Universiteit Antwerpen

Master Thesis

Model Transformation for Modelling Language Evolution

Author:
Simon Van Mierlo

Promoter:
Prof. Dr. Hans Vangheluwe

Supervisor:
Bart Meyers

A thesis submitted in fulfilment of the requirements for the degree of Master of Science: Computer Science

in the

Modelling, Simulation and Design Lab
Department of Mathematics and Computer Science

June 2013

UNIVERSITEIT ANTWERPEN

Abstract

Faculty of Science
Department of Mathematics and Computer Science

Master of Science: Computer Science

Model Transformation for Modelling Language Evolution

by Simon Van Mierlo

Model-Driven Engineering (MDE) is a recent approach to designing, developing, and maintaining large, software-intensive, and safety-critical systems. In MDE, a domain expert encodes his knowledge in models, which are abstractions of the problem domain. Using domain concepts instead of computation concepts as central entities in the development process increases the understandability and maintainability of software artefacts. Current MDE solutions allow a language expert to create modelling languages and allow domain experts to encode their knowledge in models and transformations in these modelling languages, using a textual or visual concrete syntax. Modelling languages are a faithful abstraction of the problem domain; if the domain changes, the modelling language needs to be adapted as well. Modelling languages are often assumed to be static, however, and most MDE solutions do not explicitly support modelling language evolution. Modelling language adaptations could invalidate existing modelling artefacts and leave the MDE system in an inconsistent state. It is therefore critical for the advancement of MDE that a systematic, efficient, and semi-automatic method supporting modelling language evolution is developed. In the last decade, the co-evolution of modelling languages and modelling artefacts has been researched extensively. In this thesis, we propose a novel approach to support co-evolution, which uses model transformations to co-evolve modelling languages and their artefacts. The proposed approach supports modelling language evolution by explicitly modelling evolution and co-evolution operations as model transformations and supports both model and transformation co-evolution through the use of higher-order transformations. The solution is implemented and integrated in Ark, the metamodelling kernel for the MDE tool AToMPM, which is currently under development in the MSDL research group.

UNIVERSITEIT ANTWERPEN

Samenvatting

Faculteit Wetenschappen
Departement Wiskunde-Informatica

Master in de Wetenschappen: Computerwetenschappen

Modeltransformatie voor de Evolutie van Modelleertalen

door Simon Van Mierlo

Model-Driven Engineering (MDE) is een recente methode voor het ontwerpen, ontwikkelen en onderhouden van grote, software-intensieve en veiligheidskritische systemen. In MDE codeert een domeinexpert zijn kennis in modellen, die abstracties van het probleemdomein vormen. Door het gebruik van domeinconcepten als centrale entiteiten in het ontwikkelingsproces in plaats van implementatieconcepten verhoogt de begrijpbaarheid en onderhoudbaarheid van de software-artefacten. Huidige MDE oplossingen ondersteunen het creëren van modelleertalen, en maken het mogelijk voor een domeinexpert om zijn kennis te coderen in modellen en tranformaties in deze modelleertalen, gebruik makende van een tekstuele of visuele concrete syntax. Modelleertalen zijn een trouwe abstractie van het probleemdomein; als het domein wijzigt, moet de modelleertaal ook aangepast worden. In de meeste MDE oplossingen worden modelleertalen echter als statische entiteiten beschouwd en er is vaak geen expliciete ondersteuning voor de evolutie van modelleertalen. Wijzigingen aan een modelleertaal kunnen bestaande artefacten ongeldig maken, wat het MDE systeem in een inconsistente staat brengt. Het is daarom van kritiek belang voor de verdere vordering van MDE dat een systematische, efficiënte en semi-automatische methode wordt ontwikkeld die de evolutie van modelleertalen ondersteund. In het voorbije decennium is de co-evolutie van een modelleertalen en bijhorende modelleerartefacten uitgebreid onderzocht. In deze thesis stellen wij een nieuwe benadering voor die deze co-evolutie ondersteund, door gebruik te maken van modeltransformaties om modelleertalen en hun artefacten te co-evolueren. De voorgestelde benadering ondersteund de evolutie van modelleertalen door het expliciet modelleren van evolutie en co-evolutie operaties als modeltransformaties en ondersteunt zowel model- als transformatie co-evolutie door het gebruik van hogere orde transformaties. De oplossing is geïmplementeerd en geïntegreerd in Ark, de metamodelleerkernel van de MDE tool AToMPM, die op dit moment ontwikkeld wordt binnen de onderzoeksgroep MSDL.

Acknowledgements

First of all, I would like to thank my promoter, Hans Vangheluwe. In the last two years, he has been my mentor for both my research internships at the MSDL research group and my thesis. He has introduced me to the academical world, sparked my interest in Model-Driven Engineering, and has given me the opportunity to contribute to interesting research projects.

I would also like to thank Bart Meyers, whom I could always count on to answer my questions and help me whenever I needed it. He has been a great source of knowledge and I could always count on him to review my work.

Lastly, I would like to thank my friends and family, who have been a great support for me throughout my study career.

Contents

List of Figures

Abbreviations

MDE	**M**odel-**D**riven **E**ngineering
DSL	**D**omain-**S**pecific **M**odelling
DSML	**D**omain-**S**pecific **M**odelling **L**anguage
HOT	**H**igher-**O**rder **T**ransformation
LHS	**L**eft-**H**and **S**ide
LHS	**R**ight-**H**and **S**ide
NAC	**N**egative **A**pplication **C**ondition
CRUD	**C**reate, **R**ead, **U**pdate and **D**elete
Ark	**A**ToMPM **r**eusable **k**ernel
ArkM3	**A**ToMPM **r**eusable **k**ernel **M**eta-**M**eta-**M**odel

Chapter 1

Introduction to Modelling Language Evolution

1.1 Model-Driven Engineering

The systems which are analysed, designed and constructed today are characterized by an ever increasing complexity. While development and production costs must be kept minimal, the demand on quality grows. Model-Driven Engineering (MDE) [1] is a recent approach to tackle the inherent complexity faced when designing and constructing large and complex software-intensive systems. MDE raises the level of abstraction by focusing on domain concepts, rather than implementation or computing concepts. In doing so, it strives to lower the 'accidental' (as opposed to 'essential') complexity [2] by capturing only the essence of a problem. In particular, in Domain-Specific Modelling (DSM) [3], domain experts encode their knowledge in *models* in a Domain-Specific Modelling Language (DSML) which separates domain knowledge from implementation knowledge. The grammatical structure of a modelling language is defined by a *metamodel*, which contains the concepts of the language and the relations between them. This metamodel defines the abstract syntax and static semantics of the modelling language and must be accompanied by a definition of the concrete syntax, which can be textual or visual. Instance models of the metamodel (which are models created in the modelling language defined by the metamodel) are said to *conform to* the metamodel. The *dynamic semantics* of a language are defined by *model transformations*, which transform a model

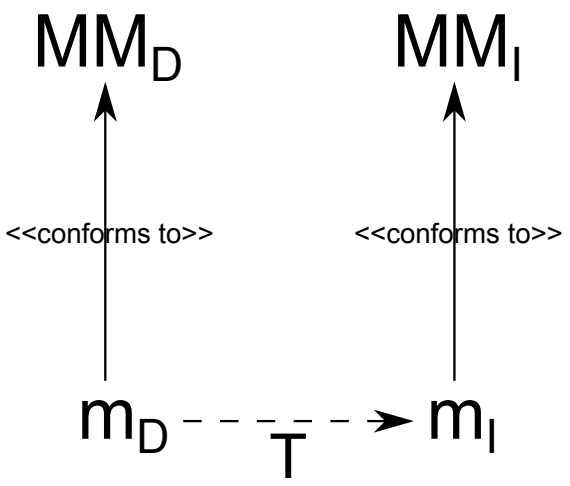


FIGURE 1.1: An example of a modelling system.

into another model. Once the semantics of a modelling language are defined, models created in that language can be used, amongst others, for verification of properties [4], automatic test case generation [5], and code synthesis [6].

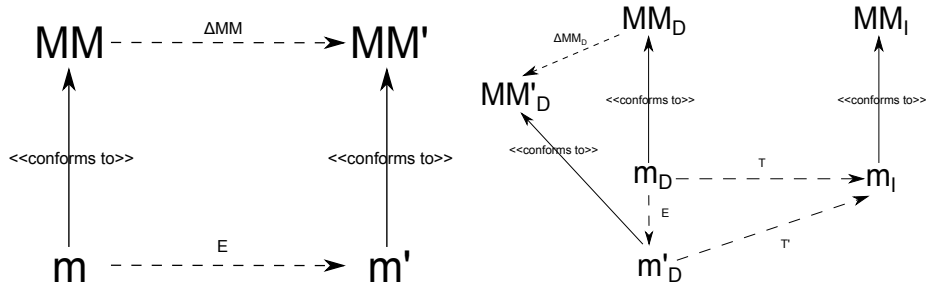
Figure 1.1 shows a typical example of a modelling system. It consists of two modelling languages, whose syntax and static semantics are defined by the metamodels MM_D and MM_I. Two models, m_D and m_I, conform to these metamodels and are related by the transformation T. This transformation can either be *endogenous* when MM_D and MM_I are the same metamodel or *exogenous*, when they differ [7]. An endogenous transformation is typically used to build a simulator, able to generate execution traces of the modelling system. An exogenous transformation is used to attach semantics to a model by transforming the model to a model in a formalism with known semantics (such as PetriNets [8]) or for full code generation, where program code is regarded as a model written in a general-purpose programming language and conforming to the grammar of that language. An important note to make is that a transformation can be regarded as a special kind of model, conforming to a transformation metamodel. This enables the transformation of transformations by so-called Higher-Order Transformations (HOTs). This is discussed in detail in Chapter 5.

1.2 Modelling Language Evolution

In software engineering, evolution is common. Requirements, data, programs, ... may all evolve during the life cycle of a system [9]. In MDE, evolution occurs at the level of models, when models that comprise a system are adapted to meet the project demands. Evolution also occurs at the level of modelling languages, when the metamodel of the language is adapted. This is contrary to general-purpose programming languages, which rarely evolve, and if they do, they avoid evolving in a non-breaking way. Functions which are no longer supported are often marked as *deprecated*, which signals users of the language that the function should no longer be used. Deprecated functions still work, to avoid breaking programs which are no longer maintained. A DSML has to be a faithful abstraction of the problem domain. If the problem domain or the implementation target domain changes, it must be reflected in the DSML. A DSML is often created for a specific application (such as the process management system of a factory, or a management application for rail road networks) and as such is closely tied to the development process of the application. It is therefore necessary and useful to make breaking changes to these languages and fix the problems that arise in existing artefacts. A DSML should also be kept minimal, to avoid keeping concepts in the language which are no longer in the domain and to decrease maintenance costs.

Changes made to a modelling language can break the conformance relation between models created in the initial version of the language and the evolved metamodel. To bring the modelling system in a consistent state, these conformance relations have to be repaired. This is called the (syntactic) *co-evolution* of models and metamodels, as models have to be *co-evolved* together with their metamodels. The term *co-evolution* emphasises the simultaneous nature of the evolution process: models and metamodels are evolved *at the same time*. The term *model migration* is used when models are migrated to the new version of the language *after* language evolution has taken place.

Syntactic evolution brings the modelling system in a consistent state, where each model conforms to its metamodel. As we explained in the previous section, the *semantics* of a modelling language are defined by a transformation, also called a semantic mapping function. This function maps each model of a modelling language to a model in a semantic domain. This mapping function allows us to evaluate certain properties of the model, which are equal to the properties evaluated in its semantic domain. When the original



(a) A metamodel adaptation and model co-evolution example

(b) A domain metamodel adaptation, resulting in the co-evolution of a conforming model and a transformation

FIGURE 1.2: Two language evolution scenarios.

model is migrated, those properties should stay the same, except for those semantic changes which were introduced on purpose as part of the evolution of the language. When two versions of a modelling system are (1) equal, modulo their intended syntactic and semantic changes and (2) syntactically consistent, the evolution of a system is said to be *continuous* [10]. Only continuous evolutions of a system are useful evolutions, as no unwanted alterations to the semantics of the system are made.

Figure 1.2 shows two example evolution scenarios. In Figure 1.2(a), a metamodel MM is evolved to metamodel MM'. In both cases, the adaptations are captured in the *difference model* ΔMM. For each change performed on the original metamodel, a suitable migration operation has to be found for all modelling artefacts. All migration operations are captured in a *migration transformation*. In Figure 1.2(a), this transformation is called E, which makes sure model m conforms to the new version of the metamodel.

Figure 1.2(b) depicts a more involved evolution scenario. The modelling system consists of two modelling languages, whose syntax and static semantics are defined by the metamodels MM_D and MM_I. A transformation T transforms models conforming to the former into models conforming to the latter. The domain metamodel of the transformation, MM_D, is adapted, resulting in a new version of the metamodel called MM'_D. As in Figure 1.2(a), the models conforming to the original metamodel have to be adapted in order to repair the conformance relation. The transformation T also has to be migrated, however, as its domain metamodel has changed. The migrated transformation is called T' and has been adapted to the new version of the metamodel.

1.3 Research Agenda

The research goal of this thesis is to provide techniques for modelling language evolution, by making extensive use of model transformation. The aim is to support both metamodel and model co-evolution as well as metamodel and transformation co-evolution. Our focus will be on providing tool support to explicitly model evolution and co-evolution operators. As such, we can formulate one central research goal:

To construct a metamodelling framework, capable of explicitly modelling and executing model transformations and to use this framework to support the evolution of modelling systems which consists of metamodels, models, and transformations, by modelling migration operators and combining them into a migration transformation, which can either be a regular model transformation or a higher-order transformation.

The remainder of this thesis is constructed as follows. Chapter 2 details the current state-of-the art in metamodel and model/transformation co-evolution. It presents the solutions which have been proposed, along with their advantages and disadvantages. At the end of the chapter, proposals for future research are made, some of which have been included in our research goal. Chapter 3 presents a running example which is used to demonstrate the techniques presented in this thesis. Chapter 4 introduces Ark, our metamodelling framework of choice. Chapter 5 explains how model transformations are developed in Ark. Chapter 6 explains how the example modelling language evolution scenario is implemented in Ark. Lastly, Chapter 7 concludes the thesis and makes suggestions for future work.

Chapter 2

Related Work

The evolution of modelling languages and the co-evolution of modelling artefacts has been studied extensively in the last decade. In this chapter, an exhaustive literary review of these studies is presented. It is an unmodified reproduction of our work in [11] and is included in this thesis for completeness.

2.1 A Classification of Solutions

The solutions proposed for co-evolving metamodels, models and transformations differ significantly. In this section, we propose a set of criteria to categorize and differentiate between these solutions. This will allow us to decide where current solutions lack functionality or further validation of these solutions is needed and make suggestions for future work. The first criterion, which is discussed in Section 2.1.1, is how authors categorize the adaptations performed on a metamodel. If such a categorization is present, it is based on how the metamodel is adapted and what the consequences are for the related models and transformations. If a categorization is used in the paper and a solution is proposed, we will mention which of the adaptations the authors acknowledge are supported in their solution. Section 2.1.2 lists which migrations the proposed solutions support, which is either model migration, transformation migration or both. In Section 2.1.3 the automatibility and completeness of each solution will be discussed. This is more often than not a trade-off: either the solution is complete but not fully automatic, or the solution is fully automatic but not complete. Section 2.1.4 discusses

how intention (i.e. properties) of models are preserved during migration. Section 2.1.5, lastly, looks at tool support of the proposed solutions. This will prove to be one of the most important criteria of all, because a proposed solution cannot be properly tested without a tool that implements it.

2.1.1 Metamodel Adaptations

Classifying metamodel adaptations is done by most authors that propose solutions for the co-evolution problem. On the one hand, it is useful as a way to determine the effects of different types of adaptations on related artefacts like models and transformations, which leads to different types of actions to migrate these artefacts to the new version of the metamodel. On the other hand, it forces authors to acknowledge which types of adaptations are supported by the proposed solution. If no classification is present, it is no trivial task to discover whether the solution supports all possible adaptations performed on a metamodel. In the next paragraph the classifications found in the literature are discussed. If a solution uses a certain type of classification, we will also discuss whether all types of adaptation are supported.

1. Non-Breaking and Breaking Changes: In [12] and [13], Gruschko et al. introduce a classification of metamodel adaptations into three categories: **Not Breaking Changes**, **Breaking and Resolvable Changes** and **Breaking and Unresolvable Changes**. The first category consists of adaptations which occur in metamodels but do not break the conformance relation between models and the metamodel. As such, models that conform to the initial version of the metamodel also conform to the new version of the metamodel. An example of this type of adaptation is the addition of a non-mandatory class in the metamodel, as all models not containing an instance of this class conform to the new version of the metamodel as well. The second category consists of adaptations which breaks the conformance relation between the models conforming to the initial version of the metamodel and the new version of the metamodel, but can be resolved by automatic means. An example adaptation is the renaming of a class. This change results in all models containing an instance of the renamed class to not conform to the new version of the metamodel any more. It is however possible to create an automatic renaming mechanism that renames all instances of the renamed class to

reflect the name change. The last category consists of adaptations that break the conformance relation between models and metamodel and are not resolvable automatically. An example change is the addition of a mandatory class to the metamodel. None of the models conforming to the initial version of the metamodel will contain an instance of this class, which means an instance of the class will need to be added to every model. However, we cannot automatically decide how and where to add an instance of a class and as such a developer will have to manually edit each model.

This change classification has become a popular choice for authors to use and is the basis for many of the proposed solutions. In the paper by Gruschko et al. a solution is proposed which uses the change classification to take different types of actions based on the type of metamodel adaptation. It consists of a change recording phase in which a difference model is built, a classification phase in which the adaptations represented in the change model are classified in the categories presented here and a manual edit phase for unresolvable changes. After that a transformation is constructed which migrates instance models. The solution proposed by the paper thus supports all adaptations using the categorization the authors have introduced.

In [14], Cicchetti et al. use a similar approach. There, a difference model is constructed as well and the adaptations are classified. However, this classification leads to two non-overlapping difference models which are used to construct two migrations (one automatically, the other with the intervention of the user) which are executed concurrently to migrate models. This technique can be applied if all adaptations in the two difference models are independent of each other. If a dependency exists, certain migration steps need to be executed before others. The authors resolve this problem in [15], which makes this solution complete in the sense that it supports all categories of changes.

The classification has also been used for solutions that migrate transformations. In [16], Garcia and Díaz use the classification to support transformation migration. A difference model is created to represent the metamodel adaptations, which is used by an ATL higher-order transformation to migrate the transformation. All categories of changes are discussed and supported by their solution.

In [10], Meyers and Vangheluwe build a framework that supports the evolution of modelling languages in general. This encompasses metamodels, models, transformations and their visual representations. They make use of the classification proposed by Gruschko and support all of the categories of changes. They achieve this by breaking down the evolution of languages into primitive scenarios, which encompass the consequences of

evolution and the required remedial actions. By combining these primitives in a high level framework, all possible evolutions are supported.

Van Den Brand et al. extend the original classification in [17] by dividing **Breaking and Unresolvable Changes** into **Breaking and Semi-Resolvable Changes** and **Breaking and Human-Resolvable Changes**. The first category encompasses metamodel adaptations that cannot be resolved automatically but can be resolved by configuring the evolution process. The adaptations in the second category can only be resolved by a user in a differences-resolution environment. All of these changes are supported by the solution proposed in the paper.

2. Metamodel and Model Independent and Specific Changes: In [18], Hermannsdoerfer et al. introduce COPE, a tool used to incrementally evolve a domain-specific language by employing coupled operators. They use a change classification which is similar to the one described in the previous paragraph. Metamodel adaptations are classified into four categories: **Metamodel Only Changes**, **Metamodel Independent Changes**, **Metamodel Specific Changes** and **Model Specific Changes**. The first category encompasses changes that only have an effect on metamodels. This category corresponds to **Not Breaking Changes** discussed above which means no co-evolution has to be performed on models. For the second category, generic and reusable coupled operators can be defined. This is a core concept in COPE: the more reusable coupled operators that are defined, the more useful the solution becomes, because it saves the user from having to define his own specific operators. An example of such an operator is the renaming of a property of a class in the metamodel. It is possible to define a generic, reusable coupled operator for this adaptation as the operations performed on the metamodel and conforming models are always the same, irrespective of the specific metamodel the operator is applied on. In [19], an extensive catalogue of coupled reusable operators is defined to demonstrate the usability of the operator-based approach. The third category encompasses metamodel adaptations which need the definition of a custom coupled operation because the operator cannot be reused for other metamodels. They can, however, be defined in a model-independent way. COPE allows the user to define metamodel-specific coupled operators by applying a set of primitives for both metamodel adaptation and model migration. This set of primitives is complete

which means every possible metamodel adaptation and model migration can be specified with them. The last category consists of metamodel adaptations that have to be resolved on a per-model basis. This means no coupled operator can be defined (either reusable or custom) that evolves the metamodel and migrates the instance models automatically. The developer has to manually intervene in the migration process, which was not supported in the first version of COPE. In [20], Herrmannsdoerfer et al. introduce constructs that allow the user to choose for each model which action has to be taken to migrate it. As such, COPE supports all types of changes described by this classification.

3. Metamodel Isomorphism, Extension, Projection, Factoring: In [21], Hössler et al. classify metamodel adaptations based on relations between the two versions of the metamodel. The relations discussed in the paper are **Metamodel Isomorphism**, **Metamodel Extension**, **Metamodel Projection** and **Metamodel Factoring**, which are defined as follows. Two metamodels are said to be **isomorphic** if the set of their instance models is equivalent. All instances conforming to the initial version of the metamodel also conform to an **extension** of that metamodel. This resulting metamodel is then called a **super-metamodel** of the original metamodel. Conversely, the original metamodel is a **sub-metamodel** of the resulting metamodel. A metamodel is a **projection** of another metamodel if the size of the set of instance models reduces by deleting a metamodel element. **Metamodel Factoring** encompasses more complex patterns of metamodel evolution, including property movement and amalgamation and class splitting and amalgamation. The paper mentions how to deal with each of the relations, but no proof is presented that every possible metamodel adaptation can be categorized using this classification. Because of this, we cannot conclude that the solution is complete.

4. Construction, Destruction and Refactoring In [22], Wachsmuth proposes one of the most elaborate classifications of metamodel adaptations. First, preservation properties for metamodels are presented by defining metamodel relations like equivalence, variation, sub- and super-metamodel. These are then used to derive semantics- and instance-preservation properties for metamodel relations. Then, a set of edit operations are classified into **Construction**, **Destruction** and **Refactoring** operations. Each operation is associated with a semantics-preservation property which defines what

the result on the metamodel and its instance models is when the operation is executed. For each of these categories, co-evolution operations are defined which migrate models in response to the changes performed on the metamodel. In that sense, all adaptations which are described are supported by the solution. However, as will be seen further on, it may be possible that not all possible metamodel adaptations can be categorized using this classification. Performing an additive change may not necessarily mean that the set of instances becomes larger. For instance, if a relation is added between two classes with multiplicity 1..1, all instance models containing instances of those classes will have to be adapted, adding a relation between the instances. This means the set of valid instances reduces instead of increases. None of the metamodel relations defined by Wachsmuth describe this.

5. Fully Automated, Partially Automated and Fully Semantic In [23] a solution is proposed for the semi-automatic migration of model transformations. There, a classification is made which consists of three categories: **Fully Automated Changes**, **Partially Automated Changes** and **Fully Semantic Changes**. The first category consists of adaptations for which migrations can be automatically constructed, like the renaming of an attribute. The second category consists of adaptations for which information is lacking to migrate transformations. The example of deleting an attribute is given. If the attribute is used in a transformation, we do not know how the transformation should be adapted to compensate for the missing attribute. It is the task of the developer to fill in this missing semantic information. The last category consists of adaptations for which the developer needs to supply all semantic information. An example for such an adaptation is the addition of an element. No transformation rules are yet defined for this new element and as such the developer will have to supply all needed information in the transformations. The proposed solution performs an automated pass for adaptations in the first category, followed by a manual pass for adaptations in the last two categories.

In Table 2.1 a set of commonly occurring metamodel adaptations are classified according to the five aforementioned classifications. It attempts to clarify the presented classifications and by no means forms a complete set of metamodel adaptations. As can be

seen, some classifications cannot deal with certain metamodel adaptations. These cells are marked with a question mark.

2.1.2 Supported Migrations

When applying an MDE development process and using its full potential, a number of models and transformations are created. As previously stated, co-evolution for both models and transformations should be supported in order for MDE to become an accepted development method. In this section, the proposed solutions will be categorized according to the types of migration they support. This is either model only, transformation only or both.

Model Only: In [24] a domain-specific visual language is presented which supports the evolution of domain-specific modelling languages. This language is used to define patterns which describe the migration steps to perform when migrating models from one version of a metamodel to the next.
A similar method was used in the construction of the tool *Lever* [25]. Language evolutions in *Lever* are textually specified and used for both metamodel evolution and model migration. Model migrations created by *Lever* transform the abstract syntax graph representation of a model created in any version of the language to the last version of the language by keeping track of the history of the language and the evolution operations performed on the metamodel of the language.
In [26] a domain-specific transformation language (DSTL) is derived from the metametamodel the metamodel conforms to automatically. This language allows to specify evolution scenarios for the language. To support co-evolution, a user-defined mapping between primitive evolution operators in the DSTL and corresponding model migration steps has to be provided. From the evolution scenario and user-defined mapping a model migration transformation is constructed automatically.
In [27] the model change language (MCL) is used to define metamodel evolution and model migration patterns. MCL is a visual language which is used to specify the patterns which define the evolution scenario. These patterns contain information to evolve the metamodel and co-evolve conforming models.
Epsilon Flock [28] is a pattern language which is used to specify patterns that define

Adaptation	**1**	**2**	**3**	**4**	**5**
Generalize Metaproperty	Not Breaking	Metamodel-Only	Super-Metamodel	Construction	Fully Automated
Add (non-obligatory) Metaclass	Not Breaking	Metamodel-Only	Super-Metamodel	Construction	Fully Automated
Add (non-obligatory) Metaproperty	Not Breaking	Metamodel-Only	Super-Metamodel	Construction	Fully Automated
Extract (abstract) Superclass	Not Breaking	Metamodel-Only	MM Isomorphism	Refactoring	Fully Automated
Eliminate Metaclass	Breaking and Resolvable	Metamodel-Independent	MM Projection	Destruction	Partially Automated
Eliminate Metaproperty	Breaking and Resolvable	Metamodel-Independent	MM Projection	Destruction	Partially Automated
Push Metaproperty	Breaking and Resolvable	Metamodel-Independent	MM Factoring	Destruction	Partially Automated
Flatten Hierarchy	Breaking and Resolvable	Metamodel-Independent	MM Factoring	Destruction	Partially Automated
Rename Metaelement	Breaking and Resolvable	Metamodel-Independent	MM Isomorphism	Refactoring	Fully Automated
Move Metaproperty	Breaking and Resolvable	Metamodel-Independent	MM Factoring	Refactoring	Partially Automated
Extract Metaclass	Breaking and Resolvable	Metamodel-Independent	MM Factoring	Refactoring	Fully Semantic
Inline Metaclass	Breaking and Resolvable	Metamodel-Independent	MM Factoring	Refactoring	Partially Automated
Add Obligatory Metaclass	Breaking and Unresolvable	Model-Specific	?	?	Fully Semantic
Add Obligatory Metaproperty	Breaking and Unresolvable	Model-Specific	?	?	Fully Semantic
Pull Metaproperty	Breaking and Unresolvable	Model-Specific	MM Factoring	Construction	Fully Semantic
Restrict Metaproperty	Breaking and Unresolvable	Model-Specific	Sub-Metamodel	Destruction	Fully Semantic
Extract (non-abstract) Superclass	Not Breaking	Metamodel-Only	MM Factoring	Construction	Fully Automated

Table 2.1: Adaptation Classifications

model migration scenarios. The language developer creates a model migration transformation using Flock which is capable of migrating models from one version of the metamodel to the next.

In [29] existing in-place transformation languages are used to define model migrations. The first step in the process is to merge the two versions of the metamodel, as both elements from the initial version and the evolved version will be used in the transformation patterns. Using this merged metamodel, a user can define the necessary rules for instantiating metamodel elements introduced in the evolved version of the metamodel. The last step in the process is to remove elements that are no longer present in the evolved version of the metamodel automatically.

In [12] and [13] model migration is approached differently. In these papers, a process model is created with which a metamodel and its instance models are co-evolved (semi-)automatically. First, a change model is computed which represents the metamodel adaptations. These adaptations are classified after which user input is gathered if needed. Then, the necessary model migration algorithms are determined and the transformation is executed.

In [14] and [15] a similar method is employed. The main difference is that after change classification, the difference model is split into two non-overlapping difference models based on the automatibility of the migration derivation. Two migration transformations are derived from these change models and executed in parallel to migrate models.

In [30] metamodel adaptations are detected by comparing the two versions of the metamodel and employing a set of user-chosen and/or user-defined heuristics which has to be configured for each evolution scenario. The resulting difference model is then mapped onto a migration transformation.

In [22], metamodels are evolved by stepwise adaptation. This is done by so-called coupled operators, which consist of the metamodel adaptation and corresponding model migration transformation. The same approach is used in COPE [18, 20], where reusable coupled operators are defined in a library. A facility to create custom coupled operators is also provided.

Transformation Only: There are only a few solutions providing only transformation migration. In [16] a semi-automatic method is developed which is based on a generated difference model. The metamodel adaptations are classified, after which transformations

are generated automatically or with the help from the user if semantic information is missing.
In [23] the MCL is used to specify metamodel adaptations. Then, transformation migrations are derived automatically where possible or provided by the user in the form of MCL patterns.

Model and Transformation: In [10] both model and transformation migration are discussed. As previously mentioned, the authors create a framework for the evolution of languages which is as complete as possible. To evolve models and transformations a pipeline is built which defines the different steps of the evolution process. These steps can be found in other solutions as well: change detection and representation, automatic transformation generation together with user input gathering for unresolvable changes and transformation execution.
[31] discusses both model and interpreter migration. The authors present an approach in which a specification of the metamodel adaptation is mapped onto model and interpreter migration transformations.

2.1.3 Automatibility and Completeness

Automatibility of a solution pertains to how much user intervention is needed: either the solution is fully automatic, only based on user input or a combination of the two. Completeness is a characteristic of a solution which is closely tied with automatibility. We regard a solution as complete if all possible evolution scenarios are supported. It is closely tied with automatibility because a solution which is fully automatic often is not complete and a solution which is only based on user input often is complete, but requires a lot more effort from the user.

Fully Automatic: Fully automatic solutions usually start from a model that captures the adaptations performed on the metamodel. This model can either be the result of a change recording mechanism that keeps track of all adaptations performed on the metamodel or an algorithm that compares the two versions of the metamodel after the changes have been performed. This change model is then used to derive a migration transformation which is subsequently executed on models to migrate them to the new

version of the metamodel.

In [30], Garcés et al. attempt to fully automatically migrate instance models. There, two versions of the metamodel are compared by an algorithm which is the composition of a set of heuristics. These heuristics have to be chosen by the language developer on a per-metamodel basis to get the best resulting difference model. It is however a non-trivial task to decide a priori which of the heuristics are required to produce the best results. In the paper, the derivation of the adaptation transformation is given little attention and it is assumed this mechanism works correctly. These arguments lead to the conclusion that the solution cannot be deemed complete.

In [31], a generative approach to intepreter evolution is proposed. In their solution, the authors map a specification of the metamodel adaptations onto interpreter and model migration transformations. They require that a formal specification of metamodel changes has to be present, either provided by the user or computed by a differencing technique from the two metamodel versions. The authors state that more research has to be performed to evaluate the feasibility of the approach. It is thus not complete.

Only User Input: This category consists of solutions that require the user to define a model migration transformation manually. This migration is either specified in an existing transformation language or in a newly created domain-specific language for the domain of model migration.

In [24] a domain-specific visual language is created to specify visually how models should be migrated in response to metamodel adaptations. A sequence of transformations is created by the user and concatenated in a user-specified order. The models are then migrated by performing each transformation in this order. Primitives for the creation, deletion and mapping of model elements are provided in the language. It is not clear, however, whether these primitives form a complete set and can be used to specify arbitrary complex migration scenarios. In the example given a rather trivial co-evolution problem is solved and there is, for example, no change present that could be described as 'breaking, not resolvable' which requires user input for each model that is to be migrated. As such, we can conclude this solution is not complete.

In [25], the tool *Lever* is created. In *Lever*, model migrations are defined textually by the user. The tool keeps track of the history of the metamodel, which allows a model created in every version of the language to be loaded into the tool by first interpreting it using

the version of the language it was created in and then migrating it using the user-defined migration transformations. This solution, however, does not include mechanisms to deal with model-specific changes, i.e. those changes for which user intervention is required on a per-model basis. As no categorization of metamodel adaptations is employed in the paper, it is a challenging task to prove the solution can deal with every possible metamodel adaptation. Therefore, we can conclude that this solution is not complete.

In [27], the model change language (MCL) is introduced. MCL is used to define patterns for model migration using both versions of the adapted metamodel. A pattern in MCL consists of a left hand side, defining which element of the initial version of the metamodel should be matched and a right hand side, where modified elements in the new version of the language can be used. The user defines relations between these elements and a model migration is subsequently derived. The paper is not extensive and it is not clear whether all complex migration scenarios are supported by MCL. We therefore conclude that this solution is not complete.

In [32] a different approach altogether is investigated. The authors observe that models conforming to the old version of an evolved metamodel often cannot be loaded into the normal development environment used to create and edit models after installing the new version of the metamodel. To counter this, the authors propose a solution where models are bound to a generic metamodel. This makes it possible to load all models, even non-conforming ones. When an inconsistency is detected in a model, the user is notified and markers are placed in the places which cause the inconsistency. The user can then edit the model which has been transformed to the Human Understandable Textual Notation (HUTN). The solution is complete but may require a lot of effort from the user as each model has to be migrated manually.

Epsilon Flock, introduced in [28] is a model-to-model transformation language which can be used to define migration patterns. The authors analyse ATL, COPE and Ecore2Ecore to derive requirements for their tool, which tries to combine the advantages of each tool and avoid their disadvantages. The result is a pattern language which consists of two elements that either migrate types or delete them. It automatically copies elements conforming to the new version of the metamodel language, which is called a conservative copy strategy in the paper. Flock is complete as it uses the Epsilon Object Language, which is expressive enough to perform any evolution scenario.

Combination: Solutions that are a combination of the previous two categories usually use the same approach as can be found in the fully automatic solutions, but consist of an automatic phase and a phase in which the user can provide missing semantic information. These approaches are logical consequences from the act of classifying metamodel adaptations based on their effect on conforming models. As we have seen in Section 2.1.1, authors always make the distinction between adaptations for which the remedial action can be deducted automatically and those for which user intervention is required. If one can then accurately detect all adaptations performed on the metamodel and classify those adaptations, it is possible to divide the generation of the migration transformation into two non-overlapping parts.

A second category consists of operator-based solutions, whereby metamodels are adapted by the successive application of coupled operators. Some of these operators can be reused and information regarding model migration is attached to these reusable operations, permitting the automatic deduction of a migration transformation. Other operators are so specific to a certain language evolution scenario that they have to be defined by the user. That is why operator-based solutions are classified as solutions which are a combination of automatic solutions and solutions based on user input: a user evolves the metamodel by applying operators which are either reusable or user-defined, and a migration transformation is (semi-)automatically deduced.

In [12] and [13] a process model is constructed which performs the steps of the first approach discussed in this paragraph. First, a change model is constructed, either by change recording during the editing of the metamodel or by direct comparison of the two versions of the metamodel. Then the changes are categorized and transformations are constructed, either automatically for resolvable changes or by user input gathering for unresolvable changes. Then the transformation is executed to migrate the model. This solution is complete in the sense that it should be able to handle all types of changes, yet it is incomplete because the change detection algorithms and the algorithms to automatically derive transformations from metamodel adaptations are largely omitted.

