne Jørgensen, “What contributes to the success of IT projects?: success factors, challenges and lessons learned from an empirical study of software projects in the Norwegian public sector.,” *Proceedings of the 39th International Conference on Software Engineering Companion,* pp. 371-373, 2017.

[8] J. D. Haan, “10 Misperceptions and challenges of Model Driven Development,” 21 January 2009. [Online]. Available: http://www.theenterprisearchitect.eu/blog/2009/01/21/10-misperceptions-and-challenges-of-model-driven-development/.

[9] A. R. d. Silva, “Model-driven engineering: A survey supported by the unified conceptual model,” *Computer Languages, Systems & Structures,* vol. 43, pp. 139-155, 2015.

[10] Marco Brambilla, Jordi Cabot, Manuel Wimmer, Model-Driven Software Engineering in Practice: Second Edition, Morgan & Claypool, 2017.

[11] Tomaž Kosar, Nuno Oliveira, Marjan Mernik, Maria João Varanda, Matej Črepinšek, Daniela da Cruz, Pedro Rangel Henriques, “Comparing General-Purpose and Domain-Specific,” University of Maribor, Faculty of Electrical Engineering and Computer Science, Maribor, Slovenia, 2000.

[12] M. S. Puccini, “Executable models for Extensible Workflow Engines,” University of los Andes, Bogotá, 2011.

[13] F. Macias, “MultEcore,” Bergen university college, [Online]. Available: http://prosjekt.hib.no/ict/multecore/.

[14] T. Kühne, “Contrasting classification with generalisation,” in *APCCM '09 Proceedings of the Sixth Asia-Pacific Conference on Conceptual Modeling*, Wellington, New Zealand, 2009.

[15] Marco Brambilla, Jordi Cabot, Manuel Wimmer, Model-Driven Software Engieering in Practice 1st edition, Morgan & Claypool Publishers, 2012.

[16] “www.obeodesigner.com,” Obeo, [Online]. Available: https://www.obeodesigner.com/en/product/sirius.

[17] “Xtext,” [Online]. Available: https://eclipse.org/Xtext/index.html.

[18] Adrian Rutle, Wendy MacCaull, Hao Wang, Yngve Lamo, “A metamodelling approach to behavioural modelling,” in *Proceedings of the Fourth Workshop on Behaviour Modelling - Foundations and Applications*, Kgs. Lyngby, 2012.

[19] A. Rutle, Diagram Predicate Framework, A Formal Approach to MDE, Bergen: University of Bergen, 2010.

[20] Alessandro Rossini, Juan de Lara, Esther Guerra, Adrian Rutle, Uwe Wolter, “A Formalisation of Deep Metamodelling,” *Formal Aspects of Computing,* vol. Volume 26, no. Issue 6, p. 1115–1152, 2014.

[21] Colin Atkinson, Thomas Kühne, “Rearchitecting the UML infrastructure,” *ACM Transactions on Modeling and Computer Simulation (TOMACS),* vol. 12, no. 4, pp. 290-321, 2002.

[22] “Wikipedia: Meta-Object_Facility,” [Online]. Available: https://en.wikipedia.org/wiki/Meta-Object_Facility.

[23] Juan de lara, et al, “Model-Driven Engineering with Domain-Specific Meta-Modelling Languages,” in *Proceedings of the 8th European conference on Modelling Foundations and Applications*, Kgs. Lyngby, Denmark, 2012.

[24] Colin Aktinson, Thomas Kühne, “The Essence of Multilevel Metamodeling,” *≪UML≫ 2001 — The Unified Modeling Language. Modeling Languages, Concepts, and Tools,* vol. 2185, pp. 19-33, 2001.

[25] Fernando Macias, Uwe Wolter, Adrian Rutle, Francisco Duran, Roberto Rodriguez-Echeverria, “Multilevel Coupled Model Transformations for Precise and Reusable Definition of Model Behaviour,” 2017.

[26] Francisco Durán, Antonio Moreno-Delgado, Fernando Orejas, Steffen Zschaler, “Amalgamation of domain specific languages with behaviour,” *Journal of Logical and Algebraic Methods in Programming,* vol. 86, no. 1, pp. 208-325, 2015.

[27] Anneke Kleppe, Jos Warmer, Wim Bast, MDA Explained: The Model Driven Architecture : Practice and, Boston: Addison Wesley, 2003.

[28] “WIkipedia: ATLAS_Transformation_Language,” [Online]. Available: https://en.wikipedia.org/wiki/ATLAS_Transformation_Language.

[29] Macias, Rutle, Stoltz, “Formalisation of Flexible Multilevel Modelling,” *Emisa,* 2017.

[30] Marc Andries, Gregor Engels, Gabriele Taentzer, et al., “Graph Transformation for Specification and Programming*,” *Science of Computer Programming,* vol. 34, no. 1, pp. 1-54, 1999.

[31] H. Wiemann, “Theory of Graph Transformations,” University of Bremen, Bremen, 2005.

[32] H. Ehrig, R. Heckel, M. Korff, M. Löwe, L. Ribeiro, A. Wagner, A. Corradini, “Algebraic approaches to graph transformation: Part II: Single pushout approach and comparison with double pushout approach,” *Handbook of Graph Grammars and Computing by Graph Transformation,* vol. 1, pp. 247-312, 1997.

[33] P. Barvik, “Model to Model Transformation Tool for the DPF Workbench,” Bergen University College, Bergen, 2014.

[34] “Melanee,” [Online]. Available: http://www.melanee.org/.

[35] “MetaDepth,” [Online]. Available: http://metadepth.org/.

[36] Nafiseh Kahani, James R. Cordy, “Comparison and Evaluation of Model Transformation,” Queen’s University, Kingston, Ontario, 2015.

[37] Tom Mens, Krzysztof Czarnecki, Pieter Van Gorp, “A Taxonomy of Model Transformations,” *Electronic Notes in Theoretical Computer Science (ENTCS),* vol. 152, pp. 125-142, 2006.

[38] “Tefkat homepage,” [Online]. Available: http://tefkat.sourceforge.net/.

[39] Arend Rensink, Iovka Boneva, Harmen Kastenberg, Tom Staijen, User Manual for the GROOVE Tool Set.

[40] Jon Oldevik, Oystein Haugen, “Higher-Order Transformations for Product Lines,” in *Software Product Line Conference*, Kyoto, 2007.

[41] Gabriel Valiente, Conrado Martinez, “An algorithm for graph pattern-matching,” *In Proc. 4th South American Workshop on String Processing, volume 8 of Int. Informatics Series,* pp. 180-197, 1997.

[42] Michael R. Garey, David S. Johnson, Computers and Intractability: A Guide to the Theory of NP-Completeness, New York: Freeman, 1979.

[43] M. W. Saltz, A Fast Algorithm for Subgraph Pattern Matching on Large Labeled Graphs, Master's thesis, Athens: University Of Georgia , 2013.