In [14] a similar approach is used. However, the classification of changes in the proposed approach is made explicit by dividing the change model into two non-overlapping change models, one for the automatically resolvable changes and one for the non-resolvable changes. From these models, two transformations are generated which can be run concurrently to migrate models to the new version of the metamodel. A challenge recognized by the authors with this approach is dealing with dependent changes. They therefore

propose an algorithm in [15] to resolve these dependencies by scheduling transformation executions correctly. This approach is complete in the sense that all possible scenarios should be supported. However, the authors also mention the need for validation on a large set of metamodels and models, which has yet to be performed.

In [22] the concept of coupled operations is first explored. These operations are first described on the metamodel level as QVT Relations that define the adaptations performed. These can be used for the stepwise adaptation of metamodels. Next to these adaptations, model co-adaptations are defined in the form of parametrized QVT Relations. A metamodel adaptation transformation can then call one of these co-adaptation transformations with the correct parameters to create a model migration transformation. It is possible for users to intervene in this process by defining new adaptation transformations, co-adaptation transformations or OCL queries to fill in missing semantic information. It is therefore a complete process in the sense that any scenario *could* be supported, although only a handful of adaptation transformations are presented as examples in the paper.

COPE is a tool that supports the co-evolution of metamodels and models using coupled operators. It is introduced in [18] and expanded in [20]. In [19] an extensive catalogue of reusable coupled operators is presented with which a language developer should be able to specify most common language evolution scenarios. A coupled operator is an operator which performs the metamodel adaptation and contains the migration transformation steps which need to be carried out in order for models to be migrated to the adapted metamodel. In COPE, a series of adaptations is performed by executing coupled operators on a metamodel. During this process, the corresponding model migrations are recorded. Once a developer is done editing the metamodel, these model migration steps are combined into one migration transformation which can be executed to migrate models to the new version of the metamodel. As COPE supports the definition of custom coupled operators and is capable of asking for user input when additional semantic information is needed during the migration process, this is a complete solution.

Vermolen et al. employ the use of a domain-specific language for evolving metamodels in [26]. The DSL is created starting from the metametamodel which the metamodel conforms to. This languages contains primitives for adding, removing and modifying metamodel elements. Next to the DSL, a user-defined mapping has to be supplied that maps metamodel adaptations to the required model migrations. Using these elements, a metamodel adaptation is created in the DSL by the language developer, after which

a model migration transformation is automatically calculated. The solution is not complete, as the definition of the user-supplied mapping is not given any attention and as such it is not clear whether every possible mapping can be specified. There is no mention of support for user input during the migration process either (for model-specific adaptations).

In [16] a solution is provided which evolves transformations semi-automatically in response to a metamodel adapation. Only adaptations of the source metamodel are considered, as such it is not complete. The approach classifies changes and either automatically generates the full transformation code or only a skeleton which requires the user to fill in the missing semantics.

Another approach to semi-automatically migrate transformations is described in [23]. There, the MCL is used to describe the adaptations performed on the metamodel. Subsequently, migration transformations are semi-automatically deduced. A classification is presented which divides adaptations into categories based on their automatibility. Some of the adaptations, like the renaming of a metamodel element, can be resolved automatically. For others, only the skeleton in the resulting transformation can be generated. For others, the user needs to specify the complete transformation from scratch. The solution is not complete as the authors do not cover this last class of adaptations, which are called fully semantic changes in the paper.

In [29] the use of existing in-place transformation languages for specifying model migration transformations is explored. The user has to specify rules for elements that have been changed in the new version of the metamodel. For this purpose, the two versions of the metamodel are merged as to allow the use of both outdated and new elements of the metamodel in these rules. After executing the user-defined rules, outdated elements are removed from models automatically. There is no support for automatic generation of rules for resolvable changes and the authors mention this as future work. However, the solution is complete as it should be possible for users to specify every pattern needed to perform migration of models. The approach is not formally validated in the paper.

In [10] the authors attempt to provide a complete framework for the evolution of modelling languages. The framework supports the semi-automatic migration of models and transformations in response to metamodel adaptations. In the paper, every possible evolution scenario is exhaustively explored. The solution uses a migration pipeline which consists of all necessary steps to be performed in order to migrate all artefacts related to an evolved metamodel. First, an intermediate metamodel is created which is the result

of merging the two versions of the metamodel. Next, from the automatically resolvable adaptations, migration transformation steps are constructed. If needed, user input is gathered and the migration transformation is executed. It is complete, as it is able to handle every possible evolution scenario using the proposed approach.
In [17] a generic metametamodel is constructed. All metamodels can be transformed in such a way that they conform to this metametamodel. After this transformation, metamodels can be regarded as a special kind of model. This allows the use of existing model differencing techniques to obtain a difference model by comparing the two versions of the metamodel. Using this difference model, the authors generate a difference model for each model that needs to be migrated. This difference model captures the adaptations that have to be performed on the model in order for the conformance relation to be reestablished. This approach is complete, as it can deal with all possible types of changes and is extensible by providing user-defined transformations. A large validation study is present which shows the solution is applicable in real-life language evolution scenarios.

2.1.4 Intention Preservation

Intention preservation is a somewhat overloaded concept. In [10], the authors introduce the concept of *continuity* of software language evolution. This means that the system is consistent (models conform to their metamodel after metamodel adaptation) and semantically equal to its previous version, modulo intended changes. *Semantics* in this context mean the properties a certain model has and not, as was previously defined, the denotational or operational semantics attached to a modelling language by model transformations. The properties of a model correspond to the properties the model has in its semantic domain. For instance, a possible property for a model created using a modelling language which is mapped onto Petri Nets is *liveness*, meaning it never reaches a deadlock state. In the paper it is mentioned that in order to support continuity, there has to be some mechanism in place in order for the properties to be checked. As an example, a constraint on a toy language for constructing train networks is given: if a constraint exists on the initial version of the language that no two trains should ever be on the same rail, this should be the case in every subsequent version of the language. This could be checked by transforming the models to a Petri Net model and running a reachability analysis. However, this is a rather ad-hoc way of checking properties and cannot serve as a general mechanism to ensure continuity. As properties are checked after

migration is performed, it is not possible to decide whether a migration transformation will preserve the properties of a model before it is executed. The set of models on which properties are checked is furthermore only a subset of the set of all models which can be created with the modelling language, and it is not possible using this method to formally prove that a transformation preserves properties, before or after migration has been performed. More research has to be done in this area to discover possible solutions to overcome the challenge of continuity.

Other methods exist to ensure semantic equivalence of models before and after migration. However, they often defer the responsibility to someone or something else. In the case of manual specification methods (see Section 2.1.3) the user is responsible for providing the correct migration patterns and ensuring semantic equivalence. In [24] this fact is mentioned and the only solution provided is to spend a lot of effort on the development of the migration transformation. There is, again, no way to check a priori whether a migration will preserve certain semantic properties of a model.

When using (semi-)automatic methods of generating a transformation migration, the correct working of the algorithm depends largely on the change recording or differencing technique which is used to build the difference model. If all changes are represented faithfully, which requires the difference metamodel to be complete and the technique to build the difference model to be correct, it should be possible to define remedial actions for every possible scenario and ensure correctness with respect to semantic preservation. However, lots of differencing techniques have issues with quite trivial edit operations. Let's say we want to move a property from a class to its superclass. We therefore delete the property of the class and create a new one in the superclass with the same name and data type. To ensure semantic equivalence, we would like that the value of each instance of the property be retained in the migrated models. However, the difference model might not be able to differentiate between this edit operation and two unrelated edit operations: removing a property and creating a property. If it detects these last two, there is no way for the system to know that it should keep the value of the properties and semantic information will be lost. In [30] a substantial effort has been made to accurately detect metamodel adaptations. However, the process presented in that paper has to be configured on a per-case basis. The process uses a set of heuristics to compute the difference model. One of the main characteristics of a heuristic is that it can approach perfection but never attain it. This may be unacceptable in certain applications that require rigorous checking of properties using a formal method.

We can conclude that semantic preservation of models is currently not supported by language evolution solutions and needs further researching.

2.1.5 Tool Support

Tool support is one of the deciding factors whether a proposed technique will be successful and a way to check its functionality. In this section, we will look at which solutions are implemented in tools, which we divide into prototypes and fully functional tools.

Prototypes: As was discussed earlier, in [25] the tool *Lever* is constructed. This is a prototypical implementation of the techniques presented in the paper which tries to automatically maintain a layered DSL which undergoes adaptations. The authors have performed a small case study using the tool, but it is not extensive enough to conclude that this tool is mature and ready to be used on industrial-sized projects.
In [12] a prototype is built which implements the three-step process of change recording and exporting, classification and transformation generation and transformation execution. The prototype only supports the renaming of structural features.
In [26] a prototype is constructed which implements the architecture presented in the paper and has been applied to the domain of data modelling. The solution in the paper automatically derives a DSL in which evolution patterns for metamodels can be defined and derives model migration transformations from these patterns and a user-defined mapping which specifies how metamodel adaptations should be mapped onto model migrations.
Cicchetti et al. have implemented their approach, which is described in the previous sections, in a prototype tool [14] [15] on the AMMA [33] platform.
The AMMA platform was also used by Garcés et al. to create a prototype for their solution in [30]. They use the AtlanMod Model Weaver (AMW) [34] to work with matching models and ATL is used to implement the heuristics and the higher-order transformation which is responsible for generating the migration transformation.
In [17] a prototype tool is built which is tested on 10 metamodels, each having 10 conforming models. However, the amount of tested metamodel adaptations is rather small.

Fully Functional: A fully functional tool based on the use of coupled operators is COPE [18, 20]. It is integrated into the EMF metamodel editor and provides all the necessary functionality for metamodel and model co-evolution. Users can edit a metamodel using a context-aware editor which presents the library of reusable coupled operators. Custom operations can be created, edited and subsequently applied on metamodels. The tool keeps track of the history of the metamodel adaptations. Once a user is done editing the metamodel, he releases a new version of the language which automatically creates a migrator capable of migrating models to the new version of the language. COPE has been tested thoroughly and appears in a couple of industrially sized case studies [35, 36].
In [32], a tool is built which binds models conforming to an EMF metamodel to a generic metamodel. The authors then generate a report which states whether the model is inconsistent with its metamodel and if so, which parts of the model are responsible. The user can edit the inconsistent models as to re-establish the conformance relationship.
Evolving transformations can be accomplished by the tool built in [23]. The tool is implemented in the GME/GReAT toolset and has been tested in an industrial environment. Future work which is mentioned in the paper consists of providing tool support for the addition of missing semantic information. In the current version of the tool, this information has to be added manually, which may mean a lot of work.
Lastly, Epsilon Flock [28] is a model-to-model transformation language built atop of the Epsilon Object Language (EOL) [37], which is the core of the Epsilon platform [38]. Flock is used for defining model migration patterns and has proved its worth in industrial sized case studies [36].

2.2 Analysis

In this section the results of the previous section are analysed. We will attempt to discover open research questions which should lead to a better support for the co-evolution of metamodels and their related artefacts when discussed in future work. In each of the following subsections, four pairs of evolution criteria will be the basis for four matrices. The matrices consist of an x-axis and a y-axis, each one used for a particular criteria. For each criteria, a scale has been introduced which rates the solution based on the findings in the previous section. The criteria used as axes are *Migration Support* (Model, Transformation, Both), *Tool Support* (Prototype, Implemented, Tested),

Automation (Only User Input, Fully Automatic, Combination), *Algorithm* (Manual, Operator Based, Differencing, Change Recording) and *Example* (Toy, Full Fletched). In Section 2.2.1, migration support and tool support of solutions are combined into a matrix. In Section 2.2.2 the same is done for automation and tool support. In Section 2.2.3, the matrix axes are automation and tool support. Using these three matrices, we can conclude which proposed solutions lack tool support. Lastly, in 2.2.4, we take a look at the types of supported migrations of each solution and whether an example is present.

2.2.1 Migration Support vs. Tool Support

A solution can either support model migration, transformation migration or both. In this section, a matrix is created (see Figure 2.1) which lists the solutions discussed in Section 2.1 along the x-axis according to this criterion and along the y-axis according to whether tool support is present. If tool support is present, three categories of tools are distinguished: a prototype is a tool which implements the solution but has only been used to migrate toy examples or examples from which cannot be derived that the tool can deal with arbitrarily complex metamodel adaptations. An implemented tool is a tool which implements the solution faithfully and should be able to handle most or all evolution scenarios. A tested tool is a tool which is implemented and tested on a large, industrial-sized project.

As we can deduct from Figure 2.1 major progress has been made developing tools that support model migration. This is to be expected as most of the literature on metamodel evolution deals with the migration of models only. Implementations that have been tested thoroughly are COPE and Epsilon Flock. In [36] these two tools appear in a study which compares four tools capable of co-evolving metamodels and models. The tools are used on two co-evolution scenarios: one toy example (a Petri Net metamodel) and a larger example taken from a real-world model-driven development project.
Transformation migration is an area which has not been explored extensively. In [16] a semi-automatic method is proposed, generating ATL transformations from a difference model resulting from the comparison of the two metamodel versions. [23] uses the MCL to describe metamodel adaptations and corresponding transformation migrations.

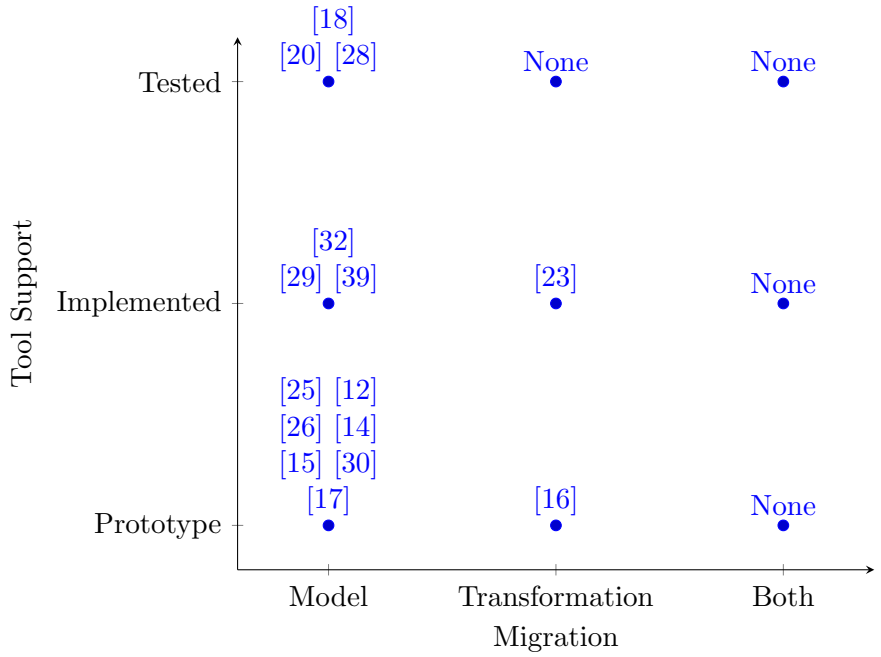


FIGURE 2.1: Migration Support vs. Tool Support

Transformation migrations are constructed semi-automatically in two phases: an automatic phase and a phase in which the user can supply missing semantic information. In [10] a third approach to transformation migration is explored. The authors use function composition to create the evolved transformation from the original transformation and the migration transformation generated to migrate models conforming to the initial version of the metamodel. For example, in the case of image evolution, the resulting transformation T' = $E \circ T$. However, because of the *projection problem* T' = $E \circ T$ does not always hold for image evolution. The projection problem states that E does not necessarily map models of the old version of the language to the complete set of models in the new version of the language. For instance, if a non-obligatory concept is introduced in the language, E will not migrate models to models containing instances of this class. As such, $E \circ T$ will not explore the full power of the new version of the image language, which may be undesirable. User input is required in these cases to fill in missing semantic information. In the paper, however, only a conceptual framework is built and therefore it is not listed in the matrix presented in this section.

This section can be concluded by stating that no verified implementation of transformation migration has been presented yet. No attempt has been made to create a tool which supports both model and transformation migration.

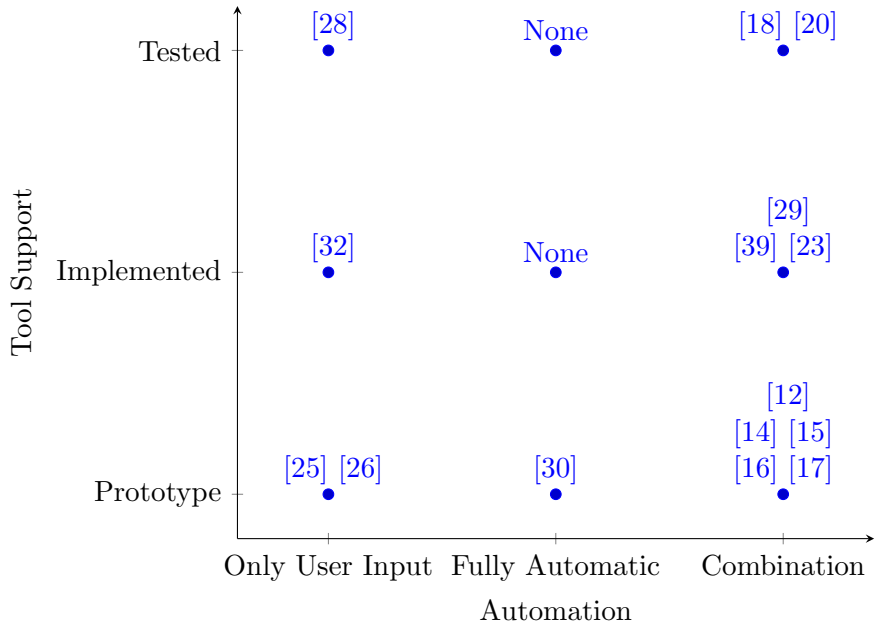


FIGURE 2.2: Automation vs. Tool Support

2.2.2 Automation vs. Tool Support

In this section, we will discuss tools and which level of automation they support. The automation levels of algorithms we distinguish are user input only, fully automatic and a combination of these two, which are semi-automatic solutions. These levels correspond to the ones discussed in Section 2.1.3. Unlike some of the criteria presented in this paper, a solution is not inherently better if it uses a certain level of automation instead of another one. As mentioned in Section 2.1.3, each level of automation brings its challenges and advantages: a fully automatic solution requires no user input, but it is challenging to create a complete solution using this technique. A solution which requires the user to manually specify the migration steps requires a lot of involvement of the user, but it is easier to achieve completeness.

In Figure 2.2 all solutions are presented in a matrix which uses these two criteria as axes. As can be seen in this matrix, solutions which are only based on user input already have mature tool support. As in the previous section, COPE and Epsilon Flock are both implemented and tested on industrial-sized projects. There exists only one prototype supporting fully automatic migration of instance models, presented in [30]. From previous sections and this matrix we can conclude that this approach is not popular because it is a challenging task to completely automate the migration process: some metamodel

adaptations are simply too specific to resolve automatically by a generic algorithm. Authors have therefore attempted the creation of an algorithm which can automatically detect metamodel adaptations and create a model migration transformation with, for some types of adaptations, transformation steps provided by the user. However, it remains to be investigated whether these tools are capable of supporting co-evolution in industrial-sized projects.

2.2.3 Algorithm vs. Tool Support

This section will discuss tool support for the different types of algorithms identified in solutions to the co-evolution problem. These algorithms have not been explicitly discussed in the previous sections, but have been mentioned a number of times. We distinguish four different algorithms. The first are manual algorithms, which require the user to manually specify the migration transformation steps (in a language like Epsilon Flock). Second are operator-based solutions, which allow the user to specify metamodel adaptations and model/transformation migration steps using coupled operators. The last two categories are used in solutions which co-evolve metamodels, models and transformations semi-automatically and are in need of a method to build a difference model. This difference model can either be constructed by comparing the two versions of the metamodel using a tool like EMF Compare [40] or by keeping track of the changes performed on the metamodel and generating a change trace.

In Figure 2.3 the resulting matrix is presented (note that some of the solutions are listed twice: these solutions can either use a differencing or change recording algorithm). A few interesting results can be concluded from this matrix. Firstly, as is the case in the preceding matrices as well, COPE and Epsilon Flock are the only tools that have been thoroughly tested for the manual and operator based approaches, respectively. Secondly, only prototypes for the differencing and change recording algorithms have been created. This means these two method of co-evolving metamodel and conforming models and transformations have never been validated. Most of the approaches which use a combination of automation and user input employ a three-step algorithm: change recording/detecting, mapping of change model on model migration steps (either automatically or through user input collection) and execution of migration transformation.

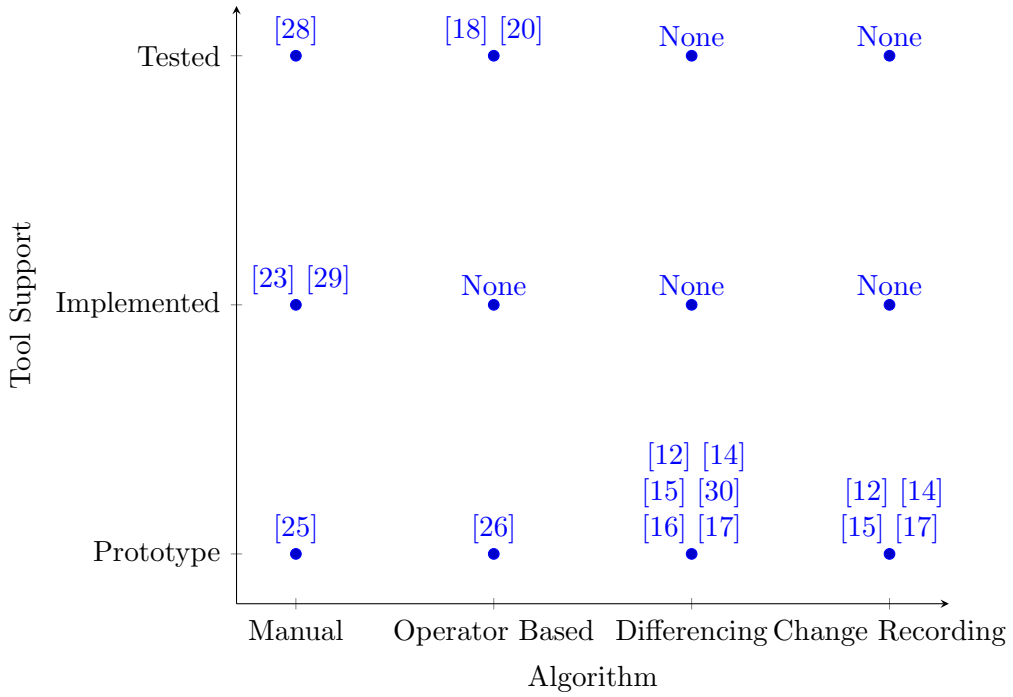


FIGURE 2.3: Algorithm vs. Tool Support

A tool which implements this approach and is tested on industrial-sized projects has yet to be created.

2.2.4 Migration Support vs. Example

Examples are often used to clarify concepts and algorithms used in a paper. In the case of metamodel and model co-evolution, examples are often given with respect to categorization of metamodel adaptations (see Section 2.1.1 for the different types of categorizations employed by authors) and the algorithms used by the proposed solution. We differentiate between two different types of examples: toy examples and full fledged examples. Toy examples are examples which either only demonstrate a classification or algorithm on trivial evolution scenarios or demonstrate the functionality of the solution on a toy language. A full fletched example demonstrates the working of an algorithm by executing it on a set of non-trivial evolution scenarios or on an industrial-sized project.

In Figure 2.4 migration support of solutions and the examples presented in the corresponding paper which demonstrate the functionality of the solution are combined into a matrix. The results are explained below, where we present each type of migration support and which papers present which type of example.

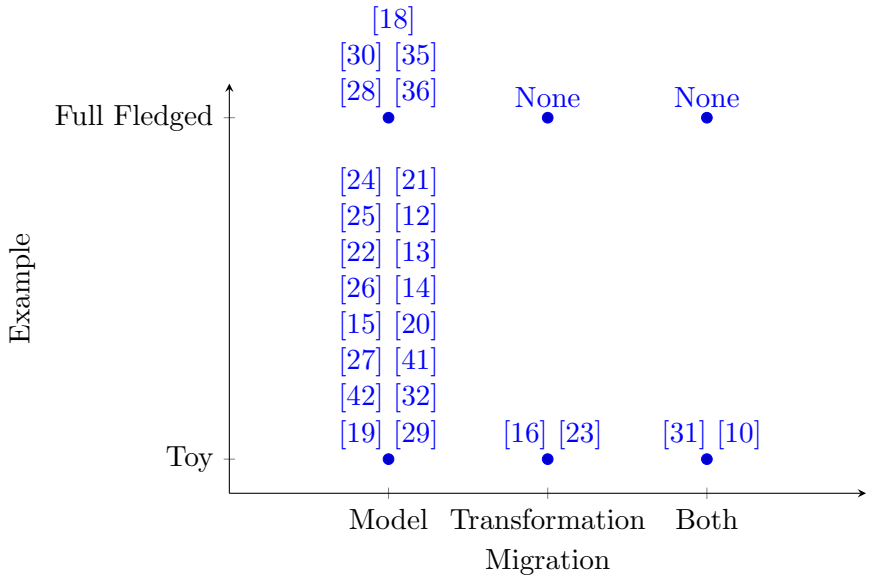


FIGURE 2.4: Migration Support vs. Example

Model: In [24] an example model migration transformation is constructed using the DSL introduced in the paper. The language being evolved is used to create models in the domain of signal processing. In its original version three concepts are defined: *Ports*, *Connections* and processing blocks which are connected to each other using *Ports* and *Connections*. In the evolved version of the language *Ports* are specialized into *InputPorts* and *OutputPorts*. As a specific semantic meaning to these concepts is attached, the authors define a migration transformation which maintains the semantic meaning of models after migration.

[21] presents a classification of metamodel adaptations and presents model migration algorithms for each identified class of changes. Example metamodel adaptations are presented visually. However, the examples given neither form a complete set of metamodel adaptations nor have they been applied to a real-life project.

In [25] a toy example is presented to demonstrate the functionality of *Lever*. It concerns an expression language that translates sums to stack machine code. The authors transform the language from infix to postfix notation by specifying a metamodel evolution which can automatically migrate instance models as well.

[12] introduces a change classification and algorithm based on this classification scheme. To demonstrate their approach, the authors present an example metamodel of a file system. This metamodel is evolved by performing certain adaptations such as renaming, subclassing and creating or moving of containment references. To capture the semantic intention behind these changes, the method proposed in the paper constructs migration

transformations which are capable of maintaining this semantic information.

In [22] an extensive categorization of metamodel adaptation is presented. For each category, a set of example adaptations are modelled as coupled operators using QVT Relations. An example Petri Net metamodel evolution is presented, clarifying the general concept of metamodel and model co-evolution.

In [26] the concept of metamodel and model co-adaptation is applied to the domain of data modelling. A set of metamodel adaptations is presented and corresponding model migration steps are derived. Then, these examples are used to construct a general architecture which consists of an automatically constructed DSL for the construction of metamodel adaptations. The examples are then represented using this general architecture to show its functionality.

The authors in [14] validate their approach by applying it on an evolving Petri Net metamodel. The metamodel is evolved in two steps using a set of metamodel adaptations such as 'restrict metaproperty', 'introduce superclass' and 'introduce property'. Only fragments of the resulting model migration transformations are shown in the paper. The same Petri Net metamodel is used by the authors in [15] which explains how to deal with dependent metamodel adaptations.

COPE has been tested on a large number of evolution scenarios. As it was not possible to deal with model-specific adaptations in the first version of COPE, the authors present the needed constructs in [20]. A toy example is presented to demonstrate these constructs which evolves an automata metamodel to support initial states. This is a model-specific adaptation as the modeller has to choose an initial state for each evolved model.

The MCL is used in [27] to co-evolve metamodel and models. A set of example MCL patterns is presented for metamodel adaptations such as the introduction of subclasses and the changing of containment hierarchy.

In [41] a theoretical overview of domain-specific languages is presented. The authors use their definitions to present the different types of evolution that can be performed on a system created using a DSL. An example language is used to clarify the concepts introduced and to argue which metamodel adaptations can be trivially mapped to model migrations and which are in need of user intervention.

In [42], Rose et al. compare different approaches to model migration. Several examples are given to clarify migration scenarios for the different approaches (the authors

distinguish between manual specification, operator based and metamodel matching approaches).

Rose et al. introduce a model migration approach in [32] which is based on binding models to a generic metamodel, after which inconsistencies are presented to the user in a resolution environment. A toy example is presented to clarify the approach and show the inconsistency report generated by the tool.

In [19], Herrmannsdoerfer presents an extensive catalog of reusable coupled operators. A number of these coupled operators are clarified by presenting example evolution scenarios.

In [29] a running metamodel evolution scenario is used throughout the paper to clarify the approach of using in place transformation languages to specify model co-evolution. The example is used at every step of the process: first, the original and evolved metamodel are used to create a merged metamodel. Then, the co-evolution rules are expressed as graph transformations, which make use of this merged metamodel. These graph transformations are then mapped onto ATL transformation rules.

COPE, as has been mentioned in the previous sections, has been tested thoroughly. In the paper which introduces the tool [18], the authors already demonstrate the full functionality by presenting different adaptation scenarios of a simple domain-specific language. These examples are complete in the sense that every supported type of operation (reusable and user-defined) is explained in these examples. Next to these small examples, a case study is performed on two EMF-based metamodels: one is part of the Graphical Modelling Framework (GMF), the other is part of the Palladio Component Model (PCM). Both of these metamodels already have a rich evolution history and this history has been recreated through the use of COPE. The results have been analysed in the paper to test the viability of the operator-based approach. A similar case study has been performed in [35] on two industrial-sized metamodels. The adaptations performed on these metamodels have been analysed and classified according to the classification of Herrmannsdoerfer (see Section 2.1.1). From these results, a number of requirements for automated coupled evolution are formulated, which reappear in the subsequent papers on COPE.

In [30] a method is proposed based on precise detection of metamodel adaptations through the use of heuristics. To clarify the approach, a running example is used based

on an evolution scenario of a Petri Net metamodel. The approach is validated by implementing a prototype tool and testing it on six versions of a Petri Net model and eight versions of the Netbeans Java metamodel. The matching strategies composed of a set of heuristics for both metamodels are presented and the results are analysed to demonstrate that the approach detects most metamodel adaptations correctly.
In [28] Epsilon Flock is introduced. Its functionality is demonstrated using two example metamodel adaptation scenarios. The first uses a Petri Net metamodel which is included as a comparison to other approaches. The other uses the UML metamodel and the adaptations performed from UML 1.5 to UML 2.0. Flock also appears in [36], where it is compared to COPE, amongst others. The tools are compared by first applying them on an example migration of a Petri Nets metamodel. Then they are applied to the larger example of evolution of the GMF metamodel. It shows both COPE and Epsilon Flock can deal with the complex evolution of large metamodels.

Transformation: As for the two papers that introduce methods for transformation migration, they both clarify their solutions by presenting toy examples.
In [16] an example transformation is described by a number of ATL rules. The transformation has as its domain an exam language which describes an exam by a list of questions and as its image an MVC language. When a metamodel adaptation is performed, the transformation is migrated by a higher-order transformation. Missing semantic information may need to be provided manually by the developer.
In [23] the MCL is used to migrate transformations. A case study is presented for the domain of hierarchical signal flows, which are mapped onto an actor-based language without hierarchy by a transformation. The domain metamodel is evolved and transformation instances co-evolved by specifying a mapping in MCL. Adaptations that can be automatically mapped to model migrations are generated and missing semantic information should be provided by the developer where necessary.

Both: In [31] both model and interpreter evolution is discussed. An architecture is presented which should be able to migrate models and interpeters starting from a metamodel evolution specification. A few example metamodel adaptations are presented together with their remedial actions in models and interpreters. In all of the cases, a trivial search/replace algorithm is sufficient to migrate these instances.

In [10] an example language is used throughout the paper. This DSL is used to specify rail road networks, which are mapped onto Petri Nets by a transformation which attaches semantics to the constructed models. The language is used to clarify concepts introduced in the paper and show limitations of transformation migration (which is called the projection problem). An example evolution is performed on the metamodel of the language whereby a migration pipeline is built for the evolution of instance models and transformations created using the initial version of the language. In the evolution scenario, only a few operations are performed on the metamodel and as such it is classified as a toy example.

2.3 Conclusion

This paper has discussed the currently available literature on metamodel, model and transformation co-evolution using several criteria: change classification and support, migration support, automatibility and completeness, intention preservation and tool support. Then, these criteria were combined into a number of matrices which lead to directions for future work in this area. The following was concluded:

- Additional research is required to construct tools which support transformation migration and tools which support both model and transformation migration.
- Validation studies are needed to test the correct functioning of tools that (semi-)automatically migrate instance models or transformations through the construction of a difference model, as these tools have not been tested on industrial-sized projects.
- Full fledged examples for solutions that migrate transformations or migrate both models and transformations have to be presented. Currently, no large case studies have been performed that validate these solutions.
- The ability to support intention preservation and continuity has to be further investigated in future tool implementations.
- A standard case study should be created which can be used to test the functionality of a proposed approach. This case study should include a complex metamodel evolution scenario which consists of all possible types of changes.

- It is not possible to construct a fully automatic solution for co-evolving metamodels, models and transformations due to the fact that the intention of the language developer has to be captured in some way. Researchers should focus on semi-automatic methods instead.

Further research into these fields should increase the general level of understanding of the subject and lead to full support of the co-evolution of metamodels and their related artefacts, allowing the adoption of MDE as a mainstream software development method.

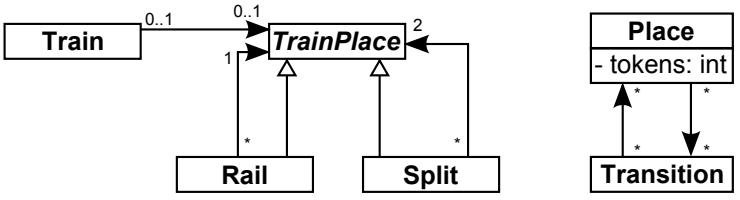
Chapter 3

Running Example - The TrainSim Modelling Language

In this chapter, the TrainSim modelling language is introduced. It will be used as a running example throughout the rest of the chapters, to demonstrate the techniques for modelling language evolution. The example has been adopted from [10], where Meyers and Vangheluwe use it to introduce their framework for modelling language evolution. We build upon their work and use parts of their proposed framework as a basis for our research.

3.1 The TrainSim Modelling Language

The TrainSim modelling language is used to model railroad networks consisting of trains and rail segments. A rail segment can either be a simple rail, or a split, which has one



(a) (b)

FIGURE 3.1: The metamodels used in defining the TrainSim modelling language: (a) shows the TrainSim metamodel and (b) shows the PetriNet metamodel.

incoming rail and two outgoing rails. A train can occupy a rail segment. The metamodel of this language is shown in Figure 3.1(a).

The TrainSim metamodel defines the language's abstract syntax (*i.e.*, the concepts of the language) and its static semantics (*i.e.*, the valid combinations of the language's concepts). Attaching semantics to the modelling language is done by defining *model transformations* which can either be endogenous or exogenous, as explained in Chapter 1. In the example, the latter approach is chosen, and a transformation is developed which maps TrainSim models onto PetriNet [8] models. A PetriNet model consists of places which contain tokens and transitions between these places. The metamodel of the PetriNet language is shown in Figure 3.1(b). PetriNets have well-defined dynamic semantics and are used to prove safeness and deadlock properties or simulate the behaviour of the system. As such, the semantics of a TrainSim model are defined by the semantics of the PetriNet model resulting from applying the TrainSim-to-PetriNet transformation. All properties of the PetriNet model are also properties of the original TrainSim model (*i.e.*, if the PetriNet model is *safe*, the TrainSim model is safe). In other words, the PetriNet language is the *semantic domain* of the TrainSim language.

In Figure 3.2, an example TrainSim model is shown, together with its semantic mapping onto PetriNets. Figure 3.2(a) is a graphical representation of the model, containing a small railroad network. Figure 3.2(b) shows the definition of the transformation that transforms TrainSim models into PetriNet models, given in rules. A rule consists of a Left-Hand-Side (LHS), which contains the pattern that is to be matched in the source model, a Right-Hand Side (RHS), which contains the resulting pattern in the target model, and zero or more Negative-Application Conditions (NACs), which contain forbidden patterns in the source graph. Once a rule matches its LHS and none of its NACs in the source model, it creates the pattern contained in its RHS in the target model. The rules are executed according to a scheduling strategy. In the example, rules are executed according to their priority, where rules at the top have highest priority, and rules on the bottom have lowest priority. As an example, the topmost rule states that a rail which does not have a train on it is transformed into two PetriNet places and two PetriNet transitions. The place with label *free* contains a token when the rail is free, and the place with label *rail* contains a token when there is a train on the rail.

For brevity reasons, only the rules mapping the individual TrainSim structures on

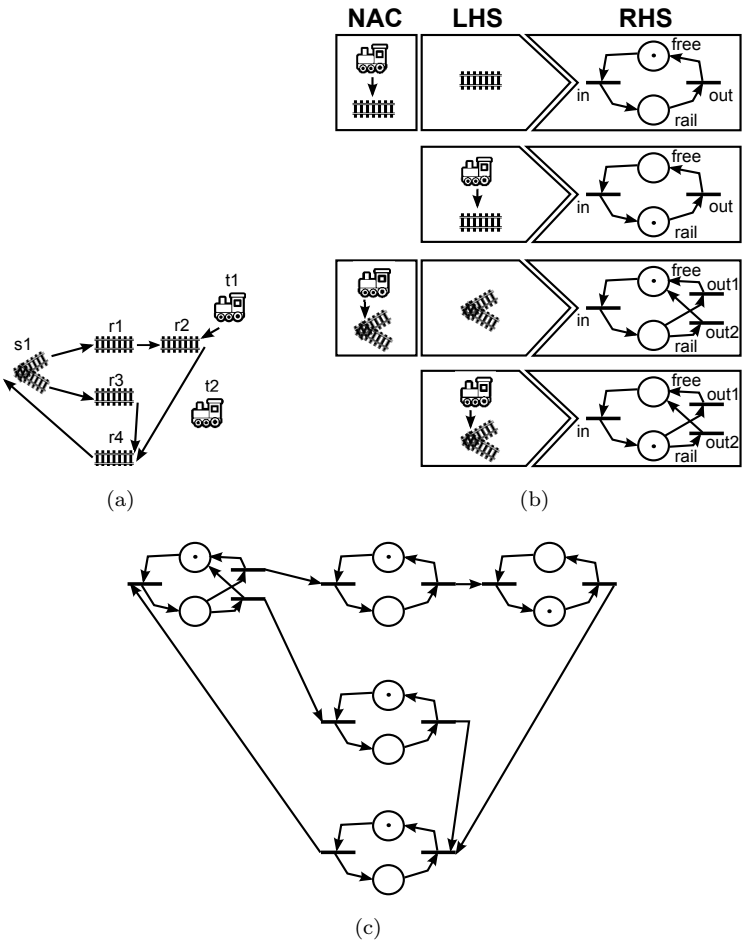


FIGURE 3.2: (a) An example TrainSim model, (b) The TrainSim-to-PetriNet transformation (semantic mapping), in the form of rules, (c) The resulting PetriNet model.

PetriNet structures are shown. The PetriNet which is the result of applying the semantic mapping transformation on the model of Figure 3.2(a) is shown in Figure 3.2(c). There, we see that the individual PetriNet structures are connected by their transitions. The rules which create these connections are not shown in Figure 3.2(b), although they are part of the transformation.

3.2 An Example Evolution Scenario

In the development process of the TrainSim language, adaptations have to be made to answer to the demands of users, developers and domain experts and to react to changes in the domain, as well as the implementation target domain. In this section, an example evolution scenario is introduced, in which the metamodel of the TrainSim language is adapted in a way which breaks the conformance relation between models created in the

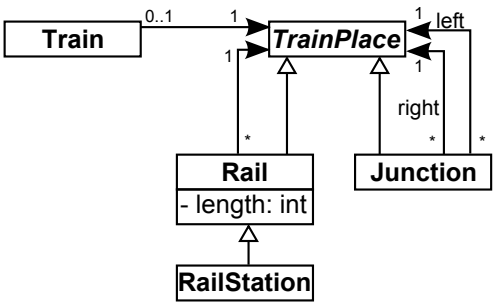


FIGURE 3.3: The evolved TrainSim metamodel.

initial version of the language and the metamodel of the evolved TrainSim language. As the TrainSim language is used as the domain language for a transformation (the TrainSim-to-PetriNet transformation), that transformation has to be adapted as well. Refer to Figure 1.2(b) for a general overview of domain metamodel evolution.

Figure 3.3 shows the evolved metamodel. Below, all adaptations made are listed and categorized into non-breaking, breaking and resolvable and breaking, unresolvable changes, as proposed in [12].

- **Non-Breaking Changes**
  - Addition of the *RailStation* class as a subclass of the *Rail* class.
- **Breaking, Resolvable Changes**
  - Renaming of the *Split* class to *Junction*.
- **Breaking, Unresolvable Changes**
  - Changing the cardinality of the relation between *Train* and *TrainPlace*: now, every train needs to be on a train place.
  - Splitting the relation between *Junction* and *TrainPlace* into two relations: a 'left' and 'right' relation.
  - Addition of the length attribute in the *Rail* class. Note that for this change, instance models can be migrated automatically with the construction of a migration transformation (which replaces each sequence of n rails by a single rail with length n), but this transformation has to be constructed manually, it cannot be automatically generated.

This classification is only valid for metamodel and model co-evolution. A different classification has to be made when considering metamodel and transformation co-evolution. For instance, additive changes can no longer be classified as non-breaking changes: in [23], Levendovszky et al. introduce the category of *fully semantic operations*: if we want the transformation to cover the whole input domain (which, in this case, we want, because each concept should be mapped onto a concept in the semantic domain), then a rule has to be added manually to the semantic mapping transformation.

In the remaining chapters, this evolution scenario is developed in the metamodelling kernel Ark. In Chapter 4, the TrainSim language and the examples given in this chapter are modelled in Ark. In Chapter 5, we delve deeper into the modelling of transformations, and in particular, we develop the techniques needed to develop transformations in Ark. In Chapter 6, the evolution scenario presented in this chapter is developed in Ark, using the techniques introduced in Chapter 4 and Chapter 5.

Chapter 4

Metamodelling in Ark

In this chapter, we will introduce the metamodelling and modelling facilities of Ark, the metamodelling kernel used throughout this thesis. In Section 4.1, Ark is introduced. In Section 4.2, we explain the concept of the metaverse, which is a central repository of modelling artefacts. In Section 4.3, the example TrainSim language, introduced in the previous chapter, is metamodelled in Ark. In Section 4.4, the semantics of the ArkM3 action language are discussed. In Section 4.5, we explain how each ArkM3 structure is physically represented. Lastly, Section 4.6 explains how ArkM3 structures are serialised.

4.1 Ark, the Metamodelling Kernel for Domain Specific Modelling

AToMPM, A Tool for Multi-Paradigm Modelling, is currently under development in the MSDL research group. It is the successor of AToM^3, A Tool for Multi-paradigm and Meta-Modelling [43]. For AToMPM, the new metamodelling kernel Ark (AToMPM Reusable Kernel) was developed [44]. The kernel allows to model all aspects of an MDE system, from metamodels, to models and transformations, to action and constraint code which is contained in these models. This is a key feature of the kernel: every aspect of a modelling system is explicitly modelled. This makes it possible to design uniform methods for transforming all elements of a modelling system, and ultimately to support evolution of modelling artefacts using model transformation. The metamodel of Ark, ArkM3 (AToMPM Reusable Kernel Meta-Meta-Model) serves as the metametamodel,

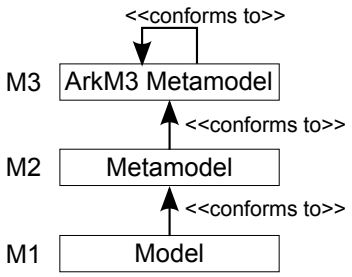


FIGURE 4.1: The metamodelling structure of Ark.

which the metamodels of all languages developed in AToMPM conform to. It is on the M3 level in the OMG/MOF standard [45].

In Figure 4.1, the metamodelling structure of Ark is shown. The ArkM3 metamodel is on the *M3* level, and conforms to itself. Metamodels of modelling languages created in Ark conform to the ArkM3 metamodel. Models created in a language created in Ark conform to the metamodel of that language.

4.2 The Metaverse, and CRUD Operations

The metaverse is the collection of all modelling artefacts (metamodels, models, transformations, actions, ...) which have been created by modellers, developers and users. It includes every view and version of these elements, and as everything of interest (design documents, requirements, tests, ...) is explicitly modelled using the appropriate formalism(s) at the right level of abstraction, the metaverse literally covers every aspect of an MDE system. It is unbounded (i.e., it can contain a virtually infinite amount of MDE systems, including all versions of these systems) and accessible at all times.

Figure 4.2 is a visual representation of the metaverse. A central aspect of the metaverse is its support for Create, Read, Update and Delete (CRUD) operations. By using these four basic operations, a modeller can manipulate the metaverse in different ways:

- **Create**: Creates an element in a specified location. To create an element, we need to know what kind of element it is (a package, a type, an instance of a type, ...), what the type of the element is (either a type defined in ArkM3 or a user-defined type), the location where the element is to be created and any parameters specific to the kind of element we are creating (for instance, the two endpoints of

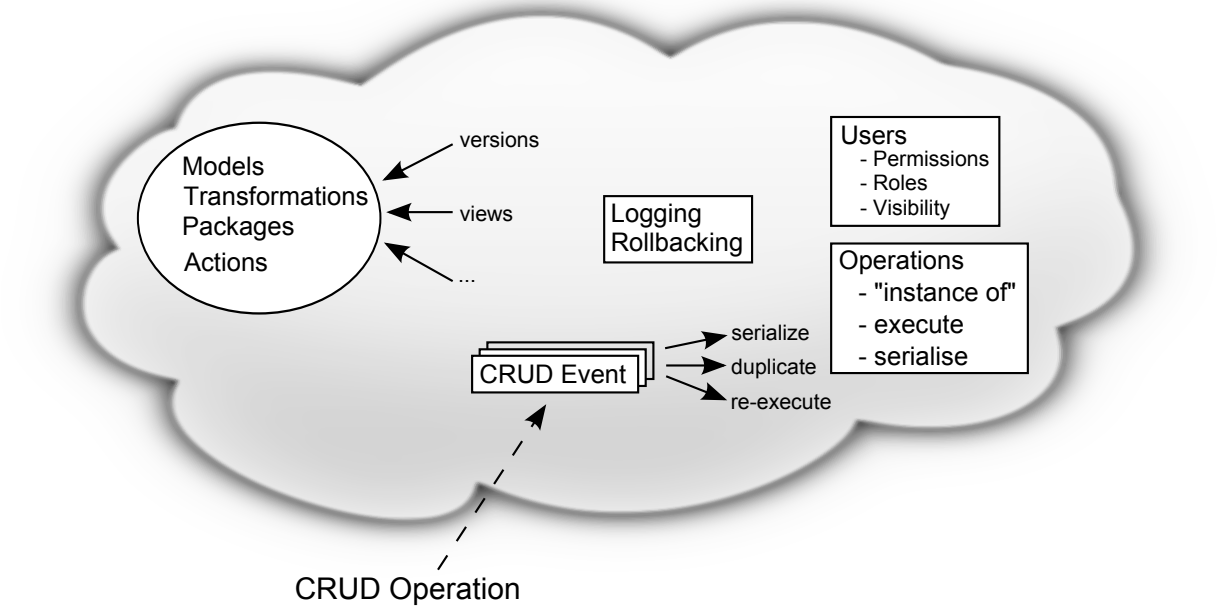


FIGURE 4.2: The conceptual metaverse.

an association instance). By creating an element, the correct ArkM3 elements are instantiated in the metaverse and they are directly accessible by other operations.

- **Read**: Obtain a reference to a previously created element. To read an element, it is sufficient to know its location. Once a reference to an element is obtained, it can be manipulated through other operations.
- **Update**: Update any meta-property of an element.
- **Delete**: An element can be deleted from a specified location. Once the element is deleted, it is no longer possible to obtain a reference to it.

A CRUD operation is currently received and executed immediately by the metaverse. A CRUD operation is irreversible and there is no logging of operations implemented. Future work consists of implementing an event-based approach to CRUD operations. Then, a CRUD operation is received by the metaverse and stored as an event. A number of such CRUD events can form an event trace, which can be serialised to disk, or duplicated and re-executed on another machine. This method of grouping events into event traces and saving then eventually makes it possible to support logging and rollbacking of operations.

From the modeller's point of view, all elements in the metaverse are directly accessible by referencing its location. For instance, if a modeller wants to create a model conforming to a metamodel, he needs to know the location of the metamodel. There is no need for an explicit import statement, as everything is available in the metaverse from the point it is defined by referencing its location.

```
formalisms.ExampleLanguage.A

models.ExampleLanguage.a

models.ExampleLanguage.a.attr
```

Listing 4.1: Paths in Ark.

In Listing 4.1, three example paths are shown. A path is used to navigate through the package structure of the metaverse. For instance, the path on line 1 refers to the class *A* found in the package *ExampleLanguage*, which in its turn can be found in the top-level package *formalisms*. On line 2, a path is shown that refers to the instance *a* in the *ExampleLanguage* package, which is located in the *models* package. Lastly, on line 3, a reference is made to the attribute *attr* of the *a* instance. In this way, we can navigate the package, class, and property structure of the metaverse.

```
formalisms.ExampleLanguage.A a
```

Listing 4.2: An example instantiation of a class in A.

When a path is used to refer to an element in the metaverse, a *reference* is constructed internally, containing the string representation of the path. Only when the element referred to by the path is needed, is it read from the metaverse to obtain the actual element referenced by the path. This is a form of *lazy evaluation*. For instance, in Listing 4.2, class *A* is instantiated, and the instance is called *a*. Behind the scenes, the instance *a* only contains a reference to its type, and that reference only contains the path to the class *A*. On the moment an operation is executed on *a* that requires its type to be accessed (for instance, a *get_type* operation), the kernel obtains a reference to that type. This is a useful feature, as elements may dynamically be adapted. In this way, always the most recent version of the element referred to by a path is obtained.

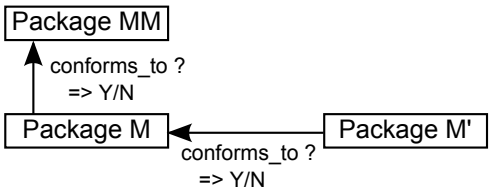


FIGURE 4.3: Conformance checking in Ark.

4.3 Metamodels and Models

Central to any MDE system are metamodels, which describe the concepts of a modelling language and the relations between them, and models, conforming to the metamodels of the language they were created in. In Ark, the "conforms-to"-relation is not enforced by the kernel. In other words, it is possible to create non-conforming models. Allowing non-conforming models has a number of advantages. First, it allows the modeller to incrementally build a model, saving intermediate (possibly non-conforming) versions of the model. Second, it allows the loading of models which no longer conform to their metamodel, which is necessary when the metamodel of a language evolves and the model is migrated at a later point in time.

Metamodels and models are represented in Ark as a package. The package contains all elements (classes, associations and properties) of the metamodel or model. In the case of a metamodel, it contains all definitions of concepts in the defined modelling language and their relations. In the case of models, it contains instances of the concepts and relations in the metamodel of the language in which the model is created. As the conformance relation is not enforced at creation time, it will be possible (in the future) to check algorithmically whether one package (the model's package) conforms to another package (the metamodel's package). This is visually represented in Figure 4.3. Note that the check can be performed on any two packages: this enables a model to also act as a metamodel (in the example shown in the figure, the package **M** could both conform to package **MM** *and* be a metamodel, to which package **M'** conforms).

```
1 package formalisms.trainsim
2   package trainsimMM
3     class Train
4
5     abstract class TrainPlace
6     class Rail(TrainPlace)
```

```
7     class Split(TrainPlace)
8
9     association T_on_TP
10      t: from Train<0..1>
11      tp: to TrainPlace<0..1>
12
13    association R_to_TP
14      r: from Rail<0..*>
15      tp: to TrainPlace<1..1>
16
17    association S_to_TP
18      s: from Split<0..*>
19      tp: to TrainPlace<2..2>
```

LISTING 4.3: The TrainSim metamodel in the textual syntax of Ark.

At this point in time, (meta)models in Ark are specified using a textual syntax [46]. In Section 3.1, we introduced the TrainSim modelling language. Listing 4.3 shows the TrainSim metamodel, modelled in the textual syntax of Ark. The location of the metamodel is the package with path **formalisms.trainsim.trainsimMM**, as defined on lines 1-2. In that package, the concepts of the language (Train, TrainPlace, Rail, and Split) are modelled as classes (lines 3-7). The relations between classes are modelled as associations (lines 9-19). As the metamodel is an instance of (conforms to) the metametamodel ArkM3, constructing a metamodel is achieved by instantiating concepts of that metametamodel: classes, associations and properties.

```
1  package models.trainsim
2    package example
3      formalisms.trainsim.trainsimMM.Rail r1
4      formalisms.trainsim.trainsimMM.Rail r2
5      formalisms.trainsim.trainsimMM.Rail r3
6      formalisms.trainsim.trainsimMM.Rail r4
7
8      formalisms.trainsim.trainsimMM.Split s1
9
```

```
10    formalisms.trainsim.trainsimMM.Train t1
11    formalisms.trainsim.trainsimMM.Train t2
12
13    formalisms.trainsim.trainsimMM.R_to_TP r1_to_r2
14      r = r1
15      tp = r2
16    formalisms.trainsim.trainsimMM.R_to_TP r2_to_r4
17      r = r2
18      tp = r4
19    formalisms.trainsim.trainsimMM.R_to_TP r3_to_r4
20      r = r3
21      tp = r4
22    formalisms.trainsim.trainsimMM.R_to_TP r4_to_s1
23      r = r4
24      tp = s1
25
26    formalisms.trainsim.trainsimMM.S_to_TP s1_to_r1
27      s = s1
28      tp = r1
29    formalisms.trainsim.trainsimMM.S_to_TP s1_to_r2
30      s = s1
31      tp = r2
32
33    formalisms.trainsim.trainsimMM.T_on_TP t1_on_r2
34      t = t1
35      tp = r2
```

LISTING 4.4: An example TrainSim model in the textual syntax of Ark.

In Listing 4.4, the example model which was presented in Section 3.1 (see Figure 3.2(a)) is modelled in the textual syntax of Ark. The location of the model's package is **models.trainsim.example**. As this is a model in the TrainSim language, it contains instances of the concepts defined in the TrainSim metamodel. An instance of a class defined in a metamodel is created by explicitly referencing the location of the class.

The main difference between the metamodel shown in Listing 4.3 and the model shown in Listing 4.4 is that in the metamodel, ArkM3 concepts such as classes and associations are instantiated, while in the model, concepts from the metamodel are instantiated, such as *Rail* and *Train*. This is to ensure that the model conforms to the metamodel and the metamodel conforms to ArkM3. The main similarity is that under the hood, instances of classes and associations are represented in the same way as their types, which are classes and associations (both classes and their instances are instantiations of the ArkM3 class *Clabject*). This makes it possible to build a model conforming to the model in Listing 4.4, effectively changing the role of the latter to that of a metamodel. In a model, it is also possible to define constraints and actions on instances, or to define properties on instances which are not present in their type. It is not, however, possible to define new cardinalities in instances of associations.

4.4 Action Language Semantics

```
1  package formalisms.petrinet
2    package petrinetMM
3      class Place
4        int tokens
5
6        constraint t_c:
7          return tokens >= 0
8
9      class Transition
10
11     association P_to_T
12       p: from Place<0..*>
13       t: to Transition<0..*>
14
15     association T_to_P
16       t: from Transition<0..*>
17       p: to Place<0..*>
```

Listing 4.5: The PetriNet metamodel in the textual syntax of Ark.

Ark is designed in a way which enables every aspect of an MDE system to be modelled explicitly. It includes a metamodel of an action language, which is used to define actions and constraints on models and modelling elements. As an example, consider the PetriNet metamodel shown in Listing 4.5. There, a constraint is defined on the class *Place*, stating that a place can only contain a positive amount of tokens at all times. The kernel can check this constraint when the model is created, edited or transformed. The constraint itself is metamodelled explicitly and conforms to the ArkM3 metametamodel, which contains classes such as *ReturnStatement*, effectively making the constraint a model itself. This enables instances of the action language (actions and constraints) to be the subject of structural model transformation, without the modeller resorting to other methods of transforming action code, such as using regular expressions, which would be necessary if actions are modelled as a regular string value. The structure of an action or a constraint is similar to an abstract syntax tree: the textual definition is parsed, and an abstract syntax tree, consisting of ArkM3 elements, is created from this textual definition. In Ark, transformations can be developed which match the structure of the action or constraint and transform it structurally, by creating, deleting or updating the ArkM3 structures that make up the action or constraint.

```
1  package formalisms.examples
2    package example_function_MM
3      class A
4        int val = 0
5
6        {A, int : int} do_plus
7        do_plus = {A self, int p : int}:
8          return self.val + p
9
10     class B
11       int val = 0
12       {B, A : bool} is_valid
13
14   package example_function_M
15     B b:
16       val = 10
```

```
17      is_valid = {B self, A a : bool}:
18        return a.do_plus(self.val) > 15
19
20    do_computation = {A a, B b : void}:
21      if (b.is_valid()):
22        b.val = a.do_plus(b.val)
```

LISTING 4.6: Example uses of the action language in Ark.

The action language is used for other purposes as well. It is possible to define functions (in packages) and methods (in classes, as properties of those classes). In Listing 4.6, examples uses of the ArkM3 action language are shown.

A method or a function always has to be declared and defined. In class *A*, a method called *do_plus* is declared, accepting two parameters (an instance of class A and an integer), and returning an integer. On line 7 and 8, the function is defined. To define a function, the parameters are given names and a function body is provided. This function can be called on every instance of A and will always perform the same actions. In other words, the function *do_plus* cannot be redefined in instances of class A. It can, however, be overridden in subclasses of A.

In class *B*, a function called *is_valid* is declared, but not defined. In this case, instances of class B are allowed to provide the function definition, which can be seen on line 17 and 18. On line 18, moreover, we see a call to the function *do_plus* on the instance of class A which is passed as a parameter.

A function's definition in a package includes its declaration. It can be directly defined, as can be seen on line 20. The function is accessible from all elements in the metaverse by referencing its path. The function *do_computation* accepts two parameters and returns nothing. It calls the previously defined methods on instances of classes A and B.

Methods, functions, actions and constraints written in Ark action language can be executed. The textual definition is parsed and an abstract syntax tree is generated. It is possible to either *interpret* the action language, or *compile* it to, for instance, Python code. The latter is more optimized than the former, but it has a few challenges to deal with in the dynamic environment of Ark. As explained before, it is possible to transform action language fragments using model transformations. If a compiled version of

that fragment exists, it has to be recompiled in order for the transformed version to be executed instead of the original version. A solution is provided by Ark, by allowing to remove the compiled version of a fragment written in the Ark action language. During execution, the kernel will check whether a compiled version is present. If it is, that version is executed. If it is not, either the fragment is interpreted, or the fragment is compiled and run.

4.5 Physical Representation of ArkM3 Structures

At the conceptual level, from the modeller's point of view, ArkM3 consists of all needed meta-classes to create metamodels, models and other modelling artefacts of an MDE system. All ArkM3 structures also have a physical representation. This representation is used for serialisation (see Section 4.6) and can be operated upon by algorithms. The typed, attributed graph structure Himesis [47] was chosen for the physical representation of ArkM3 structures. In [48], we describe how each ArkM3 structure is mapped onto a Himesis structure.

A Himesis graph consists of a number of nodes and edges. Himesis graphs, nodes and edges all contain attributes, and the value of these attributes can be of any data type. Every ArkM3 structure has a representation in Himesis, and each ArkM3 element 'knows' what its physical representation looks like. In essence, the ArkM3 elements form a thin layer (interface) on top of Himesis structures, while they query and update this structure through the CRUD operations they receive from the user. Only ArkM3 packages are Himesis graphs. Other elements, belonging to that package, add their Himesis representation (consisting of a number of nodes and/or edges) to the package's Himesis graph.

The physical representation of an ArkM3 element contains all meta-information related to that element. The name and location of the ArkM3 class of the element is stored, along with all its attributes. This makes it possible to build an ArkM3 structure, starting from its physical representation in Himesis.

The result is that each modelling artefact created in Ark is a graph structure, consisting of typed, attributed nodes and edges. This makes it possible to develop efficient

algorithms that perform graph matching and graph rewriting, which is required for the efficient execution of transformations. This will be further explained in Chapter 5.

4.6 Serialisation

It is possible to serialise ArkM3 structures, in order to send them to a remote host or save the state of the metaverse at any point in time. The serialisation process is not tailored to ArkM3: the Himesis compilation algorithm is reused. As a result, the serialised version of an ArkM3 structure is its physical representation in Himesis, compiled to a file. The serialisation process is defined in more detail in [48]. In short, this is the serialisation procedure, followed by the de-serialisation procedure:

1. The Himesis serialisation algorithm is capable of serialising Himesis graphs. As only ArkM3 packages are represented physically as a Himesis graph, only packages can be serialised. When a serialisation request is received, the package delegates the call to its Himesis representation. The graph, containing all nodes and edges representing the elements in the package, is compiled to a file.

2. The file can be stored on an external device as a backup, or sent to a remote machine.

3. When the user wants to de-serialise the file, the ArkM3 structure represented by the serialised Himesis graph has to be rebuilt. The de-serialisation process iterates over all the nodes of the graph and builds the Himesis classes which are represented by these nodes and edges. By using the meta-information stored in the nodes and edges of the Himesis graph, it is able to rebuild the correct ArkM3 structure.

Chapter 5

Developing Model Transformations in Ark

Model transformations are called the heart and soul of MDE systems [49]. They have different purposes and uses in an MDE system, as explained in [7]. They are used, amongst others, for model refactoring, refinement, simulation and code synthesis. In this chapter, we explain how model-to-model transformations, either endogenous (the source and target model are the same) or exogenous (the source and target model differ) are developed in Ark. The approach presented here supports both types of transformation, but we focus on exogenous model transformations. This chapter is organized as follows. Section 5.1 explains how to explicitly model rule-based model-to-model transformations, meaning that transformations can be regarded as a special model, modelled in a domain-specific modelling (transformation) language, whose syntactic structure is defined in a metamodel. A positive consequence of this approach is that Higher-Order Transformations (HOTs) are facilitated - how these are modelled is explained in Section 5.2. Lastly, Section 5.3 explains how we developed a (naive) graph matching and rewriting algorithm which is capable of executing a model-to-model transformation. The algorithm is first developed in Python, and later in the ArkM3 action language.

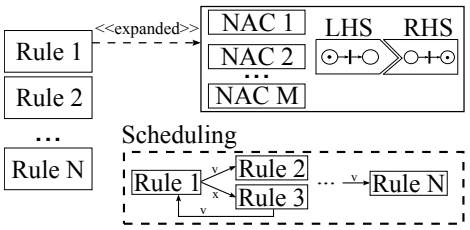


FIGURE 5.1: A rule-based specification of a transformation.

5.1 Explicit Modelling of Rule-Based Model-to-Model Transformations

In [50], Kühne et al. advocate the explicit modelling of transformations. By doing so, the advantages of metamodelling in general apply to the modelling of transformations as well: (1) the specification is not hidden in the code of a tool, making it easier to understand and correct, (2) one can reason about the specifications and the instance models they describe, (3) one may synthesize modelling environments from the specification and (4) this makes it easy for users to alter the specification instead of requiring a new tool release.

Model-to-model transformations can be specified in a number of ways. We focus on rule-based model-to-model transformations. This is visualized in Figure 5.1. A transformation consists of a number of rules and the order in which they need to be applied (known also as scheduling). A rule consists of a Left-Hand-Side (LHS), containing the pattern to be matched in the source model, a Right-Hand-Side (RHS), specifying how the matched part of the model should be rewritten and zero or more Negative Application Conditions (NACs), specifying the patterns that, when found, should stop the rule from being applied.

Of course, the patterns which appear in the LHS, RHS and NACs are very similar to the models (created in a particular modelling language) we want to transform. Therefore, we would like to reuse the metamodels of the domain and image modelling language for these pattern languages, instead of creating them from scratch. Starting from the original metamodel of the language creates a highly specialized transformation language, which only allows patterns specific to the modelling languages involved in the transformation, including the language-specific (visual) notations. The metamodel of the transformation language allows for the generation of a syntax-directed editing environment, which is

more constrained than a textual freehand-editing environment. At this moment, Ark does not allow for the generation of these editing environments, so all examples are still developed textually.

We cannot simply copy the metamodels of the source and target languages and use them as a pattern language, however. Firstly, patterns that appear in LHS, RHS and NACs are not necessarily well-formed models in the original modelling language. For instance, in the TrainSim modelling language an instance of the *Split* class needs exactly two outgoing associations to two *TrainPlaces*. However, it may be interesting to have a pattern which only contains an instance of a Split (as can be seen in Figure 3.2(b), where one of the rules' LHS contains only an instance of a Split). In the original TrainSim language, that is not a well-formed model. This means that the metamodel's constraints should be *relaxed*, in order for such patterns to be specified. Secondly, a few additions have to be made to the elements of the metamodel. It should be possible to identify elements across LHS, RHS and NACs: for endogenous transformations, we may want to update the attribute of an element in a rule, which means that we must have a way to denote that an element in the LHS corresponds to an element in the RHS. This is done by *augmenting* the metamodel by adding labels to the elements of the pattern metamodel. Thirdly, the data type of model element properties should be modified to allow the definition of constraints on properties (in precondition patterns) as well as actions to compute the new value of a property (in postcondition patterns). These are the three main concepts of RAMification: Relaxation, Augmentation and Modification. The authors of the paper describe an automatic process which creates a customized pattern language with minimal effort, starting from the original metamodel of the language.

```
1 package formalisms.transformations
2   package transformation
3     abstract class Step
4       bool isStart = False
5
6     class Transformation(Step)
7       string location
8     class Rule(Step)
9       string location
```

```
10    class Exhaust(Step)
11      {string} locations
12    class ExhaustRandom(Step)
13      {string} locations
14
15    association OnFailure
16      from_step: from Step<0..1>
17      to_step: to Step<1..1>
18
19    association OnNotApplicable
20      from_step: from Step<0..1>
21      to_step: to Step<1..1>
22
23    association OnSuccess
24      from_step: from Step<0..1>
25      to_step: to Step<1..1>
```

LISTING 5.1: The transformation metamodel in Ark. Adapted from [51].

We applied this pattern to our example TrainSim language, modelled in Ark. First, we developed a transformation metamodel which covers the scheduling part of a transformation, shown in Listing 5.1. A transformation consists of a number of *Steps*. A step is specified by its location, which is the package in which the step is modelled. This means that a transformation has explicit references to its steps, but the steps are modelled independently of the transformation. This encourages reuse of steps in other transformations. A step can either be another transformation (which means that the step executes the referenced transformation), a rule, or a set of rules which are executed until none of them matches. A difference is made between random and sequential execution of these rules. Steps are connected to each other by three different relations:

- *OnFailure*, which signifies that the target Step should be executed when there is a failure in the source Step (for instance, an exception which prevents the Step from being executed);

- *OnNotApplicable*, which signifies that the target Step should be executed when the source Step cannot be applied (for instance, there is no match to be found for the LHS of a rule);
- *OnSuccess*, which signifies that the target Step should be executed when the source Step has been successfully executed: *i.e.*, the source model has been rewritten according to the specifications in the LHS and RHS of a rule.

This allows a modeller to specify complex model transformations as a combination of rules.

```
1  package formalisms.transformations
2    package rule
3      abstract class PreConditionPattern
4        {{int: arkm3.object.Clabject}: bool} p_constraint
5      abstract class PostConditionPattern
6        {{int: arkm3.object.Clabject}: void} p_action
7
8      abstract class PatternElement
9        int __pLabel
10     abstract class PreConditionPatternElement(PatternElement)
11     abstract class PostConditionPatternElement(PatternElement)
12
13     abstract association PatternAssociation
14       int __pLabel
15     abstract association PreConditionPatternAssociation(PatternAssociation)
16     abstract association PostConditionPatternAssociation(PatternAssociation)
17
18     class NAC(PreConditionPattern)
19       string name
20     class LHS(PreConditionPattern)<1..1>
21     class RHS(PreConditionPattern)<1..1>
22
23     composition PreConditionPatternContents<0..*>
24       pattern: from PreConditionPattern<1..1>
```

```
25      element: to PreConditionPatternElement<0..*>
26
27    composition PostConditionPatternContents<0..*>
28      pattern: from PostConditionPattern<1..1>
29      element: to PostConditionPatternElement<0..*>
```

LISTING 5.2: The rule metamodel in Ark.

We need to be able to specify rules as well. In Listing 5.2, the rule metamodel as we modelled it in Ark is shown. There are two types of patterns: *PreConditionPatterns* (LHS and NACs) and *PostConditionPatterns* (RHS).

A precondition pattern has a *p_constraint* attribute, which is a function, receiving a match object (which is a mapping between pattern labels and the matched elements in the source model) as parameter and returning a boolean value. In this function, a global condition is specified which has to be satisfied in order for the pattern to match. A postcondition pattern has a *p_action* attribute, which specifies the action to be executed after the LHS has been rewritten according to the RHS pattern.

```
1  package example
2    package MM
3      class A
4        int test
5
6    package MM_Pre
7      class A(formalisms.transformations.rule.PreConditionPatternElement)
8        {{int: arkm3.object.Clabject}: bool} test
9
10   package MM_Post
11     class A(formalisms.transformations.rule.PostConditionPatternElement)
12       {{int: arkm3.object.Clabject}: void} test
13
14   package rule:
15     # # # #
16     # LHS #
17     # # # #
```

```
18    formalisms.transformations.rule.LHS lhs
19      p_constraint = {{int: arkm3.object.Clabject}: bool}:
20        ((example.MM.A) match[0]).test + ((example.MM.A) match[1]).test > 10
21
22    example.MM_Pre.A a0_lhs
23      __pLabel = 1
24    example.MM_Pre.A a1_lhs
25      __pLabel = 2
26
27    formalisms.transformations.rule.PreConditionPatternContents
28      pattern = lhs
29      element = a0_lhs
30
31    formalisms.transformations.rule.PreConditionPatternContents
32      pattern = lhs
33      element = a1_lhs
34
35    # # # #
36    # RHS #
37    # # # #
38    formalisms.transformations.rule.RHS rhs
39      p_action = {{int: arkm3.object.Clabject}: bool}:
40        ((example.MM.A) match[0]).test = ((example.MM.A) match[1]).test
41
42    example.MM_Post.A a0_rhs
43      __pLabel = 1
44    example.MM_Post.A a1_rhs
45      __pLabel = 2
46
47    formalisms.transformations.rule.PostConditionPatternContents
48      pattern = rhs
49      element = a0_rhs
50
```

```
51      formalisms.transformations.rule.PostConditionPatternContents
52        pattern = rhs
53        element = a1_rhs
```

LISTING 5.3: An example usage of rule constraints and actions.

An example usage of the *p_constraint* and *p_action* functions is shown in Listing 5.3. On lines 18-20, an LHS object is instantiated and its *p_constraint* property is given a value. As noted before, the function receives a mapping between the labels of pattern nodes (*i.e.*, values of the *__pLabel* attribute - refer to the explanation in the next paragraph) and the matched elements in the host graph as a parameter, and returns a boolean value. In this case, the constraint checks whether the sum of the values for the property *test* of the matched objects is greater than ten. Only if this constraint evaluates to *true* is the match valid.
On lines 38-40, an RHS object is instantiated and its *p_action* property is given a value. This function is executed after the matched elements in the LHS have been replaced by the pattern in the RHS. In the example, the value of the *test* property of the object which is matched with label 2 is assigned to the *test* property of the object which is matched with label 1.

The contents of the patterns are *PatternElements*, which have a *__pLabel* attribute, enabling them to be identified across LHS, NACs and RHS patterns. The same goes for associations. All concepts of the pattern languages subclass either the class *PatternElement* or *PatternAssociation*. Both are connected to their pattern by the associations *PreConditionPatternContents* and *PostConditionPatternContents*. We model the LHS and NAC classes as subclasses of the PreConditionPattern class, where a name attribute is added to the NAC class to be able to identify instances when there is more than one NAC in a rule, and the RHS class as subclass of the PostConditionPattern class.

```
1  package formalisms.trainsim
2    package trainsimMM_Pre
3      class Train(formalisms.transformations.rule.PreConditionPatternElement)
4
5      class TrainPlace(formalisms.transformations.rule.PreConditionPatternElement)
6      class Rail(TrainPlace)
7      class Split(TrainPlace)
```

```
8
9     association T_on_TP(formalisms.transformations.rule.PreConditionPatternAssociation)
10      t: from Train<0..1>
11      tp: to TrainPlace<0..1>
12
13    association R_to_TP(formalisms.transformations.rule.PreConditionPatternAssociation)
14      r: from Rail<0..*>
15      tp: to TrainPlace<1..1>
16
17    association S_to_TP(formalisms.transformations.rule.PreConditionPatternAssociation)
18      s: from Split<0..*>
19      tp: to TrainPlace<0..2>
20
21  package trainsimMM_Post
22    class Train(formalisms.transformations.rule.PostConditionPatternElement)
23
24    class TrainPlace(formalisms.transformations.rule.PostConditionPatternElement)
25    class Rail(TrainPlace)
26    class Split(TrainPlace)
27
28    association T_on_TP(formalisms.transformations.rule.PostConditionPatternAssociation)
29      t: from Train<0..1>
30      tp: to TrainPlace<0..1>
31
32    association R_to_TP(formalisms.transformations.rule.PostConditionPatternAssociation)
33      r: from Rail<0..*>
34      tp: to TrainPlace<1..1>
35
36    association S_to_TP(formalisms.transformations.rule.PostConditionPatternAssociation)
37      s: from Split<0..*>
38      tp: to TrainPlace<0..2>
```

Listing 5.4: The RAMified TrainSim metamodel in Ark.

```
1  package formalisms.petrinet
2    package petrinetMM_Pre
3      class Place(formalisms.transformations.rule.PreConditionPatternElement)
4        {{int: arkm3.object.Clabject}: bool} tokens
5
6      class Transition(formalisms.transformations.rule.PreConditionPatternElement)
7
8      association P_to_T(formalisms.transformations.rule.PreConditionPatternAssociation)
9        p: from Place<0..*>
10       t: to Transition<0..*>
11
12     association T_to_P(formalisms.transformations.rule.PreConditionPatternAssociation)
13       t: from Transition<0..*>
14       p: to Place<0..*>
15
16   package petrinetMM_Post
17     class Place(formalisms.transformations.rule.PostConditionPatternElement)
18       {{int: arkm3.object.Clabject}: void} tokens
19
20     class Transition(formalisms.transformations.rule.PostConditionPatternElement)
21
22     association P_to_T(formalisms.transformations.rule.PostConditionPatternAssociation)
23       p: from Place<0..*>
24       t: to Transition<0..*>
25
26     association T_to_P(formalisms.transformations.rule.PostConditionPatternAssociation)
27       t: from Transition<0..*>
28       p: to Place<0..*>
```

Listing 5.5: The RAMified PetriNet metamodel in Ark.

The transformation metamodel and rule metamodel are reusable metamodels: the transformation metamodel defines a general transformation language, which is used to schedule rules and other transformations. The rule metamodel is reused in the RAMification

process. For each input and output language of a transformation, a RAMified version has to be constructed. In Listing 5.4, the RAMified TrainSim metamodel is shown. One metamodel, called *trainsimMM_Pre*, defines the precondition pattern language, the other, *trainsimMM_Post*, the postcondition pattern language. The RAMified PetriNet metamodel is shown in Listing 5.5. Both metamodels contain exactly the same classes as the metamodels they originate from, with a few modifications, dictated by the RAMification process:

- *Relaxation*: Firstly, the *TrainPlace* class has been made concrete, instead of abstract. This allows the abstract class to appear in patterns. For instance, we might want to match *all* train places (Rail and Split instances) in an LHS pattern. Allowing this abstract class to appear in patterns avoids multiple definitions of the same rule for each concrete class. Secondly, the lower multiplicity of the *S_to_TP* association's target train place has been removed. This allows instances of the Split class to appear in patterns without any outgoing associations.

- *Augmentation*: Each class in the precondition pattern metamodel subclasses the *PreConditionPatternElement* class, and each class in the postcondition pattern metamodel subclasses the *PostConditionPatternElement* class. Similarly, associations subclass *PreConditionPatternAssociation* or *PostConditionPatternAssociation*. This adds the *__pLabel* property to each class and association in the pattern languages. It also allows the RAMified classes to appear in patterns, as they can now be connected by the *PreConditionPatternContents* and *PostConditionPatternContents* associations of the rule metamodel.

- *Modification*: The type of all properties of each class and association is changed. In the PetriNet precondition metamodel, the type of the property *tokens* is changed to a function which accepts a match object (which is a mapping between labels and matched elements in the source model) and returns a boolean value. This allows to model a condition on the value of the *tokens* value for each *Place* instance in a pattern. In the PetriNet postcondition pattern metamodel, the data type is similarly changed, but now allows to define an action which computes a new value for the property.

While the RAMification process can be fully automatised, we developed the examples in this thesis by hand. In fact, the RAMification process can be described as a model transformation, which transforms a metamodel (which conforms to the ArkM3 metamodel) into its RAMified version. This is future work.

The transformation, rule, and RAMified metamodels allow the modeller to specify rules, containing the necessary patterns, and the scheduling of the rules. The kernel is then responsible for finding all matching submodels of the input model, as specified in the LHS and NACs, and rewriting them as specified in the RHS. In Section 5.3, we will develop a graph matching and rewriting algorithm responsible for executing transformations in Ark.

As an example, the TrainSim-to-PetriNet transformation (see Figure 3.2(b)) is modelled in Ark in Listing A.1. On lines 2-8, the scheduling of the transformation is defined using the *Exhaust* construct, which executes its rules in the order they are listed. The four rules which the transformation uses, *rail_to_petrinet*, *rail_train_to_petrinet*, *split_to_petrinet*, and *split_train_to_petrinet* are modelled in subpackages of the *formalisms.trainsim. trainsim_to_petrinet* package.

5.2 Higher-Order Transformations

In the previous section, we explained how we enabled the explicit modelling of rule-based model-to-model transformations in Ark. As a result, transformations in Ark are models themselves, which in turn means they can be transformed. A transformation that has a transformation model as input and/or output is called a Higher-Order Transformation (HOT). HOTs are an important part of our proposed solution for modelling language evolution, as they are used for transformation migration (see Chapter 6). In this section, we will explore how HOTs are modelled in Ark.

Figure 5.2 shows an example HOT. In that example, we want to transform the LHS of a rule which contains a PetriNet place (for the PetriNet metamodel, see Figure 3.1(b)). The LHS of the HOT matches a LHS which contains a PetriNet place. Keep in mind that the source model we are rewriting is now a pattern model, which contains instances of concepts of a RAMified metamodel. This means that the properties we are matching are constraints or actions. In other words, the *tokens* attribute of the PetriNet place in the

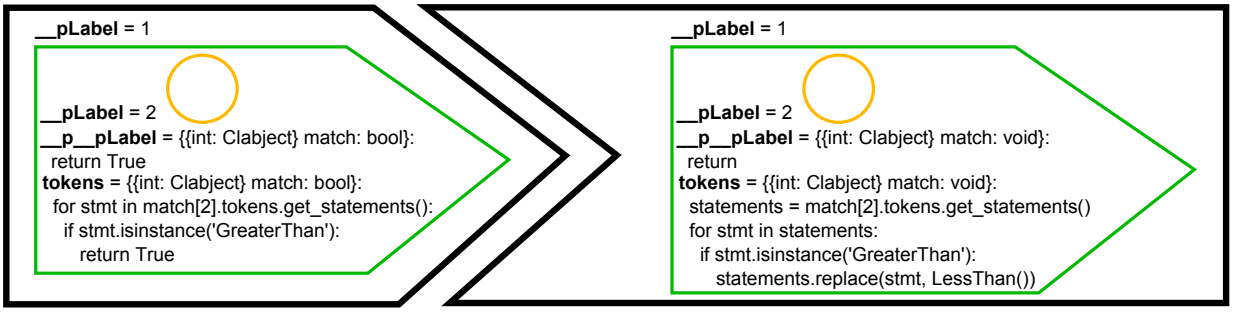


FIGURE 5.2: An example of a HOT.

HOT is a constraint over a constraint. The constraint matches those constraints which contain a 'greater than' statement. It iterates over all statements of the constraint, and when it finds an instance of the ArkM3 class *GreaterThan*, it returns *true*. The difference between *__pLabel* and *__p__pLabel* is explained later in this section. Basically, the *__pLabel* attribute is the label of the element in the pattern, and the *__p__pLabel* attribute is a constraint over the *__pLabel* attribute of the element in the source pattern we are rewriting. The RHS of the HOT rewrites the *tokens* constraint, by replacing the *GreaterThan* instance by a *LessThan* instance.

From the example, we can see what the necessary elements are for modelling a HOT. First, it should be possible for transformations and rule elements to appear in patterns. This means that, just as in the previous section, the transformation and rule metamodel have to be RAMified. Second, while the metamodels of the PetriNet and TrainSim languages have been RAMified already, resulting in the metamodels of the pattern languages used in the definition of rules and transformations, these metamodels have to be RAMified a second time. It is possible to apply the RAMification process a second time, because the RAMified metamodel is a metamodel conforming to the ArkM3 metamodel. This allows for the definition of actions and constraints on the actions and constraints which are defined in the patterns of the rule which is being transformed by the HOT.

```
1 package formalisms.transformations
2   package transformation_Pre
3     class Step(formalisms.transformations.rule.PreConditionPatternElement)
4       {{int: Clabject}: bool} isStart
5
6     class Transformation(Step)
7       {{int: Clabject}: bool} location
```

```
8     class Rule(Step)
9       {{int: Clabject}: bool} location
10    class Exhaust(Step)
11      {{int: Clabject}: bool} locations
12    class ExhaustRandom(Step)
13      {{int: Clabject}: bool} locations
14
15    association OnFailure(formalisms.transformations.rule.PreConditionPatternAssociation)
16      from_step: from Step<0..1>
17      to_step: to Step<0..1>
18
19    association OnNotApplicable(formalisms.transformations.rule.PreConditionPatternAssociation)
20      from_step: from Step<0..1>
21      to_step: to Step<0..1>
22
23    association OnSuccess(formalisms.transformations.rule.PreConditionPatternAssociation)
24      from_step: from Step<0..1>
25      to_step: to Step<0..1>
26
27  package transformation_Post
28    class Step(formalisms.transformations.rule.PostConditionPatternElement)
29      {{int: Clabject}: void} isStart
30
31    class Transformation(Step)
32      {{int: Clabject}: void} location
33    class Rule(Step)
34      {{int: Clabject}: void} location
35    class Exhaust(Step)
36      {{int: Clabject}: void} locations
37    class ExhaustRandom(Step)
38      {{int: Clabject}: void} locations
39
40    association OnFailure(formalisms.transformations.rule.PostConditionPatternAssociation)
```

```
41        from_step: from Step<0..1>
42        to_step: to Step<0..1>
43
44      association OnNotApplicable(formalisms.transformations.rule.PostConditionPatternAssociation)
45        from_step: from Step<0..1>
46        to_step: to Step<0..1>
47
48      association OnSuccess(formalisms.transformations.rule.PostConditionPatternAssociation)
49        from_step: from Step<0..1>
50        to_step: to Step<0..1>
```

LISTING 5.6: The RAMified Transformation metamodel in Ark.

The RAMified transformation metamodel is shown in Listing 5.6. Exactly the same relax, augment and modify operations as on the TrainSim and PetriNet metamodels have been performed, as described in the previous section.

```
1  package formalisms.transformations
2    package rule_Pre
3      class PreConditionPattern(formalisms.transformations.rule.PreConditionPatternElement)
4        {{int: Clabject}: bool} p_constraint
5      class PostConditionPattern(formalisms.transformations.rule.PreConditionPatternElement)
6        {{int: Clabject}: bool} p_action
7
8      class PatternElement(formalisms.transformations.rule.PreConditionPatternElement)
9        {{int: Clabject}: bool} __p__pLabel
10     class PreConditionPatternElement(PatternElement)
11     class PostConditionPatternElement(PatternElement)
12
13     association PatternAssociation(formalisms.transformations.rule.PreConditionPatternElement)
14       {{int: Clabject}: bool} __p__pLabel
15     association PreConditionPatternAssociation(PatternAssociation)
16     association PostConditionPatternAssociation(PatternAssociation)
17
18     class NAC(PreConditionPattern)
```

```
19        {{int: Clabject}: bool} name
20      class LHS(PreConditionPattern)
21      class RHS(PreConditionPattern)
22
23      composition PreConditionPatternContents<0..*>(formalisms.transformations.rule.
         PreConditionPatternAssociation)
24        pattern: from PreConditionPattern<1..1>
25        element: to PreConditionPatternElement<0..*>
26
27      composition PostConditionPatternContents<0..*>(formalisms.transformations.rule.
         PreConditionPatternAssociation)
28        pattern: from PostConditionPattern<1..1>
29        element: to PostConditionPatternElement<0..*>
30
31    package rule_Post
32      class PreConditionPattern(formalisms.transformations.rule.PostConditionPatternElement)
33        {{int: Clabject}: void} p_constraint
34      class PostConditionPattern(formalisms.transformations.rule.PostConditionPatternElement)
35        {{int: Clabject}: void} p_action
36
37      class PatternElement(formalisms.transformations.rule.PostConditionPatternElement)
38        {{int: Clabject}: void} __p__pLabel
39      class PreConditionPatternElement(PatternElement)
40      class PostConditionPatternElement(PatternElement)
41
42      association PatternAssociation(formalisms.transformations.rule.PostConditionPatternElement)
43        {{int: Clabject}: void} __p__pLabel
44      association PreConditionPatternAssociation(PatternAssociation)
45      association PostConditionPatternAssociation(PatternAssociation)
46
47      class NAC(PreConditionPattern)
48        {{int: Clabject}: void} name
49      class LHS(PreConditionPattern)
```

```
50      class RHS(PreConditionPattern)
51
52      composition PreConditionPatternContents<0..*>(formalisms.transformations.rule.
         PreConditionPatternAssociation)
53        pattern: from PreConditionPattern<1..1>
54        element: to PreConditionPatternElement<0..*>
55
56      composition PostConditionPatternContents<0..*>(formalisms.transformations.rule.
         PreConditionPatternAssociation)
57        pattern: from PostConditionPattern<1..1>
58        element: to PostConditionPatternElement<0..*>
```

LISTING 5.7: The RAMified Rule metamodel in Ark.

Listing 5.7 shows the RAMified rule metamodel. Again, the metamodel is relaxed, augmented and modified according to the RAMification process. However, there is one exception. In each pattern, we want to be able to identify all elements by their *__pLabel* attribute. This includes precondition patterns (LHS, NACs) and postcondition patterns (RHS) and their contents. These contents are RAMified *PatternElements*,and they already have a *__pLabel* attribute. In the RAMification process, we would normally modify the data type of this attribute to allow constraints and actions to be defined and we would leave the name of the property intact. In the rule metamodel, we cannot do that, as we wouldn't have a *__pLabel* attribute to identify the elements in the patterns of a HOT. That is why we make an exception. The original *__pLabel* attribute of the *PatternElement* class is renamed to *__p__pLabel* in the RAMified metamodel. This resolves any ambiguities between the original and the RAMified version: the *__p__pLabel* attribute is used to specify constraints and actions on the *__pLabel* attribute of elements in the host model. This allows to RAMify a metamodel a virtually infinite amount of times. RAMifying a metamodel a third time allows to transform models of HOTs. The classes and associations of this metamodel would contain at least the attributes *__p__p__pLabel*, *__p__pLabel*, and *__pLabel*. The kernel will need to be aware of this, as it needs to know that a constraint or action with the name *__p__pLabel* is actually a constraint on the *__pLabel* attribute of the element in the host model.

```
1   package formalisms.petrinet
```

```
2  package petrinetMM_Pre_Pre
3    class Place(formalisms.transformations.rule_Pre.PreConditionPatternElement)
4      {{int: arkm3.object.Clabject}: bool} tokens
5
6    class Transition(formalisms.transformations.rule_Pre.PreConditionPatternElement)
7
8    association P_to_T(formalisms.transformations.rule_Pre.PreConditionPatternAssociation)
9      p: from Place<0..*>
10     t: to Transition<0..*>
11
12   association T_to_P(formalisms.transformations.rule_Pre.PreConditionPatternAssociation)
13     t: from Transition<0..*>
14     p: to Place<0..*>
15
16 package petrinetMM_Pre_Post
17   class Place(formalisms.transformations.rule_Post.PreConditionPatternElement)
18     {{int: arkm3.object.Clabject}: void} tokens
19
20   class Transition(formalisms.transformations.rule_Post.PreConditionPatternElement)
21
22   association P_to_T(formalisms.transformations.rule_Post.PreConditionPatternAssociation)
23     p: from Place<0..*>
24     t: to Transition<0..*>
25
26   association T_to_P(formalisms.transformations.rule_Post.PreConditionPatternAssociation)
27     t: from Transition<0..*>
28     p: to Place<0..*>
29
30 package petrinetMM_Post_Pre
31   class Place(formalisms.transformations.rule_Pre.PostConditionPatternElement)
32     {{int: arkm3.object.Clabject}: bool} tokens
33
34   class Transition(formalisms.transformations.rule_Pre.PostConditionPatternElement)
```

```
35
36      association P_to_T(formalisms.transformations.rule_Pre.PostConditionPatternAssociation)
37        p: from Place<0..*>
38        t: to Transition<0..*>
39
40      association T_to_P(formalisms.transformations.rule_Pre.PostConditionPatternAssociation)
41        t: from Transition<0..*>
42        p: to Place<0..*>
43
44    package petrinetMM_Post_Post
45      class Place(formalisms.transformations.rule_Post.PostConditionPatternElement)
46        {{int: arkm3.object.Clabject}: void} tokens
47
48      class Transition(formalisms.transformations.rule_Post.PostConditionPatternElement)
49
50      association P_to_T(formalisms.transformations.rule_Post.PostConditionPatternAssociation)
51        p: from Place<0..*>
52        t: to Transition<0..*>
53
54      association T_to_P(formalisms.transformations.rule_Post.PostConditionPatternAssociation)
55        t: from Transition<0..*>
56        p: to Place<0..*>
```

Listing 5.8: The RAMified PetriNet pattern language metamodel

Listing 5.8 shows the four metamodels which are the result of RAMifying the RAMified metamodel of the PetriNet language. There are four metamodels because we want to be able to specify four different kinds of patterns in HOTs:

1. Precondition patterns (LHS, NACs) in the precondition patterns (RHS) of the HOT. The contents of these patterns are instances of the elements in *petrinetMM_Pre_Pre*.

2. Precondition patterns in the postcondition patterns of the HOT. The contents of these patterns are instances of the elements in *petrinetMM_Pre_Post*.

3. Postcondition patterns in the precondition patterns of the HOT. The contents of these patterns are instances of the elements in *petrinetMM_Post_Pre*.

4. Postcondition patterns in the postcondition patterns of the HOT. The contents of these patterns are instances of the elements in *petrinetMM_Post_Post*.

For instance, in the example rule shown in Figure 5.2, a precondition pattern (LHS) appears in the LHS of the HOT, and that same precondition pattern appears in the RHS of the HOT.

```
1  package example.hot
2    package rule
3      # # # #
4      # LHS #
5      # # # #
6      formalisms.transformations.rule.LHS lhs
7
8      formalisms.transformations.rule_Pre.LHS lhs_lhs
9        __pLabel = 1
10
11     formalisms.petrinet.petrinetMM_Pre_Pre.Place place_lhs_lhs
12       __pLabel = 2
13           __p__pLabel = {{int: Clabject} match: bool}:
14                 return True
15       tokens = {{int: Clabject} match: bool}:
16         for stmt in match[2].tokens.get_statements():
17           if stmt.isinstance('GreaterThan'):
18             return True
19
20     formalisms.transformations.rule_Pre.PreConditionPatternContents lhs_lhs_to_place_lhs_lhs
21       pattern = lhs_lhs
22       element = place_lhs_lhs
23       __pLabel = 3
24
25     formalisms.transformations.rule.PreConditionPatternContents lhs_to_lhs_lhs
```

```
26        pattern = lhs
27        element = lhs_lhs
28
29      formalisms.transformations.rule.PreConditionPatternContents lhs_to_place_lhs_lhs
30        pattern = lhs
31        element = place_lhs_lhs
32
33      formalisms.transformations.rule.PreConditionPatternContents lhs_to_lhs_lhs_to_place_lhs_lhs
34        pattern = lhs
35        element = lhs_lhs_to_place_lhs_lhs
36
37      # # # #
38      # RHS #
39      # # # #
40      formalisms.transformations.rule.RHS rhs
41
42      formalisms.transformations.rule_Post.LHS lhs_rhs
43        __pLabel = 1
44
45      formalisms.petrinet.petrinetMM_Pre_Post.Place place_lhs_rhs
46        __pLabel = 2
47            __p__pLabel = {{int: Clabject} match: bool}:
48                  return
49        tokens = {{int: Clabject} match: void}:
50          statements = match[2].tokens.get_statements()
51          for stmt in statements:
52            if stmt.isinstance('GreaterThan'):
53              statements.replace(stmt, LessThan())
54
55      formalisms.transformations.rule_Post.PreConditionPatternContents lhs_rhs_to_place_lhs_rhs
56        pattern = lhs_rhs
57        element = place_lhs_rhs
58        __pLabel = 3
```

```
59
60      formalisms.transformations.rule.PostConditionPatternContents rhs_to_lhs_rhs
61        pattern = rhs
62        element = lhs_rhs
63
64      formalisms.transformations.rule.PostConditionPatternContents rhs_to_place_lhs_rhs
65        pattern = rhs
66        element = place_lhs_rhs
67
68      formalisms.transformations.rule.PreConditionPatternContents rhs_to_lhs_rhs_to_place_lhs_rhs
69        pattern = rhs
70        element = lhs_rhs_to_place_lhs_rhs
```

LISTING 5.9: The example HOT of Figure 5.2, modelled in Ark.

In Listing 5.9, the example HOT of Figure 5.2 is shown as it is modelled in Ark. The LHS of the rule is instantiated on line 6. On lines 8-9, the LHS which is part of the pattern in the LHS of the HOT is instantiated. Note that the metamodel of this instance is *rule_Pre*, the RAMified version of the *rule* metamodel. On lines 11-18, the PetriNet place instance is instantiated. Its metamodel is the twice RAMified PetriNet metamodel, called *petrinetMM_Pre_Pre*. Similarly, the RHS of the HOT is instantiated on line 40. The LHS pattern instance of the RHS is instantiated on lines 42-43. Its metamodel is now *rule_Post*, which contains the RAMified classes of the *rule* metamodel that appear in postcondition patterns. Lastly, the PetriNet place is instantiated on lines 45-53.

5.3 Graph Matching Algorithm

The rule and transformation metamodels developed in the previous section, together with the semi-automatic RAMification process, allow the explicit modelling of rule-based model transformations. In this section, we will develop a graph matching and rewriting algorithm which is capable of executing such a transformation specification. The algorithm is then added to Ark, allowing the execution of transformation models on host models.

```
1  match_and_rewrite(lhs, nacs, host_graph, rhs):
```

```
2         for m in match(lhs, host_graph):
3                 valid_match = True
4                 for nac in nacs:
5                         if match(nac, host_graph, initial_match = m):
6                                 valid_match = False
7
8                 if valid_match:
9                         rewrite(m, host_graph, rhs)
```

LISTING 5.10: The extend function of the graph matching algorithm.

The high-level match-and-rewrite algorithm's pseudocode is shown in Listing 5.10. The algorithm iterates over all matches found for the LHS in the host model. Then, for each match, it checks whether any NAC matches. The match for the LHS is used as an initial match, which means that all nodes in the NAC with a label found in the LHS are bound to the node in the host graph found in the match of the LHS. As an example, consider the first rule in Figure 3.2(b). The algorithm will first find a match for the rail found in the LHS. Assume that *r1* in the model of Figure 3.2(a) is matched. Then, we pass this match to the algorithm when we match the NAC, which means that the rail of the NAC is also bound to *r1*. The algorithm then continues finding a match for the NAC, trying to find a train instance which is connected to that rail (which is will fail to do, as only *r2* has a train instance connected to it).

The rest of this section is concerned with developing the matching and rewriting algorithms.

The rewriting algorithm is concerned with creating, deleting and updating elements in the host model, according to the RHS pattern. The algorithm iterates over all nodes of the RHS and makes a choice between one of the following actions:

- If a node with the same *__pLabel* is found in the LHS, the properties of the matched node in the host graph are updated by executing the property actions of the RHS node.
- If no node with the same *__pLabel* is found in the LHS, a new element is created in the host graph. The type of the newly created element is the non-RAMified version of the type of the RHS node. All properties are initialized to their default values, after which they are updated by executing the property actions of the RHS node.

- All matched LHS elements who do not have a corresponding element (with the same *__pLabel*) in the RHS are deleted from the host graph.

```
1  extend(state):
2    if mappingIsComplete(state) then
3      storeMatch(state)
4    end if
5    for p, s in suggestMapping(state) do
6      if areCompatible(p, s) then
7        if areSyntacticallyFeasible(p, s) then
8          if areSemanticallyFeasible(p, s) then
9            state.storeMapping(p, s)
10           extend(state)
11           state.undoMapping(p, s)
12         end if
13       end if
14     end if
15   end for
```

LISTING 5.11: The extend function of the graph matching algorithm.

The matching algorithm is a naive combination of the VF2 [52] and Ulmann [53] algorithms, as proposed in [51]. Central to the algorithm is the class *State*, which keeps track of where the algorithm is in its execution. Basically, the algorithm considers all <PatternNode, SourceNode> pairs and stores the mapping if it is *feasible*. It continues storing feasible mappings until a complete match is found, meaning that each pattern node is matched with a source node. A mapping of a pattern node and a source element is feasible if:

1. They are compatible, meaning they have the same number of incoming and outgoing associations;
2. They are syntactically feasible: the topology of the current mapping corresponds to a sub-graph of the pattern graph;
3. They are semantically feasible: the type of the pattern element corresponds to that of the source element and the constraints imposed by the pattern element on the properties of the source element are satisfied.

These checks are performed by the *extend* function, shown in Listing 5.11. The function iterates through all suggested mappings of <PatternNode, SourceNode> pairs. If a

feasible mapping is found (meaning that a morphism is found between an element of the source model and an element of the pattern model), the mapping is added to the current state. Once the match is complete, meaning that all pattern nodes have been mapped on a source node, the match is stored, and the algorithm attempts to construct a new complete match. In this way, all combinations of source and pattern nodes are examined and all matches for the pattern are found.

The goal of this section is to develop the graph matching algorithm in the ArkM3 action language. The algorithm is a complex benchmark to assess the action language's expressiveness, proving that the language can be used to model complex constraints and actions. Furthermore, the algorithm is used to demonstrate the executability of the modelled transformations. This is a necessary precondition for the evolution of modelling languages using model transformations, as explained in the next chapter.

We developed the algorithm in three steps. First, we developed the algorithm in Python, on a simple, self-made data structure. This initial step validated the algorithm and proved it was applicable on representative match/rewrite examples. We do not elaborate further on this initial version of the algorithm. Next, we adapted the algorithm in Python to work on the concrete representation of ArkM3 data structures, which are Python classes (see Section 5.3.1). Lastly, we developed the algorithm in ArkM3 action language (see Section 5.3.2).

5.3.1 In Python, on ArkM3 Data Structures

In this section, we will explain how the graph matching and rewriting algorithm was developed in Python. The algorithm is capable of rewriting a host model created in Ark, following the specifications of a transformation model created in Ark. Refer to Appendix B for the full ArkM3 metamodel, which is an updated version from the one found in [44]. The algorithm is shown in Listing C.1.

While developing this algorithm, missing functionalities in the ArkM3 classes were identified. This is a necessary step towards implementing the algorithm in the ArkM3 action language, as we need to make sure that every function is present and works as expected in ArkM3.

In what follows, we list the functionality which is required by the algorithm, that is not yet present in ArkM3. We then explain how the functionality was added.

1. The algorithm requires a way to check the type of an element. Two type checking functions are needed: one to check the *linguistic* type of an element (*i.e.*, answering the question 'Is this element a Clabject?' - in the rest of this section referred to as the 'typeof' operation), and one to check the *ontological* type of an element (*i.e.*, answering the question 'Is this element a PetriNet Place?' - in the rest of this section referred to as the 'instanceof' operation) [54]. The linguistic type of elements is used in the algorithm to distinguish between nodes of the model (which are represented by *Clabject* instances) and edges of the model (which are represented by *Association* and *Composition* instances). The ontological type of elements is used to check whether a mapping of an element of the source model and an element of the pattern model is semantically viable: they need to be of compatible types.

2. We need access to the metaverse. On our simple data structure, we added and removed nodes and edges using the interface of the data structure. Now, we need to add and remove elements from the source model through the CRUD operations provided by the metaverse (see Section 4.2 for more details on the metaverse).

3. Access to properties and the values of properties in association and object instances has to be provided. In the initial version of Ark, this functionality was limited.

Some of these functionalities were already present in Ark, others had to be added. In what follows, we explain how we implemented these different aspects of the algorithm.

1. **'instanceof' and 'typeof'**: In Python, we can check whether an element is of a specific type by using the *isinstance* method. In the following code sample, we check whether the element *el* is of type *Clabject*:

```
isinstance(el, Clabject)
```

   The *isinstance* operation in principle is a linguistic check, but depending on the application, it can act as an ontological check as well. For instance, if we metamodel the PetriNet language manually in Python, we could use the *isinstance* operation to check whether the element *el* is a *Place*:

```
isinstance(el, Place)
```

In our trivial data structure, there was no *ontological* typing. However, in ArkM3, each element is typed by an element in the metamodel which the model conforms to. In ArkM3, we both need an operation that checks the linguistic type and one that checks the ontological type of an element.

- The linguistic check is implemented as a method of the *Element* class. It checks whether the name of the class equals the string value passed as a parameter.

```
class Element:

  def is_instance(self, class_str):

    return str(self.__class__) == class_str
```

- The ontological check is implemented by using the typing system of ArkM3. An instance of a class is typed by that class. A class can have a number of super types (and those, in their turn can have super types). The ontological check checks whether an instance is typed ontologically by a class by obtaining a reference to the class, and checking whether the class is present in the type hierarchy of the instance, as demonstrated in the following code snippet:

```
def is_instance(x, metaverse):

        clazz = metaverse.read('formalisms.transformations.rule.PatternElement').

    _object

        types = set([x.get_type()]) + x.get_type().get_all_super_class()

        return types.contains(pattern_class)
```

2. As can be seen in the ontological check, access has to be provided to **the metaverse**. The reference to the metaverse is used to perform CRUD operations on models and metamodels, and their elements. In the Python algorithm, it is passed as an argument to the various functions which need it (such as the *match* and *rewrite* functions). In the algorithm which is written in the ArkM3 action language (see the next subsection for details), access will have to be provided as well.

Having access to the metaverse not only allows us to read elements, we can now also update, remove and create them. To remove an element *el* from a model *host_graph*, the following statement is used:

```
host_graph.get_ownedElements().remove(el)
```

As a model is represented by an ArkM3 package, it is sufficient to remove the element from the owned elements of the package.

Creating elements is more challenging. We need three parameters for this (as explained in Section 4.2):

- A location where to create the element.
- The type of the element.
- The values of the properties of the element.

We have a reference to our host graph, so the first point is trivial. A method was added to both the ArkM3 Package and Classifier classes called *get_path*, which returns the full path of the element. This results in a string representation of the location where to create the element. The second point is handled by the RAMified attributes, which in the right-hand side are actions. We create the element with default properties, and then let the action fill in the correct values.

The last point is the most challenging, as right now there is no relation between the elements in the RAMified metamodel and the original metamodel. We could create a link between them, either in the form of an inheritance relation or traceability links. However, at this moment we impose the restriction that MM_Post is always named in a similar way (<*original metamodel name*>*_Post*) and has the same parent package as the original metamodel. If we then also assume that all classes in the MM_Post package are called the same as in MM, the following statement creates the correct instance:

```
a_type = node.get_type().get_parent().get_location().get_value()[:-5] + '.' + node.get_type()
    .get_name().get_value()
arkm3.create(host_graph.get_location().get_value(), ArkM3.OBJECT, '', a_type, False)
```

This approach is only temporary, until a better method is developed to create a link between the RAMified metamodel and the original metamodel. One way of doing

this would be to encode the name of the original metamodel as a property of the RAMified metamodel. That way, the RAMified metamodel explicitly references the original metamodel.

3. Classes and associations in metamodels can have a number of properties. These properties have a type, a name, and a default value. Instances of these classes and associations can then assign values to those properties. For instance, in the PetriNet metamodel (Figure 3.1(b)), the *Place* class declares a property called *tokens*, of type *integer*. An instance of the class in a model then has the option to assign a value to that property. In ArkM3, a class and an instance of a class are both represented as a *Clabject*. The instance clabject has the class clabject as its type. A clear distinction is made between properties and property values; however, they are both accessed by the name of the property. So, if x is a subclass of the *PatternElement* class in the rule metamodel, the following code would retrieve the *__pLabel* attribute:

```
x.get_property(StringValue('__pLabel'))
```

If y is an instance of the *PatternElement* class, the *__pLabel* is given a value in y. The value is accessed with the following code:

```
y.get_propertyValue(StringValue('__pLabel'))
```

The addition of these elements allowed us to develop the graph matching and rewriting algorithm successfully to work on ArkM3 data structures. It is now possible to model transformations in ArkM3 textual syntax, parse them, and execute them using this algorithm. For an example transformation modelled in Ark, refer to Listing 5.9.

This algorithm is not very efficient. For large-scale models, it takes a long time to find the matches and rewrite the host models according to the pattern in the RHS. This is because the algorithm itself is not efficient (it is a naive algorithm which considers all possible mappings and does no pruning of the search tree), and it is written on the wrong level of abstraction, namely for host models in the ArkM3 data structure. In the future, a more efficient algorithm should be constructed which works directly on the physical representation of ArkM3 structures: Himesis graphs. It is expected that the use of T-Core [51] will speed up the algorithm significantly and allow for more advanced

features, such as a configurable iterator, which can execute the rule on all matches found, or on the first match found, or on a randomly selected match.

5.3.2 In ArkM3 Action Language

As a last step, the algorithm is written in the Ark action language. It is shown in full in Listing C.2.

ArkM3 action language is statically typed. This is different from Python, which is a dynamically typed language. In the ArkM3 algorithm, we need to declare the type of each variable, and the compiler checks at compile-time whether the operations performed on elements are allowed and whether the accessed properties are declared in the type of the element. In the algorithm, we iterate over all elements of a model, but we are unable to declare the ontological type of the elements. If we want to access type-specific properties of the element, a new mechanism has to be introduced. In what follows, we explain how the three challenges identified in the previous section and the challenge with the static typing system were overcome.

1. For the linguistic check, we can reuse the method defined on the *Element* class. In fact, all methods on ArkM3 classes can be executed from within the ArkM3 action language. The following is a valid statement in the ArkM3 action language (the check will obviously return true - it is only used as an example here, to demonstrate the mechanism):

```
arkm3.object.Clabject c
c.is_instance('Clabject')
```

   The ontological check is implemented analogously to the check in Python:

```
arkm3.object.Clabject pattern_content_class = formalisms.transformations.rule.
    PreConditionPatternContents
arkm3.object.Association a
{any} types = set([a.get_type()]) + a.get_type().get_all_super_class()
types.contains(pattern_content_class))
```

First, the type hierarchy of the association instance a is retrieved. Then, we check whether any of the types is equal to the *PreConditionPatternContents* association type.

2. In ArkM3 action language, access to the metaverse is implicit (while in the Python algorithm, we need an explicit reference to the metaverse). The following statement reads the *PatternElement* element from the *formalisms.transformations.rule* package and assigns it to the *pattern_class* variable.

```
arkm3.object.Clabject pattern_class = formalisms.transformations.rule.PatternElement
```

3. When accessing a property of an instance in ArkM3 action language, we are generally interested in its value. The value of a property is accessed as in the following example:

```
class State

  formalisms.transformations.rule.PreConditionPattern p

  {void: arkm3.action.Constraint} get_constraint

  get_constraint = {void: arkm3.action.Constraint}:

    return self.p.p_constraint
```

This code snippet accesses the *p_constraint* property value of the *p* property (which is an instance of the *PreConditionPattern* class) of the *self* variable (which is an instance of the *State* class).

4. The fourth challenge, the static typing system, was overcome by introducing a casting operator. This allows an arbitrary element to be cast to an instance of a type in any metamodel. An example of the casting operator is shown below:

```
arkm3.object.Association edge

int label = ((formalisms.transformations.rule.PatternElement) (edge.get_isFrom())).__pLabel
```

Here, the association instance *edge*'s source is queried by the *get_isFrom* method. This method returns a generic *Clabject* instance. However, in this case, we know that the source element will be a *PatternElement*, because it is an element in a pattern model. We can cast it to the *PatternElement* class and query its *__pLabel* attribute.

Chapter 6

Modelling Language Evolution in Ark

In this chapter, we will use the techniques developed in the previous sections for modelling, metamodelling, and model transformation to co-evolve models and transformations conforming to the original TrainSim language according to the changes made to the metamodel of the language, as explained in Section 3.2. We will make use of model transformation to migrate both models (using regular transformations) and transformation models (using HOTs).

In Section 6.1, we will revisit the example evolution scenario and develop it in ArkM3. In Section 6.2, we will present techniques for model migration using model transformation. Lastly, in Section 6.3, transformations are migrated using HOTs.

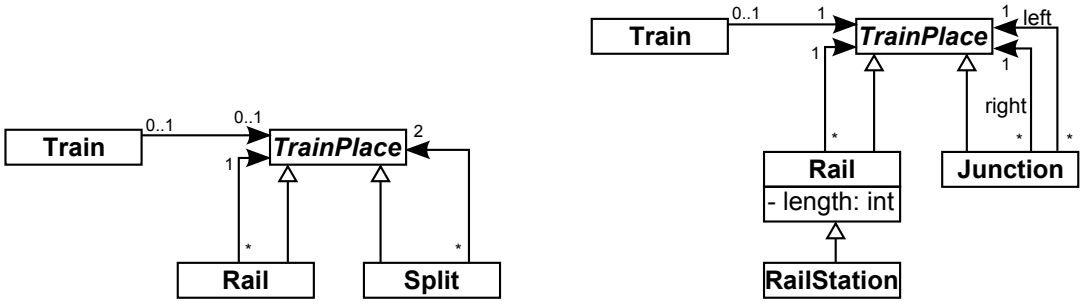


FIGURE 6.1: (a) The original TrainSim metamodel and (b) The evolved TrainSim metamodel.

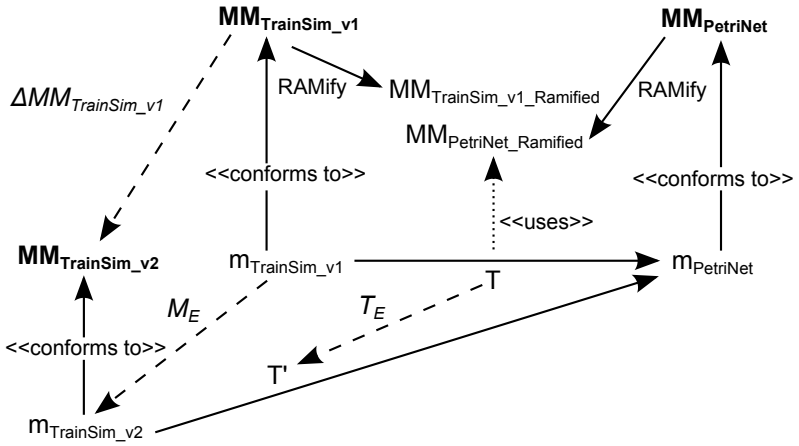


FIGURE 6.2: Architecture of our language evolution approach.

6.1 Example Evolution Scenario

```
1  package formalisms.trainsim
2    package trainsimMM
3      class Train
4
5      abstract class TrainPlace
6      class Rail(TrainPlace)
7        int length
8      class Junction(TrainPlace)
9
10     class RailStation(Rail)
11
12     association T_on_TP
13       t: from Train<0..1>
```

```
14      tp: to TrainPlace<0..1>
15
16    association R_to_TP
17      r: from Rail<0..*>
18      tp: to TrainPlace<1..1>
19
20    association S_to_TP_Left
21      j: from Junction<0..*>
22      left: to TrainPlace<2..2>
23
24    association S_to_TP_Right
25      j: from Junction<0..*>
26      right: to TrainPlace<2..2>
```

LISTING 6.1: The evolved TrainSim metamodel, modelled in Ark.

The example evolution scenario was introduced in Section 3.2. In Figure 6.1, the original and evolved metamodels are repeated for clarification. In Listing 6.1, the evolved metamodel of the TrainSim language is shown, as it is metamodelled in Ark.

In Figure 6.2, the main components of the architecture of our solution for language evolution are shown. Central to the evolution scenario are the two metamodels of our running example: $MM_{TrainSim}$ and $MM_{PetriNet}$. The transformation T maps models in the TrainSim language on models in the PetriNet language, which is its semantic domain. As we explained in the previous section, both metamodels are RAMified to obtain the pattern languages used for constructing rule-based model-to-model transformations. The transformation T uses both the RAMified metamodels of the TrainSim and PetriNet languages.

In the example evolution scenario, introduced in Chapter 3, the TrainSim language was evolved through a series of metamodel adaptations. In the figure, the adaptations are represented by the difference model $\Delta MM_{TrainSim\_v1}$. However, this is only a conceptual model: we do not make use of an explicit difference model in our approach for language evolution. Instead, the user has to choose an appropriate model migration transformation for each metamodel adaptation. This is analogous to the approach introduced by

Hermanssdoerfer et al. in the tool COPE [18]. In COPE, a language developer evolves the metamodel through the application of a series of *coupled operators*, which contain both the metamodel adaptation step and the corresponding model migration. We use a similar approach, but do not use the concept of coupled operators. We leave the adaptation of the metamodel up to the language developer, and present him with a number of model migration transformation possibilities. He can then choose the most appropriate one, or develop one himself. In the figure, the migration transformation which repairs the conformance relation between models created in the initial version of the TrainSim language and the adapted metamodel of the language is called M_E. The migration transformation, which is a HOT, that transforms the semantic mapping transformation T such that it is capable of mapping models created in the evolved TrainSim language to PetriNet models is called T_E.

As represented in Figure 6.2, the two versions of the TrainSim language have two different metamodels: this in fact creates two modelling languages. When migrating models and transformations to the new version of the language, we not only have to account for the differences between the two metamodels; the instances of the metamodel concepts which have not been changed need to be migrated as well. This has been noted before by a number of authors. In [28], Rose et al. compare existing co-evolution approaches based on this source-target relationship. In our approach, we require all unchanged elements to be migrated explicitly. This requires the development of transformations that simply copy elements of a version one model to elements of a version two model. As these transformations are trivial and can be constructed automatically, they are not shown in the rest of this chapter.

In the next sections, we will explore how to construct migration transformations to co-evolve both models and transformations in light of the changes made to the metamodel.

6.2 Model Migration

In this section, we will construct a number of transformations which migrate models conforming to the original metamodel of the TrainSim language, to restore the conformance relation with the metamodel of the new version of the language.

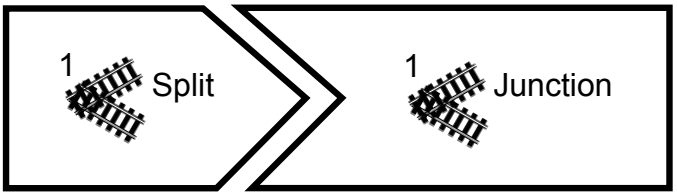


FIGURE 6.3: The migration transformation for the rename operation.

6.2.1 Addition of the RailStation Class

The first change is the addition of the *RailStation* class as a subclass of the *Rail* class. As this is a non-breaking change (the RailStation class is not mandatory, so models not containing an instance of the RailStation class are valid models, conforming to the evolved TrainSim metamodel), no migration transformation has to be constructed.

6.2.2 Renaming of the Split Class

```
1  package formalisms.trainsim
2    package migration.rule1
3      # # # #
4      # LHS #
5      # # # #
6      formalisms.transformations.rule.LHS lhs
7
8      formalisms.trainsim.trainsimMM_v1_Pre.Split s_lhs
9        __pLabel = 1
10
11     formalisms.transformations.rule.PreConditionPatternContents lhs_to_s_lhs
12       pattern = lhs
13       element = s_lhs
14
15     # # # #
16     # RHS #
17     # # # #
18     formalisms.transformations.rule.RHS rhs
19
20     formalisms.trainsim.trainsimMM_v2_Post.Junction j_rhs
```

```
21        __pLabel = 2
22
23      formalisms.transformations.rule.PostConditionPatternContents rhs_to_j_rhs
24        pattern = rhs
25        element = j_rhs
```

LISTING 6.2: The migration transformation for the rename operation, modelled in Ark.

The second change is the renaming of the *Split* class to *Junction*. Figure 6.3 is a visual representation of the rule and Listing 6.2 shows the migration transformation for this change as it is modelled in Ark. The LHS of the rule matches all *Split* instances, and the RHS makes sure they are replaced by a *Junction* instance.

6.2.3 Cardinality Change

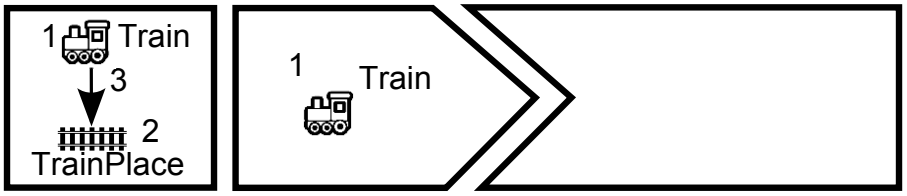


FIGURE 6.4: The first migration transformation for the cardinality change.

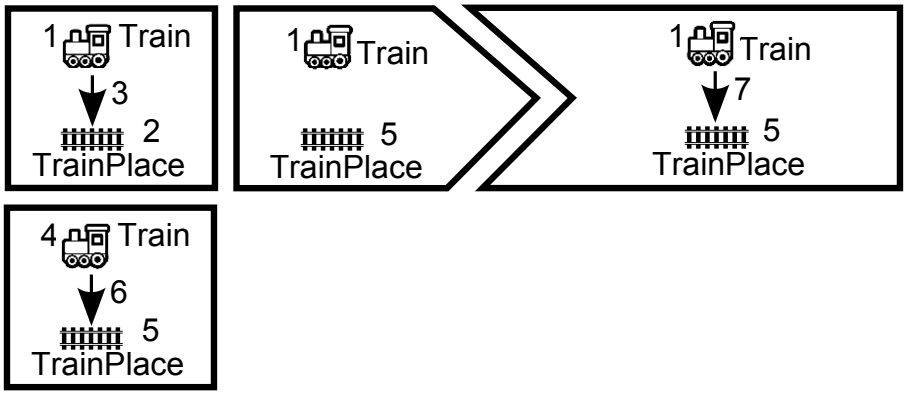


FIGURE 6.5: The second migration transformation for the cardinality change.

```
1  package formalisms.trainsim
2    package migration.rule1
3      # # # #
4      # NAC #
5      # # # #
```

```
6     formalisms.transformations.rule.NAC nac
7
8     formalisms.trainsim.trainsimMM_v1_Pre.Train t_nac
9       __pLabel = 1
10
11    formalisms.trainsim.trainsimMM_v1_Pre.TrainPlace tp_nac
12      __pLabel = 2
13
14    formalisms.trainsim.trainsimMM_v1_Pre.T_on_TP t_nac_on_tp_nac
15      t = t_nac
16      tp = tp_nac
17      __pLabel = 3
18
19    formalisms.transformations.rule.PreConditionPatternContents nac_to_t_nac
20      pattern = nac
21      element = t_nac
22
23    formalisms.transformations.rule.PreConditionPatternContents nac_to_tp_nac
24      pattern = nac
25      element = tp_nac
26
27    formalisms.transformations.rule.PreConditionPatternContents nac_to_t_nac_on_tp_nac
28      pattern = nac
29      element = t_nac_on_tp_nac
30
31    # # # #
32    # LHS #
33    # # # #
34    formalisms.transformations.rule.LHS lhs
35
36    formalisms.trainsim.trainsimMM_v1_Pre.Train t_lhs
37      __pLabel = 1
38
```

```
39      formalisms.transformations.rule.PreConditionPatternContents lhs_to_t_lhs
40        pattern = lhs
41        element = t_lhs
42
43      # # # #
44      # RHS #
45      # # # #
46      formalisms.transformations.rule.RHS rhs
```

LISTING 6.3: The first migration transformation for the cardinality change, modelled in Ark.

```
1   package formalisms.trainsim
2     package migration.rule1
3       # # # #
4       # NAC #
5       # # # #
6       formalisms.transformations.rule.NAC nac1
7             name = 'NAC_1'
8
9       formalisms.trainsim.trainsimMM_v1_Pre.Train t_nac1
10        __pLabel = 1
11
12      formalisms.trainsim.trainsimMM_v1_Pre.TrainPlace tp_nac1
13        __pLabel = 2
14
15      formalisms.trainsim.trainsimMM_v1_Pre.T_on_TP t_nac1_on_tp_nac1
16        t = t_nac1
17        tp = tp_nac1
18        __pLabel = 3
19
20      formalisms.transformations.rule.PreConditionPatternContents nac1_to_t_nac1
21        pattern = nac1
22        element = t_nac1
23
```

```
24  formalisms.transformations.rule.PreConditionPatternContents nac1_to_tp_nac1
25    pattern = nac1
26    element = tp_nac1
27
28  formalisms.transformations.rule.PreConditionPatternContents nac1_to_t_nac1_on_tp_nac1
29    pattern = nac1
30    element = t_nac1_on_tp_nac1
31
32  # # # #
33  # NAC #
34  # # # #
35  formalisms.transformations.rule.NAC nac2
36        name = 'NAC_2'
37
38  formalisms.trainsim.trainsimMM_v1_Pre.Train t_nac2
39    __pLabel = 4
40
41  formalisms.trainsim.trainsimMM_v1_Pre.TrainPlace tp_nac2
42    __pLabel = 5
43
44  formalisms.trainsim.trainsimMM_v1_Pre.T_on_TP t_nac2_on_tp_nac2
45    t = t_nac2
46    tp = tp_nac2
47    __pLabel = 6
48
49  formalisms.transformations.rule.PreConditionPatternContents nac2_to_t_nac2
50    pattern = nac2
51    element = t_nac2
52
53  formalisms.transformations.rule.PreConditionPatternContents nac2_to_tp_nac2
54    pattern = nac2
55    element = tp_nac2
56
```

```
57    formalisms.transformations.rule.PreConditionPatternContents nac2_to_t_nac2_on_tp_nac2
58      pattern = nac2
59      element = t_nac2_on_tp_nac2
60
61    # # # #
62    # LHS #
63    # # # #
64    formalisms.transformations.rule.LHS lhs
65
66    formalisms.trainsim.trainsimMM_v1_Pre.Train t_lhs
67      __pLabel = 1
68
69    formalisms.trainsim.trainsimMM_v1_Pre.TrainPlace tp_lhs
70      __pLabel = 5
71
72    formalisms.transformations.rule.PreConditionPatternContents lhs_to_t_lhs
73      pattern = lhs
74      element = t_lhs
75
76    formalisms.transformations.rule.PreConditionPatternContents lhs_to_tp_lhs
77      pattern = lhs
78      element = tp_lhs
79
80    # # # #
81    # RHS #
82    # # # #
83    formalisms.transformations.rule.RHS rhs
84
85    formalisms.trainsim.trainsimMM_v2_Post.Train t_rhs
86      __pLabel = 1
87
88    formalisms.trainsim.trainsimMM_v2_Post.TrainPlace tp_rhs
89      __pLabel = 5
```

```
90
91    formalisms.trainsim.trainsimMM_v2_Post.T_on_TP t_rhs_on_tp_rhs
92      t = t_rhs
93      tp = tp_rhs
94      __pLabel = 7
95
96    formalisms.transformations.rule.PostConditionPatternContents rhs_to_t_rhs
97      pattern = rhs
98      element = t_rhs
99
100   formalisms.transformations.rule.PostConditionPatternContents rhs_to_tp_rhs
101     pattern = rhs
102     element = tp_rhs
103
104   formalisms.transformations.rule.PostConditionPatternContents rhs_to_t_rhs_on_tp_rhs
105     pattern = rhs
106     element = t_rhs_on_tp_rhs
```

LISTING 6.4: The second migration transformation for the cardinality change, modelled in Ark.

The third change is the changing of the cardinality of the *T_on_TP* relation. In the new version of the metamodel, every *Train* has to be on a *TrainPlace*. This is an unresolvable change, but we can make a few suggestions to the modeller as to how the conformance relation can be restored. Three approaches are discussed here:

1. The first approach is to remove all unconnected trains in non-conforming models. This will ensure that all trains are connected and the constraint imposed by the cardinality is satisfied. However, semantic information may be lost by using this approach, as some of the unconnected trains that are deleted may be vital parts of the models. The approach is shown graphically in Figure 6.4 and modelled as a transformation rule in Listing 6.3. The LHS of the rule matches a train, while the NAC of the rule makes sure the matched train is not connected to a rail. The RHS is empty, which means that the train matched by the LHS will be deleted in the host model.

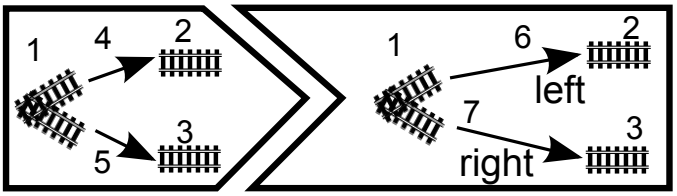


FIGURE 6.6: The migration transformation for the relation split change.

2. The second approach is to connect all unconnected trains to an unoccupied rail. This will also ensure that the constraint imposed by the cardinality is satisfied. However, all unconnected trains will be connected to a random unoccupied rail by the transformation. This may not be semantically correct. The migration transformation rule is shown graphically in Figure 6.5 and modelled in Ark in Listing 6.4. The LHS matches a train and a train place, and the RHS connects these two. The rule contains two NACs: the first ensures that the matched train is not connected to a train place, while the second ensures that no train is connected to the matched train place.

3. The last possibility is to do no migration at all. This leaves the model in a non-conforming state, which requires a modeller to go through the models and adapt them manually. While this is not optimal, it may be the only possibility when we want to make sure that the semantics of the models are not compromised and to ensure continuity of the MDE system.

6.2.4 Splitting of Relation

The fourth change is the splitting of the *S_to_TP* relation into two relations: *S_to_TP_Left* and *S_to_TP_Right*. As with the last change, this is a non-resolvable change. Again, we can leave the models in a non-conforming state and let a modeller adapt the models manually. This will ensure semantic correctness, but it is time-consuming.

```
1 package formalisms.trainsim
2   package migration.rule1
3     # # # #
4     # LHS #
5     # # # #
6     formalisms.transformations.rule.LHS lhs
```

```
7
8     formalisms.trainsim.trainsimMM_v2_Pre.Junction j_lhs
9       __pLabel = 1
10
11    formalisms.trainsim.trainsimMM_v1_Pre.TrainPlace tp_lhs_1
12      __pLabel = 2
13
14    formalisms.trainsim.trainsimMM_v1_Pre.TrainPlace tp_lhs_2
15      __pLabel = 3
16
17    formalisms.trainsim.trainsimMM_v1_Pre.S_to_TP j_lhs_to_tp_lhs_1
18      s = j_lhs
19      tp = tp_lhs_1
20      __pLabel = 4
21
22    formalisms.trainsim.trainsimMM_v1_Pre.S_to_TP j_lhs_to_tp_lhs_2
23      s = j_lhs
24      tp = tp_lhs_2
25      __pLabel = 5
26
27    formalisms.transformations.rule.PreConditionPatternContents lhs_to_j_lhs
28      pattern = lhs
29      element = j_lhs
30
31    formalisms.transformations.rule.PreConditionPatternContents lhs_to_tp_lhs_1
32      pattern = lhs
33      element = tp_lhs_1
34
35    formalisms.transformations.rule.PreConditionPatternContents lhs_to_tp_lhs_2
36      pattern = lhs
37      element = tp_lhs_2
38
39    formalisms.transformations.rule.PreConditionPatternContents lhs_to_j_lhs_to_tp_lhs_1
```

```
40        pattern = lhs
41        element = j_lhs_to_tp_lhs_1
42
43      formalisms.transformations.rule.PreConditionPatternContents lhs_to_j_lhs_to_tp_lhs_2
44        pattern = lhs
45        element = j_lhs_to_tp_lhs_2
46
47      # # # #
48      # RHS #
49      # # # #
50      formalisms.transformations.rule.RHS rhs
51
52      formalisms.trainsim.trainsimMM_v2_Post.Junction j_rhs
53        __pLabel = 1
54
55      formalisms.trainsim.trainsimMM_v1_Post.TrainPlace tp_rhs_1
56        __pLabel = 2
57
58      formalisms.trainsim.trainsimMM_v1_Post.TrainPlace tp_rhs_2
59        __pLabel = 3
60
61      formalisms.trainsim.trainsimMM_v2_Post.S_to_TP_Left j_rhs_to_tp_rhs_1
62        j = j_rhs
63        tp = tp_rhs_1
64        __pLabel = 6
65
66      formalisms.trainsim.trainsimMM_v2_Post.S_to_TP_Right j_rhs_to_tp_rhs_2
67        j = j_rhs
68        tp = tp_rhs_2
69        __pLabel = 7
70
71      formalisms.transformations.rule.PostConditionPatternContents rhs_to_j_rhs
72        pattern = rhs
```

```
73        element = j_rhs
74
75      formalisms.transformations.rule.PostConditionPatternContents rhs_to_tp_rhs_1
76        pattern = rhs
77        element = tp_rhs_1
78
79      formalisms.transformations.rule.PostConditionPatternContents rhs_to_tp_rhs_2
80        pattern = rhs
81        element = tp_rhs_2
82
83      formalisms.transformations.rule.PostConditionPatternContents rhs_to_j_rhs_to_tp_rhs_1
84        pattern = rhs
85        element = j_rhs_to_tp_rhs_1
86
87      formalisms.transformations.rule.PostConditionPatternContents rhs_to_j_rhs_to_tp_rhs_2
88        pattern = rhs
89        element = j_rhs_to_tp_rhs_2
```

LISTING 6.5: The migration transformation for the relation split change, modelled in Ark.

An automatic way of migrating the models consists of constructing a transformation which looks at the two outgoing relations of each junction and randomly chooses one as the left one and one as the right one. This will ensure that all models are syntactically correct and conforming with the new version of the metamodel. The migration transformation is shown in Figure 6.6 and modelled in Ark in Listing 6.5. The LHS of the rule consists of five elements: a junction, two train places and two associations, connecting the junction to the train places. The RHS also consists of five elements, but replaces the two associations by a left and a right association.

6.2.5 Addition of Attribute

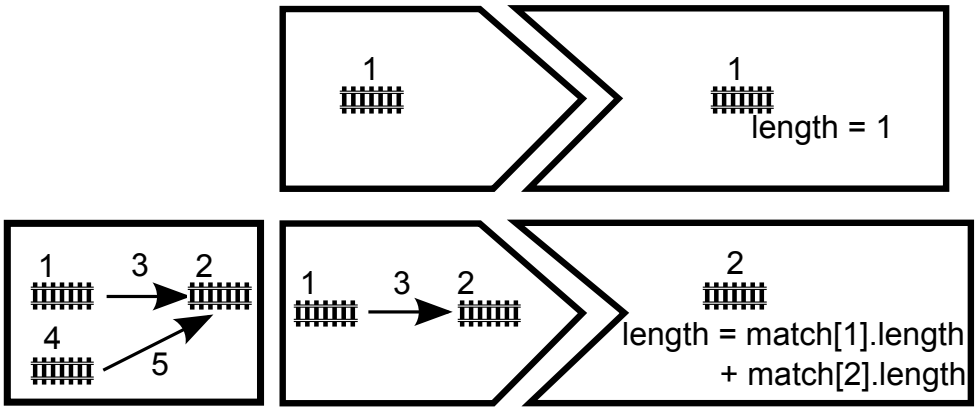


FIGURE 6.7: The migration transformation for the addition of the *length* attribute.

```
1  package formalisms.trainsim
2    package migration.rule1
3      # # # #
4      # LHS #
5      # # # #
6      formalisms.transformations.rule.LHS lhs
7
8      formalisms.trainsim.trainsimMM_v1_Pre.Rail r_lhs
9        __pLabel = 1
10
11     formalisms.transformations.rule.PreConditionPatternContents lhs_to_r_lhs
12       pattern = lhs
13       element = r_lhs
14
15     # # # #
16     # RHS #
17     # # # #
18     formalisms.transformations.rule.RHS rhs
19
20     formalisms.trainsim.trainsimMM_v2_Post.Rail r_rhs
21       __pLabel = 2
22       length = {{int: Clabject} match: void}:
23         match[2].length = 1
```

```
24
25      formalisms.transformations.rule.PostConditionPatternContents rhs_to_r_rhs
26        pattern = rhs
27        element = r_rhs
28
29    package migration.rule2
30      # # # #
31      # NAC #
32      # # # #
33      formalisms.transformations.rule.NAC nac
34        name = 'NAC_1'
35
36      formalisms.trainsim.trainsimMM_v2_Pre.Rail r_nac_1
37        __pLabel = 1
38        length = {{int: Clabject} match: bool}:
39          return True
40
41      formalisms.trainsim.trainsimMM_v2_Pre.Rail r_nac_2
42        __pLabel = 2
43        length = {{int: Clabject} match: bool}:
44          return True
45
46      formalisms.trainsim.trainsimMM_v2_Pre.Rail r_nac_3
47        __pLabel = 4
48        length = {{int: Clabject} match: bool}:
49          return True
50
51      formalisms.trainsim.trainsimMM_v2_Pre.R_to_TP r_nac_1_to_r_nac_2
52        r = r_nac_1
53        tp = r_nac_2
54        __pLabel = 3
55
56      formalisms.trainsim.trainsimMM_v2_Pre.R_to_TP r_nac_3_to_r_nac_2
```

```
57        r = r_nac_3
58        tp = r_nac_2
59          __pLabel = 5
60
61      # # # #
62      # LHS #
63      # # # #
64      formalisms.transformations.rule.LHS lhs
65
66      formalisms.trainsim.trainsimMM_v2_Pre.Rail r_lhs_1
67        __pLabel = 1
68        length = {{int: Clabject} match: bool}:
69          return True
70
71      formalisms.trainsim.trainsimMM_v2_Pre.Rail r_lhs_2
72        __pLabel = 2
73        length = {{int: Clabject} match: bool}:
74          return True
75
76      formalisms.trainsim.trainsimMM_v2_Pre.R_to_TP r_lhs_1_to_r_lhs_2
77        r = r_lhs_1
78        tp = r_lhs_2
79        __pLabel = 3
80
81      formalisms.transformations.rule.PreConditionPatternContents lhs_to_r_lhs_1
82        pattern = lhs
83        element = r_lhs_1
84
85      formalisms.transformations.rule.PreConditionPatternContents lhs_to_r_lhs_2
86        pattern = lhs
87        element = r_lhs_2
88
89      formalisms.transformations.rule.PreConditionPatternContents lhs_to_r_lhs_1_to_r_lhs_2
```

```
90        pattern = lhs
91        element = r_lhs_1_to_r_lhs_2
92
93      # # # #
94      # RHS #
95      # # # #
96      formalisms.transformations.rule.RHS rhs
97
98      formalisms.trainsim.trainsimMM_v2_Post.Rail r_rhs
99        __pLabel = 2
100       length = {{int: Clabject} match: void}:
101         match[2].length = match[1].length + match[2].length
102
103     formalisms.transformations.rule.PostConditionPatternContents rhs_to_r_rhs
104       pattern = rhs
105       element = r_rhs
```

LISTING 6.6: The migration transformation for the addition of the *length* attribute, modelled in Ark.

The fifth and last change is the addition of the attribute *length* to the *Rail* class. This change is also non-resolvable, as a migration transformation cannot be constructed automatically. However, it is possible to construct a migration transformation manually which replaces all sequences of n tracks by one single track with length n. The transformation is shown in Figure 6.7 and is modelled in Ark in Listing 6.6. It consists of two rules. The first one transforms all version one rails to a version two rail of length 1. The second rule transforms two connected version two rails of arbitrary length to one version two rail whose length is the sum of the lengths of the two connected rails. A NAC is added to this rule that prohibits two rails to be connected to the rail with label 2. We only want to transform sequences of rails with one connection between them.

6.3 Transformation Migration

In this section, transformations that have the TrainSim language as their domain language are migrated to the new version of the language. This requires the precondition patterns of the rules of the transformations to be adapted according to the changes made to the metamodel. We use HOTs to migrate the transformations.

6.3.1 Addition of the RailStation Class

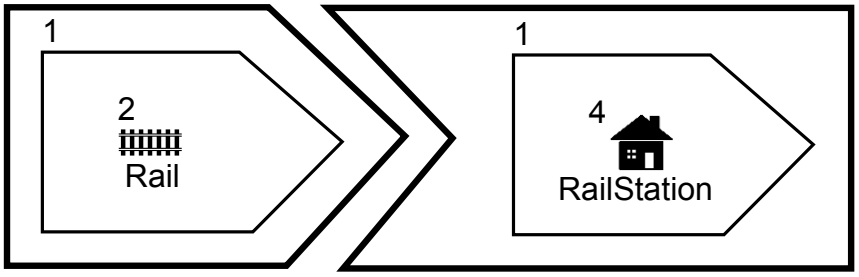


FIGURE 6.8: The first HOT for the addition of the RailStation class.

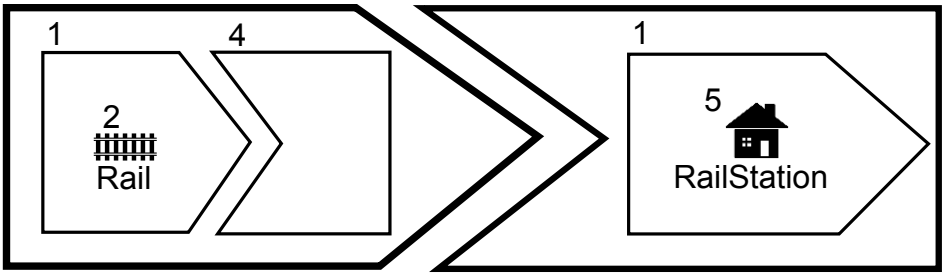


FIGURE 6.9: The second HOT for the addition of the RailStation class.

```
1  package formalisms.trainsim
2    package migration.rule1
3      # # # #
4      # LHS #
5      # # # #
6      formalisms.transformations.rule.LHS lhs
7
8      formalisms.transformations.rule_Pre.PreConditionPattern pcp_lhs
9        __pLabel = 1
10       p_constraint = {{int: Clabject} match: bool}:
11         return True
```

```
12
13      formalisms.trainsim.trainsimMM_v1_Pre_Pre.Rail r_pcp_lhs
14        __pLabel = 2
15        __p__pLabel = {{int: Clabject} match: bool}:
16          return True
17
18      formalisms.transformations.rule_Pre.PreConditionPatternContents pcp_lhs_to_r_pcp_lhs
19        pattern = pcp_lhs
20        element = r_pcp_lhs
21        __pLabel = 3
22
23      formalisms.transformations.rule.PreConditionPatternContents lhs_to_pcp_lhs
24        pattern = lhs
25        element = pcp_lhs
26
27      formalisms.transformations.rule.PreConditionPatternContents lhs_to_r_pcp_lhs
28        pattern = lhs
29        element = r_pcp_lhs
30
31      formalisms.transformations.rule.PreConditionPatternContents lhs_to_pcp_lhs_to_r_pcp_lhs
32        pattern = lhs
33        element = pcp_lhs_to_r_pcp_lhs
34
35      # # # #
36      # RHS #
37      # # # #
38      formalisms.transformations.rule.RHS rhs
39
40      formalisms.transformations.rule_Post.PreConditionPattern pcp_rhs
41        __pLabel = 1
42        p_constraint = {{int: Clabject} match: void}:
43          return
44
```

```
45      formalisms.trainsim.trainsimMM_v2_Pre_Post.RailStation rs_pcp_rhs
46        __pLabel = 4
47        __p__pLabel = {{int: Clabject} match: void}:
48          return
49
50      formalisms.transformations.rule_Post.PreConditionPatternContents pcp_rhs_to_rs_pcp_rhs
51        pattern = pcp_rhs
52        element = rs_pcp_rhs
53        __pLabel = 5
54
55      formalisms.transformations.rule.PostConditionPatternContents rhs_to_pcp_rhs
56        pattern = rhs
57        element = pcp_rhs
58
59      formalisms.transformations.rule.PostConditionPatternContents rhs_to_rs_pcp_rhs
60        pattern = rhs
61        element = rs_pcp_rhs
62
63      formalisms.transformations.rule.PostConditionPatternContents rhs_to_pcp_rhs_to_rs_pcp_rhs
64        pattern = rhs
65        element = pcp_rhs_to_rs_pcp_rhs
```

LISTING 6.7: The first HOT for the addition of the RailStation class, modelled in Ark.

```
1   package formalisms.trainsim
2     package migration.rule1
3       # # # #
4       # LHS #
5       # # # #
6       formalisms.transformations.rule.LHS lhs
7
8       formalisms.transformations.rule_Pre.PreConditionPattern pcp_lhs
9         __pLabel = 1
10        p_constraint = {{int: Clabject} match: bool}:
11          return True
```

```
12
13      formalisms.trainsim.trainsimMM_v1_Pre_Pre.Rail r_pcp_lhs
14        __pLabel = 2
15        __p__pLabel = {{int: Clabject} match: bool}:
16          return True
17
18      formalisms.transformations.rule_Pre.PreConditionPatternContents pcp_lhs_to_r_pcp_lhs
19        pattern = pcp_lhs
20        element = r_pcp_lhs
21        __pLabel = 3
22
23      formalisms.transformations.rule_Pre.PostConditionPattern pocp_lhs
24        __pLabel = 4
25        p_action = {{int: Clabject} match: bool}:
26          return True
27
28      formalisms.transformations.rule.PreConditionPatternContents lhs_to_pcp_lhs
29        pattern = lhs
30        element = pcp_lhs
31
32      formalisms.transformations.rule.PreConditionPatternContents lhs_to_r_lhs
33        pattern = lhs
34        element = r_pcp_lhs
35
36      formalisms.transformations.rule.PreConditionPatternContents lhs_to_pcp_lhs_to_r_pcp_lhs
37        pattern = lhs
38        element = pcp_lhs_to_r_pcp_lhs
39
40      formalisms.transformations.rule.PreConditionPatternContents lhs_to_pocp_lhs
41        pattern = lhs
42        element = pocp_lhs
43
44      # # # #
```

```
45    # RHS #
46    # # # #
47    formalisms.transformations.rule.RHS rhs
48
49    formalisms.transformations.rule_Post.PreConditionPattern pcp_rhs
50      __pLabel = 1
51      p_constraint = {{int: Clabject} match: void}:
52        return
53
54    formalisms.trainsim.trainsimMM_v2_Pre_Post.RailStation r_pcp_rhs
55      __pLabel = 5
56      __p__pLabel = {{int: Clabject} match: void}:
57        return
58
59    formalisms.transformations.rule_Post.PreConditionPatternContents pcp_rhs_to_r_pcp_rhs
60      pattern = pcp_rhs
61      element = r_pcp_rhs
62      __pLabel = 6
63
64    formalisms.transformations.rule.PostConditionPatternContents rhs_to_pcp_rhs
65      pattern = rhs
66      element = pcp_rhs
67
68    formalisms.transformations.rule.PostConditionPatternContents rhs_to_r_pcp_rhs
69      pattern = rhs
70      element = r_pcp_rhs
71
72    formalisms.transformations.rule.PostConditionPatternContents rhs_to_pcp_rhs_to_r_rhs
73      pattern = rhs
74      element = pcp_rhs_to_r_pcp_rhs
```

Listing 6.8: The second HOT for the addition of the RailStation class.

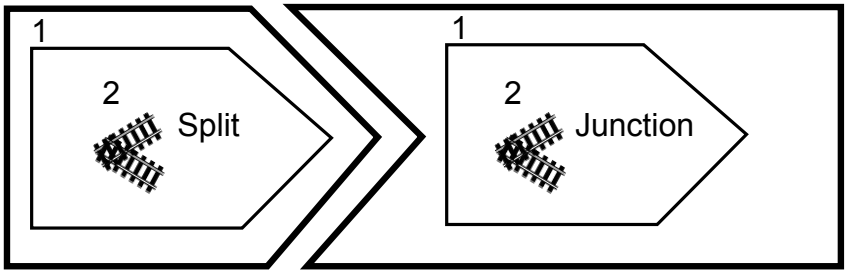


FIGURE 6.10: The HOT for the renaming of the Split class to Junction.

While the addition of the *RailStation* class is a non-breaking change for models, this is not the case for transformations. If we want the transformation to cover the whole domain language, rules have to be added which transform the new element as well. This can be done manually: a modeller creates a new rule, or several rules, that transform the new element into structures of the image language and fits them into the rule scheduler. We also provide two automated HOTs:

1. The first HOT, shown in Figure 6.8 and modelled in Ark in Listing 6.7, creates a new transformation for the *RailStation* class based on the transformation definition of its superclass *Rail*. As an example, recall the TrainSim-to-PetriNet transformation from Figure 3.2(b). It could make sense to add a rule that transforms a RailStation to the same PetriNet structure as a Rail. In our HOT, we match a precondition pattern in the LHS which contains an instance of the *Rail* class. The precondition pattern is then transformed by the RHS, and the instance of the Rail class is replaced by an instance of the RailStation class.

2. The second HOT (shown in Figure 6.9 and modelled in Ark in Listing 6.8) is similar, but it creates a transformation without an RHS. This will make the new transformation non-conforming with the *Transformation* metamodel, which will prompt the modeller to 'fix' the error. He can then fill in the RHS of the rule, making sure the semantic mapping of the RailStation class is done correctly.

6.3.2 Renaming of the Split Class

```
1 package formalisms.trainsim
2   package migration.rule1
3     # # # #
```

```
4   # LHS #
5   # # # #
6   formalisms.transformations.rule.LHS lhs
7
8   formalisms.transformations.rule_Pre.PreConditionPattern pcp_lhs
9     p_constraint = {{int: Clabject} match: bool}:
10      return True
11    __pLabel = 1
12
13  formalisms.trainsim.trainsimMM_v1_Pre_Pre.Split s_pcp_lhs
14    __pLabel = 2
15    __p__pLabel = {{int: Clabject} match: bool}:
16      return True
17
18  formalisms.transformations.rule_Pre.PreConditionPatternContents pcp_lhs_to_s_pcp_lhs
19    pattern = pcp_lhs
20    element = s_pcp_lhs
21    __pLabel = 3
22
23  formalisms.transformations.rule.PreConditionPatternContents lhs_to_pcp
24    pattern = lhs
25    element = pcp_lhs
26
27  formalisms.transformations.rule.PreConditionPatternContents lhs_to_s_pcp_lhs
28    pattern = lhs
29    element = s_pcp_lhs
30
31  formalisms.transformations.rule.PreConditionPatternContents lhs_to_pcp_to_s_pcp_lhs
32    pattern = lhs
33    element = pcp_lhs_to_s_pcp_lhs
34
35  # # # #
36  # RHS #
```

```
37    # # # #
38    formalisms.transformations.rule.RHS rhs
39
40    formalisms.transformations.rule_Post.PreConditionPattern pcp_rhs
41      p_constraint = {{int: Clabject} match: void}:
42        return
43      __pLabel = 1
44
45    formalisms.trainsim.trainsimMM_v2_Pre_Post.Junction j_pcp_rhs
46      __pLabel = 4
47      __p__pLabel = {{int: Clabject} match: void}:
48        return
49
50    formalisms.transformations.rule_Post.PreConditionPatternContents pcp_to_j_pcp_rhs
51      pattern = pcp
52      element = j_pcp_rhs
53      __pLabel = 5
54
55    formalisms.transformations.rule.PostConditionPatternContents rhs_to_pcp
56      pattern = rhs
57      element = pcp_rhs
58
59    formalisms.transformations.rule.PostConditionPatternContents rhs_to_j_pcp_rhs
60      pattern = rhs
61      element = j_pcp_rhs
62
63    formalisms.transformations.rule.PostConditionPatternContents rhs_to_pcp_to_j_pcp_rhs
64      pattern = rhs
65      element = pcp_to_j_pcp_rhs
```

LISTING 6.9: The HOT for the renaming of the Split class to Junction, modelled in Ark.

Migrating the transformation in response to the rename change is done similarly to models: only now, we use a HOT instead of a regular transformation. The HOT is

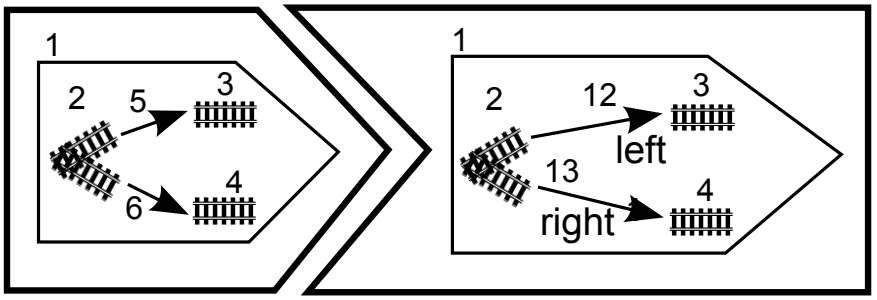


FIGURE 6.11: The HOT for the splitting of the S_to_TP association.

shown in Figure 6.10 and modelled in Ark in Listing 6.9. In the LHS, a precondition pattern is matched, containing an instance of the RAMified *Split* class. In the RHS, this instance is replaced by an instance of the RAMified *Junction* class.

6.3.3 Cardinality Change

While the change in the cardinality of the *T_on_TP* association required a model migration, transformations are unaffected. In the RAMified version of the metamodel, the minimum bound of all cardinalities is removed as part of the relaxation step. In other words, it is allowed to model an unconnected train in the patterns of a rule, even in the new version of the language. All changes performed by the RAMification process in the relaxation step will be non-breaking changes when performed as part of the evolution process. This includes cardinality changes and making a concrete metamodel class abstract: in patterns, we are allowed to create instances of abstract classes (as they are made concrete by the RAMification process).

6.3.4 Splitting of Relation

```
1 package formalisms.trainsim
2   package migration.rule1
3     # # # #
4     # LHS #
5     # # # #
6     formalisms.transformations.rule.LHS lhs
7
8     formalisms.transformations.rule_Pre.PreConditionPattern pcp_lhs
```

```
9       p_constraint = {{int: Clabject} match: bool}:
10        return True
11      __pLabel = 1
12
13    formalisms.trainsim.trainsimMM_v2_Pre_Pre.Junction j_pcp_lhs
14      __pLabel = 2
15      __p__pLabel = {{int: arkm3.object.Clabject} match: bool}:
16        return True
17
18    formalisms.trainsim.trainsimMM_v1_Pre_Pre.TrainPlace tp_pcp_lhs_1
19      __pLabel = 3
20      __p__pLabel = {{int: arkm3.object.Clabject} match: bool}:
21        return True
22
23    formalisms.trainsim.trainsimMM_v1_Pre_Pre.TrainPlace tp_pcp_lhs_2
24      __pLabel = 4
25      __p__pLabel = {{int: arkm3.object.Clabject} match: bool}:
26        return True
27
28    formalisms.trainsim.trainsimMM_v1_Pre_Pre.S_to_TP j_pcp_lhs_to_tp_pcp_lhs_1
29      s = j_pcp_lhs
30      tp = tp_pcp_lhs_1
31      __pLabel = 5
32      __p__pLabel = {{int: arkm3.object.Clabject} match: bool}:
33        return True
34
35    formalisms.trainsim.trainsimMM_v1_Pre_Pre.S_to_TP j_pcp_lhs_to_tp_pcp_lhs_2
36      s = j_pcp_lhs
37      tp = tp_pcp_lhs_2
38      __pLabel = 6
39      __p__pLabel = {{int: arkm3.object.Clabject} match: bool}:
40        return True
41
```

```
42  formalisms.transformations.rule_Pre.PreConditionPatternContents pcp_to_j_pcp_lhs
43    pattern = pcp_lhs
44    element = j_pcp_lhs
45    __pLabel = 7
46
47  formalisms.transformations.rule_Pre.PreConditionPatternContents pcp_to_tp_pcp_lhs_1
48    pattern = pcp_lhs
49    element = tp_pcp_lhs_1
50    __pLabel = 8
51
52  formalisms.transformations.rule_Pre.PreConditionPatternContents pcp_to_tp_pcp_lhs_2
53    pattern = pcp_lhs
54    element = tp_pcp_lhs_2
55    __pLabel = 9
56
57  formalisms.transformations.rule_Pre.PreConditionPatternContents pcp_to_j_pcp_lhs_to_tp_pcp_lhs_1
58    pattern = pcp_lhs
59    element = j_pcp_lhs_to_tp_pcp_lhs_1
60    __pLabel = 10
61
62  formalisms.transformations.rule_Pre.PreConditionPatternContents pcp_to_j_pcp_lhs_to_tp_pcp_lhs_2
63    pattern = pcp_lhs
64    element = j_pcp_lhs_to_tp_pcp_lhs_2
65    __pLabel = 11
66
67  formalisms.transformations.rule.PreConditionPatternContents lhs_to_pcp_lhs
68    pattern = lhs
69    element = pcp_lhs
70
71  formalisms.transformations.rule.PreConditionPatternContents lhs_to_j_pcp_lhs
72    pattern = lhs
73    element = j_pcp_lhs
74
```

```
75  formalisms.transformations.rule.PreConditionPatternContents lhs_to_tp_pcp_lhs_1
76    pattern = lhs
77    element = tp_pcp_lhs_1
78
79  formalisms.transformations.rule.PreConditionPatternContents lhs_to_tp_pcp_lhs_2
80    pattern = lhs
81    element = tp_pcp_lhs_2
82
83  formalisms.transformations.rule.PreConditionPatternContents lhs_to_j_pcp_lhs_to_tp_pcp_lhs_1
84    pattern = lhs
85    element = j_pcp_lhs_to_tp_pcp_lhs_1
86
87  formalisms.transformations.rule.PreConditionPatternContents lhs_to_j_pcp_lhs_to_tp_pcp_lhs_2
88    pattern = lhs
89    element = j_pcp_lhs_to_tp_pcp_lhs_2
90
91  formalisms.transformations.rule.PreConditionPatternContents lhs_to_pcp_to_j_pcp_lhs
92    pattern = lhs
93    element = pcp_to_j_pcp_lhs
94
95  formalisms.transformations.rule.PreConditionPatternContents lhs_to_pcp_to_tp_pcp_lhs_1
96    pattern = lhs
97    element = pcp_to_tp_pcp_lhs_1
98
99  formalisms.transformations.rule.PreConditionPatternContents lhs_to_pcp_to_tp_pcp_lhs_2
100   pattern = lhs
101   element = pcp_to_tp_pcp_lhs_2
102
103 formalisms.transformations.rule.PreConditionPatternContents
      lhs_to_pcp_to_j_pcp_lhs_to_tp_pcp_lhs_1
104   pattern = lhs
105   element = pcp_to_j_pcp_lhs_to_tp_pcp_lhs_1
106
```

```
107     formalisms.transformations.rule.PreConditionPatternContents
         lhs_to_pcp_to_j_pcp_lhs_to_tp_pcp_lhs_2
108       pattern = lhs
109       element = pcp_to_j_pcp_lhs_to_tp_pcp_lhs_2
110
111     # # # #
112     # RHS #
113     # # # #
114
115     formalisms.transformations.rule.RHS rhs
116
117     formalisms.transformations.rule_Post.PreConditionPattern pcp_rhs
118       p_constraint = {{int: Clabject} match: void}:
119         return
120       __pLabel = 1
121
122     formalisms.trainsim.trainsimMM_v2_Pre_Post.Junction j_pcp_rhs
123       __pLabel = 2
124       __p__pLabel = {{int: arkm3.object.Clabject} match: void}:
125         return
126
127     formalisms.trainsim.trainsimMM_v1_Pre_Post.TrainPlace tp_pcp_rhs_1
128       __pLabel = 3
129       __p__pLabel = {{int: arkm3.object.Clabject} match: void}:
130         return
131
132     formalisms.trainsim.trainsimMM_v1_Pre_Post.TrainPlace tp_pcp_rhs_2
133       __pLabel = 4
134       __p__pLabel = {{int: arkm3.object.Clabject} match: void}:
135         return
136
137     formalisms.trainsim.trainsimMM_v2_Pre_Post.S_to_TP_Left j_pcp_rhs_to_tp_left_rhs
138       j = j_pcp_rhs
```

```
139       tp = tp_pcp_rhs_1
140       __pLabel = 12
141       __p__pLabel = {{int: arkm3.object.Clabject} match: void}:
142         return
143
144     formalisms.trainsim.trainsimMM_v2_Pre_Post.S_to_TP_Right j_pcp_rhs_to_tp_right_rhs
145       j = j_pcp_rhs
146       tp = tp_pcp_rhs_2
147       __pLabel = 13
148       __p__pLabel = {{int: arkm3.object.Clabject} match: void}:
149         return
150
151     formalisms.transformations.rule_Post.PreConditionPatternContents pcp_rhs_to_j_pcp_rhs
152       pattern = pcp_rhs
153       element = j_pcp_rhs
154       __pLabel = 7
155
156     formalisms.transformations.rule_Post.PreConditionPatternContents pcp_rhs_to_tp_pcp_rhs_1
157       pattern = pcp_rhs
158       element = tp_pcp_rhs_1
159       __pLabel = 8
160
161     formalisms.transformations.rule_Post.PreConditionPatternContents pcp_rhs_to_tp_pcp_rhs_2
162       pattern = pcp_rhs
163       element = tp_pcp_rhs_2
164       __pLabel = 9
165
166     formalisms.transformations.rule_Post.PreConditionPatternContents pcp_to_j_pcp_rhs_to_tp_right_rhs
167       pattern = pcp_rhs
168       element = j_pcp_rhs_to_tp_right_rhs
169       __pLabel = 14
170
171     formalisms.transformations.rule_Post.PreConditionPatternContents pcp_to_j_pcp_rhs_to_tp_left_rhs
```

```
172       pattern = pcp_rhs
173       element = j_pcp_rhs_to_tp_left_rhs
174       __pLabel = 15
175
176     formalisms.transformations.rule.PostConditionPatternContents rhs_to_pcp_rhs
177       pattern = rhs
178       element = pcp_rhs
179
180     formalisms.transformations.rule.PostConditionPatternContents rhs_to_j_pcp_rhs
181       pattern = rhs
182       element = j_pcp_rhs
183
184     formalisms.transformations.rule.PostConditionPatternContents rhs_to_tp_pcp_rhs_1
185       pattern = rhs
186       element = tp_pcp_rhs_1
187
188     formalisms.transformations.rule.PostConditionPatternContents rhs_to_tp_pcp_rhs_2
189       pattern = rhs
190       element = tp_pcp_rhs_2
191
192     formalisms.transformations.rule.PostConditionPatternContents rhs_to_j_pcp_rhs_to_tp_left_rhs
193       pattern = rhs
194       element = j_pcp_rhs_to_tp_left_rhs
195
196     formalisms.transformations.rule.PostConditionPatternContents rhs_to_j_pcp_rhs_to_tp_right_rhs
197       pattern = rhs
198       element = j_pcp_rhs_to_tp_right_rhs
199
200     formalisms.transformations.rule.PostConditionPatternContents rhs_to_pcp_rhs_to_j_pcp_rhs
201       pattern = rhs
202       element = pcp_rhs_to_j_pcp_rhs
203
204     formalisms.transformations.rule.PostConditionPatternContents rhs_to_pcp_rhs_to_tp_pcp_rhs_1
```

```
205       pattern = rhs
206       element = pcp_rhs_to_tp_pcp_rhs_1
207
208     formalisms.transformations.rule.PostConditionPatternContents rhs_to_pcp_rhs_to_tp_pcp_rhs_2
209       pattern = lhs
210       element = pcp_rhs_to_tp_pcp_rhs_2
211
212     formalisms.transformations.rule.PostConditionPatternContents
        rhs_to_pcp_to_j_pcp_rhs_to_tp_right_rhs
213       pattern = rhs
214       element = pcp_to_j_pcp_rhs_to_tp_right_rhs
215
216     formalisms.transformations.rule.PostConditionPatternContents
        rhs_to_pcp_to_j_pcp_rhs_to_tp_left_rhs
217       pattern = rhs
218       element = pcp_to_j_pcp_rhs_to_tp_left_rhs
```

LISTING 6.10: The HOT for the splitting of the S_to_TP association, modelled in Ark.

This change is handled similarly for transformations as for models. Again, we can choose not to migrate transformations automatically, leaving the modeller to migrate transformations manually. The second option is to choose the left and right outgoing associations randomly for each instance of the *Junction* class. The HOT which performs this adaptation is shown in Figure 6.11 and modelled in Ark in Listing 6.10. In the LHS, a precondition pattern is matched which contains an instance of the RAMified *Junction* class, connected to two instances of the RAMifeid *TrainPlace* class. In the RHS, the associations between the junction and train places are replaced by a right and a left association. Which of the two associations is replaced by the left association and which is replaced by the right association is determined by the matcher.

6.3.5 Addition of Attribute

For the addition of the *length* attribute to the *Rail* class, no HOT can be constructed which automatically migrates transformations. The metamodel adaptation requires a

modeller to migrate each transformation manually. The only thing that can be done is migrating the version 1 pattern elements to version 2 pattern elements. As discussed in the introduction of this chapter, we will not show these trivial transformations. As we cannot know in advance what the semantic mapping of a rail with length n will be, we leave it up to the language developer to develop a suitable transformation.

Chapter 7

Conclusion

This thesis introduces a solution for modelling language evolution in the existing modelling kernel Ark. We introduce a platform that uses explicit modelling of transformations to allow for model and transformation migration in response to metamodel adaptation. Our contributions include:

1. Enabling the explicit modelling of rule-based model-to-model transformations in Ark. Our approach includes the process of RAMification to construct transformation languages specific to the input and output languages of the transformation and allows for HOTs by default.

2. Creating a graph matching algorithm in Ark which is capable of interpreting model transformation definitions and rewriting a host graph according to this definition.

3. Enabling modelling language evolution by using model transformation, including the migration of models and transformations. This approach proves that Ark is capable of supporting modelling language evolution.

Future work includes:

1. Creating a more efficient graph matching and rewriting algorithm based on T-Core, that works on the underlying Himesis structures of the ArkM3 elements.

2. Explicit modelling of metamodel adaptations. This can either be in the form of an explicit difference model which captures all adaptations performed on the

metamodel or an operator-based approach, where each metamodel adaptation is performed by a model transformation.

3. Automatic generation of migration transformations, where possible. This could be accompanied by a migration transformation library, containing frequently used migration transformations. This would allow a modeller to semi-automatically evolve a modelling language together with existing artefacts.

4. Modelling the RAMification process explicitly as a model transformation, which transforms ArkM3 metamodels.

Appendix A

TrainSim-to-PetriNet Transformation in Ark

```
1  package formalisms.trainsim
2    package trainsim_to_petrinet
3      formalisms.transformations.transformation.Exhaust exhaust
4        isStart = True
5        locations = {'formalisms.trainsim.trainsim_to_petrinet.rail_to_petrinet',
6                     'formalisms.trainsim.trainsim_to_petrinet.rail_train_to_petrinet',
7                     'formalisms.trainsim.trainsim_to_petrinet.split_to_petrinet'
8                     'formalisms.trainsim.trainsim_to_petrinet.split_train_to_petrinet'}
9
10   package rail_to_petrinet
11     # # # #
12     # NAC #
13     # # # #
14     formalisms.transformations.rule.NAC nac
15       name = 'NAC_1'
16
17     formalisms.trainsim.trainsimMM_Pre.Rail r_nac
18       __pLabel = 1
19
20     formalisms.trainsim.trainsimMM_Pre.Train t_nac
```

```
21        __pLabel = 2
22
23      formalisms.trainsim.trainsimMM_Pre.T_on_TP t_on_tp_nac
24        t = t_nac
25        tp = r_nac
26        __pLabel = 3
27
28      formalisms.transformations.rule.PreConditionPatternContents nac_to_r_nac
29        pattern = nac
30        element = r_nac
31
32      formalisms.transformations.rule.PreConditionPatternContents nac_to_t_nac
33        pattern = nac
34        element = t_nac
35
36      formalisms.transformations.rule.PreConditionPatternContents nac_to_t_on_tp_nac
37        pattern = nac
38        element = t_on_tp_nac
39
40      # # # #
41      # LHS #
42      # # # #
43      formalisms.transformations.rule.LHS lhs
44
45      formalisms.trainsim.trainsimMM_Pre.Rail r_lhs
46        __pLabel = 1
47
48      formalisms.transformations.rule.PreConditionPatternContents lhs_to_r_lhs
49        pattern = lhs
50        element = r_lhs
51
52      # # # #
53      # RHS #
```

```
54    # # # #
55    formalisms.transformations.rule.RHS rhs
56
57    formalisms.petrinet.petrinetMM_Post.Place free
58      __pLabel = 4
59      tokens = {{int: arkm3.object.Clabject} match: void}:
60        match[4].tokens = 1
61
62    formalisms.petrinet.petrinetMM_Post.Place rail
63      __pLabel = 5
64      tokens = {{int: arkm3.object.Clabject} match: void}:
65        match[5].tokens = 0
66
67    formalisms.petrinet.petrinetMM_Post.Transition in_rhs
68      __pLabel = 6
69
70    formalisms.petrinet.petrinetMM_Post.Transition out_rhs
71      __pLabel = 7
72
73    formalisms.petrinet.petrinetMM_Post.P_to_T free_to_in_rhs
74      p = free
75      t = in_rhs
76      __pLabel = 8
77
78    formalisms.petrinet.petrinetMM_Post.T_to_P in_rhs_to_rail
79      t = in_rhs
80      p = rail
81      __pLabel = 9
82
83    formalisms.petrinet.petrinetMM_Post.P_to_T rail_to_out_rhs
84      p = rail
85      t = out_rhs
86      __pLabel = 10
```

```
87
88      formalisms.petrinet.petrinetMM_Post.T_to_P out_rhs_to_free
89        t = out_rhs
90        p = free
91        __pLabel = 11
92
93      formalisms.transformations.rule.PostConditionPatternContents rhs_to_free
94        pattern = rhs
95        element = free
96
97      formalisms.transformations.rule.PostConditionPatternContents rhs_to_rail
98        pattern = rhs
99        element = rail
100
101     formalisms.transformations.rule.PostConditionPatternContents rhs_to_in_rhs
102       pattern = rhs
103       element = in_rhs
104
105     formalisms.transformations.rule.PostConditionPatternContents rhs_to_out_rhs
106       pattern = rhs
107       element = out_rhs
108
109     formalisms.transformations.rule.PostConditionPatternContents rhs_to_free_to_in_rhs
110       pattern = rhs
111       element = free_to_in_rhs
112
113     formalisms.transformations.rule.PostConditionPatternContents rhs_to_in_rhs_to_rail
114       pattern = rhs
115       element = in_rhs_to_rail
116
117     formalisms.transformations.rule.PostConditionPatternContents rhs_to_rail_to_out_rhs
118       pattern = rhs
119       element = rail_to_out_rhs
```

```
120
121     formalisms.transformations.rule.PostConditionPatternContents rhs_to_out_rhs_to_free
122       pattern = rhs
123       element = out_rhs_to_free
124
125   package rail_train_to_petrinet
126     # # # #
127     # LHS #
128     # # # #
129     formalisms.transformations.rule.LHS lhs
130
131     formalisms.trainsim.trainsimMM_Pre.Rail r_lhs
132       __pLabel = 1
133
134     formalisms.trainsim.trainsimMM_Pre.Train t_lhs
135       __pLabel = 2
136
137     formalisms.trainsim.trainsimMM_Pre.T_on_TP t_on_tp_lhs
138       t = t_lhs
139       tp = r_lhs
140       __pLabel = 3
141
142     formalisms.transformations.rule.PreConditionPatternContents lhs_to_r_lhs
143       pattern = lhs
144       element = r_lhs
145
146     formalisms.transformations.rule.PreConditionPatternContents lhs_to_t_lhs
147       pattern = lhs
148       element = t_lhs
149
150     formalisms.transformations.rule.PreConditionPatternContents lhs_to_t_on_tp_lhs
151       pattern = lhs
152       element = t_on_tp_lhs
```

```
153
154     # # # #
155     # RHS #
156     # # # #
157     formalisms.transformations.rule.RHS rhs
158
159     formalisms.petrinet.petrinetMM_Post.Place free
160       __pLabel = 4
161       tokens = {{int: arkm3.object.Clabject} match: void}:
162         match[4].tokens = 0
163
164     formalisms.petrinet.petrinetMM_Post.Place rail
165       __pLabel = 5
166       tokens = {{int: arkm3.object.Clabject} match: void}:
167         match[5].tokens = 1
168
169     formalisms.petrinet.petrinetMM_Post.Transition in_rhs
170       __pLabel = 6
171
172     formalisms.petrinet.petrinetMM_Post.Transition out_rhs
173       __pLabel = 7
174
175     formalisms.petrinet.petrinetMM_Post.P_to_T free_to_in_rhs
176       p = free
177       t = in_rhs
178       __pLabel = 8
179
180     formalisms.petrinet.petrinetMM_Post.T_to_P in_rhs_to_rail
181       t = in_rhs
182       p = rail
183       __pLabel = 9
184
185     formalisms.petrinet.petrinetMM_Post.P_to_T rail_to_out_rhs
```

```
186       p = rail
187       t = out_rhs
188       __pLabel = 10
189
190     formalisms.petrinet.petrinetMM_Post.T_to_P out_rhs_to_free
191       t = out_rhs
192       p = free
193       __pLabel = 11
194
195     formalisms.transformations.rule.PostConditionPatternContents rhs_to_free
196       pattern = rhs
197       element = free
198
199     formalisms.transformations.rule.PostConditionPatternContents rhs_to_rail
200       pattern = rhs
201       element = rail
202
203     formalisms.transformations.rule.PostConditionPatternContents rhs_to_in_rhs
204       pattern = rhs
205       element = in_rhs
206
207     formalisms.transformations.rule.PostConditionPatternContents rhs_to_out_rhs
208       pattern = rhs
209       element = out_rhs
210
211     formalisms.transformations.rule.PostConditionPatternContents rhs_to_free_to_in_rhs
212       pattern = rhs
213       element = free_to_in_rhs
214
215     formalisms.transformations.rule.PostConditionPatternContents rhs_to_in_rhs_to_rail
216       pattern = rhs
217       element = in_rhs_to_rail
218
```

```
219     formalisms.transformations.rule.PostConditionPatternContents rhs_to_rail_to_out_rhs
220       pattern = rhs
221       element = rail_to_out_rhs
222
223     formalisms.transformations.rule.PostConditionPatternContents rhs_to_out_rhs_to_free
224       pattern = rhs
225       element = out_rhs_to_free
226
227   package split_to_petrinet
228     # # # #
229     # NAC #
230     # # # #
231     formalisms.transformations.rule.NAC nac
232       name = 'NAC_1'
233
234     formalisms.trainsim.trainsimMM_Pre.Split s_nac
235       __pLabel = 1
236
237     formalisms.trainsim.trainsimMM_Pre.Train t_nac
238       __pLabel = 2
239
240     formalisms.trainsim.trainsimMM_Pre.T_on_TP t_on_tp_nac
241       t = t_nac
242       tp = s_nac
243       __pLabel = 3
244
245     formalisms.transformations.rule.PreConditionPatternContents nac_to_s_nac
246       pattern = nac
247       element = s_nac
248
249     formalisms.transformations.rule.PreConditionPatternContents nac_to_t_nac
250       pattern = nac
251       element = t_nac
```

```
252
253     formalisms.transformations.rule.PreConditionPatternContents nac_to_t_on_tp_nac
254       pattern = nac
255       element = t_on_tp_nac
256
257     # # # #
258     # LHS #
259     # # # #
260     formalisms.transformations.rule.LHS lhs
261
262     formalisms.trainsim.trainsimMM_Pre.Split s_lhs
263       __pLabel = 1
264
265     formalisms.transformations.rule.PreConditionPatternContents lhs_to_s_lhs
266       pattern = lhs
267       element = s_lhs
268
269     # # # #
270     # RHS #
271     # # # #
272     formalisms.transformations.rule.RHS rhs
273
274     formalisms.petrinet.petrinetMM_Post.Place free
275       __pLabel = 4
276       tokens = {{int: arkm3.object.Clabject} match: void}:
277         match[4].tokens = 1
278
279     formalisms.petrinet.petrinetMM_Post.Place rail
280       __pLabel = 5
281       tokens = {{int: arkm3.object.Clabject} match: void}:
282         match[5].tokens = 0
283
284     formalisms.petrinet.petrinetMM_Post.Transition in_rhs
```

```
285       __pLabel = 6
286
287     formalisms.petrinet.petrinetMM_Post.Transition out1_rhs
288       __pLabel = 7
289
290     formalisms.petrinet.petrinetMM_Post.Transition out2_rhs
291       __pLabel = 8
292
293     formalisms.petrinet.petrinetMM_Post.P_to_T free_to_in_rhs
294       p = free
295       t = in_rhs
296       __pLabel = 9
297
298     formalisms.petrinet.petrinetMM_Post.T_to_P in_rhs_to_rail
299       t = in_rhs
300       p = rail
301       __pLabel = 10
302
303     formalisms.petrinet.petrinetMM_Post.P_to_T rail_to_out1_rhs
304       p = rail
305       t = out1_rhs
306       __pLabel = 11
307
308     formalisms.petrinet.petrinetMM_Post.P_to_T rail_to_out2_rhs
309       p = rail
310       t = out2_rhs
311       __pLabel = 12
312
313     formalisms.petrinet.petrinetMM_Post.T_to_P out1_rhs_to_free
314       t = out1_rhs
315       p = free
316       __pLabel = 13
317
```

```
318     formalisms.petrinet.petrinetMM_Post.T_to_P out2_rhs_to_free
319       t = out2_rhs
320       p = free
321       __pLabel = 14
322
323     formalisms.transformations.rule.PostConditionPatternContents rhs_to_free
324       pattern = rhs
325       element = free
326
327     formalisms.transformations.rule.PostConditionPatternContents rhs_to_rail
328       pattern = rhs
329       element = rail
330
331     formalisms.transformations.rule.PostConditionPatternContents rhs_to_in_rhs
332       pattern = rhs
333       element = in_rhs
334
335     formalisms.transformations.rule.PostConditionPatternContents rhs_to_out1_rhs
336       pattern = rhs
337       element = out1_rhs
338
339     formalisms.transformations.rule.PostConditionPatternContents rhs_to_out2_rhs
340       pattern = rhs
341       element = out2_rhs
342
343     formalisms.transformations.rule.PostConditionPatternContents rhs_to_free_to_in_rhs
344       pattern = rhs
345       element = free_to_in_rhs
346
347     formalisms.transformations.rule.PostConditionPatternContents rhs_to_in_rhs_to_rail
348       pattern = rhs
349       element = in_rhs_to_rail
350
```

```
351     formalisms.transformations.rule.PostConditionPatternContents rhs_to_rail_to_out1_rhs
352       pattern = rhs
353       element = rail_to_out1_rhs
354
355     formalisms.transformations.rule.PostConditionPatternContents rhs_to_rail_to_out2_rhs
356       pattern = rhs
357       element = rail_to_out2_rhs
358
359     formalisms.transformations.rule.PostConditionPatternContents rhs_to_out1_rhs_to_free
360       pattern = rhs
361       element = out1_rhs_to_free
362
363     formalisms.transformations.rule.PostConditionPatternContents rhs_to_out2_rhs_to_free
364       pattern = rhs
365       element = out2_rhs_to_free
366
367   package split_train_to_petrinet
368     # # # #
369     # LHS #
370     # # # #
371     formalisms.transformations.rule.LHS lhs
372
373     formalisms.trainsim.trainsimMM_Pre.Split s_lhs
374       __pLabel = 1
375
376     formalisms.trainsim.trainsimMM_Pre.Train t_lhs
377       __pLabel = 2
378
379     formalisms.trainsim.trainsimMM_Pre.T_on_TP t_on_tp_lhs
380       t = t_lhs
381       tp = s_lhs
382       __pLabel = 3
383
```

```
384     formalisms.transformations.rule.PreConditionPatternContents lhs_to_s_lhs
385       pattern = lhs
386       element = s_lhs
387
388     formalisms.transformations.rule.PreConditionPatternContents lhs_to_t_lhs
389       pattern = lhs
390       element = t_lhs
391
392     formalisms.transformations.rule.PreConditionPatternContents lhs_to_t_on_tp_lhs
393       pattern = lhs
394       element = t_on_tp_lhs
395
396     # # # #
397     # RHS #
398     # # # #
399     formalisms.transformations.rule.RHS rhs
400
401     formalisms.petrinet.petrinetMM_Post.Place free
402       __pLabel = 4
403       tokens = {{int: arkm3.object.Clabject} match: void}:
404         match[4].tokens = 0
405
406     formalisms.petrinet.petrinetMM_Post.Place rail
407       __pLabel = 5
408       tokens = {{int: arkm3.object.Clabject} match: void}:
409         match[5].tokens = 1
410
411     formalisms.petrinet.petrinetMM_Post.Transition in_rhs
412       __pLabel = 6
413
414     formalisms.petrinet.petrinetMM_Post.Transition out1_rhs
415       __pLabel = 7
416
```

```
417     formalisms.petrinet.petrinetMM_Post.Transition out2_rhs
418       __pLabel = 8
419
420     formalisms.petrinet.petrinetMM_Post.P_to_T free_to_in_rhs
421       p = free
422       t = in_rhs
423       __pLabel = 9
424
425     formalisms.petrinet.petrinetMM_Post.T_to_P in_rhs_to_rail
426       t = in_rhs
427       p = rail
428       __pLabel = 10
429
430     formalisms.petrinet.petrinetMM_Post.P_to_T rail_to_out1_rhs
431       p = rail
432       t = out1_rhs
433       __pLabel = 11
434
435     formalisms.petrinet.petrinetMM_Post.P_to_T rail_to_out2_rhs
436       p = rail
437       t = out2_rhs
438       __pLabel = 12
439
440     formalisms.petrinet.petrinetMM_Post.T_to_P out1_rhs_to_free
441       t = out1_rhs
442       p = free
443       __pLabel = 13
444
445     formalisms.petrinet.petrinetMM_Post.T_to_P out2_rhs_to_free
446       t = out2_rhs
447       p = free
448       __pLabel = 14
449
```

```
450     formalisms.transformations.rule.PostConditionPatternContents rhs_to_free
451       pattern = rhs
452       element = free
453
454     formalisms.transformations.rule.PostConditionPatternContents rhs_to_rail
455       pattern = rhs
456       element = rail
457
458     formalisms.transformations.rule.PostConditionPatternContents rhs_to_in_rhs
459       pattern = rhs
460       element = in_rhs
461
462     formalisms.transformations.rule.PostConditionPatternContents rhs_to_out1_rhs
463       pattern = rhs
464       element = out1_rhs
465
466     formalisms.transformations.rule.PostConditionPatternContents rhs_to_out2_rhs
467       pattern = rhs
468       element = out2_rhs
469
470     formalisms.transformations.rule.PostConditionPatternContents rhs_to_free_to_in_rhs
471       pattern = rhs
472       element = free_to_in_rhs
473
474     formalisms.transformations.rule.PostConditionPatternContents rhs_to_in_rhs_to_rail
475       pattern = rhs
476       element = in_rhs_to_rail
477
478     formalisms.transformations.rule.PostConditionPatternContents rhs_to_rail_to_out1_rhs
479       pattern = rhs
480       element = rail_to_out1_rhs
481
482     formalisms.transformations.rule.PostConditionPatternContents rhs_to_rail_to_out2_rhs
```

```
483       pattern = rhs
484       element = rail_to_out2_rhs
485
486     formalisms.transformations.rule.PostConditionPatternContents rhs_to_out1_rhs_to_free
487       pattern = rhs
488       element = out1_rhs_to_free
489
490     formalisms.transformations.rule.PostConditionPatternContents rhs_to_out2_rhs_to_free
491       pattern = rhs
492       element = out2_rhs_to_free
```

LISTING A.1: The TrainSim-to-PetriNet transformation, modelled in Ark

s

Appendix B

ArkM3 Metamodels

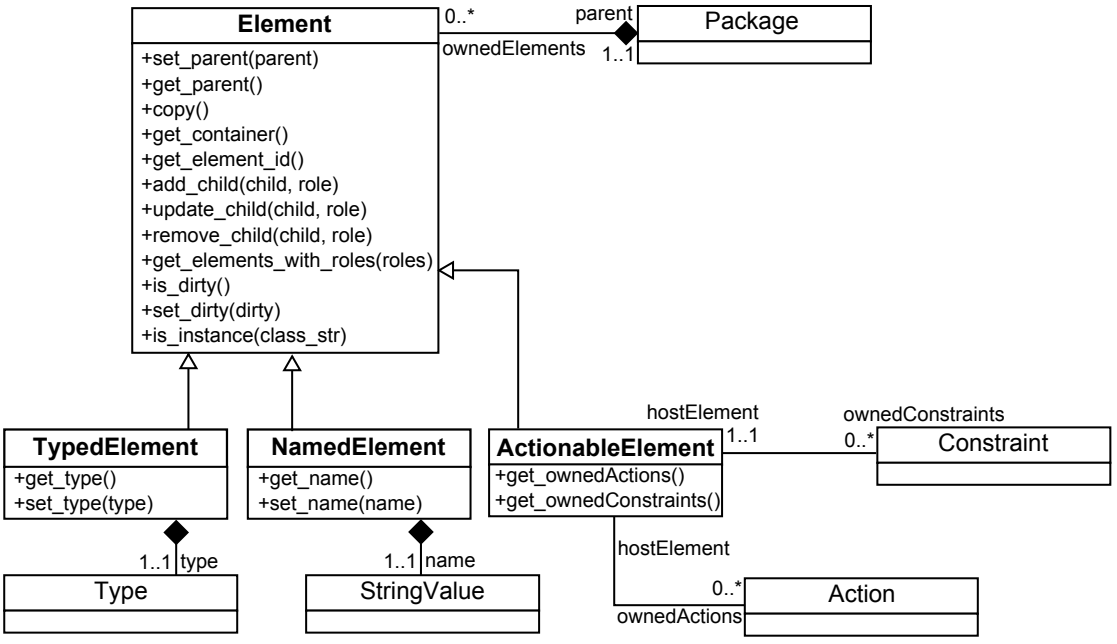


FIGURE B.1: The ArkM3 Element metamodel.

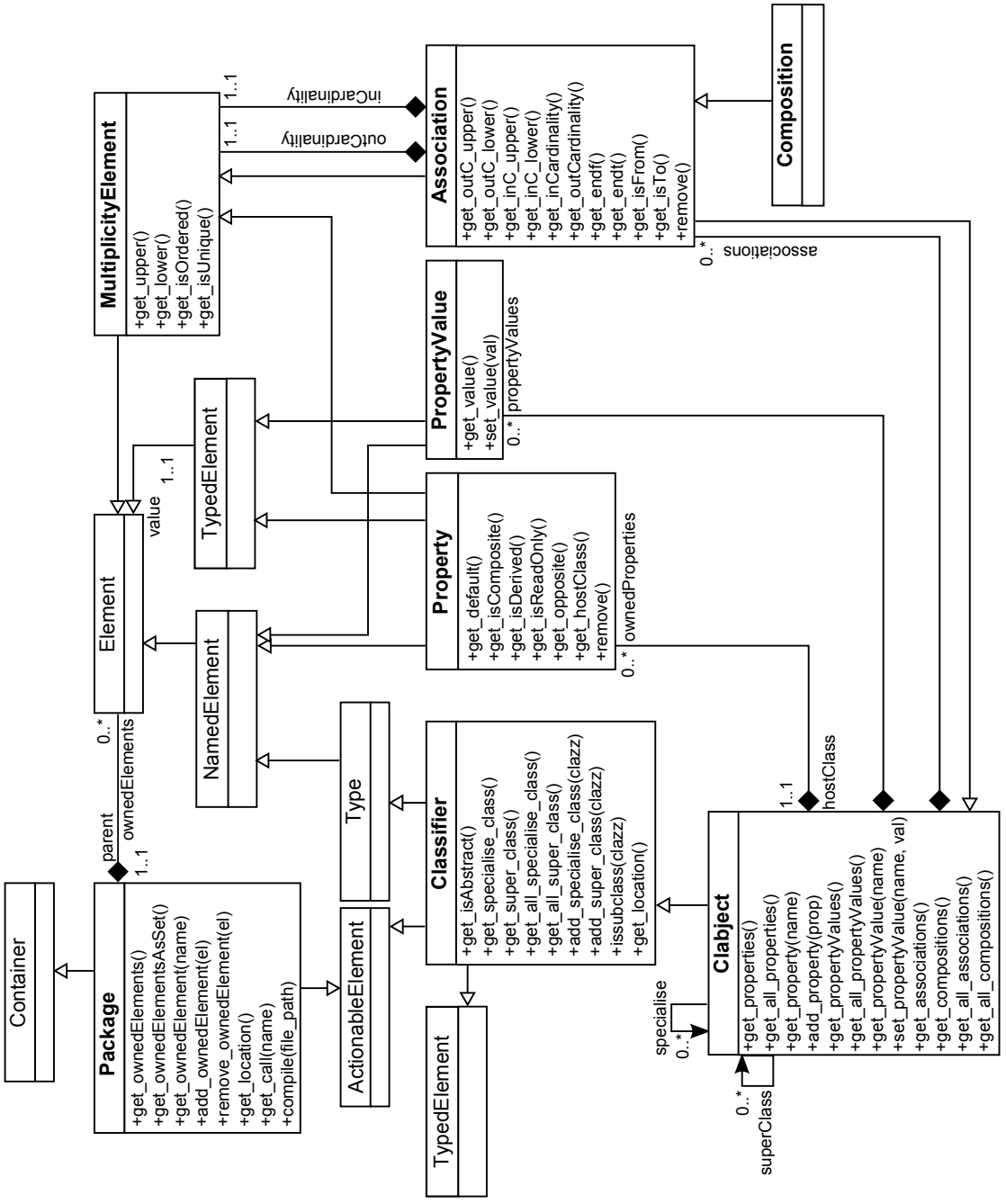


FIGURE B.2: The ArkM3 Object metamodel.

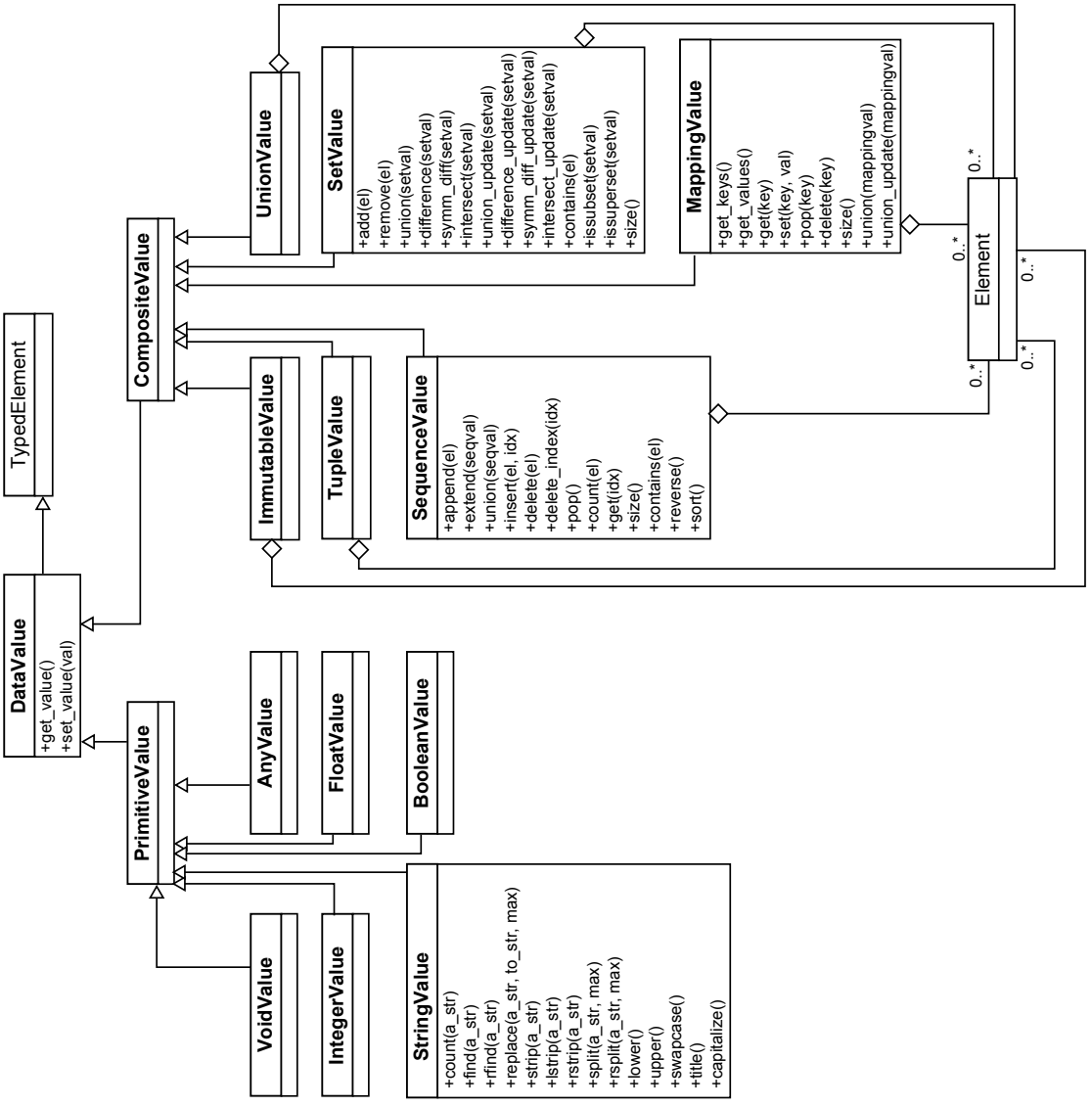


FIGURE B.3: The ArkM3 DataValue metamodel.

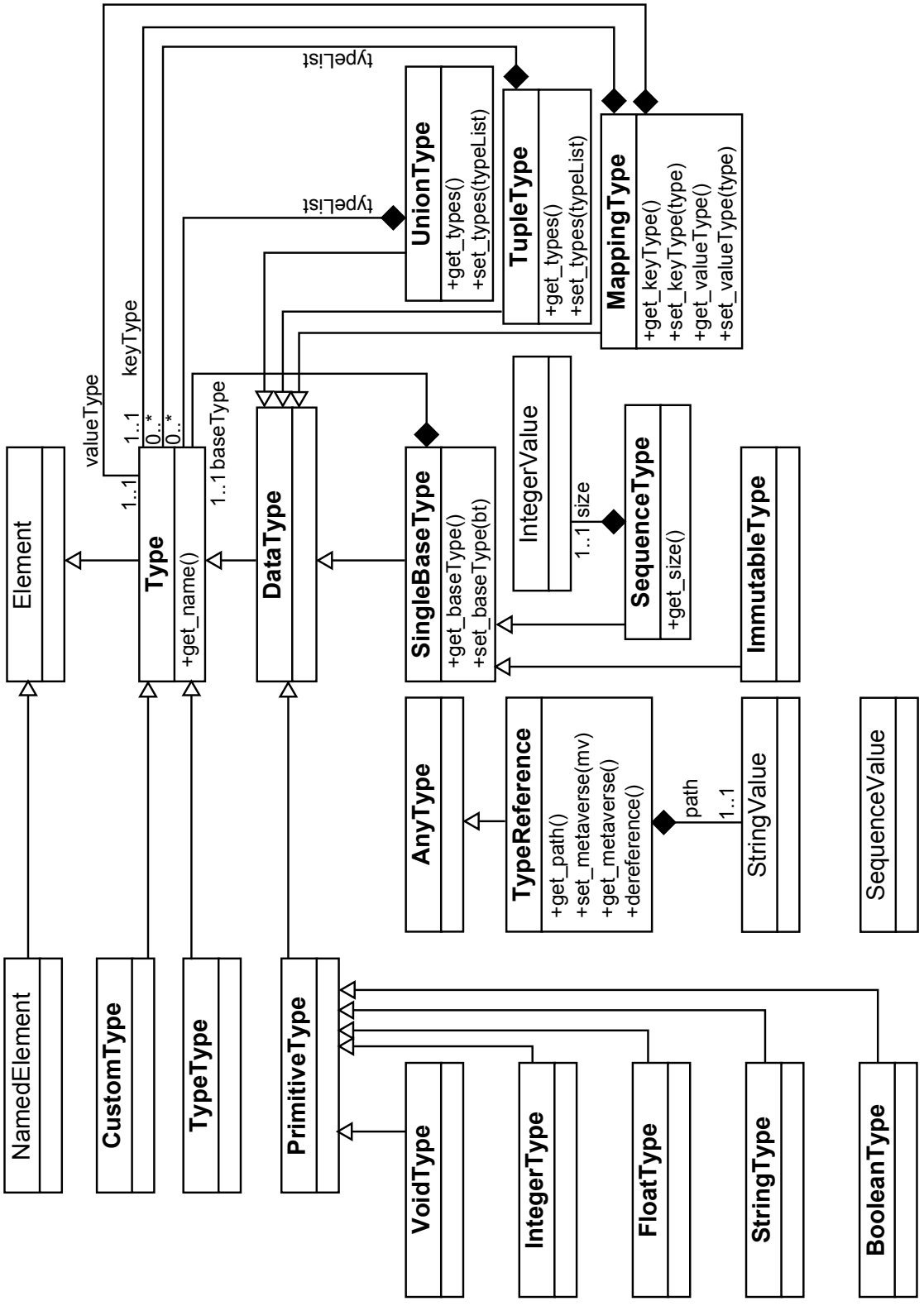


FIGURE B.4: The ArkM3 DataType metamodel.

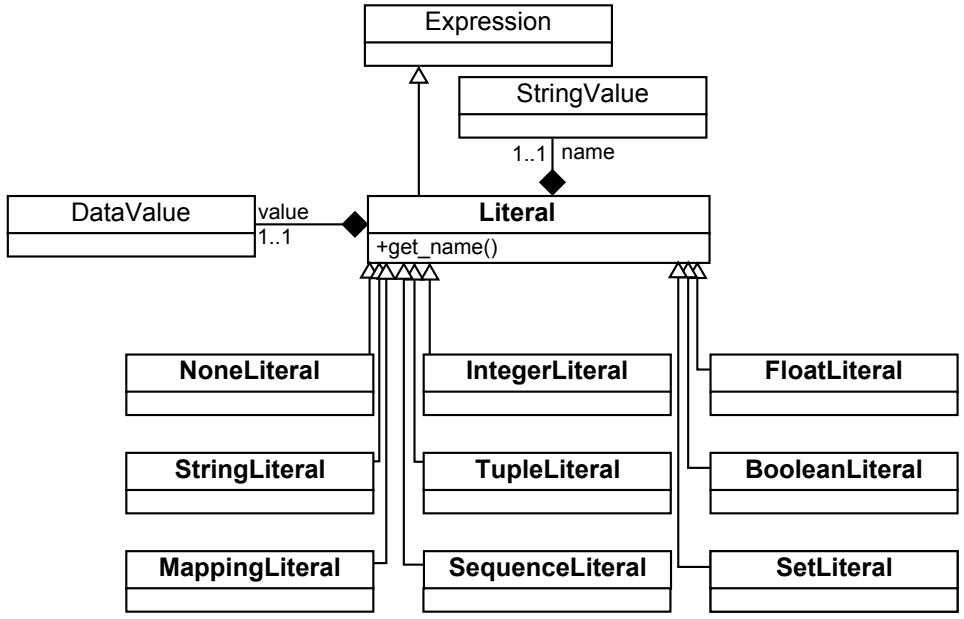


FIGURE B.5: The ArkM3 Literal metamodel.

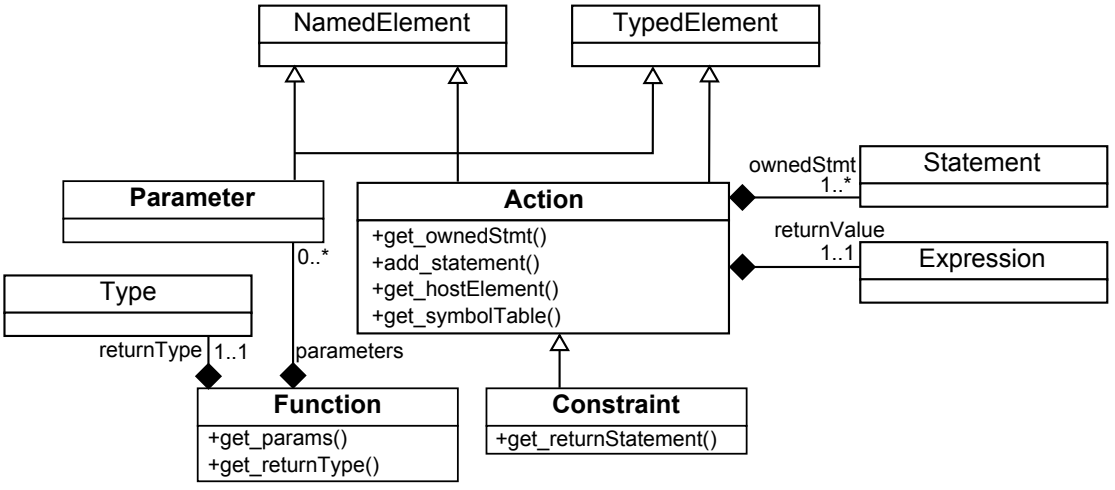


FIGURE B.6: The ArkM3 Action and Constraint metamodel.

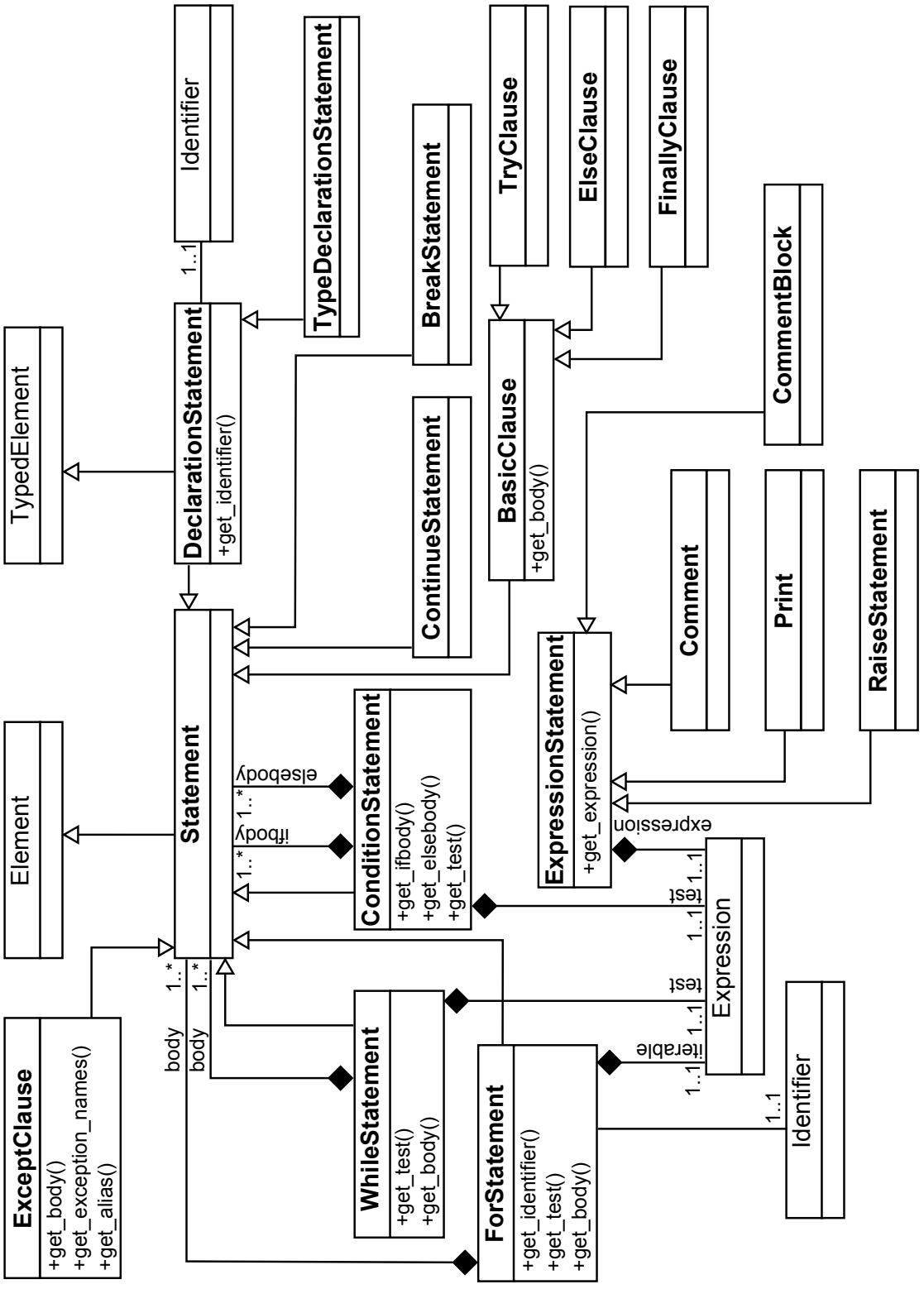


FIGURE B.7: The ArkM3 (Action Language) Statement metamodel.

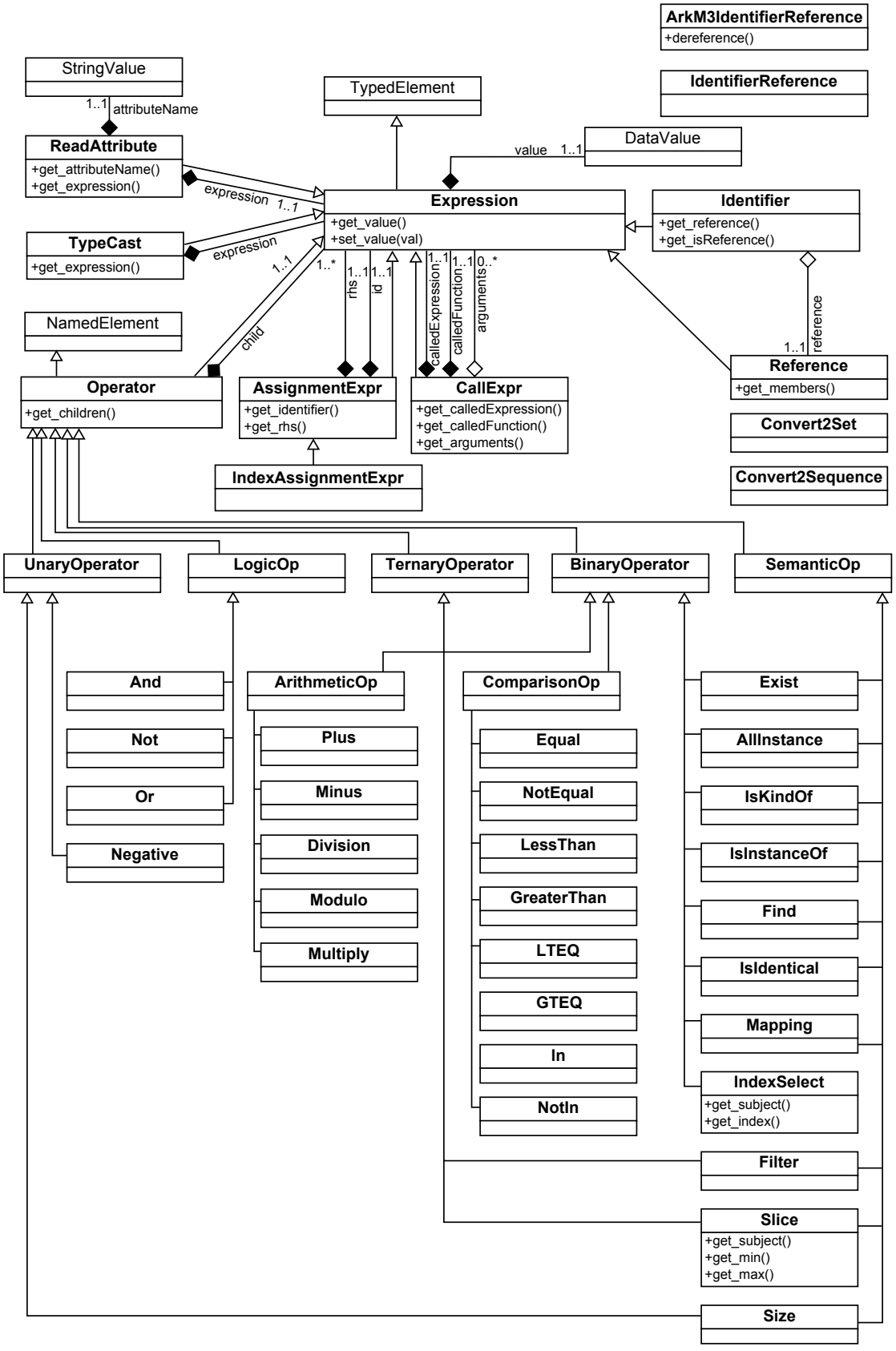


FIGURE B.8: The ArkM3 (Action Language) Expression metamodel.

Appendix C

Graph Matching Algorithm Code

```
'''
Created on 25-mrt.-2013

@author: Simon
'''
from arkm3.Action import IdentifierReference
from arkm3.ArkM3 import ArkM3
from arkm3.DataType import TypeReference
from arkm3.DataValue import StringValue
from arkm3.Object import Clabject, Association

class State:
    '''
    Initializes the state. Parameters:
      - p: The pattern (LHS, NAC) to be matched.
      - s: The source model (an ArkM3 package).
      - arkm3: A reference to the metaverse.
    '''
    def __init__(self, p, s, arkm3):
        self.m_p = []
        self.m_s = []

```

```
23          self.s = s
24          self.p = p
25
26          self.matches = []
27
28          # Read definitions of pattern element classes from the metaverse.
29          pattern_class = arkm3.read('formalisms.transformations.rule.PatternElement')._object
30          pattern_association_class = arkm3.read('formalisms.transformations.rule.PatternAssociation')
        ._object
31          pattern_content_class = arkm3.read('formalisms.transformations.rule.
        PreConditionPatternContents')._object
32
33          # Find all elements in the pattern, which are connected to p.
34          pattern_contents = []
35          for a in p.get_associations().get_value():
36              for super_type in a.get_type().get_all_super_class().get_value() + [a.get_type()]:
37                  if super_type == pattern_content_class:
38                      pattern_contents.append(a.get_isTo())
39
40          # Make a distinction between nodes (Clabjects) and edges (Associations) of the pattern.
41          self.p_nodes = []
42          self.p_edges = []
43          for c in pattern_contents:
44              if isinstance(c, IdentifierReference): c = c.dereference()
45              a_type = c.get_type()
46              if isinstance(a_type, Association):
47                  # Ontological check.
48                  super_classes = c.get_type().get_all_super_class().get_value()
49                  for super_class in super_classes:
50                      if super_class.equals(pattern_association_class):
51                          self.p_edges.append(c)
52              elif isinstance(a_type, Clabject):
53                  # Ontological check.
```

```
54                for super_class in c.get_type().get_all_super_class().get_value():
55                    if super_class == pattern_class:
56                        self.p_nodes.append(c)
57
58        # Make a distinction between nodes (Clabjects) and edges (Associations) of the source model
      .
59        self.s_nodes = []
60        self.s_edges = []
61        for c in self.s.get_ownedElements().get_value().itervalues():
62            if isinstance(c, IdentifierReference): c = c.dereference()
63            a_type = c.get_type()
64            if isinstance(a_type, Association):
65                self.s_edges.append(c)
66            elif isinstance(a_type, Clabject):
67                self.s_nodes.append(c)
68
69    '''
70    Store a <PatternElement, SourceElement> mapping. Parameters:
71      - p: The pattern element.
72      - s: The source model element.
73    '''
74    def store_mapping(self, p, s):
75        self.m_p.append(p)
76        self.m_s.append(s)
77
78    '''
79    Undo a <PatternElement, SourceElement> mapping. Parameters:
80      - p: The pattern element.
81      - s: The source model element.
82    '''
83    def undo_mapping(self, p, s):
84        self.m_p.remove(p)
85        self.m_s.remove(s)
```

```
86
87      '''
88      Stores the currently saved <PatternElement, SourceElement> (in self.m_p and self.m_s) as a
         match.
89      A match is a mapping of all pattern elements on a source element.
90      '''
91      def store_match(self):
92          match = {}
93          for i in range(len(self.m_p)):
94              match[self.m_p[i].get_propertyValue(StringValue('__pLabel')).get_value().get_value().
         get_value()] = self.m_s[i]
95
96          for i in range(len(self.m_p)):
97              p_properties = self.m_p[i].get_all_properties().get_value()
98              for prop in p_properties.itervalues():
99                  name = prop.get_name().get_value()
100                 if name.startswith('__p__p'):
101                     name = name[3:]
102                 elif name.startswith('__p'):
103                     continue
104
105                 ''' execute constraint on property '''
106                 if not self.m_p[i].get_propertyValue(name).get_value().execute(match):
107                     return
108
109         ''' execute LHS constraint '''
110         if self.matches.count(match) == 0 and self.p.get_propertyValue(StringValue('p_constraint')).
         get_value().execute(match):
111             self.matches.append(match)
112
113     '''
114     Returns True if all pattern elements have been mapped onto a source model element, else False.
115     '''
```

```
116     def mapping_complete(self):
117         return len(self.m_p) == len(self.p_nodes + self.p_edges)
118 
119     '''
120     Returns all complete matches of the pattern in the source model.
121     '''
122     def get_matches(self):
123         return self.matches
124 
125 '''
126 Suggests a <PatternElement, SourceElement> mapping.
127 '''
128 def suggest_mapping(state):
129     for p_n in state.p_nodes + state.p_edges:
130         for s_n in state.s_nodes + state.s_edges:
131             if state.m_p.count(p_n) == 0 and state.m_s.count(s_n) == 0:
132                 yield p_n, s_n
133 
134 ''' Returns True if the pattern element 'lhs_el' can be mapped on the source model element 'host_el
        '. '''
135 def is_feasible(state, lhs_el, host_el):
136     ''' Mp and Ms after lhs_el and host_el are added. '''
137     f_m_p = state.m_p + [lhs_el]  # Mp
138     f_m_s = state.m_s + [host_el] # Ms
139 
140     # A match is a mapping between pattern element labels and source model elements.
141     match = {}
142     for i in range(len(f_m_p)):
143         match[f_m_p[i].get_propertyValue(StringValue('__pLabel')).get_value().get_value().get_value()
        ] = f_m_s[i]
144 
145     f_m_p = set(f_m_p)
146     f_m_s = set(f_m_s)
```

```
147
148     # Compute the outgoing and incoming elements of the pattern node.
149     out_lhs = set() # out(p)
150     in_lhs = set()  # in(p)
151
152     if isinstance(lhs_el, Association):
153         out_lhs.add(lhs_el.get_isTo())
154         in_lhs.add(lhs_el.get_isFrom())
155     elif isinstance(lhs_el, Clabject):
156         for a in lhs_el.get_associations().get_value():
157             if a.get_isFrom() == lhs_el:
158                 out_lhs.add(a)
159             elif a.get_isTo() == lhs_el and a.get_isFrom() in state.p_nodes:
160                 in_lhs.add(a)
161
162     # Compute the outgoing and incoming elements of the source model node.
163     out_host = set() # out(s)
164     in_host = set() # in(s)
165
166     if isinstance(host_el, Association):
167         out_host.add(host_el.get_isTo())
168         in_host.add(host_el.get_isFrom())
169     elif isinstance(host_el, Clabject):
170         for a in host_el.get_associations().get_value():
171             if a.get_isFrom() == host_el:
172                 out_host.add(a)
173             elif a.get_isTo() == host_el:
174                 in_host.add(a)
175
176     # in_lhs_mapped is the set of source model elements which the elements in in_lhs are mapped to
177     in_lhs_mapped = set([match[s.get_propertyValue(StringValue('__pLabel')).get_value().get_value().
        get_value()] for s in in_lhs if s.get_propertyValue(StringValue('__pLabel')).get_value().
        get_value().get_value() in match])
```

```
178     # out_lhs_mapped is the set of source model elements which the elements in out_lhs are mapped to
179     out_lhs_mapped = set([match[s.get_propertyValue(StringValue('__pLabel')).get_value().get_value().
         get_value()] for s in out_lhs if s.get_propertyValue(StringValue('__pLabel')).get_value().
         get_value().get_value() in match])
180     inout_lhs_mapped = in_lhs_mapped | out_lhs_mapped
181     inout_host = in_host | out_host
182
183     ''' conforms_to '''
184     feasible =  lhs_el.__class__ == host_el.__class__ and \
185                 lhs_el.get_type().get_name() == host_el.get_type().get_name() and \
186                 len(out_lhs) <= len(out_host) and \
187                 len(in_lhs) <= len(in_host) and \
188                 len(out_lhs & f_m_p) == len(out_host & f_m_s) and \
189                 len(in_lhs & f_m_p) == len(in_host & f_m_s) and \
190                 len(in_lhs_mapped - in_host) == 0 and \
191                 len(out_lhs_mapped - out_host) == 0 and \
192                 len(inout_lhs_mapped - inout_host) == 0
193
194     return feasible
195
196 ''' Recursive function which computes all complete matches of a pattern in a source model. '''
197 def extend(state):
198     if state.mapping_complete():
199         state.store_match()
200         return
201
202     suggested_mappings = suggest_mapping(state)
203     for lhs_el, host_el in suggested_mappings:
204         if is_feasible(state, lhs_el, host_el):
205             state.store_mapping(lhs_el, host_el)
206             extend(state)
207             state.undo_mapping(lhs_el, host_el)
208
```

```
209 ''' Returns a list of all complete matches of the pattern 'lhs' in the source model 'host_graph'.
        '''
210 def match(lhs, host_graph, arkm3):
211     state = State(lhs, host_graph, arkm3)
212     extend(state)
213     return state.get_matches()
214
215 '''
216 Rewrites a model using the RHS of a rule. Parameters:
217   - match: A match obtained through the match() function.
218   - host_graph: The host model (an ArkM3 package).
219   - rhs: The RHS of the rule.
220   - arkm3: A reference to the metaverse.
221 '''
222 def rewrite(match, host_graph, rhs, arkm3):
223     # in_rhs keeps track of the pattern labels of the elements in the RHS.
224     in_rhs = set()
225     # new_elements is the set of newly created elements by the RHS.
226     new_elements = {}
227
228     # Read definitions of pattern element classes from the metaverse.
229     pattern_class = arkm3.read('formalisms.transformations.rule.PatternElement')._object
230     pattern_association_class = arkm3.read('formalisms.transformations.rule.PatternAssociation').
        _object
231     pattern_content_class = arkm3.read('formalisms.transformations.rule.PostConditionPatternContents
        ')._object
232
233     # Find all elements in the pattern, which are connected to rhs.
234     pattern_contents = []
235     for a in rhs.get_associations().get_value():
236         if pattern_content_class in a.get_all_super_class().get_value() + [a.get_type()]:
237             pattern_contents.append(a.get_isTo())
238
```

```
239     # Make a distinction between nodes (Clabjects) and edges (Associations) of the RHS.
240     rhs_nodes = []
241     rhs_edges = []
242     for c in pattern_contents:
243             if isinstance(c, IdentifierReference): c = c.dereference()
244             a_type = c.get_type()
245             if isinstance(a_type, Association):
246                 super_classes = c.get_type().get_all_super_class().get_value()
247                 for super_class in super_classes:
248                     if super_class.equals(pattern_association_class):
249                         rhs_edges.append(c)
250             elif isinstance(a_type, Clabject):
251                 for super_class in c.get_type().get_all_super_class().get_value():
252                     if super_class == pattern_class:
253                         rhs_nodes.append(c)
254
255     # Iterate over all RHS nodes.
256     for node in rhs_nodes:
257         in_rhs.add(node.get_propertyValue(StringValue('__pLabel')).get_value().get_value().get_value
        ())
258
259         if not node.get_propertyValue(StringValue('__pLabel')).get_value().get_value().get_value() in
         match:
260             ''' Create '''
261             a_type = node.get_type().get_parent().get_location().get_value()[:-5] + '.' + node.
        get_type().get_name().get_value()
262             new_elements[node.get_propertyValue(StringValue('__pLabel')).get_value().get_value().
        get_value()] = arkm3.create(host_graph.get_location().get_value(), ArkM3.OBJECT, '', a_type,
        False)._object
263
264         ''' Update '''
265         node_properties = node.get_type().get_all_properties().get_value()
266         for prop in node_properties.itervalues():
```

```
267             name = prop.get_name().get_value()
268             if name.startswith('__p__p'):
269                 ''' RAMified __pLabel '''
270                 name = StringValue(name[3:])
271             elif name.startswith('__p'):
272                 ''' __pLabel, should be ignored here '''
273                 continue
274
275             ''' execute action '''
276             node.get_propertyValue(name).get_value().execute(match)
277
278     # Iterate over all RHS edges.
279     for edge in rhs_edges:
280         if not edge.get_propertyValue(StringValue('__pLabel')).get_value().get_value().get_value() in
          match:
281             ''' Create '''
282             from_label = edge.get_isFrom().get_propertyValue(StringValue('__pLabel')).get_value().
        get_value().get_value()
283             to_label = edge.get_isTo().get_propertyValue(StringValue('__pLabel')).get_value().
        get_value().get_value()
284             from_node = match[from_label] if from_label in match else new_elements[from_label]
285             to_node = match[to_label] if to_label in match else new_elements[to_label]
286             a_type = edge.get_type().get_parent().get_location().get_value()[:-5] + '.' + edge.
        get_type().get_name().get_value()
287             new_elements[edge.get_propertyValue(StringValue('__pLabel')).get_value().get_value().
        get_value()] = arkm3.create(host_graph.get_location().get_value(), ArkM3.ASSOCIATIONINSTANCE, '
        ', a_type, from_node.get_location().get_value(), to_node.get_location().get_value())
288
289         ''' Update '''
290         edge_properties = edge.get_type().get_all_properties()
291         for prop in edge_properties.get_value().itervalues():
292             name = prop.get_name().get_value()
293             if name.startswith('__p__p'):
```

```
294                 ''' RAMified __pLabel '''
295                 name = StringValue(name[3:])
296             elif name.startswith('__p'):
297                 ''' __pLabel, should be ignored here '''
298                 continue
299
300             ''' execute action '''
301             edge.get_propertyValue(name).get_value().execute(match)
302
303         in_rhs.add(edge.get_propertyValue(StringValue('__pLabel')).get_value().get_value().get_value
    ())
304
305     # Find all nodes which are in the LHS, but not in the RHS.
306     in_lhs_not_rhs = list(set(match.keys()) - in_rhs)
307     for label in in_lhs_not_rhs:
308         ''' Delete '''
309         host_graph.get_ownedElements().delete(match[label].get_name())
310
311   ''' execute RHS action '''
312   rhs.get_propertyValue(StringValue('p_action')).get_value().execute(match)
```

LISTING C.1: The graph matching algorithm on ArkM3 data structures, written in Python

```
1  package rewriting
2    class State
3      arkm3.object.Package            s
4      arkm3.object.Clabject           p
5      [arkm3.object.Clabject]         m_p   = []
6      [arkm3.object.Clabject]         m_s   = []
7      [{int: arkm3.object.Clabject}]  matches = []
8      [arkm3.object.Clabject]         p_nodes = []
9      [arkm3.object.Association]      p_edges = []
10     [arkm3.object.Clabject]         s_nodes = []
11     [arkm3.object.Association]      s_edges = []
```

```
12
13    {rewriting.State, arkm3.object.Clabject, arkm3.object.Package: void} __init__
14    __init__ = {rewriting.State self, arkm3.object.Clabject p, arkm3.object.Package s: void}:
15      self.m_p       = []
16      self.m_s       = []
17
18      self.s         = s
19      self.p         = p
20
21      self.matches   = []
22
23      arkm3.object.Clabject pattern_class             = formalisms.transformations.rule.
      PatternElement
24      arkm3.object.Clabject pattern_association_class  = formalisms.transformations.rule.
      PatternAssociation
25      arkm3.object.Clabject pattern_content_class      = formalisms.transformations.rule.
      PreConditionPatternContents
26
27      # pattern contents
28      [arkm3.object.Clabject] pattern_contents = []
29      arkm3.object.Association a
30      for a in p.get_associations():
31        {any} types = set([a.get_type()]) + a.get_type().get_all_super_class()
32        if (types.contains(pattern_content_class)):
33          pattern_contents.append(a.get_isTo())
34
35      # differentiate between nodes (clabjects) and edges (associations) of the pattern
36      self.p_nodes = []
37      self.p_edges = []
38      arkm3.object.Clabject c
39      for c in pattern_contents:
40        if (c.is_instance('Association') and c.get_type().get_all_super_class().contains(
      pattern_association_class)):
```

```
41          self.p_edges.append(c)
42        elif (c.is_instance('Clabject') and c.get_type().get_all_super_class().contains(pattern_class
      )):
43          self.p_nodes.append(c)
44
45      # differentiate between nodes (clabjects) and edges (associations) of the host graph
46      self.s_nodes = []
47      self.s_edges = []
48      arkm3.object.Clabject c
49      for c in self.s.get_ownedElements():
50        if (c.is_instance('Association')):
51          self.s_edges.append(c)
52        elif (c.is_instance('Clabject')):
53          self.s_nodes.append(c)
54
55    {rewriting.State, arkm3.object.Package, arkm3.object.Package: void} store_mapping
56    store_mapping = {rewriting.State self, arkm3.object.Package p, arkm3.object.Package s: void}:
57      self.m_p.append(p)
58      self.m_s.append(s)
59
60    {rewriting.State, arkm3.object.Package, arkm3.object.Package: void} undo_mapping
61    undo_mapping = {rewriting.State self, arkm3.object.Package p, arkm3.object.Package s: void}:
62      self.m_p.delete(p)
63      self.m_s.delete(s)
64
65    {rewriting.State: void} store_match
66    store_match = {rewriting.State self: void}:
67      {int: arkm3.object.Clabject} match = map()
68
69      int i = 0
70      while (i < self.m_p.size()):
71        match[self.m_p[i].__pLabel] = self.m_s[i]
72        i = i + 1
```

```
73
74        i = 0
75        while (i < self.m_p.size()):
76          {arkm3.object.Property} p_properties = self.m_p[i].get_all_properties()
77          arkm3.object.Property prop
78          for prop in p_properties:
79            string name = prop.get_name()
80            if (name.find('__p__p') == 0):
81              name = name[3:]
82            elif (name.find('__p') == 0):
83              continue
84
85            if not self.m_p[i].get_propertyValue(name).get_value().execute(match):
86              return
87
88          i = i + 1
89
90        if self.matches.count(match) == 0 and ((formalisms.transformations.rule.PreConditionPattern)
       self.p).p_constraint.execute(match):
91          self.matches.append(match)
92
93      {rewriting.State: bool} mapping_complete
94      mapping_complete = {rewriting.State self: bool}:
95        return self.m_p.size() == self.p_nodes.size() + self.p_edges.size()
96
97      {rewriting.State: [{int: arkm3.object.Clabject}]} get_matches
98      get_matches = {rewriting.State self: [{int: arkm3.object.Clabject}]}:
99        return self.matches
100
101   suggest_mappings = {rewriting.State state: [[arkm3.object.Clabject]]}:
102     [[arkm3.object.Clabject]] mappings = []
103
104     arkm3.object.Clabject p_n
```

```
105     for p_n in state.p_nodes + state.p_edges:
106       arkm3.object.Clabject s_n
107       for s_n in state.s_nodes + state.s_edges:
108         mappings.append([p_n, s_n])
109
110     return mappings
111
112   is_feasible = {rewriting.State state, arkm3.object.Clabject lhs_el, arkm3.object.Clabject host_el:
        bool}:
113     [arkm3.object.Clabject] f_m_p_l = state.m_p + [lhs_el]
114     [arkm3.object.Clabject] f_m_s_l = state.m_s + [host_el]
115
116     {int: arkm3.object.Clabject} match = map()
117     int i = 0
118     while (i < f_m_p_l.size()):
119       match[f_m_p_l[i].__pLabel] = f_m_s_l[i]
120       i = i + 1
121
122     {arkm3.object.Clabject} f_m_p = set(f_m_p_l)
123     {arkm3.object.Clabject} f_m_s = set(f_m_s_l)
124
125     # compute outgoing and incoming elements of LHS element
126     {arkm3.object.Clabject} out_lhs = set()
127     {arkm3.object.Clabject} in_lhs = set()
128
129     if (lhs_el.is_instance('Association')):
130       arkm3.object.Association a_lhs_el = (arkm3.object.Association) lhs_el
131       out_lhs.add(a_lhs_el.get_isTo())
132       in_lhs.add(a_lhs_el.get_isFrom())
133     elif (lhs_el.is_instance('Clabject')):
134       arkm3.object.Association a
135       for a in lhs_el.get_associations():
136         if (a.get_isFrom() == lhs_el):
```

```
137           out_lhs.add(a)
138         elif (a.get_isTo() == lhs_el):
139           in_lhs.add(a)
140
141     # compute outgoing and incoming elements of host element
142     {arkm3.object.Clabject} out_host = set()
143     {arkm3.object.Clabject} in_host = set()
144
145     if (host_el.is_instance('Association')):
146       arkm3.object.Association a_host_el = (arkm3.object.Association) host_el
147       out_host.add(a_host_el.get_isTo())
148       in_host.add(a_host_el.get_isFrom())
149     elif (host_el.is_instance('Clabject')):
150       arkm3.object.Association a
151       for a in host_el.get_associations():
152         if (a.get_isFrom() == host_el):
153           out_host.add(a)
154         elif (a.get_isTo() == host_el):
155           in_host.add(a)
156
157     {arkm3.object.Clabject} in_lhs_mapped = set()
158     {arkm3.object.Clabject} out_lhs_mapped = set()
159     arkm3.object.Clabject el
160     for el in in_lhs:
161       if (match.get_keys().contains(el.__pLabel)):
162         in_lhs_mapped.add(match[el.__pLabel])
163     for el in out_lhs:
164       if (match.get_keys().contains(el.__pLabel)):
165         out_lhs_mapped.add(match[el.__pLabel])
166
167     {arkm3.object.Clabject} inout_lhs_mapped = in_lhs_mapped.union(out_lhs_mapped)
168     {arkm3.object.Clabject} inout_host = in_host.union(out_host)
169
```

```
170     bool feasible = lhs_el.get_type().get_name() == host_el.get_type().get_name()
171     feasible = feasible and out_lhs.size() <= out_host.size() and in_lhs.size() <= in_host.size()
172     feasible = feasible and ((out_lhs.intersect(f_m_p)).size() == (out_host.intersect(f_m_p)).size())
          and ((in_lhs.intersect(f_m_p)).size() == (in_host.intersect(f_m_p)).size())
173     feasible = feasible and ((in_lhs_mapped - in_host).size() == 0 and (out_lhs_mapped - out_host).
         size() == 0)
174     feasible = feasible and (inout_lhs_mapped - inout_host).size() == 0
175
176     return feasible
177
178   extend = {rewriting.State state: void}:
179     if (state.mapping_complete()):
180       state.store_match()
181       return
182
183     [[arkm3.object.Clabject]] suggested_mappings = suggest_mappings(state)
184
185     [arkm3.object.Clabject] suggested_mapping
186     for suggested_mapping in suggested_mappings:
187       arkm3.object.Clabject lhs_el = suggested_mapping[0]
188       arkm3.object.Clabject host_el = suggested_mapping[1]
189       if state.m_p.count(lhs_el) == 0 and state.m_s.count(host_el) == 0:
190         if is_feasible(state, lhs_el, host_el):
191           state.store_mapping(lhs_el, host_el)
192           extend(state)
193           state.undo_mapping(lhs_el, host_el)
194
195   match = {arkm3.object.Package lhs, arkm3.object.Package host_graph: [{int: arkm3.object.Clabject
        }]}:
196     rewriting.State state = rewriting.State(lhs, host_graph)
197     extend(state)
198     return state.get_matches()
199
```

```
200  rewrite = {{int: arkm3.object.Clabject} match, arkm3.object.Package host_graph, arkm3.object.
       Clabject rhs: void}:
201    {int} in_rhs = {}
202    {int: arkm3.object.Clabject} new_elements = map()
203
204    arkm3.object.Clabject  pattern_class         = formalisms.transformations.rule.PatternElement
205    arkm3.object.Clabject  pattern_association_class   = formalisms.transformations.rule.
       PatternAssociation
206    arkm3.object.Clabject  pattern_content_class     = formalisms.transformations.rule.
       PreConditionPatternContents
207
208    # pattern contents
209    [arkm3.object.Clabject] pattern_contents = []
210    arkm3.object.Association a
211    for a in rhs.get_associations():
212      {any} types = set([a.get_type()]) + a.get_type().get_all_super_class()
213      if (types.contains(pattern_content_class)):
214        pattern_contents.append(a.get_isTo())
215
216    [formalisms.transformations.rule.PatternElement] rhs_nodes = []
217    [formalisms.transformations.rule.PatternAssociation] rhs_edges = []
218
219    # differentiate between nodes (clabjects) and edges (associations) of the pattern
220    arkm3.object.Clabject c
221    for c in pattern_contents:
222      if (c.is_instance('Association') and c.get_type().get_all_super_class().contains(
       pattern_association_class)):
223        rhs_edges.append(c)
224      elif (c.is_instance('Clabject') and c.get_type().get_all_super_class().contains(pattern_class))
       :
225        rhs_nodes.append(c)
226
227    arkm3.object.Clabject node
```

```
228     for node in rhs_nodes:
229       in_rhs.add(node.__pLabel)
230       if not match.keys().contains(node.__pLabel):
231         # create
232         string a_type = node.get_type().get_parent().get_location()[:-5] + '.' + node.get_type().
        get_name()
233         new_elements[node.__pLabel] = create(host_graph.get_location(), OBJECT, '', a_type)
234
235       # update
236       {arkm3.object.PropertyValue} node_properties = node.get_all_propertyValues()
237       for prop in node_properties:
238         string name = prop.get_name()
239         if (name.find('__p__p') == 0):
240           name = name[3:]
241         elif (name.find('__p') == 0):
242           continue
243
244         prop.execute(match)
245
246     for edge in rhs_edges:
247       if not match.keys().contains(edge.__pLabel):
248         # create
249         arkm3.object.Clabject from_node
250         arkm3.object.Clabject to_node
251         int from_label = ((formalisms.transformations.rule.PatternElement) (edge.get_isFrom())).
        __pLabel
252         int to_label = ((formalisms.transformations.rule.PatternElement) edge.get_isTo()).__pLabel
253         if (match.keys().contains(from_label)):
254           from_node = match[from_label]
255         else:
256           from_node = new_elements[from_label]
257         if (match.keys().contains(to_label)):
258           to_node = match[to_label]
```

```
259         else:
260           to_node = new_elements[to_label]
261         string a_type = edge.get_type().get_parent().get_location()[:-5] + '.' + edge.get_type().
        get_name()
262         new_elements[edge.__pLabel] = create(host_graph.get_location(), ASSOCIATIONINSTANCE, '',
        a_type, from_node.get_location(), to_node.get_location())
263
264       # update
265       {arkm3.object.PropertyValue} edge_properties = edge.get_all_propertyValues()
266       for prop in edge_properties:
267         string name = prop.get_name()
268         if (name.find('__p__p') == 0):
269           name = name[3:]
270         elif (name.find('__p') == 0):
271           continue
272
273         prop.execute(match)
274
275       in_rhs.add(edge.__pLabel)
276
277     {int} in_lhs_not_rhs = set(match.keys()) - in_rhs
278     int label
279     for label in in_lhs_not_rhs:
280       # delete
281       host_graph.get_ownedElements().remove(match[label])
282
283     # execute RHS action
284     ((formalisms.transformations.rule.PostConditionPattern)rhs).p_action.execute(match)
```

LISTING C.2: The graph matching algorithm written in Ark action language.

Bibliography

[1] Douglas C. Schmidt. Guest editor's introduction: Model-Driven Engineering. *Computer*, 39(2):25–31, February 2006. ISSN 0018-9162. doi: 10.1109/MC.2006.58. URL http://dx.doi.org/10.1109/MC.2006.58.

[2] Frederick P. Brooks, Jr. No silver bullet: Essence and accidents of software engineering. *Computer*, 20(4):10–19, April 1987. ISSN 0018-9162. doi: 10.1109/MC.1987.1663532. URL http://dx.doi.org/10.1109/MC.1987.1663532.

[3] Steven Kelly and Juha-Pekka Tolvanen. *Domain-Specific Modeling - Enabling Full Code Generation*. Wiley, 2008. ISBN 978-0-470-03666-2.

[4] Ákos Schmidt and Dániel Varró. CheckVML: A tool for model checking visual modeling languages. In Perdita Stevens, Jon Whittle, and Grady Booch, editors, *UML 2003 - The Unified Modeling Language. Modeling Languages and Applications*, volume 2863 of *Lecture Notes in Computer Science*, pages 92–95. Springer Berlin / Heidelberg, 2003. ISBN 978-3-540-20243-1. URL http://dx.doi.org/10.1007/978-3-540-45221-8_8. 10.1007/978-3-540-45221-8_8.

[5] Peter Fröhlich and Johannes Link. Automated test case generation from dynamic models. In Elisa Bertino, editor, *ECOOP 2000 Object-Oriented Programming*, volume 1850 of *Lecture Notes in Computer Science*, pages 472–491. Springer Berlin / Heidelberg, 2000. ISBN 978-3-540-67660-7. URL http://dx.doi.org/10.1007/3-540-45102-1_23. 10.1007/3-540-45102-1_23.

[6] Zef Hemel, Lennart Kats, and Eelco Visser. Code generation by model transformation. In Antonio Vallecillo, Jeff Gray, and Alfonso Pierantonio, editors, *Theory and Practice of Model Transformations*, volume 5063 of *Lecture Notes in Computer Science*, pages 183–198. Springer Berlin / Heidelberg, 2008. ISBN 978-3-540-69926-2.

URL http://dx.doi.org/10.1007/978-3-540-69927-9_13. 10.1007/978-3-540-69927-9_13.

[7] Tom Mens, Krzysztof Czarnecki, and Pieter Van Gorp. A taxonomy of model transformation. In *Proc. Dagstuhl Seminar on "Language Engineering for Model-Driven Software Development". Internationales Begegnungs- und Forschungszentrum (IBFI), Schloss Dagstuhl*. Electronic, 2005.

[8] James Lyle Peterson. *Petri Net Theory and the Modeling of Systems*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1981. ISBN 0136619835.

[9] Tom Mens. *Software Evolution*. Springer, 2008. ISBN 978-3-540-76439-7.

[10] Bart Meyers and Hans Vangheluwe. A framework for evolution of modelling languages. *Sci. Comput. Program.*, 76(12):1223–1246, December 2011. ISSN 0167-6423. doi: 10.1016/j.scico.2011.01.002. URL http://dx.doi.org/10.1016/j.scico.2011.01.002.

[11] Simon Van Mierlo. Evolution of domain-specific languages: A literary study. 2012.

[12] Boris Gruschko, Dimitrios S. Kolovos, and Richard F. Paige. Towards synchronizing models with evolving metamodels. In *In Proc. Int. Workshop on Model-Driven Software Evolution held with the ECSMR*, 2007.

[13] Steffen Becker, Thomas Goldschmidt, Boris Gruschko, and Heiko Koziolek. A process model and classification scheme for semi-automatic meta-model evolution. In *Proc. 1st Workshop MDD, SOA und IT-Management (MSI'07)*. GiTO-Verlag, 2007.

[14] Antonio Cicchetti, Davide Di Ruscio, Romina Eramo, and Alfonso Pierantonio. Automating co-evolution in Model-Driven Engineering. In *Proceedings of the 2008 12th International IEEE Enterprise Distributed Object Computing Conference*, EDOC '08, pages 222–231, Washington, DC, USA, 2008. IEEE Computer Society. ISBN 978-0-7695-3373-5. doi: 10.1109/EDOC.2008.44. URL http://dx.doi.org/10.1109/EDOC.2008.44.

[15] Antonio Cicchetti, Davide Di Ruscio, and Alfonso Pierantonio. Managing dependent changes in coupled evolution. In Richard F. Paige, editor, *Proc. 2nd International Conference on Model Transformation (ICMT'09)*, volume 5563 of *LNCS*, pages 35–51, Zurich, Switzerland, 2009. Springer.

[16] Jokin Garcia and Oscar Daz. Adaptation of transformations to metamodel changes. *Library*, 2:1–9, 2010. URL http://dx.doi.org/10.1007/978-3-642-02408-5_4.

[17] Mark Van Den Brand, Zvezdan Protić, and Tom Verhoeff. A generic solution for syntax-driven model co-evolution. In *Proceedings of the 49th international conference on Objects, models, components, patterns*, TOOLS'11, pages 36–51, Berlin, Heidelberg, 2011. Springer-Verlag. ISBN 978-3-642-21951-1. URL http://dl.acm.org/citation.cfm?id=2025896.2025901.

[18] Markus Herrmannsdoerfer, Sebastian Benz, and Elmar Juergens. COPE - automating coupled evolution of metamodels and models. In *Proceedings of the 23rd European Conference on ECOOP 2009 — Object-Oriented Programming*, Genoa, pages 52–76, Berlin, Heidelberg, 2009. Springer-Verlag. ISBN 978-3-642-03012-3. doi: 10.1007/978-3-642-03013-0_4. URL http://dx.doi.org/10.1007/978-3-642-03013-0_4.

[19] Markus Herrmannsdoerfer, Sander D. Vermolen, and Guido Wachsmuth. An extensive catalog of operators for the coupled evolution of metamodels and models. In *Proceedings of the Third international conference on Software language engineering*, SLE'10, pages 163–182, Berlin, Heidelberg, 2011. Springer-Verlag. ISBN 978-3-642-19439-9. URL http://dl.acm.org/citation.cfm?id=1964571.1964585.

[20] Markus Herrmannsdoerfer and Daniel Ratiu. Limitations of automating model migration in response to metamodel adaptation. In Sudipto Ghosh, editor, *MoDELS Workshops*, volume 6002 of *Lecture Notes in Computer Science*, pages 205–219. Springer, 2009. ISBN 978-3-642-12260-6. URL http://dblp.uni-trier.de/db/conf/models/models2009w.html#HerrmannsdoerferR09.

[21] Joachim Hössler, Michael Soden, and Hajo Eichler. *Coevolution of Models, Metamodels and Transformations*, chapter Coevolution of Models, Metamodels and Transformations. Wissenschaft und Technik Verlag, Berlin, 2005.

[22] Guido Wachsmuth. Metamodel adaptation and model co-adaptation. In Erik Ernst, editor, *Proceedings of the 21st European Conference on Object-Oriented Programming (ECOOP'07)*, volume 4609 of *Lecture Notes in Computer Science*, pages 600–624. Springer-Verlag, July 2007.

[23] Tihamer Levendovszky, Daniel Balasubramanian, Anantha Narayanan, and Gabor Karsai. A novel approach to semi-automated evolution of DSML model transformation. In *Proceedings of the Second international conference on Software Language Engineering*, SLE'09, pages 23–41, Berlin, Heidelberg, 2010. Springer-Verlag. ISBN 3-642-12106-3, 978-3-642-12106-7. doi: 10.1007/978-3-642-12107-4_4. URL http://dx.doi.org/10.1007/978-3-642-12107-4_4.

[24] Jonathan Sprinkle and Gabor Karsai. A domain-specific visual language for domain model evolution. *Journal of Visual Languages & Computing*, 15(3-4):291–307, June 2004. URL http://chess.eecs.berkeley.edu/pubs/746.html.

[25] Markus Pizka and Elmar Jurgens. Automating language evolution. In *Proceedings of the First Joint IEEE/IFIP Symposium on Theoretical Aspects of Software Engineering*, TASE '07, pages 305–315, Washington, DC, USA, 2007. IEEE Computer Society. ISBN 0-7695-2856-2. doi: 10.1109/TASE.2007.13. URL http://dx.doi.org/10.1109/TASE.2007.13.

[26] Sander Vermolen and Eelco Visser. Heterogeneous coupled evolution of software languages. In *Proceedings of the 11th international conference on Model Driven Engineering Languages and Systems*, MoDELS '08, pages 630–644, Berlin, Heidelberg, 2008. Springer-Verlag. ISBN 978-3-540-87874-2. doi: 10.1007/978-3-540-87875-9_44. URL http://dx.doi.org/10.1007/978-3-540-87875-9_44.

[27] Anantha Narayanan, Tihamer Levendovszky, Daniel Balasubramanian, and Gabor Karsai. Automatic domain model migration to manage metamodel evolution. In *Proceedings of the 12th International Conference on Model Driven Engineering Languages and Systems*, MODELS '09, pages 706–711, Berlin, Heidelberg, 2009. Springer-Verlag. ISBN 978-3-642-04424-3. doi: 10.1007/978-3-642-04425-0_57. URL http://dx.doi.org/10.1007/978-3-642-04425-0_57.

[28] Louis M. Rose, Dimitrios S. Kolovos, Richard F. Paige, and Fiona A. C. Polack. Model migration with Epsilon Flock. In *Proceedings of the Third international conference on Theory and practice of model transformations*, ICMT'10, pages 184–198, Berlin, Heidelberg, 2010. Springer-Verlag. ISBN 3-642-13687-7, 978-3-642-13687-0. URL http://dl.acm.org/citation.cfm?id=1875847.1875862.

[29] J. Schönböck W. Retschitzegger W. Schwinger G. Kappel M. Wimmer, A. Kusel. On using inplace transformations for model co-evolution. In *Proceedings of the 2nd International Workshop on Model Transformation with ATL (MtATL 2010)*, 2010.

[30] Kelly Garcés, Frédéric Jouault, Pierre Cointe, and Jean Bézivin. Managing model adaptation by precise detection of metamodel changes. In *Proceedings of the 5th European Conference on Model Driven Architecture - Foundations and Applications*, ECMDA-FA '09, pages 34–49, Berlin, Heidelberg, 2009. Springer-Verlag. ISBN 978-3-642-02673-7. doi: 10.1007/978-3-642-02674-4_4. URL http://dx.doi.org/10.1007/978-3-642-02674-4_4.

[31] Jing Zhang, Jeff Gray, and Yuehua Lin. A generative approach to model interpreter evolution.

[32] Louis M. Rose, Dimitrios S. Kolovos, Richard F. Paige, and Fiona A.C. Polack. Enhanced automation for managing model and metamodel inconsistency. *Automated Software Engineering, International Conference on*, 0:545–549, 2009. ISSN 1527-1366. doi: http://doi.ieeecomputersociety.org/10.1109/ASE.2009.57.

[33] Jean Bézivin, Frédéric Jouault, Peter Rosenthal, and Patrick Valduriez. Modeling in the large and modeling in the small. In *Model Driven Architecture*, pages 33–46. 2005. doi: 10.1007/11538097_3. URL http://dx.doi.org/10.1007/11538097_3.

[34] M. D. Del Fabro, J. Bézivin, F. Jouault, E. Breton, and G. Gueltas. AMW: A generic model weaver. *Proc. of the 1ères Journées sur l'Ingénierie Dirigée par les Modèles*, 2005. URL http://www.sciences.univ-nantes.fr/lina/atl/www/papers/IDM_2005_weaver.pdf.

[35] Markus Herrmannsdoerfer, Sebastian Benz, and Elmar Juergens. Automatability of coupled evolution of metamodels and models in practice. In *Proceedings of the 11th international conference on Model Driven Engineering Languages and Systems*, MoDELS '08, pages 645–659, Berlin, Heidelberg, 2008. Springer-Verlag. ISBN 978-3-540-87874-2. doi: 10.1007/978-3-540-87875-9_45. URL http://dx.doi.org/10.1007/978-3-540-87875-9_45.

[36] Louis Rose, Markus Herrmannsdoerfer, James Williams, Dimitrios Kolovos, Kelly Garcés, Richard Paige, and Fiona Polack. A comparison of model migration tools. In Dorina Petriu, Nicolas Rouquette, and Øystein Haugen, editors, *Model Driven*

Engineering Languages and Systems, volume 6394 of *Lecture Notes in Computer Science*, chapter 5, pages 61–75–75. Springer Berlin / Heidelberg, Berlin, Heidelberg, 2010. ISBN 978-3-642-16144-5. doi: 10.1007/978-3-642-16145-2_5. URL http://dx.doi.org/10.1007/978-3-642-16145-2_5.

[37] Dimitrios S. Kolovos, Richard F. Paige, and Fiona A. C. Polack. The Epsilon Object Language (EOL). In *Proceedings of the Second European conference on Model Driven Architecture: foundations and Applications*, ECMDA-FA'06, pages 128–142, Berlin, Heidelberg, 2006. Springer-Verlag. ISBN 3-540-35909-5, 978-3-540-35909-8. doi: 10.1007/11787044_11. URL http://dx.doi.org/10.1007/11787044_11.

[38] D.S Kolovos. *An Extensible Platform for Specification of Integrated Languages for Model Management*. PhD thesis, University of York, United Kingdom, 2009.

[39] Davide Di Ruscio, Ralf Lämmel, and Alfonso Pierantonio. Automated co-evolution of GMF editor models. In *Proceedings of the Third international conference on Software language engineering*, SLE'10, pages 143–162, Berlin, Heidelberg, 2011. Springer-Verlag. ISBN 978-3-642-19439-9. URL http://dl.acm.org/citation.cfm?id=1964571.1964584.

[40] Dimitrios S. Kolovos, Davide Di Ruscio, Alfonso Pierantonio, and Richard F. Paige. Different models for model matching: An analysis of approaches to support model differencing. In *Proceedings of the 2009 ICSE Workshop on Comparison and Versioning of Software Models*, CVSM '09, pages 1–6, Washington, DC, USA, 2009. IEEE Computer Society. ISBN 978-1-4244-3714-6. doi: 10.1109/CVSM.2009.5071714. URL http://dx.doi.org/10.1109/CVSM.2009.5071714.

[41] Jonathan Sprinkle, Jeffrey Gray, and Marjan Mernik. Fundamental limitations in domain-specific language evolution. *IEEE Transactions on Software Engineering*, 2009.

[42] Louis M. Rose, Richard F. Paige, Dimitrios S. Kolovos, and Fiona A. C. Polack. An analysis of approaches to model migration. In *Proc. Models and Evolution (MoDSE-MCCM) Workshop, 12th ACM/IEEE International Conference on Model Driven Engineering, Languages and Systems*, October 2009.

[43] Juande Lara and Hans Vangheluwe. AToM^3: A Tool for Multi-formalism and Meta-Modelling. In Ralf-Detlef Kutsche and Herbert Weber, editors, *Fundamental Approaches to Software Engineering*, volume 2306 of *Lecture Notes in Computer Science*, pages 174–188. Springer Berlin Heidelberg, 2002. ISBN 978-3-540-43353-8. doi: 10.1007/3-540-45923-5_12. URL http://dx.doi.org/10.1007/3-540-45923-5_12.

[44] Xiaoxi Dong. Ark, the metamodelling kernel for domain specific modelling. Master's thesis, McGill University, 2010.

[45] MOF. Meta Object Facility (MOF) 2.0 core specification. Technical Report formal/06-01-01, OMG, 2001. URL http://www.omg.org/cgi-bin/doc?formal/2006-01-01. OMG Available Specification.

[46] Jelle Slowack. A human textual notation for ArkM3 (work in progress).

[47] Marc Provost. Himesis: A hierarchical subgraph matching kernel for model driven development. Master's thesis, McGill University, 2005.

[48] Simon Van Mierlo. Himesis representation of ArkM3 structures. 2013.

[49] S. Sendall and W. Kozaczynski. Model transformation: The heart and soul of model-driven software development. *Software, IEEE*, 20(5):42–45, 2003. ISSN 0740-7459. doi: 10.1109/MS.2003.1231150.

[50] Thomas Kühne, Gergely Mezei, Eugene Syriani, Hans Vangheluwe, and Manuel Wimmer. Explicit transformation modeling. In Sudipto Ghosh, editor, *Models in Software Engineering*, volume 6002 of *Lecture Notes in Computer Science*, pages 240–255. Springer Berlin / Heidelberg, 2010. ISBN 978-3-642-12260-6. URL http://dx.doi.org/10.1007/978-3-642-12261-3_23. 10.1007/978-3-642-12261-3_23.

[51] Eugene Syriani. *A Multi-Paradigm Foundation for Model Transformation Language Engineering*. PhD thesis, School of Computer Science McGill University Montreal, Quebec, Canada, February 2011.

[52] L.P. Cordella, P. Foggia, C. Sansone, and M. Vento. A (sub)graph isomorphism algorithm for matching large graphs. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 26(10):1367–1372, 2004. ISSN 0162-8828. doi: 10.1109/TPAMI.2004.75.

[53] J. R. Ullmann. An algorithm for subgraph isomorphism. *J. ACM*, 23(1):31–42, January 1976. ISSN 0004-5411. doi: 10.1145/321921.321925. URL http://doi.acm.org/10.1145/321921.321925.

[54] Thomas Kühne. Matters of (meta-)modeling. *Software and Systems Modeling*, 5(4):369–385, December 2006. ISSN 1619-1366. doi: 10.1007/s10270-006-0017-9. URL http://www-adele.imag.fr/users/German.Vega/idm/seance4/MattersOfMetaModeling.pdf.